Microsoft

Upgrading Visual Basic 6.0 Applications to Visual Basic .NET and Visual Basic 2005

patterns & practices

Microsoft®

Upgrading Visual Basic 6.0 Applications to Visual Basic .NET and Visual Basic 2005

patterns & practices

ISBN 0-7356-2298-1

Contents

Preface

Microsoft® Visual Basic® development system is one of the most popular programming languages in use today. Its relative simplicity and power have helped shape modern rapid application development, making it the language of choice for quickly and efficiently building business applications.

Like any widely used product, Visual Basic continues to be improved and adapted as the world of computer technology changes. Visual Basic .NET marks the next evolution of the language.

Visual Basic has always been fairly easy to use and has provided techniques and technologies that were current at the time of each version's release. However, new technologies and techniques have come into use since Visual Basic 6.0 was released, and these are unavailable in this and earlier versions of the language. While object-based programming has been available in the language for years, true object-oriented features have been unavailable. The ability to build distributed applications has been severely limited. Other technologies such as Web services, mobile devices, and .NET have all become prominent in recent years. The need to support these and other features has prompted the release of a new version of Visual Basic.

Visual Basic .NET marks a significant advance for the Visual Basic programming language. As part of the Microsoft .NET Framework, it supports all the latest technologies for building powerful and extensible business applications. Object-oriented programming techniques are now completely supported, making it possible to implement full object-oriented designs. New forms features make it even easier to build rich desktop applications. Moreover, Web forms make it equally easy to create Web applications.

The changes made to Visual Basic .NET have made some features of earlier versions obsolete. Although the new version does provide better ways to achieve the same result as these obsolete features, the changes break compatibility with earlier versions of the language. This makes it impossible to automate the process of upgrading applications created in earlier versions of Visual Basic to the latest version. Though it is possible to automate the process of upgrading applications created in earlier versions of the language to the latest version, entire applications cannot be upgraded through automation alone. Manual intervention will be required to complete the upgrade process.

The purpose of this guide is to provide you with the information you need to upgrade an application from Visual Basic 6.0 to Visual Basic .NET. It describes the tools and techniques that you need to upgrade your application and to start taking advantage of many new features in Visual Basic .NET and the .NET Framework.

Who Should Read This Guide

This guide is intended for software technical decision makers, solution architects, and software developers who are involved in Visual Basic 6.0 application or component development. It helps you understand the issues and risks that go along with upgrading to Visual Basic .NET. It also provides steps for preparing your applications for a successful and cost-effective upgrade. Finally, it gives ideas and pointers about how to advance your application after you successfully upgrade it to Visual Basic .NET.

For Technical Decision Makers

Technical decision makers will find a wealth of information about analyzing and assessing the costs and benefits of upgrading Visual Basic 6.0 applications to Visual Basic .NET and how best to perform this upgrade using proven processes and tools. You will learn how to identify applications or parts of applications that may benefit most from upgrading to .NET. You will also learn about the risks involved and the issues that should be carefully monitored during the upgrade process. Finally, you will learn about Visual Basic. NET improvements that you can apply to your applications after they are successfully upgraded.

For Solution Architects

Solution architects will learn how best to plan and execute the upgrade project. You will learn how to effectively assess the complexity of the upgrade project and estimate the effort that is required to complete the upgrade. You will find information about how to identify the areas of your application that will need substantial manual intervention to complete the upgrade. Furthermore, you will find a detailed discussion about the best practices for completing a successful upgrade project and how this maps to tasks and milestones within your own project.

For Developers

Developers will find information about many Visual Basic 6.0 features and how they are upgraded to Visual Basic .NET. A step-by-step process describes the most effective strategies for completing a successful upgrade. A discussion of available tools that automate some of the process is provided. You will also find information about features that are not automatically converted and suggestions for how best to manually convert these features. Code samples are provided throughout the text to provide practical examples of Visual Basic 6.0 code that is not automatically upgraded and examples of equivalent Visual Basic .NET code that you can use to replace it. Finally, you will find ideas and pointers for improving your applications after they are upgraded.

Prerequisites

As mentioned earlier, this guide is intended for software developers, solution architects, and technical decision makers who are considering or who have already decided to upgrade Visual Basic 6.0 applications to Visual Basic .NET. For this reason, this guide assumes that you have a certain degree of knowledge and experience with Visual Basic 6.0. Also, you should be generally familiar with the .NET Framework, Visual Basic .NET in particular.

This is not a how-to guide for Visual Basic 6.0, .NET application architecture, or Visual Basic .NET. The guide contains introductory explanations of many of the features of Visual Basic 6.0, Visual Basic .NET, and the .NET Framework in general, but it provides these explanations only to create a context in which techniques for upgrading between these technologies can be more appropriately discussed. If you need more information about these technologies, see the "More Information" section at the end of each chapter.

How to Use This Guide

Most chapters in this guide are independent and can be read in any order. However, it is recommended that you read in order Chapter 1, "Introduction" and Chapter 2, "Practices for Successful Upgrades." These two chapters lay the important procedural groundwork for the upgrade process that serves as a foundation for the rest of the guide.

Because this guide targets a diverse audience, from architectural specialists who oversee system design and integration to software developers who do the actual coding, readers will undoubtedly get more value from some chapters than others. The list that follows summarizes the key points of each chapter to help you determine which are more likely to focus on your particular areas of interest.

This guide has been divided into the following four parts:

- Part 1, "General Upgrade Practices," consists of Chapters 1 – 3, and focuses on the upgrade practices that are needed to make any upgrade project successful. Although the emphasis is on Visual Basic 6.0 to Visual Basic .NET upgrades, much of the information can be applied to any upgrade project. Part 1 includes the following chapters:
  - Chapter 1, "Introduction," looks at the common reasons for upgrading to Visual Basic .NET and examines some of the strategies that you might consider to ensure a successful and cost effective upgrade.
  - Chapter 2, "Practices for Successful Upgrades," presents a proven methodology that covers each step of the upgrade process from planning and application preparation, through upgrading, and on to testing and quality assurance in .NET.

- Chapter 3, "Assessment and Analysis," examines the first step of the upgrade process which focuses on assessing the scope and complexity of the planned upgrade project.

- Part 2, "Understanding the Upgrade Process," consisting of Chapters 4 – 6, provides a more technical overview of upgrading your code to Visual Basic .NET. Part 2 includes the following chapters:
  - Chapter 4, "Common Application Types," examines the different types of Visual Basic applications and presents suggestions for how to approach the upgrade of these applications, shortcuts that can be taken, and some of the difficulties and issues that can arise and should be kept in mind.
  - Chapter 5, "The Visual Basic Upgrade Process," looks more closely at the preparations and steps of performing an upgrade. The chapter includes the detailed steps involved in using the Visual Basic Upgrade Wizard, which is an add-in to the Microsoft Visual Studio® .NET development system that performs much of the upgrade work for you. This chapter shows you how to prepare your application so that your use of the upgrade wizard is more effective and the amount of code that is automatically upgraded is maximized.
  - Chapter 6, "Understanding the Visual Basic Upgrade Wizard," describes in detail the capabilities and limitations of the Visual Basic Upgrade Wizard.
- Part 3, "Manual Upgrade Tasks," consists of chapters 7 – 16, provides detailed information on those aspects of upgrading that cannot be performed automatically by the upgrade wizard and therefore require manual effort. Part 3 includes the following chapters:
  - Chapter 7, "Upgrading Commonly-Used Visual Basic 6.0 Objects," catalogs many of the objects used in typical Visual Basic 6.0 applications that are not supported in Visual Basic .NET. It gives techniques for upgrading these objects manually and suggestions for workarounds and alternatives to enable you to complete the upgrade successfully.
  - Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Features," identifies additional Visual Basic 6.0 constructs that have changed or were omitted in Visual Basic .NET. The chapter provides information on the changes that have taken place and on how to deal with them when you upgrade an application.
  - Chapter 9, "Upgrading Visual Basic 6.0 Forms Features," concentrates on the differences between Visual Basic 6.0 forms and the Windows Forms package that is used in Visual Basic .NET. It identifies upgrade problem areas associated with forms and provides information on how to overcome them.
  - Chapter 10, "Upgrading Web Applications," provides information about how to upgrade Web-based Visual Basic 6.0 applications.
  - Chapter 11, "Upgrading String and File Operations," gives information that deals specifically with string and file manipulation. It covers the types of

operations that are automatically upgraded when you apply the upgrade wizard, as well as new operations that Visual Basic .NET offers.

- Chapter 12, "Upgrading Data Access," addresses upgrading data access based on Data Access Objects (DAO), Remote Data Objects (RDO), and ActiveX Data Objects (ADO). It shows techniques that you can use to accelerate the upgrade of database access code that is not handled by the upgrade wizard.
- Chapter 13, "Working with the Windows API," discusses common issues that arise when you upgrade Visual Basic 6.0 applications that use Windows API function calls. It demonstrates how to replace common Windows API function calls with new methods that Visual Basic .NET provides.
- Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET," presents options for interoperability between these two versions of the language. It addresses techniques to access Visual Basic 6.0 components from Visual Basic .NET assemblies and to access .NET assemblies from clients built in Visual Basic 6.0. It covers many issues that arise when you use Component Object Model (COM) to interoperate between these two languages and how COM works in unmanaged and managed environments.
- Chapter 15, "Upgrading MTS and COM+ Applications," discusses how to upgrade Microsoft Transaction Services (MTS) and COM+ components. It also describes some of the deprecated functionality and how to deal with it when you upgrade an application.
- Chapter 16, "Application Completion," covers some of the remaining areas of your application that are not core parts of your application's functionality, but are fundamental to having a finished product that provides the user with a complete application. Topics include upgrading integrated help and product deployment.

- Part 4, "Beyond the Upgrade," consists of chapters 17 – 21 and covers what to do after you complete the upgrade. Part 4 includes the following chapters:
  - Chapter 17, "Introduction to Application Advancement," looks at advancement options that you can use after the application has reached functional equivalence in Visual Basic .NET. The information in this chapter includes suggestions for advancing the architecture, design, and implementation of your applications.
  - Chapter 18, "Advancements for Common Scenarios," suggests advancements that are based on specific application types, such as forms applications and enterprise services.
  - Chapter 19, "Advancements for Common Web Scenarios," is a continuation of Chapter 18 that focuses on advancements for Web-based applications.

- Chapter 20, "Common Technology Scenario Advancements," focuses on advancements that apply to almost all types of applications. These advancements include enhancements to application security, manageability, performance and scalability, and communication and state management.
- Chapter 21, "Testing Upgraded Applications," provides detailed information about strategies that you can use to verify the correctness and completeness of your application after it is upgraded and after you have implemented the advancements that you want.

Relevant Chapters for Technical Decision Makers

Technical decision makers will be most interested in information about assessment and analysis, the risks and challenges of the upgrade, and application advancement. The following chapters contain this information:

- Chapter 1, "Introduction"
- Chapter 3, "Assessment and Analysis"
- Chapter 7, "Upgrading Commonly-Used Visual Basic 6.0 Objects"
- Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features"
- Chapter 9, "Upgrading Visual Basic 6.0 Forms Features"
- Chapter 10, "Upgrading Web Applications"
- Chapter 11, "Upgrading String and File Operations"
- Chapter 12, "Upgrading Data Access"
- Chapter 17, "Introduction to Application Advancement"
- Chapter 18, "Advancements for Common Scenarios"
- Chapter 19, "Advancements for Common Web Scenarios"
- Chapter 20, "Common Technology Scenario Advancements"

Relevant Chapters for Solution Architects

Solution architects will be most interested in the information about the recommended upgrade process, assessment and analysis, best practices for managing an upgrade project, the risks associated with this process, and application advancement. The following chapters contain this information:

- Chapter 2, "Practices for Successful Upgrades"
- Chapter 3, "Assessment and Analysis"
- Chapter 4, "Common Application Types"
- Chapter 7, "Upgrading Commonly-Used Visual Basic 6.0 Objects"
- Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features"
- Chapter 9, "Upgrading Visual Basic 6.0 Forms Features"

- Chapter 10, "Upgrading Web Applications"
- Chapter 11, "Upgrading String and File Operations"
- Chapter 12, "Upgrading Data Access"
- Chapter 13, "Working with the Windows API"
- Chapter 17, "Introduction to Application Advancement"
- Chapter 18, "Advancements for Common Scenarios"
- Chapter 19, "Advancements for Common Web Scenarios"
- Chapter 20, "Common Technology Scenario Advancements"

Relevant Chapters for Software Developers

Software developers will be most interested in the material that covers how the technical features of Visual Basic 6.0 map to Visual Basic .NET. This includes the features that are converted automatically by existing tools and those that must be manually upgraded. Furthermore, developers are likely to be interested in learning how they can advance applications with features of the .NET Framework. The following chapters contain this information:

- Chapter 4, "Common Application Types"
- Chapter 5, "The Visual Basic Upgrade Process"
- Chapter 6, "Understanding the Visual Basic Upgrade Wizard"
- Chapter 7, "Upgrading Commonly-Used Visual Basic 6.0 Objects"
- Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features"
- Chapter 9, "Upgrading Visual Basic 6.0 Forms Features"
- Chapter 10, "Upgrading Web Applications"
- Chapter 11, "Upgrading String and File Operations"
- Chapter 12, "Upgrading Data Access"
- Chapter 13, "Working with the Windows API"
- Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET"
- Chapter 17, "Introduction to Application Advancement"
- Chapter 18, "Advancements for Common Scenarios"
- Chapter 19, "Advancements for Common Web Scenarios"
- Chapter 20, "Common Technology Scenario Advancements"

Document Conventions

This guide uses the style conventions and terminology shown in Table 1 on the next page.

Table 1: Document Conventions

Element	Meaning
bold font	Characters that you type exactly as shown, including commands and switches appear in bold font. Programming elements, such as methods, functions, data types, and data structures appear in bold font (except when part of a code sample, in which case they appear in monospace font). User interface elements are also bold.
Italic font	Variables for which you supply a specific value. For example, *Filename.ext* could refer to any valid file name for the case in question. New terminology also appears in italic on first use.
`Monospace font`	Code samples.
%SystemRoot%	The folder in which the Windows operating system is installed.

Feedback and Support

Every effort has been made to ensure the accuracy of the content of this guide. All sample code and procedures have been independently tested. Also, all references and descriptions of tools, specifications, and standards have been checked and are believed to be accurate at the time this guide is published.

If you have any comments or suggestions, or if you find any inaccuracies, please send them by e-mail to vbmigfb@microsoft.com.

Note: There is a community site for this guide on GotDotNet. This site contains a "Downloads" section where you can obtain the Visual Basic 6.0 Upgrade Assessment Tool that is described in this guide. It also contains forums where you can post feedback, ask questions, or provide suggestions.

Principal Authors

The content for this guide and the accompanying Visual Basic 6.0 Upgrade Assessment Tool were developed by the following individuals at ArtinSoft:

- Project Lead: Federico Zoufaly
- Guidance: César Muñoz , Paul Dermody, Manfred Dahmen, Ronny Vargas, Hendel Valverde, José David Araya, Oscar Calvo, Allan Cantillo, Alvaro Rivera, Christian Saborío, Juan Fernando Peña, Xavier Morera, Iván Sanabria
- Assessment tool development: Rolando Méndez

Contributors

The Visual Basic 6.0 Upgrade Assessment Tool and the *Upgrading Visual Basic 6.0 to Visual Basic .NET and Visual Basic 2005* guide were produced with the help of the following people.

- Program Manager: William Loeffler (Microsoft Corporation)
- Product Manager: Eugenio Pace (Microsoft Corporation)
- Architect: Keith Pleas (Guided Design)
- Test: Edward Lafferty (Microsoft Corporation); Ashish Babbar, Terrence Cyril J., Manish Duggal, Chaitanya Bijwe, Arumugam Subramaniyam, Umashankar Murugesan, Dhanaraj Subbian, Tarin R. Shah, Dipika Khanna, Gayatri Patil, and Sandesh Pandurang Ambekar (Infosys Technologies Ltd)
- Documentation and Samples: RoAnn Corbisier (Microsoft Corporation); Tina Burden McGrayne and Melissa Seymour (TinaTech Inc.); Sharon Smith (Linda Werner & Associates Inc.); Francisco Fernandez and Paul Henry (Wadeware LLC)

Acknowledgements

Many thanks to the following individuals who provided invaluable assistance during the development of this guide:

- Dan Appleman (Desaware Inc.)
- Joe Binder (Microsoft Corporation)
- Rob Copeland (Microsoft Corporation)
- Jackie Goldstein (Renaissance Computer Systems Ltd.)
- Ed Hickey (Microsoft Corporation)
- Billy Hollis
- Edward Jezierski (Microsoft Corporation)
- Chris Kinsman (Vergent Software)
- Deborah Kurata (InStep Technologies)
- Julia Lerman (The Data Farm)
- Rockford Lhotka (Magenic Technologies)
- Christian Nielsen (Volvo Information Technology AB)
- Jay Roxe (Microsoft Corporation)
- Jay Schmelzer (Microsoft Corporation)
- Scott Swigart (Swigart Consulting)
- The Visual Basic MVPs (Microsoft Valued Professionals)

More Information

For more information about the Visual Basic 6.0 Upgrade Assessment Tool and to download it, see "Visual Basic 6 to Visual Basic .NET Migration Guide" on GotDotNet:
http://www.gotdotnet.com/codegallery/codegallery.aspx?id=07c69750-9b49-4783-b0fc-94710433a66d.

For more information about ArtinSoft, see the ArtinSoft Web site:
http://www.artinsoft.com.

1

Introduction

Before investing time, money, and energy into an upgrade project, it is important to understand why an upgrade is beneficial. It is equally important to understand potential upgrade strategies to minimize costs and risks. This chapter presents the benefits of upgrading Visual Basic 6.0 applications to Visual Basic .NET, and provides several strategies that you can apply to successfully upgrade your application.

Why Consider an Upgrade Project?

Before deciding to upgrade an application from Visual Basic 6.0 to Visual Basic .NET, it is helpful to understand why you should consider upgrading at all. This section discusses why and when it makes sense to upgrade your application.

Usually, you have business requirements that compel you to upgrade an application from Visual Basic 6.0 to Visual Basic .NET. Common reasons, or drivers, to upgrade an application to Visual Basic .NET are:

- To Web-enable the application, or enhance an existing Web-enabled application with ASP.NET features, such as tracing, flexible state management, scaleable data access, and improved performance.
- To take advantage of the improvements in developer productivity and the enhanced development features that Visual Basic .NET, the .NET Framework, and Visual Studio .NET provide — particularly if new functionality is required that is more easily implemented in Visual Basic .NET, such as Web services.
- To consolidate your company's software assets. For example, if your new applications are built in Visual Basic .NET, there is often a need to upgrade other applications built with earlier versions of Visual Basic. This improves system integration and reduces the overhead of needing experts on different platforms.

- To reduce the cost of ongoing business activity. For example, the increased scalability and performance of applications that are upgraded to Visual Basic .NET, and the increased productivity of developers, can reduce the costs of regular business activity.
- To improve the maintenance of an application in any of these situations:
  - Your business does not have an in-house expert on the application.
  - Your business has a high turnover of staff.
  - Your business does not have enough resources to support the application.
  - The available documentation is limited or outdated.

Of course, a combination of these drivers may influence your decision to upgrade. These drivers can be seen as a kind of spectrum, going from *push* factors (factors that compel you to upgrade the Visual Basic application to fit a new environment) to *pull* factors (factors that result from the business wanting to seize an opportunity to grow, reach new customers, or add products and services).

Minimum Information Needed to Make a Decision

To determine whether upgrading your application is appropriate, you should perform a project feasibility analysis. The first step of your project feasibility analysis is to collect information about the current status of your Visual Basic 6.0 applications. To make the best possible decision about whether to upgrade the application to Visual Basic .NET, your analysis should address the following information.

- **Project goals and priorities**. Be sure to have clearly documented project goals, with a list of priorities for each goal. Make a point to clarify how an upgrade will meet these goals. Some questions that you want answered before making an upgrade decision are the following:
  - What is the expected life of the application? Is this a temporary solution or something more long term? An application with a short expected lifetime because of changing business needs or processes is unlikely to be worth upgrading. However, an application whose functionality remains core to the business and will remain so for a period that is sufficiently longer than the length of the upgrade project is a good candidate for upgrading.
  - When is the upgraded application needed for a production environment? If the upgrade project will take longer than the schedule allows, it may be more appropriate to replace the application.
  - Do you have realistic expectations for improvements in performance, security, scalability, and other potential benefits? Do you also understand that additional modifications may be required to achieve these improvements? It is important to realize that the upgrade process alone may not improve the performance or scalability of your application. Improvements like these often

occur during application advancement, which is discussed in Chapter 20, "Common Technology Scenario Advancements."

- How many users will this application support in its lifetime? Do you clearly understand how Visual Basic .NET will help meet increased user demand? Changes to applications with a large user base will have a significant impact on those users. However, the impact may be a positive one if the changes are for the better. Improved performance features and modern technologies available in Visual Basic .NET may improve the application experience for the user.
- Do you plan to add new features to the application, such as integration with other systems or access from the Web? Again, these improvements are part of application advancement and require effort beyond upgrading the application.

- **Business value of the applications**. Is the functionality that is provided by this application unique, or are there third-party applications that can be used to do the same job? Be sure that you can demonstrate the specialized nature of the application you propose to upgrade.

  A related question is whether the application is a reasonable fit to your business needs. You need to demonstrate that the application does provide a function that fits the needs of your business. If the application has been around for a long time, you need to make sure that the functionality of the application is not outdated, and that it still closely fits the processes and operations required by your business.
- **Development environment**. Moving to Visual Basic .NET also means updating your development environment. Your developers will have to learn a new integrated development environment (IDE), and a new language. You will also have to purchase software licenses for the tools you need to perform the upgrade and to maintain the upgraded code.
- **Developer skills**. Do your developers have knowledge or experience with the .NET Framework in general and with Visual Basic .NET specifically? Although upgrading an application is a good way to learn a new language, you should ensure that your developers have basic knowledge of the .NET Framework and Visual Basic .NET before beginning the upgrade project.
- **Quality assurance environment**. It is likely that the upgrade project will affect your quality assurance procedures, especially if they are automated. Make sure that you also consider any effort that will be required to upgrade your testing environments.
- **Application architecture and complexity**. Most applications for large enterprises have multiple dependencies on other systems within the company, as well as on systems and libraries from third-party vendors. Make sure you understand the nature of these dependencies and consider any changes that the systems require before they will work with the upgraded application.

Also, consider the effects that an upgraded application will have on other applications that depend on it. How will they be affected by the move to Visual Basic .NET?

Finally, is the application of good quality? Good quality indicates both its quality as a business tool and the quality of the source code that implements it. This is an assessment of the effective value of the application from the view of the user and the developer. The user requires an application that is stable, that does the job well, and that is easy to use. A developer requires an application with code that is understandable, well-documented, and not unnecessarily complex.

Making the following three assessments will help you gather the information that you need to make a decision about upgrading your application. They will also help you accurately estimate the effort that is needed to select and complete the appropriate upgrade process:

- Business value assessment
- Code quality assessment
- Development environment assessment

Business Value Assessment

When you are considering investing time and money into an upgrade project, it is important to evaluate the importance of the existing application's functionality to your business. Is there opportunity for return on investment if you do upgrade the application? This can be shown by determining how much has been invested in the application so far. Also, you might highlight unique features of the application that cannot be replaced by a third-party off-the-shelf product. Answering these questions will help you determine the value of the application to your business:

- Are the application's basic inputs and outputs based on legacy file formats? Applications with inputs and outputs based on legacy file formats that are not easily translatable may be problematic to upgrade. On the other hand, moving an application to Visual Basic .NET and upgrading data to a more general format, such as XML, may leverage your application for future enhancements or changes in technology.
- What kind of data does the application process, and are there other applications that can do the same job? Applications that are responsible for processing important business data are highly valuable to your business. If there are no other applications, such as third-party off-the-shelf products, available to process this data, ensuring the lifespan of your application by upgrading it to Visual Basic .NET may be a logical step.
- How does this application interact with its users, whether they are human or computer-based clients? Providing a clean, friendly interface for human users increases the value of an application. New or improved user interface features of

a new language version, such as Visual Basic .NET, may increase user appreciation. Similarly, an application that coordinates with other applications must have clean data transmission. It should use dependable data formats and communication protocols. Newer languages often support improved transmission protocols or data formats that make inter-application communication simpler and more reliable.

- What other systems depend on your application, and are there third-party products that could provide the same data or function? If other systems are dependent on your application, and there are not any third-party products to replace your application, the value of your application is high. Ensuring the future of the application becomes more important in this situation.
- What business rules are inherent to your application that are difficult to reproduce elsewhere? An application with a large number of business rules that are unavailable in other applications is very valuable to a company.
- What role does this application play in current business processes in your company? What would happen to those processes if the application was removed? An application that automates a small number of mundane tasks is not likely to be very valuable. In contrast, an application that processes large amounts of business data will be highly valuable. Even applications designed to assist employees by automating a large number of data-oriented tasks can be valuable, if such an application improves user efficiency.

Another driver for upgrading an application is reducing the total cost of ownership (TCO). The .NET Framework provides many features that increase developer productivity, while also lowering deployment and maintenance costs to decrease TCO.

Code Quality Assessment

An important factor in determining whether to upgrade your application is the current quality of its code design and implementation. Detailed analysis of your application's source code will provide information on code quality, design complexity, and component dependencies. You can use the Visual Basic Upgrade Assessment Tool to help capture some of this information. For a complete description of the tool, see Chapter 3, "Assessment and Analysis." The Visual Basic Upgrade Assessment Tool is provided with this guide on the companion CD or you can download it from the GotDotNet community site.

The assessment tool was created to help you estimate the effort that is required to upgrade an application, but it can also help you understand the structural complexity of the application and its design quality. For this discussion, design quality indicates the suitability of the code implementation given the functionality that is required by the application. For example, applications that have gone through multiple iterations over several years with different programmers often contain excessive redundant dependencies and unused code. This indicates poor design quality.

The assessment tool collects the following metrics:

- **Size metrics**. These include metrics such as the number of lines of code, forms, designers, modules, classes, components, user controls, and data sources.
- **Usage metrics**. These include functions, types, and properties.
- **Complexity metrics**. The assessment tool recognizes the same things that cause the upgrade wizard to generate errors, warnings, and issues (EWIs). These EWIs imply specific modifications that are required to resolve them. The assessment tool assigns complexities to these EWIs. This helps you estimate the work that needs to be performed to remedy all of the reported issues.
- **Dependency diagrams**. The assessment tool generates the call graphs for each function and the resulting dependencies between modules and files. In addition, the tool lists missing libraries and files. Use this information to determine possible upgrade roadmaps — that is, the order in which modules should be upgraded.

There are many types of applications and different technologies that can be used to build them. If you are considering upgrading your applications with the Visual Basic Upgrade Wizard, you should be aware of some of the tool's limitations. The following types of applications are a sample of the those with features that are only partially supported, or are not supported at all, by the upgrade wizard and some that are not supported at all by Visual Basic .NET:

- A complex distributed application with several different layers of objects that communicate through COM.
- A Visual Basic 5.0 or earlier application that has not made the transition to Visual Basic 6.0.
- An Internet project that uses Microsoft ActiveX® technologies, Web classes, or DHTML. None of these development technologies is supported in Visual Basic .NET.
- A database project that is based on the Data Environment Designer, which is also not supported in Visual Basic .NET.
- A database project that uses data binding to bind data sources to controls.
- An ActiveX control or ActiveX DLL project.
- A database client application with user controls.

Although these features and applications are not directly supported in Visual Basic .NET, there is always an alternative (and often better) way to achieve the same functionality within Visual Basic .NET. A more complete list of features that are not automatically upgraded by the upgrade wizard is given in Chapters 7 through 11.

If your applications fit into any of these scenarios, your project plan should allow time to identify and analyze upgrade alternatives for the technologies involved. You should also allocate time to review, re-design, and realize your target architecture.

Code quality is also affected by the status of the development work on the original application. If any development initiatives are currently in progress, the code will be less stable. The code may contain bugs, incomplete features, or even invalid code that will not compile.

The best time to upgrade is when the application code base is stable but is due for enhancements. This strategy gives you the opportunity to combine the upgrade effort with feature enhancements. In contrast, it is best to avoid upgrading your applications when undergoing a period of intense change, unless you have a very strongly controlled code management process. Upgrading a large application may take several weeks or even months. During this time, changes made to the original code base will also be required in the upgraded code base. The process of maintaining two code bases that are each undergoing changes is prone to error and is likely to result in a disparity between the two versions of the application. If you decide that it is necessary to make changes to the original applications while you also upgrade them, it is recommended that you minimize disruption by choosing modules to upgrade that are stable and that progressively cover all of your application.

Code quality can affect the decision to upgrade in many ways. A stable application with a well designed, implemented, tested, and documented code base that has drifted away from the business needs (either through revisions in the application or changes to the business) is a prime candidate for upgrade. On the other hand, such a stable application that does fulfill the business needs and does not require improvements is probably better left in its original form. An application of poor quality with a weak design, complex implementation, and/or poor documentation is also a good candidate for an upgrade. Upgrading an application of poor quality to a new development language, such as Visual Basic .NET, gives you the opportunity to correct these shortcomings. This will make it easier to maintain and will possibly advance the application.

Development Environment Assessment

To accurately assess your development environment, you need to consider the skills of your development team. The productivity of developers in an upgrade project depends on their knowledge of the existing source code and the target technologies. It is necessary to assess whether the team's current skills are capable of solving each issue that is found during the upgrade analysis. In general, it is recommended that the developers undergo full training in Visual Basic .NET before they attempt an upgrade project.

You must also consider other factors, such as the existence of test cases and how well your development team understands the application that will be upgraded. If the application does not have well defined test cases, it will be more difficult to demonstrate the completion of an upgraded application. When test cases do exist, they may

also need to be upgraded to Visual Basic .NET. This is especially true if the testing process is performed with automated tools.

If the developers do not have previous knowledge of the application, you must factor extra time for them to understand the design and implementation of the application. The time that is required also increases if the application does not have an adequate set of documentation.

Upgrade, Reuse, Rewrite, or Replace?

When it is necessary to update an application, you have four broad options to choose from: upgrade, reuse, rewrite, or replace. These options are described here:

- **Upgrade**. Apply the upgrade process, adding functionality and business reach as required. The advantages of upgrading are that a great deal of your previous investment is leveraged (especially in the areas of business logic). Also, compared to rewriting your application, an upgrade is likely to introduce fewer bugs, particularly when you use automated upgrade processes.
- **Reuse**. Instead of updating the application, you can add a layer over the application to enable it to interoperate with newer Visual Basic .NET systems in your organization. However, if you are updating your application for consolidation reasons, reuse is not an option because with it you will have to maintain both Visual Basic 6.0 and Visual Basic .NET systems.
- **Rewrite**. Sometimes the only way to update an application is to start over and completely rewrite it. This means disposing of the original application and starting again from scratch. If you determine that gradual changes in your business operating procedures have caused the application to slowly lose relevance and it no longer serves your business needs, you may consider rewriting most or all of the application. Another reason for rewriting the application is that it has become overly complex with un-maintainable, redundant code, and has out-of-date documentation. This can happen when multiple versions of an application were written by different developers who are no longer available to work on the latest version.

  The advantage to rewriting an application is that you have the freedom to create a completely new design with the most current technologies. The disadvantage is that writing a new application, even if it is a redevelopment of an existing application, is more costly and more risky than the other options. Also, when you rewrite a new application, you are effectively discarding all investment in the existing application.
- **Replace**. Look for a suitable package to replace the application, or outsource the work it does. Be prepared to make changes to your business model to meet the package half-way because it is unlikely that it will exactly suit your current needs.

Use the following lists of indicators to help you choose an option for updating all or part of your application:

- **Indicators for choosing to upgrade**. These include:
  - The existing application is a good fit with business needs.
  - The existing application needs only moderate functionality changes.
  - A new application would require adding significant functionality to it, and it would need to be closely integrated with the existing application.
  - High operational costs are associated with the existing application.
  - It is necessary to upgrade to .NET for strategic reasons.
  - A future vision of the application includes Web services.
  - It is difficult and costly to find resources to maintain the existing application.
- **Indicators for choosing reuse**. These include:
  - The business rules are satisfactory.
  - The existing application has low operational costs.
  - It is difficult to separate logic from persistent data and presentation layers.
  - Only simple Web access is required, which allows you to use a wrapping solution
  - You have sufficient resources to maintain core legacy code.
  - Off-the-shelf software is central to the existing application, and it is possible to rely on third-party support and maintenance.
- **Indicators for choosing rewrite**. These include:
  - The business rules are satisfactory but you will need to add additional functionality to the application.
  - The available off-the-shelf solutions do not come close to meeting your needs.
  - The existing application has poor quality code.
  - The existing application has high maintenance costs.
  - The time, cost, and disruption of rewriting the application are acceptable.
- **Indicators for choosing replace**. These include:
  - The existing application is significantly out of line with business needs.
  - Making changes to the business model to fit an off-the-shelf solution is acceptable.
  - The time, cost, and disruption of replacing the application are acceptable.

Figure 1.1 on the next page provides a visual summary of these indicators. To use this graph, you need to first determine where your current application stands in terms of business value and quality.

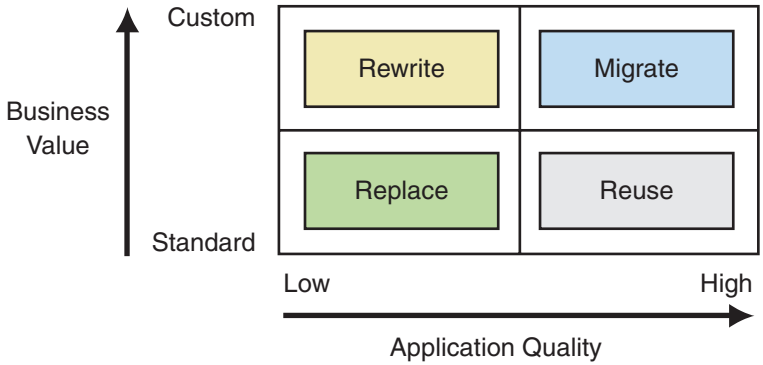

Figure 1.1
A visual guide for deciding when and how to upgrade an application

Business value is the vertical access on the left. If your application provides some level of "standard" functionality that is easily found in other applications, particularly third-party off-the-shelf products, it will be located lower down on this axis. If your application is a custom built application whose functionality is unique to your business needs, it will appear higher up on this axis.

The horizontal axis at the bottom represents the quality of an application from the points of view of both the user and the developer. If your application is difficult to use, slow, unstable, or gives inconsistent results, the user will see it as a low quality application. If the code is complex and difficult to maintain, the developer will see it as low quality. In both cases, the application will be located to the left of the application quality axis. If both users and developers are happy with the application, it will be located to the right of this axis.

After you determine where the application stands on both axes, you can determine the best course of action for updating your application. In general, applications that provide "standard" functionality are candidates for replacement or reuse while applications that provide "custom" functionality are candidates for rewriting or upgrading. Low quality applications should be replaced or rewritten while high quality applications should be reused or upgraded.

When indicators are in favor of an upgrade, choosing to upgrade to Visual Basic .NET will best leverage the existing code base. For more information about some of these benefits, see the "Moving from Visual Basic 6.0 to Visual Basic .NET" section later in this chapter.

Partial Upgrade

So far, this chapter has discussed the updating of your application as an all or nothing undertaking. You can upgrade it or you can leave it as it is. You can rewrite it or you can replace it. However, for many applications it is not a straightforward question of choosing one of the four options previously listed. In some cases, you

need to look at a combination of the upgrade, reuse, rewrite, and replacement options. For example, a combination of upgrading and reusing the application is possible and involves upgrading only part of your application. You can use standard interoperability mechanisms to integrate the new and legacy components. This approach is known as a partial upgrade and it makes sense when you determine that certain parts of your application will benefit from the move to Visual Basic .NET, but it is difficult to justify the move for others. In particular, if you examine the indicators previously listed and find that the indicators for choosing to upgrade apply to part of your application while the indicators for choosing to reuse apply to other parts, you may want to consider a partial upgrade.

If you determine that a partial upgrade is the way to update your application, you might find that you will have less upgrade work to do because you are not upgrading your entire application. However, you may have additional work to do to integrate your new application with the legacy components that have been left in Visual Basic 6.0. For more information about interoperability, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET."

Upgrade Strategies

If you decide that the upgrade option is the right approach to update your application, you also need to choose an appropriate upgrade strategy. For information about possible strategies, see the "Selecting an Upgrade Strategy" section in Chapter 2, "Practices for Successful Upgrades."

Moving from Visual Basic 6.0 to Visual Basic .NET

It is important to understand when to upgrade to Visual Basic .NET, and it is just as important to understand why you should upgrade. It is important to understand the benefits of upgrading before you begin. All of the features in Visual Basic 6.0 do not have a one-to-one mapping with features in Visual Basic .NET, and you cannot automatically upgrade all of the code. A natural question then arises: What reasons are there for upgrading functional Visual Basic 6.0 applications to Visual Basic .NET? To fully understand the advantages of upgrading, it is necessary to first understand the benefits of Visual Basic .NET, the .NET Framework, and the standard .NET integrated development environment, Visual Studio .NET.

Increased Productivity

The Visual Basic .NET language contains many new language features that increase developer productivity. This section explores some of these features, and explains how they benefit application development.

Visual Basic .NET is built around the .NET Framework. Many of the benefits of Visual Basic .NET are derived directly from the .NET Framework itself.

Microsoft .NET Framework is based on a specification named the Common Language Specification (CLS). This specification defines the rules that determine a minimum set of features that all .NET languages, including Visual Basic, must have. Included among this minimum set of features are certain object-oriented features, such as inheritance and exception handling. These features are not available in earlier versions of Visual Basic. It also specifies which format the compilers create for the executables, and it ensures that your application will run on any .NET engine on any platform without recompilation.

The result is that code is compiled to a special format named Microsoft Intermediate Language (MSIL) instead of the usual platform-specific machine code. MSIL executables are then compiled to a platform-specific machine code at run time through just-in-time (JIT) compilation. The benefit of this approach is that a single compilation of your application is optimized at run time based on the actual processor and/or operating system on which the code is currently executing. For example, a .NET application is optimized and run as a 32-bit application on the standard 32-bit edition of the Microsoft Windows® XP operating system. The same application is optimized and run as a 64-bit application, without the need for recompilation, on the 64-bit edition of the Microsoft Windows Server™ 2003 operating system. Because this is a published specification, it is possible that new processors and operating systems can have .NET run-time engines created for them as they are released. This means that an application written for the .NET Framework can run on and be optimized for new hardware or operating systems without any recompilation or upgrading of the application. A multi-platform version of an application or component means developers can solve new problems, instead of investing time porting existing components.

Additionally, the MSIL contains metadata about the classes that it contains. The result is that developers do not have to create "include files," type libraries, Interface Definition Language (IDL) code, or any other additional files to use these classes in another application. The client application can automatically extract this information from the MSIL. This makes sharing and using the MSIL a lot easier. Productivity increases because it is easier to take advantage of existing classes.

The main component of the CLS implementation is named the common language runtime (CLR), which is the engine on which all .NET applications run and which enables the interoperability and integration between all .NET languages. The management of resources (such as memory, threads, exceptions, and security) for .NET applications is handled automatically by the CLR. Applications that are built using only .NET technologies are referred to as managed applications or managed code. The term *managed* derives from the fact that resources in such applications are

managed by the CLR. Developers who build managed code benefit from the delegation of responsibilities to the CLR because it frees them to focus on the application logic instead of low level logic of resource management. This allows software developers to be far more productive because they can concentrate on solving business problems.

The data typing rules are defined by a section of the CLS named the common type system (CTS). This specification defines all of the .NET data types and their behaviors. As a result, all languages in .NET share a common set of base data types whose storage requirements and functional behaviors are identical across all languages. For example, a string is the same regardless of whether it is defined in a Visual Basic .NET module, a managed Visual C++® library, a Visual C#® component, or a Visual J#® package. The benefit is that the specification ensures that a parameter that is passed from a Visual Basic .NET application to a Visual C# library will be interpreted correctly within the library, and the return value will behave as expected back in the Visual Basic .NET application. The fact that the application and the library it uses are written in two different languages is transparent to the developers of each. This benefit results in increased productivity because Visual Basic .NET developers can now more easily integrate existing libraries and components into their Visual Basic .NET code, even when those libraries are developed in other languages.

An additional benefit of the CTS is that it enables any class written in a .NET language to inherit behavior from a class written in any other .NET language. For example, you can create a Visual Basic .NET class that inherits behavior from another class that was developed in Visual C#. Moreover, you can extend the .NET Framework itself from Visual Basic .NET. This significantly expands the set of components that you can use and the ways in which you can use them. The increase in productivity results from code reuse, even among different languages.

Applications built for the .NET Framework have access to a comprehensive class library named the .NET Framework Class Library (FCL). It contains a large number of classes that enable all categories of client and server application development. The FCL contains classes for programming security, cryptography, Windows Forms, Windows services, Web services, Web applications, I/O, serviced components, data access to SQL Server™ and many other database systems, directory services, graphics, printing, internationalization, message queues, mail, common Internet technologies and protocols, culture-specific resources, COM interoperability, .NET remoting, serialization, threading, and transactions. The benefit of the FCL is that Visual Basic .NET developers can use and expand existing components built on the FCL without having to start from the beginning.

In Visual Basic .NET, the CLR is responsible for memory management. In particular, the CLR manages the work of freeing up memory for objects that are no longer referenced in an application. This process, garbage collection, happens automatically any time the CLR runs low on memory. The benefit of this process is that developers do not have to destroy unnecessary objects to free memory resources. This results in

increased productivity because a developer can invest more time in solving business problems than in resource management issues.

Although Visual Basic 6.0 provided an object-based programming model, it lacks some features that would make it a full object-oriented language. In particular, Visual Basic 6.0 does not provide full inheritance. Although it supports interface inheritance, this limited form prevents developers from taking advantage of the all of the benefits of this object-oriented programming. For example, this type of inheritance does not allow for code reuse and it limits the ability to fully implement object-oriented designs.

Visual Basic .NET supports full implementation inheritance. You can create new classes that inherit from existing classes. In fact, you can perform implementation inheritance regardless of the .NET programming language that was used to create the super class. This means that you can now more fully reuse existing classes through inheritance and can truly implement object-oriented designs.

In addition to implementation inheritance, Visual Basic .NET still supports interface inheritance. An interface is defined with the **Interface** keyword. Your classes then inherit from an interface by implementing its methods.

Furthermore, you can create forms that inherit from existing forms. This type of inheritance is named visual inheritance, and it allows you to apply the layout, controls, and code for existing forms to new forms. Through this form of inheritance, developers can build standard base forms. These forms can then be inherited by applications to give them a similar look and style, but it still gives the application developers the freedom to expand the form by adding new controls or changing a particular control's behavior. The result of visual inheritance is that developers can take existing forms and customize them for new applications, instead of having to recreate new forms for each application.

The benefit of inheritance is that it enables code reuse. Developers can take existing classes and forms, and through inheritance, customize them to meet an application's needs. The result is increased productivity because less time is invested in building completely new components.

Visual Basic .NET includes support for a new form development system named Windows Forms. The Windows Forms framework provides features that make the creation of powerful forms-based applications easy. These features include all standard Windows application form controls (such as buttons and text boxes), built-in support for connecting forms to XML services, and ActiveX controls support. These are only a few of the many features available in Windows Forms. The benefit of this framework is that it allows developers to rapidly build forms applications.

Control anchoring frees developers from writing code to deal with form resizing. Now, controls on a form can be visually anchored so that they remain a fixed length from the edge of the form and resize whenever the form resizes, with no effort on the part of the developer. This frees the developer to focus on business logic, instead of form manipulation, which makes the developer more productive.

For Web applications, Visual Basic .NET includes built-in support for Web Forms and XML. These features increase productivity by simplifying the work that is required to include them in your applications. You can even extend a functional application into a Web service. Moreover, the Visual Studio .NET IDE includes design tools that help to quickly and efficiently design HTML documents, as well as XML documents and schemas. All these features provide you with an environment in which the effort to build a Web application development is comparable to developing a Windows-based application.

In addition to the beneficial features in the Visual Basic .NET language itself, there are productivity benefits to using the standard .NET IDE, Visual Studio .NET. Some of these features are highlighted here.

The Windows Forms Designer gives developers all the tools they need to rapidly design forms. Controls such as buttons, check boxes, and more can be added to a form by dragging them to the form.

Visual Studio .NET also provides a WYSIWYG (what you see is what you get) designer for Web pages. This enables developers to quickly design Web-based solutions. Also, as previously mentioned, XML designers provide tools for rapidly building XML enabled solutions.

Visual Studio .NET provides a Task List display, which shows the outstanding tasks the developers need to complete. The Task List indicates any **ToDo** comments that developers include in code as a way to remind themselves of these outstanding tasks. The Task List provides a centralized view of these reminders, and allows developers to jump immediately to the comment when the time to address the task arrives.

The Task List also displays any compilation errors that need to be corrected. The Visual Basic .NET compiler continuously works in the background during coding. Compilation errors are flagged in real time with blue squiggle underlines in the code. The Task List also updates in real time the list of compilation errors to give developers a complete list of corrections that they need to make.

The Task List can also be filtered to show only particular types of outstanding tasks, such as showing only compilation errors or only **ToDo** comments. Figure 1.2 on the next page shows a typical Task List in a Visual Basic .NET solution.

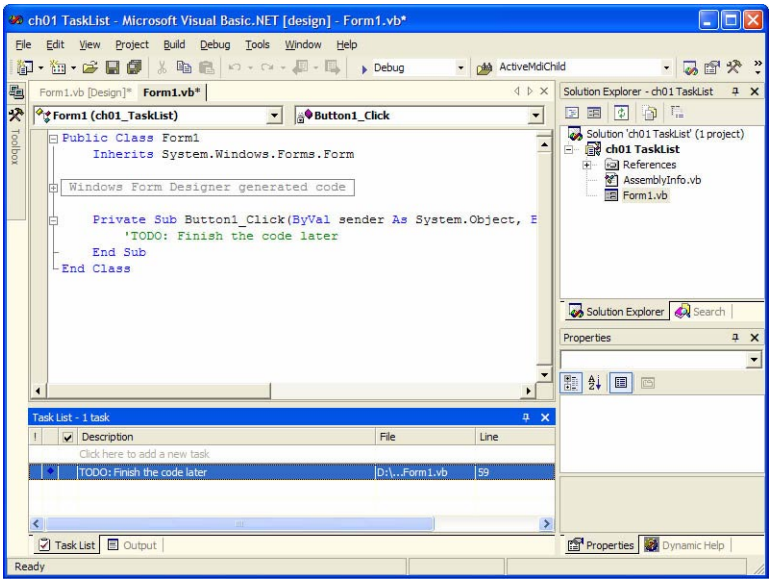

Figure 1.2
***ToDo** comments in the Task List*

The background compilation and Task List features enable developers to be more productive by reminding them of unfinished tasks.

As mentioned earlier, the .NET Framework is designed for cross-language interoperability. Controls and components written in one language can easily be used in another language because the CLR unifies all data types. Anyone who has struggled to get a Visual Basic 6.0 user control to work in a Visual C++ project can recognize the benefits of this interoperability. Not only can you easily use a component written in another .NET language, you can also inherit from a class written in any other .NET language. You can also create components in other languages that inherit from your Visual Basic .NET classes. Surprisingly, this interoperability does not complicate development. Visual Studio .NET provides a single environment for developing and compiling multi-language applications. A Visual Studio .NET solution can contain several projects, where each project can be developed using a different .NET language. Using the unified debugger, you can step through the various components, even if they were written in different languages. You can also debug COM+ services and step into SQL Server stored procedures.

A new feature of Visual Studio .NET is the **Processes** dialog box that allows you to attach to and debug an already running process. Certain types of applications require specialized execution environments that Visual Studio is unable to provide. For example, a Windows service execution is handled by the Windows Services Manager. Such applications do not have graphical user interfaces. Although it is possible in some cases to provide a Windows service with a **Main** method and an

appropriate test framework to enable debugging, it is often impossible to reproduce a problem through any means except running the service in its natural environment as a Windows service. Another situation in which the **Processes** dialog box is useful is when you are executing an application outside Visual Studio .NET and the application starts to behave in some unexpected way that is difficult to reproduce. The Visual Studio .NET debugger provides a **Processes** option that allows the debugger to attach to a currently executing process. This feature provides a means to debug applications that are running outside of the IDE; this gives developers the flexibility to handle these kinds of situations.

Visual Studio .NET provides a very high level of integration with SQL Server. You can connect to SQL Server using Server Explorer. After you are connected, you can browse the database and even create new objects. In particular, you can create and edit stored procedures written in the native SQL Server procedure language TSQL. After they are created, you can execute stored procedures from within Visual Studio .NET. You can step into the stored procedures just as you can step into Visual Basic .NET methods to debug them. You can set breakpoints so that SQL Server returns control to Visual Studio .NET when a certain line of code is reached during the execution of a stored procedure, function, trigger, or extended stored procedure.

Visual Studio .NET provides the ability to add references to other projects to your solutions. When you do this, Visual Studio automatically calculates the dependencies and the build order for your project and referenced projects. It also ensures that all of the referenced assemblies are always up to date and that they are available in the appropriate directories when they are needed.

The Immediate windows from earlier versions of Visual Studio have been enhanced and are now named Command windows. You can use these windows to read values from and write values to variables in a paused application, evaluate expressions, and execute statements. You can also use them to execute any of the Visual Studio commands that are available from menus, keyboard shortcuts, or toolbars. For example, you can issue commands to add new or existing items to the project or solution, perform search and replace operations using regular expressions, toggle break points, add watches, examine the call stack and memory, and display the current list of threads in a program. The Command window supports IntelliSense, even for file names, which makes using the Command window much more efficient.

Better Integration

Visual Basic .NET is designed to easily interact with components built on the .NET Framework and with legacy components. This section explores how improved integration can benefit applications.

Assemblies are the fundamental unit of deployment in .NET. An assembly is a collection of types and resources that form a logical unit of functionality, such as a component or application, in the form of a dynamic link library (DLL) or executable

(EXE). An assembly also maintains information used by the CLR, such as the version number of the assembly, the names of other assemblies it is dependent on, and requests for security permissions.

An important part of being able to maintain and control dependencies between components of an application is the ability to publish public interfaces that define how the components can be used. This is important because it allows the developer of a component to modify it, to improve it, or fix problems, without breaking code written by clients of the component. Assembly isolation allows the component developer to ensure that only a predefined subset of the component can be used directly by clients. This allows the developer to make changes to the implementation of the component without causing problems for or requiring changes to the clients, as long as the public interface does not change.

Another benefit of assembly isolation is that it allows side-by-side execution. Side-by-side execution refers to the ability to install multiple versions of an application, third-party components used by an application, or the .NET Framework itself and to have clients choose which version of the framework and/or components it uses at run time. This results in better integration because developers no longer need to worry about corrupting existing applications with the release of newer versions of a module, component, or framework. Assemblies cannot corrupt each other, and clients always know the correct version of an assembly to refer to.

Visual Basic .NET includes features that allow interoperability with COM components. COM interop allows developers to create a bridge between managed Visual Basic .NET code and the unmanaged code of a COM component for all basic data types, such as numeric types and strings. Data types of parameters passed to and return values received from COM components are converted to the proper equivalents between the managed and unmanaged code. This mechanism, referred to as data marshaling, ensures clean interoperability between the managed code of your upgraded application and the unmanaged code of the COM component. Note that in some instances, custom code for performing marshaling may be required for user-defined types.

Application Extendibility

Extending an application with new technologies helps to leverage the investment in that application. Visual Basic .NET includes many features that ease the process of extending an application with additional technologies.

Visual Basic .NET supports multiple distributed technologies. One such technology is Web services, an open standard that enables developers to expose a set of software services over the Internet through XML. The .NET Framework provides built-in support for creating and exposing Web services. Software services exposed in this way are both scalable and extensible and embrace open Internet standards — HTTP, XML, SOAP, and Web Services Description Language (WSDL) — so that a service

can be accessed and consumed from any client or Internet-enabled device. Support for Web services is also provided by ASP.NET.

Another example is a new technology unique to the .NET Framework named .NET remoting. Remoting is similar to DCOM. It enables communication between applications. This communication can occur between applications on the same computer, between applications on different computers on the same network, and even between applications on different computers across different networks. Support for .NET remoting is available in Visual Basic .NET.

Improved Reliability

Some of the new features in Visual Basic .NET improve the reliability of components and applications. This section describes some of these features and how they result in improved reliability.

In Visual Basic .NET, data types are bound statically. This means that the type of a variable must be known at compile time. Using this strategy, the compiler can verify that only valid operations can be applied to a particular variable or value. The strict data typing rules that are enforced by Visual Basic .NET and the CTS are not just to provide a high level of compatibility between different languages. These rules also create a development platform where errors caused by incorrect use of data are identified as early as possible in the development life cycle, typically at compile time. The increased interoperability of data types across components and strict enforcement of data typing rules in Visual Basic .NET result in more reliable applications and components.

Visual Basic .NET helps you reduce programming errors by supporting stronger type checking. For example, using the wrong **enum** value for a property or assigning an incompatible value to a variable is detected and reported by the compiler. Also, with ADO.NET, you can add typed datasets to your application. If code refers to an invalid field name, it is detected as a compile error instead of a run-time error. The earlier an error is found during the development cycle, the easier it is to correct. Identifying type errors at compilation time instead of at execution time makes them easier to correct.

It is possible to force even stronger type checking at compile time by using **Option Strict On** in your application code. This option completely prohibits late binding and requires you to use conversion functions whenever you assign to a variable a value that is of a different type. This helps developers identify points in their code in which subtle errors due to conversion may occur, such as the loss of a value due to truncation when assigning a higher-precision value to a lower-precision variable (for example, assigning a **Long** value to an **Integer** variable).

Another new feature of Visual Basic .NET that results in improved reliability is a security feature that enforces the adherence to object-oriented concepts such as the

encapsulation of data and behavior. This feature, named *type safety*, protects memory locations by restricting access to private members of an object. It ensures that when a field or method is marked as private, it may be accessed only within the class to which it belongs.

The term type safety also has a broader meaning that refers to language features that reduce the number of run-time errors that relate to incorrect data types. In this context, type safety refers to the compile-time checks that ensure operations are performed on appropriate data type values.

In contrast, the type safety that is used to ensure .NET security is checked by the compiler and can optionally be verified at run time using verification. Verification is required because the runtime has no control over how various assemblies in the application were compiled, but they may be skipped if the code in question has permission to bypass verification. This kind of type safety guarantees that a class's public interfaces are respected, and that assemblies can be isolated from other assemblies in the application domain. This improves reliability and security because no other code can adversely affect execution by altering data or invoking functionality that has been marked private. For more information about type safety and verification in .NET, see "Type Safety and Security" in the *.NET Framework Developer's Guide* on MSDN.

Structured exception handling is another new feature of Visual Basic .NET that can improve application reliability. In addition to supporting the familiar **On Error GoTo** error catching mechanism from Visual Basic 6.0, Visual Basic .NET provides a new structured error handling mechanism: the **Try/Catch/Finally** block and the **Throw** statement.

A **Try/Catch/Finally** construct encloses code that may result in errors or other exceptional conditions, and it provides handlers for each potential exception. A software developer has full control over the granularity of the exception handling, from an individual line of code, to an entire block of code, to all the code in a function. The developer also has control over the types of exceptions that are handled, from specific exceptions to general catch-all exceptions. The optional **Finally** portion of the construct provides a way to specify code that should execute regardless of whether or not an exception occurred.

The **Throw** statement is a way for a developer to signal the occurrence of an erroneous or exceptional condition. This allows the developer to signal to a client that an unexpected condition has occurred, and it gives the client the opportunity to decide how to deal with the condition. A client may be able to recover from the condition or provide a strategy to allow an application to gracefully degrade.

Structured exception handling gives developers a way to either recover from or gracefully exit on error conditions. These strategies can help to prevent data loss; this improves an application's reliability even in the event of error.

Improved Security

Application security is an important feature of Visual Basic .NET. The .NET Framework provides a set of sophisticated security features without overcomplicating requirements for developers. Security management in the CLR prevents others from modifying the assemblies that contain an application or its dependent components. It also ensures that components cannot access memory that they are not authorized to access.

Developers can control the permissions of applications and components that use configuration files. A developer can control access to a resource based on the privilege level of the user running an application or based on the trust level of the application itself. This can restrict untrusted code from accessing a resource even if the user of the code would ordinarily have full access. For example, even if untrusted code runs within a super-user account, its access to a resource can still be restricted.

Improved Deployment Options

Before .NET, deploying client applications typically required applying registry settings and/or copying DLLs to appropriate directories. An application could easily break if the registry settings were corrupted or if another application replaced a DLL with an incompatible version.

Visual Basic .NET has many new features that have been designed to simplify rich client application deployment. These features include the following:

- The .NET Framework provides application isolation. DLLs for one application do not affect other applications. This is referred to as zero-impact installation.
- Components and DLLs by default are installed privately for the application. These components must be shared explicitly to make them visible to other applications.
- Multiple versions of a DLL can be shared transparently. An application's assembly contains all the required version information for any DLLs used. As a result, even if multiple versions of the DLL exist on the target computer, only the correct version will be used by the application.
- Self-contained applications and components can be deployed without registry entries or dependencies. All that is required is copying the appropriate files to the application's bin directory. This type of installation is referred to as XCOPY installation.
- Applications can take advantage of features available in the Microsoft Windows Installer, such as advertisement, publishing, repair, and install-on-demand.
- Incremental installs provide for smaller downloads. Furthermore, if installation fails, the target computer is fully restored to the stage it was in before the installer began.

- A new type of installer called a merge module provides a much simplified way to package and install components that will be shared by multiple applications. Merge modules (.msm files) allow you to share components between multiple deployment projects. You can create your own merge modules with Visual Studio .NET, or you can use existing merge modules that are available for many standard components from Microsoft, as well as from third-party vendors. A merge module is created once and is included in all applications that use it. It handles all the details of remembering which applications are using the component and when the last application that needs it has been uninstalled. For more information about merge modules and other installation and deployment options, see Chapter 16, "Application Completion."

One of the results of improved deployment features in Visual Basic .NET is that installations can no longer corrupt other application installations.

Increased Performance

Some new features of Visual Basic .NET can be used to increase the performance of an application. This section explores some of these features and how they can be used to increase performance.

A typical Visual Basic application is single threaded, which means that only a single task at a time can be performed. However, Visual Basic .NET now provides support for multithreaded applications, so that multiple tasks can execute concurrently. Multithreading can improve application performance in the following ways:

- Program responsiveness increases because the user interface can remain active while other work continues.
- Idle tasks can yield their designated processor time to other tasks.
- Processor-intensive tasks can periodically yield to other tasks.

As an example, consider a server application that provides information to clients from a database. In a single-threaded application, the server is capable of servicing a single client at a time. This means that if multiple clients need information at the same time, only one is serviced. The other clients have to wait until the first client's request is complete before they receive attention. In a multithreaded application, the server has multiple independent threads of execution servicing individual client requests simultaneously. As a result, each client receives immediate attention from the server.

Technical Support

As of March 2005, Microsoft ended the mainstream phase of support for Visual Basic 6.0, and entered into a period of extended phase support. The current extended phase support is available until March 2008, at which point Visual Basic 6.0 will no longer be supported.

The impact of these support policies is that free support for Visual Basic 6.0 is no longer available. During the extended phase, customers can still receive professional product support for Visual Basic 6.0 through telephone and online incident support channels on a per-incident fee basis or through Premier Support. Similarly, critical updates and hotfixes are only available for a fee during the extended phase.

For more information about the support policies for Visual Basic 6.0, see "Product Family Life-Cycle Guidelines for Visual Basic 6.0" in the Microsoft Visual Basic Developer Center on MSDN.

Benefits of the Visual Basic 6.0 to Visual Basic .NET Upgrade Wizard

If you choose to upgrade your application, you can best achieve this by using the Visual Basic 6.0 to Visual Basic .NET Upgrade Wizard included in the Visual Studio .NET IDE. By using the upgrade wizard to upgrade your application, you can:

- Produce source code that is both easily extendable and maintainable.
- Acquire functional equivalence, with all of the built-in features of the original program intact.
- Gain a robust and intelligent automatic restructuring of the code.
- Keep original code comments.
- Maintain cross-references for all the application components.

Some of these benefits may be reduced or lost if an application uses many technologies that are not supported in Visual Basic .NET. The Visual Basic 6.0 Upgrade Assessment Tool can help to identify an application's dependence on such technologies, and this guide provides the solutions for addressing dependence on unsupported technologies in Visual Basic .NET.

Summary

When you decide whether to upgrade Visual Basic 6.0 applications to Visual Basic .NET, you have many factors to consider. Some applications are not well suited to an upgrade, either because they will not benefit from .NET technologies or because the cost or complexity of the upgrade is prohibitive. Other applications practically compel the move because the benefits far outweigh the costs. The main point of this chapter is that an upgrade should be considered on a per-application basis. There are many convincing reasons for upgrading applications to Visual Basic .NET, but there are often just as convincing reasons to leave an application alone. Upgrade an application only if and when it makes sense to do so. If an upgrade is warranted, risks and costs can be minimized by preparing an upgrade plan. The remaining chapters of this guide provide you with the tools and techniques to prepare such a plan, and to help you perform the actual upgrade as efficiently as possible.

For More Information

For more information about the Visual Basic Upgrade Assessment Tool or to download it, see "Visual Basic 6 to Visual Basic. NET Migration Guide" on the GotDotNet community site:
http://www.gotdotnet.com/codegallery/codegallery.aspx?id=07c69750-9b49-4783-b0fc-94710433a66d.

For more information about type safety and verification in .NET, see "Type Safety and Security" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcontypesafetysecurity.asp.

For more information about the support policies for Visual Basic 6.0, see "Product Family Life-Cycle Guidelines for Visual Basic 6.0" in the Microsoft Visual Basic Developer Center on MSDN:
http://msdn.microsoft.com/vbasic/support/vb6.aspx.

2

Practices for Successful Upgrades

This chapter examines the upgrade process in detail, and further discusses two key concepts that were introduced in Chapter 1: functional equivalence and application advancement. This chapter also presents upgrade practices that apply to these two concepts and explains some best practices for managing a successful upgrade project.

Functional Equivalence and Application Advancement

When you upgrade an application from Visual Basic 6.0 to Visual Basic .NET, you can either choose to stop the upgrade process when you have a Visual Basic .NET version of an application that looks and behaves the same as the original, or you can choose to add new features that are available in the new language. This section explains these two options, functional equivalence and application advancement.

Functional Equivalence

Functional equivalence is an important concept for any upgrade project. In the context of a Microsoft Visual Basic upgrade, achieving functional equivalence means that the exact functionality of a Visual Basic 6.0 application is retained in the Visual Basic .NET application after it has been upgraded, but before new features are added. Achieving functional equivalence guarantees that your application logic does not change as a result of the upgrade and that your existing business rules are functional in the upgraded application.

By making functional equivalence a goal for your upgrade project, your organization can benefit in several ways:

- Because your Visual Basic .NET application is functionally equivalent to the original Visual Basic 6.0 application, you have a perfect specification for the application upgrade project. This means that you can easily identify discrepan-

cies in application behavior and in output, and you can clearly measure the success of the upgrade project.

- With a functionally equivalent application, you will not encounter scope increases or changes, and you can predict and facilitate the testing phases of the project. You will still need to prepare specifications for any new functionality that is added after functional equivalence is achieved.
- The goal of achieving functional equivalence gives you a clear, easy-to-understand means of tracking the progress of the project. As you upgrade parts of your application, you can compare the functionality with the original application.
- A functionally equivalent application eliminates the need for existing users to adapt to an upgraded application and reduces any change management processes to the IT department.
- Achieving functional equivalence is the fastest way for an organization to transition from older to newer technology. It is also the most efficient way for an organization to regain autonomous control and to resume normal incremental maintenance of the application.

Application Advancement

Achieving functional equivalence provides you with a working .NET version of your application, but it is usually considered an intermediate step in the process of upgrading an application. As described in Chapter 1, "Introduction," a principal benefit of upgrading your application to the .NET Framework is that you have access to significantly improved application performance, integration, productivity, extensibility, reliability, and security. Using this new technology to add or improve functionality in your application is referred to as application advancement.

Application advancement involves identifying, and then modifying or extending, the areas of your application that would benefit from the many improvements that the .NET Framework offers. In fact, application advancement could be considered to be the key reason for upgrading to Visual Basic .NET in the first place. However, it might seem that you are wasting effort by first achieving functional equivalence in an application that you ultimately plan to change, but by doing this, you are actually providing a solid basis for a more controlled approach to application advancement.

Organizational Structure and the Software Life Cycle

Whether your upgrade project is focused on functional equivalence, functional equivalence followed by application advancement, or pure application advancement, you should view the application upgrade as a standard software development project. Although the design of your application has been determined already (for the most part) and it is already implemented, you still need to plan for the usual

stages of the development cycle. These stages — managing expectations, design, implementation, and testing — are vital elements in the upgrade process.

While testing is very important to an upgrade project, you might be able to reuse test processes that you developed for the earlier version of the application. For example, you may find that some of the testing assets — such test plans, test cases, and test code — that were used for the Visual Basic 6.0 application are still useful after the upgrade. This topic is covered in more detail later in the chapter.

Overview of the Upgrade Process

The upgrade of your Visual Basic 6.0 application is not a trivial task. As with any development project, it is important to have a clear concept of process. This will help you keep the project on schedule and within your budget.

The process described here was developed as a result of many Visual Basic upgrade projects performed by Microsoft and its partners. This chapter looks at the upgrade process as a series of phases that represent milestones within the upgrade. Some of these phases, such as planning, must be completed for the entire project before any other phases can start. Other phases can be applied to individual pieces of the application at the same time while other sections of the application are already in later phases. For example, if you have chosen a staged upgrade strategy, it is possible that some of the sections of your Visual Basic application are being prepared for upgrade while others have already been upgraded and are ready for testing.

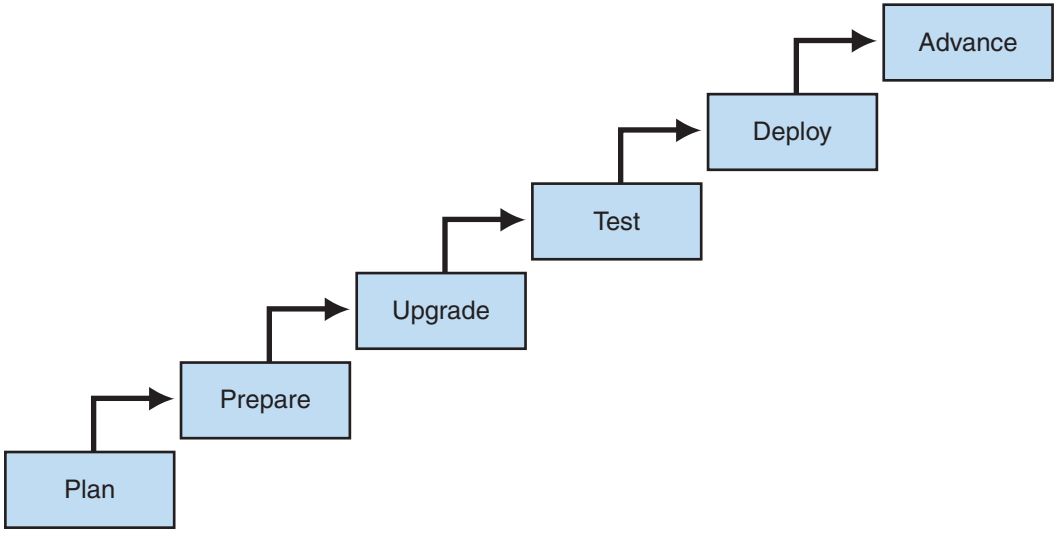

Figure 2.1
A graphical representation of the upgrade process.

The main phases of the upgrade process are planning, preparation, upgrade, testing, deployment, and advancement. They are briefly described here and will be explored in more detail later in this chapter:

- **Planning**. Planning is the first phase of the upgrade project. It involves defining goals, expectations, and constraints. Planning requires that you assess the application to be upgraded so that you can evaluate risks and determine the steps needed to upgrade the application. It is particularly important for you to determine the order in which the modules should be upgraded. During the planning phase, you will also need to evaluate the cost of upgrading the application.
- **Preparation**. The second phase involves preparing your application for the upgrade. Although the preparation phase is optional, it is strongly recommended because by preparing your application in advance you will save a great deal of time during the upgrade phase. Preparing your application includes modifying certain implementation details of the application so that the process of using the Visual Basic Upgrade Wizard is as efficient as possible. This, in turn, reduces the work you need to do after the upgrade wizard has been used.
- **Upgrade**. During the upgrade phase, you use the upgrade wizard to upgrade as much as possible of the Visual Basic 6.0 code to Visual Basic .NET code. Afterward, you manually upgrade the remaining code. The result of this phase should be a Visual Basic .NET application that is functionally equivalent to the Visual Basic 6.0 application.
- **Testing**. Usually, testing is the basis for demonstrating functional equivalence in your application. Although listed here as a phase to follow the upgrade phase, testing should be performed throughout the process, even in the early stages of the upgrade. For example, unit testing can be applied to individual components as they are upgraded to ensure they are correct before testing their behavior in the entire system.
- **Deployment**. Because deployment also must be tested, the deployment phase often overlaps with the upgrade and testing phases. This is especially true for projects that require a successfully deployed application before the testing phase can be completed.
- **Advancement**. The final phase involves advancing your functionally equivalent application by implementing new .NET features or by improving existing features with .NET functionality. Keep in mind that any new features will also have to undergo testing.

Before you begin the project in earnest, you might consider using a subsection of your application in a test run of this process. A test run can hone your cost estimations and possibly reduce upgrade risks by identifying issues that could block the progress of the actual project.

Proof of Concept

When planning your project, you should evaluate the need to test any major upgrade issues with a proof of concept or with a prototype. A proof of concept is evidence that demonstrates whether a project is ideal or feasible. It, or a prototype, can help you to avoid wasting time on poor coding decisions and can allow you to validate solutions in an isolated condition. A proof of concept will help you to determine the real cost of the most common issues that appear in the upgrade wizard's upgrade report. For more information about the upgrade wizard, see Chapter 5, "The Visual Basic Upgrade Process."

The proof of concept for an upgrade project will help you familiarize yourself with details of the upgrade process. A good way to develop a proof of concept is by upgrading a practice application. When choosing a practice application, consider a simple application that is a good candidate for use with the upgrade wizard and one that is not tightly integrated with other technologies. (For more information, see the "Performing an Application Analysis" section later in this chapter). After you have chosen and then upgraded the practice application, you and your team can gain valuable experience by analyzing and solving generic issues in the upgraded code. This activity will help you to identify issues that can affect the execution of your more complex, real-world project. Learning to upgrade is like learning anything else: by starting with a simple project, you learn in manageable pieces. Also, because you will probably have the application upgraded and running quickly, your team will build the confidence that is needed to tackle a larger project. Finally, this practice upgrade will help you to assess the feasibility of upgrading a larger, more complicated application.

If the proof of concept convinces you of the feasibility of upgrading a more complex, real-world application, choose one that will benefit from the new features of the .NET Framework so that you can review the process from the point when the application has achieved functional equivalence through the final phase of application advancement. Think about the way the architecture is going to be modified. For large applications, the principle of divide and conquer applies. You should upgrade the application module-by-module. For example, you may start with the client modules, and then move up the dependency hierarchy by next upgrading the business logic and data tier modules. (For more information about upgrade strategies, see Chapter 1, "Introduction.") As you move along in the process, be sure that you test each module of the application as it is upgraded so you know that you have quality code before you move forward.

After you and your team have had some experience with the upgrade process and you have chosen a real-world application to upgrade, you will be ready to begin the work of defining the scope of the project.

Note: If you plan to upgrade your application one piece at a time, you must apply interoperability techniques that allow the components that remain in Visual Basic 6.0 to communicate with the upgraded Visual Basic .NET components. For more information about interoperability, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET."

Planning the Upgrade

In addition to the tasks your organization typically performs during a software development project, your plan for the upgrade project should include the following activities:

- **Defining the project scope**. Define the objectives of the upgrade project. Be clear about which components and features will be upgraded and what you expect from the upgrade.
- **Performing an application analysis**. Profile the application you want to upgrade to determine the best upgrade strategy.
- **Assessing the current and the target architecture**. Assess the design and implementation of the application being upgraded. Clarify any architectural aspects that need to change during the move to Visual Basic .NET.
- **Analyzing and designing new functionality**. If expectations include functionality that is not in the current application, plan for the analysis and the design of the new features.
- **Selecting an upgrade strategy**. Based on the design of your application, you will probably opt for one upgrade strategy over another. When you make your choice, you must create measurable milestones and testing processes for intermediate releases of the application.
- **Making an inventory of source code**. Specify the amount of source code that needs to be upgraded. Be sure to include all other dependencies on the application that will be required for its compilation and execution.
- **Preparing the source code**. Plan enough time to prepare the code for the upgrade. Resist the urge to dive into the upgrade phase before your team or the code is ready.
- **Preparing to handle upgrade issues**. Based on the results of your application assessment, schedule the appropriate amount of time to manually complete functionality that cannot be fully upgraded by the upgrade wizard. You will also need to spend time on functionality that was successfully upgraded but may behave differently in the .NET environment.
- **Unit testing**. Although this is an upgrade project, you will still need to perform testing at the unit level to ensure that even low-level code behaves as expected. Doing this ensures that integration and acceptance testing are more efficient.

- **Validating functional equivalence and acceptance**. If it is necessary for you to prove that the new application does indeed behave exactly like the original application — or within a predetermined margin of error — you need to define the criteria that will be used to measure functional equivalence and application acceptance.
- **Implementing and testing new functionality**. Schedule time and resources to implement and test all new functionality that you will add to the upgraded application.
- **Deploying the Visual Basic .NET application**. No upgrade is complete until the new application has been deployed and is in production. Make the necessary plans to transition from deploying the original application to deploying the newly upgraded .NET version.

Defining the Project Scope

Defining the scope of your project is the all-important stage of project planning where you establish the project goals and the manageable tasks that will accomplish these goals. You will also define the milestones and criteria that you will use to measure whether you have succeeded in meeting them.

Upgrade project plans include certain goals that are not usually required in other software development projects. Primarily, the project scope should include criteria for measuring whether the upgraded application is functionally equivalent to the original application.

You need to specify the criteria that you will use to measure the success of reaching milestones and to prepare the mechanisms that you will use to validate them. Start by creating test cases. Test cases are an important tool for measuring progress during the upgrade process and also for measuring functional equivalence. As with any software development project, test cases help you maintain a level of high quality in your application.

By answering the following questions, you can further define the scope of your project:

- **Do you expect the first version of the upgraded application to have additional functionality?** Your first aim should be to achieve functional equivalence. If you want to add functionality, keep this goal separate by defining specific milestones for the tasks that involve application advancement. Additional functionality is not an automatic result of upgrading an application; you must plan for it.
- **Do you expect to see improvements in the performance of the functionally equivalent application?** Better performance in an application is not an automatic result of upgrading it. However, performance is a valid requirement for the first iteration of your upgraded application so these requirements should be well-documented early.

- **Do you expect improvements in security, usability, or reliability?** These improvements are additional features that you need to implement after the application has achieved functional equivalence. Their implementation is part of application advancement.
- **Do you plan to take advantage of new techniques for deployment and internationalization?** Again, the implementation of these new features is part of application advancement and should begin after the application has achieved functional equivalence.
- **Are you aware of new dependencies that come with Visual Basic .NET and the .NET Framework and how they may affect your deployment requirements?** The deployment of a functionally equivalent application is a reasonable expectation, and planning for any new dependencies is vital to its success.
- **Do you expect the upgraded application to look identical to the original application? Are you prepared for differences in appearance?** For some types of applications, there will be unavoidable changes in appearance, so you must plan to document these changes for users. For information about some of the visual and behavioral differences you can expect when you upgrade your application to Visual Basic .NET, see Chapter 9, "Upgrading Visual Basic 6.0 Forms Features."

Performing an Application Analysis

An important step in managing risk and estimating costs for an upgrade project is having a team of experts (which may be internal or external to your business) perform a technical analysis of the application. This will determine the most appropriate upgrade strategy for the project. The goal of this activity is to ensure that the project adapts to your real business needs and that your expectations are realistic about the outcome of the project. The analysis should produce an outline of the project's time and cost estimates, as well as a technical profile of the application. Both the outline and the profile are imperative to the success of the project.

Begin the profile by identifying and analyzing the application code that will be upgraded. A good way to start is by producing an inventory of the application source code, including references to libraries and resources. Then, you need to identify and classify possible issues that you will have to handle during later stages of the upgrade. For this task, an initial test run using the upgrade wizard would be very valuable. When doing this, you may want to distinguish those features in the source code that will be automatically handled from those that may require manual intervention or that you might have to consider re-architecting later on as you move through the process.

As part of the analysis, review the upgrade report that the upgrade wizard produces. The upgrade report contains information about the upgrade process and a list of issues uncovered during the upgrade that will need to be addressed before your project can be compiled and run. Study the different issues in the upgrade report

and the solutions that are recommended for resolving them. Remember, you can reduce the number of issues that need to be addressed by preparing your Visual Basic 6.0 code before applying the upgrade wizard. For more information about using the upgrade wizard, see Chapter 5, "The Visual Basic Upgrade Process."

The purpose of the application analysis is to generate information that will be used to estimate the effort of the upgrade project and to create a technical profile of the application. The output of application analysis should cover the following items:

- Current and target architecture
- Inventory to upgrade
- Source code metrics
- Handling of unsupported features
- Assessment of upgrade report information

Assessing the Current and the Target Architectures

As part of the analysis, you will want to clearly understand the current architecture of the application and the way its components will be replaced in the target architecture. For instance, you may plan to replace ActiveX Data Objects (ADO) with ADO.NET and COM components with native .NET Framework components. The resulting target architecture should also include the new components that will support the new application requirements. These new components are identified as part of the new requirements analysis and design you create.

It is likely that your upgraded application will interact with other existing Visual Basic applications, modules, or components, and any portion of the application that you may decide will be reused but not upgraded. Identify those interactions within the application architecture and clearly differentiate between the components that will interact and those that will require interoperability. Also, identify the components that can be replaced with equivalent .NET components.

Analyzing and Designing New Functionality

Upgrade projects can be an ideal time to introduce new functionality or features to an existing application. If there are features that you want to add, document the requirements for these new features, specify how they will interact with existing features of the application, and design their inclusion in the upgraded application.

Because this guide is concerned with reaching functional equivalence, details for analyzing and designing new functionality are not discussed here. However, for information about possible new features you might consider adding to your applications, see Chapters 17 – 21 of this guide.

Selecting an Upgrade Strategy

Chapter 1, "Introduction," examined the options you have available to you if you need to update your application. If you decide that an upgrade is the right approach to updating your application, you should consider the following two upgrade strategies: a complete upgrade and a staged upgrade.

The complete upgrade strategy suggests that you upgrade all your application as a whole and not release any code until the entire application is upgraded and running on the new platform.

The staged upgrade strategy allows you to upgrade some parts of your application before you upgrade others. This is less risky, but it requires more effort to enable interoperability between the upgraded and non-upgraded parts of the application.

Do not confuse staged upgrade with partial upgrade, which was discussed in Chapter 1. A partial upgrade is an alternative to updating an entire application. A partial upgrade means that you chose to move some of your application or applications to Visual Basic NET while others remain in Visual Basic 6.0.

After you have chosen to upgrade — whether it is to a partial or a full upgrade — you can choose to upgrade your application as a whole (complete upgrade) or one piece at a time (staged upgrade).

Complete Upgrade

With a complete upgrade, all components of your application are upgraded and deployed as a whole. This does not mean they are upgraded in parallel; it means only that no effort is made to deploy the application in a production environment until all components have been moved to .NET. This strategy may require significant effort, and therefore it can be very expensive. It not only involves upgrading code from Visual Basic 6.0 to Visual Basic .NET, but also upgrading all of the technologies to their .NET equivalents when they exist, or applying alternative technologies when they do not. For applications that do not make significant use of deprecated technologies, a complete upgrade can be very fast and inexpensive. The more dependent an application is on dated technologies, the more expensive and time consuming the upgrade process becomes.

This strategy has the following advantages:

- **Upgraded applications can be advanced with new functionality that uses .NET technologies and techniques**. Examples and strategies of this can be found in Chapter 17, "Introduction to Application Advancement," Chapter 18, "Advancements for Common Scenarios," Chapter 19, "Advancements for Common Web Scenarios," and Chapter 20, "Common Technology Scenario Advancements."

- **Upgraded applications can be integrated into new .NET solutions, making them usable in more scenarios**. This integration can be accomplished regardless of the programming language used to develop the new application and without requiring any changes to the existing application.
- **Fully-upgraded applications can be used on new hardware and new versions of Windows**. They can be used without requiring any upgrade or recompilation if the new hardware or the new version of the operating system has a fully functional .NET Framework engine.

Despite the advantages of complete upgrade, there are also disadvantages, as listed here:

- **Some Visual Basic 6.0 features cannot be automatically or easily upgraded to Visual Basic .NET**. Although alternatives for such features exist in Visual Basic .NET (and are often better), upgrading this kind of application requires that you change it to achieve functional equivalence. These changes can be time consuming and costly.
- **A completely upgraded application could have a different user interface or may have different behavior**. This will result in the need for complete testing of the application as a new application. It may also require retraining of users.
- **A Visual Basic .NET application requires a compatible version of the .NET Framework to be installed on target computers**. This requires additional planning and deployment by IT professionals, particularly in large companies.
- **Developers responsible for performing the upgrade or for maintaining the resulting application may have to learn new skills to perform the upgrade**. This has an associated cost in both time and money.

A complete upgrade can be an expensive strategy, but it is often the most desirable alternative because it positions the application for the future. It is important that you fully assess your application before making a decision. In many cases, the assessment will demonstrate that the benefits of a complete upgrade far outweigh the costs, and you will find that moving the application to Visual Basic .NET is the right decision for your business.

Staged Upgrade

Unlike a complete upgrade, where all pieces of the application are upgraded in a single complex process, a staged upgrade strategy allows a more controlled, gradual upgrade where the application is upgraded a part or component at a time. Each newly upgraded component can be rolled out as it is upgraded, instead of waiting for the complete upgrade of the entire application. This strategy is only possible when the existing application is composed of multiple distinct components, each

developed as a separate project. Also, interoperability techniques must be applied for the upgraded and unchanged components to function together.

A staged upgrade provides a reasonable compromise to a complete upgrade. Staged upgrades are likely to be the best alternative for large-scale legacy applications. This strategy has the following advantages:

- Upgrading the application in stages allows you to have more control over the progress and the cost of the upgrade project and allows you to minimize risk because you returns the application to a stable production-quality state after each stage. Because your development team will gradually become better acquainted with .NET at each stage, their productivity will improve — as will your estimates of effort and cost.
- Upgraded components can often immediately benefit from the performance and scalability features of .NET, without having to wait for the entire application to be upgraded.
- Upgrading of the low priority or costly portions of an application can be postponed indefinitely, resulting in a partial upgrade. A staged approach like this effectively gives you the freedom after the upgrade project has begun to opt for a mixed upgrade/reuse solution as described in the "Upgrade, Reuse, Rewrite, or Reuse?" section in Chapter 1, "Introduction."

The disadvantages of staged upgrade include the following:

- Performance improvements will not be as noticeable as in a full upgrade because of the added overhead required for COM interoperability. COM interop between .NET components and COM components is not as fast as managed components working together in the .NET Framework or COM objects in Visual Basic 6.0 working together. However, the difference in performance should only be noticeable in applications that perform a large number of COM method calls, such as those with tens of thousands of transaction requests.
- Applications with a dependency on COM components are harder to deploy and maintain than applications that are fully upgraded to .NET because COM objects need to be registered and have versioning issues that .NET objects do not.
- A staged upgrade requires the implementation of wrappers or interfaces to provide the interoperability mechanisms between the .NET and the legacy code. These wrappers will often be temporary and will be discarded later.

Despite the disadvantages, staged upgrades may be the most appealing strategies for large-scale applications with multiple components.

Staged upgrades can be further classified as vertical and horizontal:

- **Vertical upgrades**. These involve isolating and upgrading all *n*-tiers of a single module of your application without modifying other parts of the application.
- **Horizontal upgrades**. These involve upgrading an entire tier of your application without modifying the other tiers.

Whether you choose a horizontal or a vertical approach to a staged upgrade will depend on the architecture of your application. Therefore, the solution architects in your team should be highly involved in the selection of the strategy. You also will find advantages and disadvantages to each strategy based on the application you are upgrading. Usually an assessment of the design and implementation of the application will point to the best strategy to adopt.

The next sections examine the horizontal and vertical upgrade approaches to help you choose the one most suitable for your application's architecture.

Vertical Upgrade

In the vertical upgrade strategy, one or more of the application components that cut through several, if not all, application tiers are identified and selected to be upgraded. Essentially, this involves extracting a piece of your application that has minimal interaction with other pieces and upgrading it independently of the rest of the application.

If parts of your application are well isolated from others, you have an outstanding candidate for a vertical upgrade. An ideal candidate will typically share very little state information with the rest of the application and can easily be upgraded with little impact on the rest of the system.

The following scenarios describe application architectures that are well suited to the vertical upgrade strategy:

- **Application tiers are tightly integrated**. If you make heavy use of ADO recordsets between tiers, a vertical upgrade is worth examining. Many applications pass disconnected ADO recordsets from the data and business tiers to the presentation tier. The presentation tier then iterates through the recordsets and generates HTML tables. This type of application is well suited to a vertical upgrade because you can focus on specific aspects of the application individually. For example, an ADO to ADO.NET upgrade and interoperability requires special consideration. Upgrading vertically can minimize the work involved in achieving interoperability with ADO.
- **You are planning to redesign the application**. Vertically upgrading parts of your application to new architecture provides a good test bed for a new application design. The .NET Framework makes it easier to provide the functionality that newer architectures are built on. For example, you can use **HttpHandlers** to perform many of the tasks that you would previously use ISAPI extensions for, but with a much simpler programming model.

After the chosen portion is upgraded, any remaining interfaces between the new managed code and the unmanaged code will function through COM interoperability. You will need to do some development and testing to make sure that the new and old pieces of the application work together, share data, and provide a seamless experience to the end user or client developer. Afterwards, you will

have the basic infrastructure that will make it possible to continue with the upgrade of other applications using the same vertical approach — or to extend the updated portion of the original application.

Horizontal Upgrade

In a horizontal upgrade strategy, you upgrade an entire tier of your application to Visual Basic .NET without immediately upgrading the other tiers. By upgrading a single tier at a time, you can take advantage of the specific features that the .NET Framework provides for that particular tier, in many cases without modifying application code or affecting the operation of other application tiers. Deciding which tier to upgrade first is the first step in a horizontal upgrade strategy.

Middle-tier COM components can be upgraded to Visual Basic .NET with few or no changes to the presentation tier. You should consider the following points when deciding if a horizontal upgrade strategy is appropriate, and if so, which tier is the most suitable to upgrade first:

- **Your application is on a large number of Web servers**. A prerequisite for the deployment of a Visual Basic .NET application is that the common language runtime (CLR) must be installed on each Web server. If your application is deployed on a large number of servers in a Web farm configuration, this can be an issue. If you have relatively fewer servers in middle tiers than in other tiers, consider using a horizontal upgrade and then upgrading the middle tier first.
- **Your application uses a large amount of shared code**. If your Visual Basic application uses a large number of DLLs or other types of shared code, constants, or custom libraries in the middle tier, choosing to upgrade this tier first can help you manage the upgrade of your entire application. However, if you do this, you will destabilize a large number of components that use the shared code and thereby affect the testing process.
- **Your application has a complex middle tier**. Complex object hierarchies in the middle tier should be kept as a unit. Deciding where to isolate an application with a complex middle-tier object hierarchy is difficult, and upgrading only parts of a complex middle tier typically necessitates numerous interoperability calls between environments, resulting in performance degradation.

You should also consider the following issues when replacing the middle tier:

- To transparently replace middle-tier components with .NET components without affecting client code, you must maintain the original GUIDS and ProgIDs of your COM components. In addition, you must properly handle replacement of the class interface generated by Visual Basic components when you attempt to transparently replace a COM component.
- You will have to translate the ADO.NET datasets that are returned from your upgraded middle-tier components to ADO recordsets that are used in your original Visual Basic code.

- You will need to deploy the interoperability assemblies for the middle tier components.
- Interoperability between Visual Basic .NET and Visual Basic 6.0 can become prevalent in this type of upgrade, depending on how you choose to define the application tiers. If the distance between the tiers is widened by making one side of the communication a .NET assembly, you may be increasing the amount of marshaling that is occurring in your application, which can negatively affect performance.

After you decide on an upgrade strategy, you can increase the efficiency and chance for success in the upgrade process by producing a project plan for the upgrade. For more information, see the "Producing a Project Plan" section later in this chapter.

Testing

To make sure that your application achieves functional equivalence regardless of the upgrade strategy you choose, you need to produce comprehensive unit and system test plans for the entire system, as well as for individual upgrade modules and tiers. For detailed information about testing in upgrade projects, see Chapter 21, "Testing Upgraded Applications."

Making an Inventory of Source Code

Identify the relevant source code that will be upgraded. This should include libraries, resources and components. You can use the application's Call Graph view to make a dependency diagram in the form of a tree view to help you recognize the relevant Visual Basic 6.0 projects and files. For any dead code you find, you will need to decide to make it part of the inventory or exclude it partially or entirely. Tools such as the Visual Basic 6.0 Upgrade Assessment Tool can help you identify dependencies and dead code in your source code. For more information about these tools see Chapter 5, "The Visual Basic Upgrade Process."

In addition, identify the application components that will be upgraded and any that will be left alone and reused, including all of the components that the application currently uses. In the case of third-party components, determine the availability of existing .NET equivalents.

Preparing the Source Code

Investing time to prepare your source code for the upgrade will increase the efficiency of the upgrade project. For information about how to do this, see the "Preparing the Visual Basic 6.0 Source Code" section later in this chapter.

Preparing to Handle Upgrade Issues

If you identify potential issues before you begin the upgrade, you will be better prepared to address them during the upgrade process than if you wait until you are deeply involved in implementing the upgrade changes. This section provides guidance on how to obtain the information you need to help you build a catalog of issues.

Obtaining Source Code Metrics

Obtain metrics information from the application source code. This information should include size metrics, such as the number of lines of code, projects, forms, designers, modules, classes, components, user controls, data sources; and usage metrics, such as functions, properties and events used.

You also need to obtain metrics on how frequently each component is used in the application. Knowing in what way and how frequently a component is used helps to determine whether you can replace each component with an equivalent .NET version after the application has been upgraded.

Figure 2.2 shows partial source code metrics for a sample stock management application. It gives information about the source code inventory.

Project	Total Files	Classes	Modules	UserControls	Non upgradable	Designers	Forms
CoreComps	53	45	8	0	0	0	0
StockServer	41	10	25	0	0	0	6
MarketCommAPI	35	24	11	0	0	0	0
StockNegotiator	422	415	7	0	0	0	0
InvPlanServer	30	3	27	0	0	0	0
AccessNet	262	250	12	0	0	0	0
PlanCommShared	158	140	18	0	0	0	0
SOAPConnector	27	3	24	0	0	0	0
InvTasks	115	41	30	6	0	0	38
Total	1,143	931	162	6	0	0	44

Figure 2.2
Partial source metrics for a stock management application

The figure shows partial information about data types and controls occurrences, the source code inventory (in the form of a tree view), and the location of specific uses of data types; in this case the **Integer** and **String** types. The assessment tool provides all this information in easy-to-read HTML reports. For more information about the assessment tool, see Chapter 3, "Assessment and Analysis."

Identifying Unsupported Features

Identify all of the Visual Basic 6.0 features that are not supported in Visual Basic .NET, and note how often they are used in the application. You should investigate and make prototypes of Visual Basic .NET functionality that could replace these unsupported features. This is also a good time to identify where additional functionality is needed. For more information about early version and obsolete Visual Basic 6.0 features and the alternatives that are available in Visual Basic .NET, see Chapter

7, "Upgrading Commonly-Used Visual Basic 6.0 Objects", Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features," and Chapter 9, "Upgrading Visual Basic 6.0 Forms Features."

Assessing the Upgrade Report Information

After you run the upgrade wizard on your application, it generates a report that identifies the upgrade issues it encountered and the number of times that they occur in the application. Each type of issue is identified by an issue number, a description, a link to related documentation, and the steps you can take to possibly resolve the issue. For example, the issue number 1037 corresponds to the issue "Late-bound default property reference could not be resolved," and the report will link you to documentation for resolving this issue.

The following list summarizes the most probable causes for common issues identified in the upgrade report:

- A small number of Visual Basic 6.0 core functions have no direct .NET equivalent or core Visual Basic 6.0 functions behave differently when they are upgraded. However, even for the few functions that have no direct equivalent or those that behave differently when the are upgraded, Visual Basic .NET provides alternatives for achieving the same functionality in your application. These alternatives typically require manual modification of the code and cannot be automatically upgraded.
- Certain ActiveX controls are not supported in Visual Basic .NET.
- The upgrade wizard found language changes in the upgraded code and prompts you to manually review them. An example of this is the case default properties in Visual Basic 6.0. Whenever possible, the upgrade wizard will resolve default properties and explicitly expand them in the upgraded code; however, you should review these expansions to verify that the correct property was substituted for the default property in the original code.

A useful way to begin to analyze the upgrade issues in your application is to organize them into the following two categories:

- **Not upgraded and unsupported issues**. These issues indicate the presence of unsupported features, the use of control properties or methods, or the existence of statements that require manual intervention. All of the issues in this category prevent successful compilation of the upgraded application. After unsupported features are addressed, the vast majority — as much as 95 percent — of the remaining issues in this category are resolved by using simple code analyses to find solutions in Visual Basic .NET. For example, some classes and methods of the COM+ interface, such as **ObjectContext** and **Commit** are not upgraded. Equivalent members and functionality can be found in the .NET namespace **EnterpriseServices**. The remaining 5 percent of the issues require either minor changes in the application logic or in the implementation of supporting functionality.

- **Different behavior issues**. These issues indicate that the upgraded code might have behavior that differs from the original behavior in the Visual Basic 6.0 application. With these issues, you should manually review the possible consequences of the new behavior under the code context because you may need to modify the code. Issues in this category do not prevent the upgraded application from compiling successfully, but they may produce run-time errors and cause the upgraded application to fail. Experience with this category of issues has shown that only a few of them require manual intervention. Their descriptions can also provide guidance during unit testing of the upgraded application by describing the behavioral differences. By using this information, testers will know what behavioral differences to expect and can test whether or not the differences break the functional equivalence of the upgraded application. For this reason, you should not remove them from the code until after application has been successfully unit-tested.

For each category, note the different issue numbers and the frequency of their occurrence. Beginning with the most frequently occurring issues, determine the best solution and the effort required to resolve them. Because these issues occur the most frequently, you should work on them first. Next, determine solutions and effort estimations for any remaining not upgraded or unsupported issues. Then, finish up by determining the resolution and effort required for issues that relate to behavior differences. Not upgraded and unsupported issues are usually more critical to resolve than behavior-related issues. A practical way to estimate how much effort is needed to resolve an issue is to perform a test run of the upgrade where many of the issues can be resolved, and use data from this experience as a benchmark.

Producing a Project Plan

As with traditional software development projects, it is important to plan and schedule time for the following phases of an upgrade project:

- Project management (administrative tasks)
- Training for the upgrade process and for .NET
- Setting up the development environment, including setting up databases, organizing source code, and installing necessary applications
- Code preparation
- Performing the upgrade process
- Converting and validating test cases
- Code and unit testing
- Functional testing
- Transferring information and the deployment of upgraded code

As with any software development process, carefully plan and schedule so that you can meet the milestones for each deliverable. This will help you to manage the expectations of your internal and external customers. Also, be sure that when you schedule your upgrade project, you consider the rest of the projects being conducted within your company to ensure that your project has the IT support it needs.

Estimating Cost

Estimating the cost requirements for a software development project is typically a complex task because of the wide range of quality in software implementation and the varying programming abilities of developers. Estimating the cost requirements for an upgrade project is no different. Many factors affect the amount of effort that will be required to upgrade an application. Similarly, these factors affect the cost of the application; therefore, estimations for cost will vary widely depending on the application.

Some factors that affect cost estimations are common to all upgrade projects. These common factors include:

- The methodology of the Visual Basic 6.0 code; whether the code was written in an object-oriented fashion, the extent that it relies on a common set of services shared across a family of applications, and the interdependencies between parts of it that will be upgraded and parts that will remain in Visual Basic 6.0.
- The maintenance history of the original code. Applications with a history of requiring frequent fixes are likely to require frequent fixes during the upgrade.
- The complexity of the application requirements.
- The degree to which the application is integrated with other systems.
- Security considerations.
- The size and experience of your development team.
- Any third-party toolsets that are used within the application.
- The testing infrastructure and commitment to quality control.

Other factors that could affect your cost estimation are specific to the code and the platform of the application. Factors such as the use of mechanisms and constructs such as RDO/DAO data binding, late-bound objects and ActiveX documents affect estimations of cost. For ASP code, factors include the use of render functions (as opposed to Response.Write), the use of multiple languages per page, the use of nested include files, reliance on 16-bit integer size, default parameters, and GOSUB, among other specifics.

The following tasks will also affect the final estimations for the costs of your upgrade project:

- **Resolving unsupported features**. It is necessary to estimate the costs for different issues that affect unsupported features. These issues include:
  - The research costs for alternative solutions.
  - The costs of developing and testing the chosen solution.
  - The cost of replacing an unsupported feature increases the more it is used in an application. It will increase depending on the particular feature and how it is used. Multiple uses of a feature might need individual attention or in some cases may be resolved by simply using a find-and-replace mechanism.
- **Resolving reported issues**. For every issue in the upgrade report, you need to estimate the cost to resolve it. A good way to estimate these costs is to perform a test run of the upgrade. By testing the upgrade of a select portion of the application you will see an example of the type of issues that the upgrade wizard will find. With this information, you can then obtain accurate information on the cost of resolving the issues. The test run will also provide valuable insight into your team's capabilities in the upgrade process.
- **Testing**. You need to estimate the cost of testing the upgraded application. A test run (mentioned in the previous bullet) will provide valuable information on the testing requirements that you will need. By doing a test run you can evaluate whether your current testing processes will work, or if they need extensions or adjustments to certify that the upgraded application has achieved functional equivalence. You will also need to estimate the cost of testing additional functionality that will be implemented during the application advancement phase of the project.

  Also, realize that because you will be testing the entire upgraded application and not just new features (as you might do for an application that is being updated) the testing process can take longer and possibly require several iterations, which will increase your usual costs for testing.
- **Manually replacing Visual Basic 6.0 components with .NET equivalent components**. For each Visual Basic 6.0 component that cannot be automatically replaced with a .NET equivalent by the upgrade wizard, you will need to estimate the cost of manually replacing it. When you do this, be sure to consider the frequency that this feature will appear in the upgraded application.
- **Third-party .NET components**. From each third-party component that your application relies on, you will need to obtain the license cost of the .NET component that you plan to use a replacement in the upgraded application.
- **Secondary features**. If your application has secondary features such as an integrated help system, an installer, and manuals, you will need to consider the costs of adding these to your new application.

- **Deployment**. Estimate the cost of deploying the upgraded application.
- **New requirements**. You will need to estimate the costs of designing, developing and testing new features that you want to add after the application has been upgraded. Estimating the cost for the application advancement phase of the project is comparable to estimating the cost of a typical software development project.

A sensible way to estimate the cost for upgrading an application is by using your organization's experience in estimating the costs for traditional software development projects. Clearly, traditional development projects also consider general and specific factors, such as the ones previously mentioned, to ensure success. When estimating the cost for an upgrade project, keep in mind that some typical project phases may not be required and should not be included in your cost estimates.

For example, traditional software development projects include phases such as analysis, design, development, testing and deployment. In an upgrade project common phases usually include a shorter analysis phase in addition to the upgrade, testing, and deployment phases. The typical analysis and design phases are rarely needed because the source code itself is the specification.

Cost estimations for the upgrade phase of the project are based primarily on the output of the application analysis: the cost of solving different upgrade issues found during the application analysis. If appropriately addressed, solving these issues should be much less expensive than rewriting code. Pay special attention to the impact that non-supported, or obsolete features of the application may have on your costs. You can minimize these impacts by preparing or cleaning up the source code before proceeding with the upgrade process. Additionally, make sure you assess carefully the abilities of developers to handle upgrade issues.

Estimating the cost of the testing phase of an upgrade project primarily depends on the quality and the status of the infrastructure that will be used to test the application, or the portions of the application that will be upgraded. Remember that achieving functional equivalence is directly related to test cases being able to certify the expected behavior of the upgraded code and will typically define the project acceptance criteria. Therefore, if automated testing is available, consider any work needed to revise it so that it runs on the .NET Framework. Allocate budget resources to extend testing coverage where it is needed. Also, understand the depth of testing that the upgraded .NET application might require in contrast to the usual (possibly localized) testing performed on a Visual Basic 6.0 release. You may find that applying your full regression test cases to the upgraded application can take longer and require several testing iterations, which might affect the usual testing costs. In general, testing costs have two parts: creating the test cases and running them. Because the goal here is achieving functional equivalence, the test case creation

phase can be skipped if your original tests cases are adequate and it is only necessary to adapt them to the new platform. As for any development project, the execution of test cases must be done in an iterative way as you get closer to completing the project.

Finally, estimating costs for the deployment phase of an upgrade project is comparable to the deployment costs of traditional software development projects. Of course, you need to train your team to understand and deal with the issues of .NET deployment. The good news is that generally, .NET deployment consists of a much more streamlined mechanism than COM-based applications.

Using the information in this section as a guide, you should be able to estimate costs for the work-hours that are required for the upgrade project. After you complete the first estimate, you can further refine the figures and make adjustments as necessary. This process can also apply to the upgrade plan. When you estimate project costs in this way, you will notice that the main difference in expenditures for a traditional software development project and those for an upgrade project involve the testing phase. The testing process for a traditional project represents approximately 20 percent to 30 percent of the total cost. The testing process for an upgrade project can easily represent 60 percent to 70 percent of the total cost. However, the overall work-hours required for an upgrade project are considerably fewer than those required for a traditional software development project.

Upgrade tests performed by Microsoft and its partners found that when code is adequately prepared prior to the upgrade, the cost of upgrading from Visual Basic 6.0 to Visual Basic .NET is typically 15 percent to 25 percent of the total development cost. This figure takes into account the learning curve of Visual Basic developers who move from Visual Basic 6.0 to Visual Basic .NET. The next section investigates how to best prepare your application for the upgrade.

Preparing for the Upgrade

After finish the planning stage of the project, the actual implementation of the plan will begin. The first step in the implementation of this plan is not necessarily a head-first dive into applying the upgrade wizard, and then resolving the individual errors, warnings and issues (EWIs) produced by this tool. Although this will work, it is not the best course for upgrade projects other than those that involve the simplest of applications.

More complex applications will benefit from the initial preparation of searching the source code for specific and common practices (such as the use of default properties) and then modifying the code. The modifications allow the upgrade wizard to understand the code more easily and as a result, make much better decisions about correctly translating the code to Visual Basic .NET. At first glance, this preparation

may seem like additional work, but in the end, these early modifications will save a great deal of work.

Before preparing the source code for the upgrade, it is important to first prepare the development environment that you will use to upgrade the application.

Preparing the Development Environment

This section describes the basic set of tools that are needed by the developers who will work on the upgrade of the application. The tools are introduced in this section and some are described in more detail in other chapters. The introductions identify the tools you need, explain why the tools are needed, and where to get more information about the tools.

Visual Studio 6.0

Visual Studio 6.0 will be needed to prepare the Visual Basic 6.0 code for the upgrade and to ensure that the code will compile before you upgrade it using the upgrade wizard. This IDE can be installed on any computer without conflicting with the other tools you will need for the upgrade.

During the preparation of the source code, you will use a list of tasks that will have been produced by the Visual Basic 6.0 Code Advisor, as discussed later. These tasks involve modifying the source code and then recompiling. All this work must be performed inside Visual Studio. NET.

With Visual Studio 6.0 source code, you will use a list of tasks that will have been produced by the Visual Basic 6.0 Code Advisor, as discussed later. These tasks involve modifying the source code and then recompiling. All this work must be done inside Visual Studio 6.0.

It is also quite common to run the upgrade wizard, review the upgrade report, and then to return to the Visual Basic 6.0 to make further modifications.

Although these tasks can be performed on a different computer, you may find it more convenient to do it all on a single computer, especially if you find yourself switching between the original code and the upgraded code.

Visual Basic 6.0 Code Advisor

The Visual Basic 6.0 Code Advisor is an add-in for Visual Studio 6.0. It can be used to scan your Visual Basic 6.0 source code for practices that do not comply with configurable coding standards. These standards are based on practices developed by Microsoft to produce robust and easy-to-maintain code. The code advisor will also highlight issues in your code that make it difficult for the upgrade wizard to automatically convert your code to Visual Basic .NET.

The code advisor must be downloaded and installed on your computer before you can use it. It is not installed automatically with Visual Studio 6.0. For more information about the code advisor, see Chapter 5, "The Visual Basic Upgrade Process."

Visual Studio .NET

Microsoft Visual Studio .NET is the development environment that you will use to develop and maintain Visual Basic .NET applications. You will also use it to run the upgrade wizard that will convert your Visual Basic 6.0 projects to Visual Basic .NET projects.

Visual Studio .NET is an advanced integrated-development environment that has many new and improved features that were not available in Visual Studio 6.0, as well as most of the features that you are used to.

You can build applications for multiple languages and platforms within the same IDE and even within the same workspace or solution. You can build applications in Visual C++, Visual Basic .NET, Visual C#, and Visual J# for desktops, servers, the Web, and mobile devices. There are also great new features that ease deployment such as built-in Windows Installer technology for both Windows and Web deployment. Maintenance has never been easier with features, such as side-by-side application execution that cuts down on, or even eliminates, DLL versioning issues.

Another big addition to this version of Visual Studio is the support for Web services. Creating a Web service requires only a little more effort than any other class in Visual Basic .NET. The advanced support for deploying and debugging Web services allows you to focus your efforts where they are most valuable: on building your application.

A feature that is not available in Visual Studio .NET is **Edit and Continue**. This feature allowed you to modify code in the debugger as you continue to run the application with the new code without having to restart the application. This feature is very helpful in Visual Basic 6.0 because you can efficiently make small changes to code during debugging. This feature will be added to a future version of Visual Studio.

For more information about Visual Studio, see the Microsoft Visual Studio Developer Center on MSDN.

Visual Basic Upgrade Wizard

The Visual Basic Upgrade Wizard is the tool that you will use to perform the automatic steps of converting your Visual Basic 6.0 source code and project files into Visual Basic .NET. The upgrade wizard is built into Visual Studio 2003 and runs automatically when you try to open a Visual Basic 6.0 project. There are no special steps required to install or run it.

For more information about the upgrade wizard, see Chapter 5, "The Visual Basic Upgrade Process."

ASP to ASP.NET Migration Assistant

The ASP to ASP.NET Migration Assistant is an add-in to Visual Studio .NET that can be used to help you upgrade your ASP pages and applications to ASP.NET.

This migration assistant will rename files, update hyperlinks, and it will also fix syntax differences in the VBScript between ASP and ASP.NET. It will also attempt to resolve the correct type of any variable that is not explicitly typed in VBScript code and therefore relies on late-binding to resolve default properties that are not supported in ASP.NET.

For more information about the migration assistant, see Appendix C, "Introduction to Upgrading ASP."

Preparing the Visual Basic 6.0 Source Code

Almost without exception, if you take steps to prepare your application in Visual Basic 6.0 before upgrading it, the process will be more efficient. Certain common practices in Visual Basic 6.0 produce code that is correct and humanly readable but that the upgrade wizard cannot appropriately convert into Visual Basic .NET.

Several tools can be used to provide a list of issues that will come to your attention when you pass your code through the upgrade wizard. Such a list can help determine areas of your code that may be easier to modify in its original source code, especially if your developers know Visual Basic 6.0 better than they do Visual Basic .NET.

The Visual Basic 6.0 Code Advisor, described earlier in this chapter, should be the first tool that you use when you start preparing your application for the upgrade. The second tool should be the upgrade wizard. It produces a report that contains all of the errors, warnings, and issues that it finds during the upgrade. The report frequently includes issues that the code advisor does not identify. Addressing the issues indicated by the upgrade report may require several iterations to best prepare the source code before you can achieve functional equivalence. Frequently, several issues reported by the upgrade wizard are related to the same problem that when fixed, simultaneously eradicates many other issues. For example, if you use the default property in a late-bound variable, specifying the type of the variable explicitly will remove all issues related to the use of the default property. Experience using the upgrade wizard shows that, on average, fixing a single issue in the Visual Basic 6.0 source code prior to the upgrade resolves five to eight issues that the upgrade wizard would have reported. Figure 2.3 on the next page shows a graphical representation of the iterative process for modifying code based on suggestions in the upgrade report.

For more information about how to prepare your code for the upgrade, see Chapter 5, "The Visual Basic Upgrade Process."

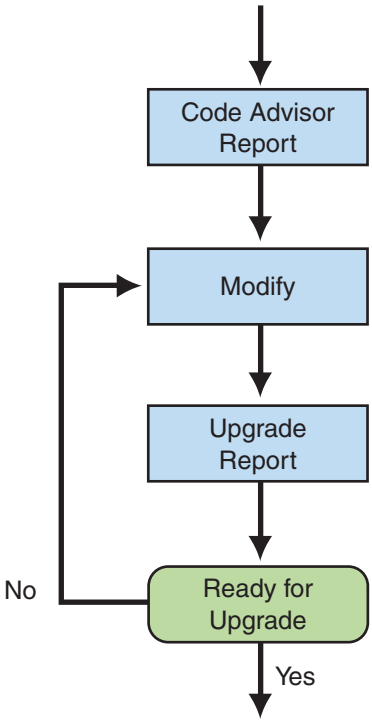

Figure 2.3
A graphical representation of the code preparation phase

Upgrading Applications Written in Earlier Versions of Visual Basic

The upgrade wizard is designed to upgrade Visual Basic 6.0 applications. For projects written in Visual Basic versions 1.0 through 5.0, it is necessary that you first upgrade them to Visual Basic 6.0 before upgrading to Visual Basic .NET. To upgrade a project that was developed in a version earlier than Visual Basic 6.0, simply open the project in the Visual Basic 6.0 IDE and save it. If Visual Basic 6.0 prompts you to upgrade controls to Visual Basic 6.0, choose **Yes**. If the project contains Visual Basic 5.0 ActiveX controls, it is often best to replace these controls with Visual Basic 6.0 versions. This is because these controls use a different threading model than models used by Visual Basic 6.0 controls. The earlier threading model is not supported in Windows Forms.

For 16-bit projects written in Visual Basic versions 1.0 through 4.0, you may need to make extra modifications to the application to convert it to Visual Basic 6.0. Some

VBX controls will not be automatically converted. You will also have to replace Win16 Windows APIs with their Win32® counterparts.

Visual Basic versions 2.0 and 3.0 often require an extra step. Visual Basic 6.0 can only open files in text format, whereas Visual Basic versions 2.0 and 3.0 support two file formats: binary and text. Before upgrading these projects, ensure the entire application is saved in text format by using the following procedure.

► **To convert Visual Basic 1.0 and 2.0 files to text format**

1. On the **File** menu, click **Save As**.
2. In the **Save** dialog box, select the **Save As Text** check box.

Because Visual Basic 1.0 can only save files in binary format, all of these projects will first need to be opened in Visual Basic 2.0 or 3.0 and then saved as text before they can be converted to Visual Basic 6.0. After converting the project to Visual Basic 6.0, you can begin the process of upgrading it to Visual Basic .NET.

Verifying Compilation

The upgrade wizard is designed to work with valid Visual Basic projects, so it is very important to verify that the code is syntactically correct before trying to upgrade it. If files have errors the upgrade wizard might have problems when it is trying to parse the code and you may get some misleading error messages or, worse, unexpected results. Correct syntax can be verified by compiling the Visual Basic 6.0 project before applying the upgrade wizard.

This also helps you ensure that all the necessary source code is available. The upgrade wizard first analyzes all the code, and then uses the information it obtains to make decisions. It then starts the conversion of the code. If there are missing files, some of the decisions that the upgrade wizard makes may prove to be incorrect.

Compiling the original source base in Visual Basic 6.0 before applying the upgrade wizard will help ensure that the tool will have all the information it needs to make the best choices possible. The better the decisions the upgrade wizard makes, the less manual intervention will be required to achieve functional equivalence.

Upgrading the Application

After the code is prepared, it is time to convert the code and produce the functionally equivalent .NET application. This is done in two steps.

First, you use the upgrade wizard to load the prepared Visual Basic 6.0 code into Visual Studio .NET. If you have prepared your application as outlined in the previous section, this step will be almost effortless because any issues of missing files or compilation problems are resolved. The code that is produced by the upgrade wizard during this step will be used as the baseline for all further development of

the upgraded application. At this point, the generated code is referred to as *green code* because it is fresh out of the upgrade wizard and has not yet been manually modified.

In the second step, you modify the green code so that it will compile and replace all remaining code that was not completely upgraded with valid Visual Basic .NET equivalents. Samples of the kinds of modifications that are necessary are listed in the "Preparing to Handle Upgrade Issues" section earlier in this chapter.

For more information about these two steps, see Chapter 5, "The Visual Basic Upgrade Process."

After the application successfully compiles and all issues that were reported by the upgrade wizard have been addressed, you are ready to begin testing. Testing is discussed in the following section.

Testing and Quality Assurance

Testing ensures that an application and its components behave according to their individual specifications. Does your program do what is expected of it? Does it perform as fast and as well as expected? A good testing process will ensure all of these requirements are met.

Continuous testing is a vital part of software development. This is equally true in an upgrade project. Continuous testing allows you to find bugs early, it can prevent the reappearance of bugs, and it can create confidence in the software and the development effort because developers can be sure that they are not introducing new bugs. All of this saves time and money, and increase customer satisfaction because customers experience fewer bugs when they receive the final product.

Any change to code has the possibility of introducing bugs. As the application is upgraded and the new version evolves, you will need an appropriate set of regression tests to guard against bugs.

For detailed information about testing your upgraded application, see Chapter 21, "Testing Upgraded Applications."

Deployment

After you have upgraded and tested your Visual Basic .NET application, it is time to start the deployment of the completed application. The main objective of deploying .NET applications is the same as deploying Visual Basic 6.0 applications: you need to install your application and get it running on the client's computer or on the production servers. However, the way you do this, the dependencies that your application has, and the mechanisms for handling updates and security have all changed. The

reason for this is the .NET assembly — the new packaging mechanism for .NET applications.

The primary new dependency that you will have to confirm when deploying your .NET application is whether the .NET Framework is installed on the target computers. If there is a chance that the .NET Framework is not installed on the target computers, you should consider providing a way for your users to install it.

Assemblies

The smallest deployable unit of a .NET application is called an assembly. An assembly is a collection of classes, configuration files, resource files, and any other files that your application needs at run time. An assembly can be an executable (EXE) or a dynamically linked library (DLL).

What makes an assembly different from previous types of executable files and DLLs is that the assembly contains extra information, called metadata, about itself and about the classes it contains. Among other things, the extra information that an assembly contains helps distinguish different versions of the same assembly.

The assembly also contains security related information that can ensure that the assembly comes from who you think it comes from and that it has not been modified since the original creator released it.

Private and Shared Assemblies

By default, assemblies are installed into the same directory as the owning application. This type of assembly is called a private assembly. Because it is installed in the application directory, there is no chance of it interfering with other applications installed on the same computer.

You can also create shared assemblies that are installed into a system wide repository for assemblies called the global assembly cache. These assemblies can be used by multiple independent applications. The advantage of these assemblies is that they use less disk space and less memory.

Versioning

You can easily install multiple versions of the same assembly in the global assembly cache and the global assembly cache will maintain these assemblies separately. These versions can have the same name and the same GUID (for exposure to COM) but have different versions. The .NET Framework will ensure that only the version of the assembly that your application was compiled against will be linked to it at run time. For you to change the version of a DLL that an application uses without reinstalling the application, you need to insert special directives into the configuration file for that application. This installs an incompatible version of a DLL, thereby

effectively eliminating the problem of new applications interfering with applications that are already on your computer.

The global assembly cache also distinguishes assemblies that have the same version but were compiled for different architectures or different processors. For example, you can install the same .NET assembly for 32-bit and 64-bit platforms and the global assembly cache will automatically choose the most suitable assembly for your application. You can even optimize your assemblies for different models of 64-bit processor and the global assembly cache will automatically pick the one that is most appropriate.

Side-by-Side Execution

The benefit of allowing different versions of the same assembly to be installed on the same computer and shared by different applications is that you can now easily install multiple versions of your .NET applications on the same computer and allow them to run side by side without interfering with each other.

Configuration

Deployment and side-by-side execution are also facilitated by the fact that .NET applications do not need to keep any configuration information in the Windows registry. This eliminates the registry as a source of collisions between different versions of an application.

Configuration information for .NET applications is kept in XML-based configuration files in the same directory as the application itself. Another file exists that applies to all the versions of the same application, while a third exists at the computer level.

Update Deployment

As previously mentioned, an installed application will only normally link to the same version assembly that it was compiled against. However, it is possible to add a configuration setting to the application's configuration file to tell it to stop using the version of a DLL that it was installed with and to start using an updated DLL. This redirection can only occur with explicit redirection. This makes it easy to modify installed applications with service packs or application updates. In this situation, deleting the old DLL and installing a new version is not enough; this will simply stop the application from running.

Microsoft Windows Installer

Visual Studio .NET provides special project templates that allow you to build installers for different types of applications.

After you have chosen a project, you can customize the installation procedure by modifying the registry, adding file types, adding shortcuts and any arbitrary files (including documentation), and specifying conditions that must exist on the target

computer before the installation will continue. In addition, you can add your own custom installation actions and application specific dialog boxes to the installation process.

The setup projects offered by Visual Studio .NET are usually built to create Windows Installer files (.msi) that can be stored and distributed on a CD, on a Web site, or on a shared network directory for installation across a network.

Visual Studio .NET provides the following deployment projects for installing different types of applications:

- **Setup project**. This is a generic Windows application setup project that you can add arbitrary files and actions to.
- **Web Setup project**. This kind of project is intended for installing Web applications and can automatically handle issues with registration and configuration.
- **Merge Module project**. This type of project is used for deploying shared components by incorporating them into other installation projects.
- **Setup Wizard**. This is not a type of project, but a wizard that helps you build an initial setup project from existing Visual Studio .NET application projects. These projects can be customized later.
- **Cab project**. A Cab project allows you to assemble a collection of files and project outputs for compression and for copying onto other computers. This type of project is commonly used for packaging ActiveX controls.

Advancing an Application

After testing is complete, you should have a working application that is functionally equivalent to your original Visual Basic 6.0 application. In many cases, the reason for upgrading an application in the first place was to add new features to it. After you have functional equivalence, you can begin application advancement to achieve the results you envisioned. All new development will be done in Visual Basic .NET.

Application advancement is the improvement of the upgraded application beyond the specifications of the original application. It often includes adding features that are easier to implement in Visual Basic .NET than they were in earlier versions of Visual Basic. For some features, such as threading, it may be the first time it is possible to implement the feature in your application.

For detailed information about application advancement, see Chapters 17, “Introduction to Application Advancement,” Chapter 18, “Advancements for Common Scenarios,” Chapter 19, “Advancements for Common Web Scenarios,” and Chapter 20, “Common Technology Scenario Advancements.”

Managing an Upgrade Project

In terms of management, upgrade projects differ from standard development projects in ways that are important to understand. This section discusses many of the best practices and techniques to ensure that your upgrade project goes as smoothly as possible.

The practices and techniques described here come from experience with many Visual Basic upgrade projects. Microsoft and their partners have learned some valuable information by upgrading millions of lines of Visual Basic 6.0 code to Visual Basic .NET. This section shares some of those lessons with you to give you the benefit of this experience.

Change Management

You will find that source management issues are more complicated during an upgrade project than they are during typical application development projects. If you do not have a source control system you should consider purchasing one; it will save a lot of headaches caused by multiple programmers working on the same set of files and overwriting each other's work.

Using Source Control

A source control system is a software program that manages the work done by a group of two or more developers who are working on a single set of source files. It prevents the developers from modifying the same source files at the same time only to lose valuable work as a result.

Different source control systems can have widely varying features for supporting different types of files and different numbers of users. However, most source control systems will have the following basic set of features:

- **Storage of the entire source tree and related files**. The source control system includes a directory structure that contains all the source code and related files for your application. This directory structure is referred to as a *source tree*. The source tree is centrally managed by the source control system and is stored on a server that is accessible by all developers.
- **Exclusive access to files**. Users can *check out* a file for temporary exclusive write access. While checked out a file can only be modified by the user that checked it out. Other users cannot check out the file or modify it during this time; they can continue to read the file, but they only see the version of the file that existed before it was checked out. After the user has finished modifying the file they can check it back in and the changes are available to all other users. Another user can then check out the file to work on. This way a developer is much less likely to overwrite changes made by another developer.

- **Version history**. As files are changed and the application progresses, source control systems keep histories of the files that allow you to see how the application looked at any point in time. This feature makes source control systems useful even for single developers who need to keep track of previous versions of their source code. For example, in most systems you can label versions of source files so that you can easily extract a version of the source control that matches the version that a particular customer has. With the exact version you can more easily reproduce bugs.
- **Source branching**. Most source control systems allow you to have multiple branches in your source tree where different teams of programmers are working on different versions of the application simultaneously. Usually, at some point, the different versions are merged into a single branch again. For example, in the case of the customer who has a bug in their version of the application, you may need to replace the application with a more current version. However if you have continued to develop the application and it is not yet stable or you do not want to give them new features that are in the current version, you can branch the code at the version of the source that they have and fix the bug within their version. The resolution of this bug can then be merged back into the main source branch later.

Some of the advantages of using a source control system are:

- You minimize lost work due to programmers overwriting each other's work.
- You always know who is working on each source file and can coordinate multiple changes to those files.
- You simplify the coordination of multiple programmers working on the same application.
- You always have access to any previous version of your application.
- You have a controlled way to release a single programmer's source file changes to the rest of the development team.

Dangers of Multiple Check-Outs and File Merging

Managers of source control systems sometimes allow developers to check out the same source files at the same time. When this happens you might find that two developers have made independent changes to the same file. Many source control systems allow this and provide tools to help merge those changes. Usually the developers will coordinate with one another to minimize the impact by modifying different parts of the source files.

When development work begins on the code generated by the upgrade wizard, many of the source files for the application will not compile. Also, some of these files will have complex interdependencies that prevent them from compiling until the files that they depend on are compiled first.

It is tempting to solve this problem by allowing multiple check-outs of source files. If you do, developers can work on the source files independently to resolve issues so that dependent files can compile. This should be avoided in Visual Basic upgrade projects, especially at the beginning when the project is first being compiled. In most cases two developers working on the same file will have to make the same set of changes to that file so that it can compile. This results in duplicated work and sometimes a more difficult merge because the same lines of code have been changed by more than one developer.

Source Branches

As previously discussed, many source control systems will allow you to create independent branches in your source code where there are multiple versions of your application being developed independently. Depending on the source management policies of your office, you will probably use several different source branches during the upgrade project. The following source branches may be included:

- **Original source branch**. The original source code will usually have its own branch. If there is any chance that you will need to continue development of the original Visual Basic 6.0 application, you should leave the original version of the application in its own branch and create a new branch for the upgrade.
- **Code preparation branch**. As previously discussed, code preparation involves modifying the original Visual Basic 6.0 source code to prepare it for use with the upgrade wizard. Although the application remains a valid Visual Basic 6.0 application, the modifications involve changes to the source code that can introduce bugs.

  Some development centers have strict restrictions about changes made to the source code and rules about how those changes are tested and even checked in. For example, it is common for many companies to insist on an initial set of tests before files are even checked in; these are referred to as check-in tests.

  In these cases it is useful to create a source branch especially for the code preparation. In this way all of the changes that you make to the original application are kept separate from the main branch of development and the original application can continue to evolve.
- **Green code and functional equivalence branch**. After the code is prepared you will pass it through the upgrade wizard and a new Visual Studio .NET solution will be created. As previously explained, this new solution is referred to as green code.

  The green code is a brand new version of your application, independent of the source code branches that contain the original and the prepared code source trees. When you are happy with the output of the upgrade wizard, all further development will apply to the green code.

Instead of using a new branch from the original application, it is common to create a brand new source tree. This new source tree will be used as the basis for the next step in the upgrade process: compilation.

In fact, after the upgrade wizard has generated the source tree, the green code is typically used for the remaining steps of the upgrade. The first step is to achieve functional equivalence with the original Visual Basic 6.0 application and the second step is application advancement that involves adding new features and improving existing functionality. All of this work is serial and can be performed on the same source tree just as with any other development process.

At times, you may want to create reports of the changes that were made to the green code. As previously mentioned, your source control systems provide a mechanism to generate a copy of the original green code files and a copy of the source code at any point. With these reports, you can determine the exact changes made to the source code by comparing the files of the green code versions to the functionally equivalent version.

Managing the Project after Using the Upgrade Wizard

After you use the Visual Basic Upgrade Wizard, your application will have a large number of source code files that do not compile. It can be a challenge to determine the best way to manage the compilation of the files, especially if you have a large, complex project. Certain goals will be obvious but how to reach these goals may not be. The following tips can be helpful in managing the compilation of source code files after you use the upgrade wizard:

- Make sure that all bottlenecks are identified early. Avoid having one developer work on a single file that the rest of the team must wait for before they can do their work.
- Start work with the files that have the fewest dependencies on other files but that also have the greatest impact on the application.
- Make sure that the same problems are not being fixed multiple times by different developers.
- Minimize the number of bugs that are introduced by developers making unnecessary changes to the code.
- Minimize repetitive and laborious tasks that can be avoided.
- Minimize the effort that is required to achieve functional equivalence for your application.

Handling Non-Compiling Files with Complex Dependencies

If you take a random file from your green code and attempt to compile it you will find references to objects in many other files that are not yet compiling. You will find that you cannot complete the file you are working on until the other files that it

references are also compiling. If you have multiple developers working on separate files you will find that they all will have dependencies on common files and they will have to wait while one person fixes the file they all need.

The obvious way to minimize occurrences of this scenario is to determine the files that have the fewest number of dependencies on other files and to start correcting those. Another characteristic to consider is how many other files depend on the files that you start working with. The files that are most important to start with are those that depend on no other files but have many dependants. This way when the file does compile you enable the compilation of as many other files as possible at the same time.

The assessment tool provides a lot of valuable information and even suggests the order that files should be handled during compilation. For more information about the assessment tool, see Chapter 5, "The Visual Basic Upgrade Process."

Prioritizing File Compilation

Obviously, before your application reaches functional equivalence all of the files must compile. To keep your developers productive, make it a priority to fix files so that they can compile, even at the expense of temporarily losing some functionality. Needless to say, the interdependence between files can make compiling those with dependencies on other files a frustrating task.

You will often find that to proceed with work on some files, certain other files must be able to compile first. Occasionally, you will also determine that some groups of files must be worked on simultaneously. This might be because of co-dependencies, where files depend on each other, or where you simply need to start making progress on some files even though the files they are dependent on are not yet ready. In these cases, an iterative approach is typically the best way to manage compilation work on several files at the same time.

The first step is to address all of the *local* compilation issues in the file. These are issues that are caused by, as well as fixed by modifications to the file itself. For example, these can be syntax issues, missing functionality issues, or uses of default properties. Issues that are caused by problems in an external file can be skipped during this step.

Each developer should fix all of the issues that they find in their respective files and then check in the modified files for other developers to use.

After all of the local compilation issues have been addressed, the developers can address the *global* or *external* compilation issues. These are issues that are caused by a problem in a different file that is being edited by a different developer. Global compilation issues are problems with the code that can only be resolved by editing files that are external to the file being compiled. Many of these issues will be resolved automatically by the developers that solve local compilation issues in their own files. The rest of these issues will have to be specifically addressed.

As developers resolve local issues and check in their files, other developers should perform additional searches for local issues in their own files. Often a change in one file will fix the problem in another file, effectively converting what was once a global issue into a local issue.

With this approach, you will gradually move your files into a state where they will compile successfully. This process will be easier if the files have as few external dependencies as possible

Shortcuts for Getting Files to Compile

A common strategy for getting files to compile as soon as possible is to comment out functionality that has not yet been upgraded completely. Commenting out code to make it compile is effectively postponing the resolution of problems until later. This can be good and bad.

If the problem you are avoiding is a superficial fix to the GUI, or involves a piece of isolated functionality in the behavior of the file, or application, it may be okay to comment out. However, if it is a fundamental piece of functionality whose absence is difficult to notice during routine testing, such as logging messages, inserting or modifying data in a database, or including certain data financial calculations, it may be costly to comment out because the missing functionality might go unnoticed until much later when it will be more costly to fix.

If you do decide to comment out functionality to quickly get files to compile, it is important that you do not forget about it, and thereby introduce new bugs into the application. You should record what you comment out, either in a task list in Visual Studio .NET or by having the application print a message each time that area of the code is executed. Record it in a way that the commented-out functionality will be noticed and can be addressed later.

Divide and Conquer

As with managing most big tasks, a good approach is the classic divide-and-conquer strategy. There are two ways to apply this concept to your upgrade project. The first way is to delegate the work based on projects or files. Assign individual project or source code files to your developers and have them focus on getting them to compile first. As previously mentioned, the assessment tool (available for download from the community site for this guide, as discussed in the "Feedback and Support" section of the Preface) will help you determine the optimal order for working on the files. The goal here should be to get all of the files to compile. Whether you do this with projects, groups of files, or individual files really depends on the sizes of the files.

After all of the files are able to compile, there are usually remaining issues that concern missing functionality that need to be resolved. Either code was commented out, or green code was marked as code that potentially behaves differently. These issues usually result in a list of tasks in the Visual Studio .NET Tasks window and a list of EWIs and other code that was commented out while making the files compile.

You can continue to delegate work based on project or source code files. Alternatively, it may now make sense to delegate work based on EWIs or tasks. You can assign specific EWIs to developers or you can assign them selected lists of tasks. You may want to do this when some issues need specialized knowledge and it makes sense to assign them to appropriately capable developers. The advantage of this is that it allows your developers to specialize in certain aspects of Visual Basic .NET to speed up progress on the project.

The following common tasks can be assigned to developers instead of assigning them files to compile:

- **Typing variables that have no type or are typed as objects**. The upgrade wizard that ships with Visual Studio .NET does not attempt to induce the type for variables that are not typed explicitly or are assigned some generic type. (The ArtinSoft Visual Basic Upgrade Wizard Companion will do this. For more information about the upgrade companion, see Chapter 5, "The Visual Basic Upgrade Process.") This leads to variables that have inappropriate types and the inability to recognize the use of default properties. An important task that needs to be done is to find variables in each file whose types have been upgraded to **Object** and assign them a more suitable type.
- **Manually fixing code that did not convert correctly**. Typically, functionality that was not upgraded correctly by the upgrade wizard in one file also appears in other files in your application. An API such as the printing API, and the use of the Windows Forms collection that has no direct equivalent in Visual Basic .NET, are examples of functionality that can not be upgraded and will appear in multiple files in your application. To fix these problems, you can assign developers specific APIs and have them review and upgrade the uses of those API's in your application.
- **Testing ActiveX/OCX components**. Many ActiveX and OCX components will work with the wrappers that are generated by the upgrade wizard however, some will not. It is useful to know the ones that will not work at the outset of your upgrade project. Before you start your upgrade project, determine the Active X and OCX components that will not work in the .NET environment. You should do this outside the application's context to simplify the fixing of bugs in the component. This ensures that any problems are a result of incompatibility between the component and the wrapper, or between the component and the .NET Framework, and not because of issues within the upgraded application.

- **Removing dead code, redundant declares, structures, and consts**. It is common when using functionality from DLLs to copy and paste code that contains all the necessary **Declare** statements from Web pages or help files and to include a lot of code that is not required because you do not need the functionality they provide access to. This leads to a lot of redundant external references that can clutter up your code and make the upgrade more difficult. These statements can be cleaned out of your green code at any point, if they were not removed during code preparation.

Upgrade Project Complications

Even well prepared code may have complications that appear during the upgrade process. This section investigates common complications and how to address them in your upgrade projects.

Managing a Continuously Developing Visual Basic 6.0 Application

If your company has active technical support and customer service departments along with a large user base for you application, you may find that your application is under continuous development. Typically, such development cannot be put on hold during an upgrade. The primary solution to this problem is to create a source branch in your source control system (or simply create a copy of the source code) and to upgrade a version of the application from some specific point in time when the application is functionally stable.

After the application is upgraded, you will have a Visual Basic .NET application that is functionally equivalent to an older version of your Visual Basic 6.0 application. If all of the changes that were made to the original application up to that point are recorded in sufficient detail, you should be able to quickly add the new functionality to the .NET application to bring it up to date.

Practically speaking, if this scenario applies to you, the process of moving your user base from the old version of the application to the new version of the application will be a major undertaking in itself and will overshadow any minor bugs reported while the application was being upgraded.

Dealing with Strict Compilation Requirements on Checked-In Source Code

This section only applies to you if you are already using a source control system and your office policies restrict changes that can be made to your application's source code. Policies like this might require check-in tests to be performed before you check in code or they may require expensive quality assurance cycles after you check it in. If this is the case for you, you can create an independent branch in your source control system that you can use to perform the code preparation. Any changes you make to your source code are independent from the main branch, and you are free to check in the source code as you want.

Handling Multiple Visual Basic 6.0 Projects with Shared Files

A problem can occur with files that are shared between projects because the upgrade wizard will not recognize these files as shared. It will upgrade them repeatedly, once for each project that uses them. This is easy to spot because the upgraded files will have the same names and namespaces in Visual Basic .NET; however, they will not show up until compilation.

You can create a new Visual Studio .NET solution to hold all the projects that you will need to manually review. In it, you can see the files that were shared and are now repeated in different projects to determine which project the files really belong to. With this information, you can fix the references accordingly.

Care should be taken when you do this because there are likely minor differences in the shared files because of the context differences in the way they were upgraded.

Handling Circular Dependencies in Visual Basic 6.0 Projects

Circular dependencies are a sign of bad programming practice, but they do exist. They are caused by groups of two or more projects being co-dependent. For example, two projects are co-dependent if each project had a reference to a DLL in the other.

If this happens, the upgraded projects will suffer from the same co-dependence. To fix this issue you will have to locate the sources of the dependencies and try to resolve them so that they are no longer circular. You can usually do this by moving code from one project to another. It is preferable to make these changes during the code preparation phase of the upgrade because it will lead to better source code in the upgraded application.

Managing a Partial Upgrade

If you have decided to perform a partial upgrade (meaning that some of your application will remain in Visual Basic 6.0 while you upgrade the rest of it), you need to decide how to handle the source code management of this process. Obviously, the issue is that some components of the application are being removed and replaced by new .NET components while others are not.

Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET," examines in detail interoperability between Visual Basic 6.0 and Visual Basic .NET. Refer to this chapter to familiarize yourself with the concepts and their limitations before undertaking a partial upgrade.

In terms of the management of source code in a partial upgrade, you have two approaches to choose from; both will probably apply.

In the first approach, you replace a component, such as a COM object or DLL, with a .NET DLL. If you can manage to do this without changing the code that uses the

DLL, you will have a much easier process. In this case you simply create a new .NET project that will eventually replace your old DLL. Whether or not you are using a source control system, it is a simple process because you create the new DLL independently of the current DLL and start using the new DLL when it is ready.

The second approach involves replacing the user of a component, such as a COM object or DLL, with a .NET equivalent. The development of the client is independent of the original client and the new client is used when it is ready. If you can avoid changing the shared component, the process is much easier. If the component must be changed, try to make the change to the original project before starting the upgrade instead of during code preparation.

Using Historical Information

Every application is different. It is impossible to know how many and what kind of upgrade issues your application will have. To provide some guidance for you, this section provides a few general statistics that are based on a large number of successful upgrades that have already been performed. Please note that all of the statistics in this section are from projects that were first prepared in Visual Basic 6.0 for the upgrade to Visual Basic .NET. If you do not prepare your application in Visual Basic 6.0 first, you can expect your actual numbers to be much higher. However, do not be afraid of a large number of issues being reported by the upgrade wizard. Most of them will be very simple to resolve after you understand why they have occurred and how to correct them.

The five most common upgrade issues generate 88 percent of the total issues. Table 2.1 lists them with the percentage that each generally occurs:

Table 2.1: Top Five Upgrade Issues

Issue	Occurrence
Could not resolve default property of object "<objectname>"	52%
Property/method was not upgraded	13%
Property/method/event is upgraded but has a different behavior	12%
COM expression not supported	7%
Use of **Null**/**IsNull()** detected	4%

These are all *micro* issues. Micro issues are commonly-occurring, easy-to-fix problems. From these projects, 41 percent of the issues occur in the design of forms, 59 percent of the issues occur in modules, classes, code-behind-forms, and other project items. On average, modules, classes and code-behind-forms have one upgrade issue for every 106 lines of code; forms have one upgrade issue for every five controls.

Best Practices for Performing Your Upgrade

The following recommendations can help you maintain better control over the execution of your project and will also help you manage potential risks associated with the upgrade:

- Software applications are usually living assets in their companies, so freezing the code development on an application is hard to accomplish. Do your best to perform the upgrade during a period when you can restrict the coding of that application to solving major bugs. This allows you to accelerate the upgrade process, reduce the probability of having to make changes to the original application as well as the upgraded one, and simplify the source code control during the upgrade. If it is not possible to halt development, you must upgrade a frozen copy of the source code and record changes and bug fixes that must be applied to the application during or after upgrade.
- Try to assemble an upgrade team composed of developers and testers who know the Visual Basic 6.0 application, and developers and testers who have experience with the .NET Framework. This combination of team members will handle technical aspects of the upgrade, while team members who have the required business skills but not necessarily the technical knowledge can ramp up to speed.
- Make your upgrade project a priority and ensure that your developers are dedicated to it. This support helps your team focus and begin working on the new code base.
- Understand and categorize the upgrade issues that need manual intervention. While many of them can be fixed quickly and mechanically, others require closer attention. Assess your team's abilities to fix the different categories of issues, including their potential affect on the project schedule.
- Share upgrade knowledge with your entire team. A single solution may apply to all occurrences of several issues found during the upgrade. Keep your developers up-to-date with the solutions that are being implemented by others. You can do this with knowledge base software, documents that are posted to your intranet, regular team meetings, and any other knowledge dissemination mechanism that your company uses.
- Make achieving functional equivalence a goal. When your Visual Basic .NET application is functionally equivalent to the Visual Basic 6.0, you have a solid basis for adding new functionality. Even if the upgraded application does not entirely follow the recommended practices for the .NET Framework, or it has not been fully optimized, you have the opportunity to validate its functionality. Then, in a controlled way, you can begin improving the application with fine-tuning, security implementation, and remote deployment, in addition to many of the other features that are offered by the .NET Framework.

- When solving a problem during the upgrade, keep in mind that you want to preserve the functionality, not the code. Sometimes it is better to replace an entire piece of code with functionality that is provided by the new platform than to try to re-implement the old features in the new platform just to keep things exactly as they were in the original application.
- Divide the testing phases into partial testing and full testing. Partial testing can be understood as code and unit testing; it is focused on isolatable pieces of the application such as components, classes, a particular method, and so on. Full testing — also referred to as system testing — comprises the testing of the functionality of individual components, the functionality of the entire application, and the application's integration with other applications. Partial testing will find errors before you integrate the entire system. It is not unusual to find bugs in the upgraded application that also exist in the original application but that have not yet been detected. Differences in the execution of the applications in the two environments can identify such errors. Resolving these bugs may be simpler when they are found during unit (localized) testing instead of during full system testing.
- Test cases are a great measurement tool for quantifying the progress of the upgrade process. They also confirm when the application reaches functional equivalence. As with any software project, test cases help to maintain a high level of quality in your application.
- Dedicate resources to testing. If you do not have dedicated testers, assign one set of developers to perform the upgrade and a different set of developers to test the upgraded code. This will improve the effectiveness of bug detection. Assign team members who know the business perhaps better than they know the code to testing.

Avoiding a Common Pitfalls

During the first phases of the upgrade, resist the temptation to redesign the application. While preparing the code for the upgrade it is easy to find ways to improve the application. If you start tinkering with the code at this stage, the upgrade process can easily spin out of control, causing you to over extend your schedule and budget. During the stages of compilation and achieving functional equivalence you should focus only on these tasks — not on redesign. The algorithms worked before so stick with them for now, but all the while do keep track of areas you can improve after the application has reached functional equivalence.

Also, if the code generated by the upgrade wizard works, do not try to improve it just because there is a .NET alternative for it. Leave these tasks for the application advancement stage.

Remember that the primary goal of your upgrade project should be to achieve functional equivalence. Only when functional equivalence is reached should you begin the tasks for improving and optimizing the application.

Summary

Deciding whether reaching functional equivalence is sufficient or if you need to add new features to an application is an important decision that you should consider before you begin an upgrade project. Although you can start by first achieving functional equivalence before adding new features, you should plan ahead for future advancements so that any necessary hooks can be included during the upgrade to facilitate future development work.

Understanding the different upgrade strategies and when each should be applied is important so that you can make decisions, plan the upgrade, and the associated estimate costs and effort. Should you upgrade your entire application, or only parts of it (leaving parts in Visual Basic 6.0, which will require the use of interoperability techniques)? Should you perform the upgrade all at once, or do it stages one component at a time? These are questions you will have to answer before you can even begin upgrading.

Finally, a successful and efficient upgrade project relies on a methodical approach with clearly defined stages. The process outlined in this chapter provides you with the knowledge you need to prepare and to carry out realistic upgrade project plans.

More Information

For more information about Visual Studio, see the Microsoft Visual Studio Developer Center on MSDN:
http://msdn.microsoft.com/vstudio/.

3

Assessment and Analysis

Introduction

Assessment and analysis usually means two things: determining how best to upgrade your applications to the Microsoft .NET Framework and how much it will cost to do this. This chapter discusses the best ways to gather this information.

The "Evaluating Upgrade Objectives" section discusses many of the common expectations that come with an upgrade to Microsoft Visual Basic .NET and discusses the realities behind those expectations.

The "Gathering Data" section provides information about what you should have ready to help you make important decisions for the upgrade process and how best to gather that information.

The "Application Analysis" section examines the Visual Basic 6.0 Upgrade Assessment Tool (included with this guide) and how to use it to get statistical, structural, and functional information from your applications. You will also learn how to use this information to help you make decisions about the best way to upgrade your applications.

The "Estimating Effort and Cost" section explains how to estimate the time and cost of your upgrade project.

Each section describes how to collect the information you need and the value of this information to the project. You will see how best to use the information to reduce costs and risks and to increase your return on investment (ROI).

To best determine the strategy and cost for upgrading your application, you need to ensure your scope and plan your project appropriately, and then test the upgraded application fully after it is a .NET Framework application. You have to be aware of

all the assumptions you are making and the expectations you have and to ensure that they are all correct. This chapter helps you learn how to do so.

Project Scope and Priorities

Probably the most important information that you can get from analyzing your application, and the information that has potentially the greatest effect on the effort and cost of the upgrade project, is the information that is needed to define the scope of the project. When you are defining the scope or your project, the considerations in this section will be helpful.

Identifying Components That Must Be Upgraded

Many factors will contribute to deciding which parts of your application must be upgraded to .NET. For example, it may be that you want to consolidate your server-side technology, but you may decide that upgrading the client is secondary. Alternatively, you may want to improve the integration with the Internet or Web services or add this functionality. In these scenarios, it may be necessary to upgrade only those portions that will interact directly with the improved or newly added functionality.

These requirements will mean that certain components of your application must be upgraded to Visual Basic .NET. Through normal dependencies, other components will also need to be upgraded. However, because of the interoperability possibilities between Visual Basic 6.0 and Visual Basic .NET, you will not need to upgrade all your code. For more information about interoperability, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET." For more information about upgrade strategies, see Chapter 1, "Introduction."

Also, the parts of your application that you upgrade may be limited by available resources, such as how many developers are available, how much time you have to perform the upgrade, and how much effort would be needed to complete the upgrade.

Later in this chapter you will learn how to track the dependencies between components of your applications. You will also learn how to measure the size of your applications and estimate the effort required to upgrade the individual components. This information will prove valuable in determine the scope of your upgrade project.

Identifying Obsolete Components

When assessing your application for upgrade, you will find areas of functionality that are no longer used. This is especially true in earlier applications that were developed by a series of development teams over several years. Often, features are added to applications and then later those same features are no longer used because the business changes focus or operating procedures. Frequently, these features can be removed before the upgrade, potentially saving significant amounts of work.

Typically, the older a feature is, the more likely it is that it depends on some behavior of an earlier version of Visual Basic that is no longer supported in Visual Basic .NET — a behavior that would be difficult to upgrade to Visual Basic .NET.

There are several analysis techniques that you can use to determine the features that are no longer used and can be left out of the upgrade. Use case analysis and input/output analysis are two such techniques that are discussed later in this chapter.

Managing Upgrade Expectations

Managing expectations is important in any development project. It is even more important in an upgrade project because people have different ideas of what an upgrade is and different reasons for performing the upgrade. Later in this chapter, you will find a list that includes many of the common expectations that people have for an upgrade project and the objectives that they believe the upgrade is attempting to reach. This list helps you manage the expectations of various people interested in your project and prevent unwelcome surprises.

Planning

To create an effective project plan, you have to know exactly what has to be done, who is available to do it, and how long it will take. Some of this was described in the "Project Scope and Priorities" section earlier in this chapter. The following sections discuss some other considerations that you should keep in mind during planning and that can be clarified with careful application of the Visual Basic 6.0 Upgrade Assessment Tool (hereafter referred to as the "assessment tool").

Identifying Components Requiring Significant Upgrade Effort

Some components can be upgraded to .NET more easily than others. For example, you may not have access to source code for third-party components. Other components may rely on features that require a lot of work to be upgraded; for example, components that make extensive use of Data Access Objects (DAO) or Remote Data Objects (RDO) for data access. These components should be given special attention during the upgrade planning stages because they correspond to the riskier areas of the upgrade.

Later in this chapter, you will find detailed information about the assessment tool (included with this guide) and how to use it to identify the components that will not upgrade easily to Visual Basic .NET.

Quantifying the Amount of Code to Upgrade

The number of lines of real code in your application, not counting comments or blank lines, is a useful indication of the complexity of an application. In fact, the number of lines of code is not used directly to estimate effort; instead, it is used as

a weight that affects other elements of the estimation. For more information, see "Estimating Time and Cost" later in this chapter.

For example, the number of lines of code affects the amount of time required to create an inventory of the application, define the order of the upgrade, review the upgrade report generated by the first run of the Visual Basic Upgrade Wizard, and so on.

The assessment tool reports how many lines of code there is in the application and in each of the files and components that comprise the application.

Determining Upgrade Order

Because of the complex interdependencies that usually exist between components and files in an application, it is normal to wonder which files, components, and modules you should approach first in a large upgrade project. Physically, the files that make up your application contain dependencies that force you to approach the upgrade of the files in a predetermined order, unless you are prepared to redesign some of your application. However, logically and even strategically, it makes more sense to upgrade your applications on a component-by-component basis, independent of the relationships between the files.

Your own business and technical needs will determine the order in which to upgrade the individual components of your application. When there is doubt, the assessment tool can shed light on current physical dependencies between components to help support your decisions at the component level. It is common to find dependencies between components that were unexpected and highlight weaknesses in the design or implementation of an application. After you choose the order in which to upgrade the components, and depending on how many files make up the component, you have to decide the order in which to upgrade individual files. The recommended approach is to start with the files that depend on the fewest number of other files, and then work your way through the dependency hierarchy. The assessment tool helps a great deal with this because it shows exactly what dependencies each file has and even suggests an upgrade order based on this information.

Assessing the Necessary Technical Expertise

When planning what needs to be done and in what order, you must also decide who will be responsible for each of the tasks that you identify. The assessment tool produces a report in the MainReport.xls file to assist with making these kinds of decisions. The **Config – Resources** tab of this file provides a list of recommended resource categories with a description and cost-per-hour for each category. Although default costs for each category are provided, these values can be modified to fit the actual costs for your company.

The MainReport.xls file also provides a fairly complete set of information about the kinds of tasks that need to be completed to upgrade your project. Furthermore, it hints at the resource categories needed to complete those tasks.

For example, each error, warning, or issue (EWI) that is found by the upgrade wizard has an associated suggested resource, such as developer or architect, that implies a minimum level of experience for the individual that is responsible for resolving that EWI. These are default suggestions provided by the assessment tool, but these can be adjusted in the **Config – EWI** tab if you disagree with any of the suggestions.

Also, all the different features of Visual Basic 6.0 code that require manual upgrading have associated with them suggested resources in the **Config – General** tab of the MainReport.xls. These values can also be adjusted, which allows you to select your own resource if you disagree with the suggestion.

The suggestions made for the type of resource required to deal with the various upgrade tasks are based on experience with numerous Visual Basic 6.0 upgrade tasks. However, your situations may vary depending on the resources available in your company. The configurability of the reports allows you to adjust them to suit your environment.

Estimating Time and Cost

These two estimates are usually the most important and traditionally the most difficult to determine. Among other things, they depend on the available resources, developer skills, experience, and knowledge, and quality and size of the application. The MainReport.xls file produced by the assessment tool provides assistance with cost and effort estimates by providing suggestions of the types of resources needed for various upgrade tasks, the cost for each type of resource, and the length of time that resource will need to perform the task.

The MainReport.xls file contains several tabs the report the estimated cost and effort for upgrade tasks, as listed here:

- **Effort – Total**: This tab provides a report of the complete estimated effort and cost.
- **Effort – By Resource Type**: This tab summarizes the effort estimate for each resource category.
- **Effort – By Task**: This tab summarizes the effort estimate for each upgrade task.
- **Effort – EWI**: This tab summarizes the effort estimate for each EWI.

Each of these tabs is discussed in more detail in the "Estimating Effort and Cost" section later in this chapter.

The estimates shown in these tabs are based on configurable values found in other tabs. As mentioned in the previous section ("Assessing the Necessary Technical

Expertise"), you can use the **Config – Resources** tab to adjust the cost for each type of resource. The values you supply will automatically be used in the effort reports. The **Config – EWI** and **Config – General** tabs suggest the type of resource and the length of time required to perform various upgrade tasks. These values can also be modified to suit your needs, and any changes you make will be reflected in the effort reports that depend on them. Thus, the task of estimating cost and effort for your upgrade project is simplified because you need only specify modifications in the configuration tabs and all effort reports are automatically adjusted to reflect your modifications.

The configurability of the various estimates in the MainReport.xls file will help your organization improve their accuracy over time. The more upgrade projects your developers participate in, the more information you gather about the costs and effort required to handle different aspects of upgrading. This information can then be used to adjust cost and effort estimates in the assessment tool reports in future projects.

Defining Priorities

Another important goal of the type of application analysis is defining upgrade project priorities. This is important because it helps focus effort and direct decisions. Priorities depend on the business and technical objectives at the root of the upgrade project, but to some degree, it also depends on the result of the analysis of the content, the operation of the application, and on the time and budget available.

Depending on the content of the application, priorities might include risk management by requiring a pilot project or initial testing of certain parts of the application.

One example scenario is where new business requirements include specific increases in performance or scalability to a handle a larger user base. This requires a close analysis of the current architecture and bottlenecks and a good understanding of the features of Visual Basic .NET that can increase performance and scalability. Chapters 1 and 14 discuss some of these features. Prioritizing bottlenecks helps to speed the upgrade.

Another example scenario is the addition of a new Web service interface in as little time as possible. If interoperability is an acceptable solution, a partial upgrade may be all that is necessary because you only need those components that provide content to the Web service; the other components can remain in Visual Basic 6.0. Here, prioritizing only the additional service would achieve the most efficient solution in terms of time and cost.

One final example is a scenario where the priority is consolidation. In this case, a partial upgrade is not an option and focus must be placed early in the project on those components that are more difficult to upgrade because they are the potential show-stoppers.

Evaluating Upgrade Objectives

This section examines some of the common reasons that people have for upgrading from Visual Basic 6.0 to Visual Basic .NET and the expectations that often accompany the upgrade project.

The reason should be clear for clarifying your expectations and the expectations of your sponsors, users, and clients. You want to ensure that you are upgrading your applications for the right reasons. You also want to make sure that the goals of your project are achievable and that you are clear about the amount of work needed to achieve those goals.

Most of these expectations come from one simple fact. When an application is upgraded using the automated tools provided by Microsoft Visual Studio .NET, most of the code used to generate the new application is based on the code that implemented the original application. This is usually the case. However, in some cases no code can be generated. For example, this can happen when the behavior is not in the source code of the original application, but instead, the behavior is in a third-party library that is no longer available.

The following sections list common objectives and the associated expectations that lead people to believe that the objectives will be met. In each case, the reasoning behind the expectations is discussed.

Business Objectives

Upgrade projects will have an impact on business objectives of your company. This section addresses some of the key considerations and expectations of an upgrade project from the perspective of business objectives.

Minimizing Organizational Disruption

An automated upgrade helps ensure that the look and feel of the original application is maintained in Visual Basic .NET. This usually leads to the expectation that the users do not need any training for the new application. This can sometimes be true, but some applications fail to meet this level of equivalence for a variety of reasons. The most common reason for this is third-party components that are not available in .NET. In this situation, you are forced to write your own component or replace it with one that is a little different in Visual Basic .NET. This can result in slightly different behavior.

Any changes in performance can affect the apparent behavior of an application and can confuse users. For example, if your .NET application performs differently, a user may be surprised by this difference and be left wondering whether his or her transaction was completed properly.

After assessment, you may determine that certain features are no longer needed or used. This is a good opportunity to reduce upgrade time and clean up the new application by removing code and features that are no longer used. However, you have to be careful when making decisions like this because other features may have undocumented dependencies on the features you remove. Also, you may find that some users do in fact use those features although they did not report that they use them or they use them for reasons other than those for which they were originally intended. For example, it is common to read information from reports that is actually secondary to the prime intent of the report simply because this report is easier or faster to create than the one that should be used.

Other changes that can cause disruption come from the fact that any procedures you had in place for testing and deployment will likely have to change. Because the application is no longer based on Visual Basic 6.0, you may have to modify your testing procedures, especially if they were automated. Also, deployment requirements will change, so you must also update your deployment procedures.

Finally, although the new application will contain most of the business logic from your original application, this does not mean that your programmers have nothing new to learn. Visual Basic .NET is a new language. It has new syntax rules, a new API, supports a new programming methodology, and no longer supports many common practices of Visual Basic 6.0 programmers. Upgrading your original application is a great way for them to learn this new language, but they will need some level of familiarity with Visual Basic .NET before the upgrade begins.

Leveraging Existing Knowledge Capital

Because the business logic encapsulated within an application represents a major investment of time and money, throwing it all away and starting a rewrite from the beginning is not typically the best solution. Part of the reason for choosing an upgrade instead of a rewrite is the possibility of maintaining the most possible investment in the original code. When an application is upgraded, very little of the existing business logic needs to be manually rewritten in the new language. Because much of this upgrade is automated, it can be done by developers who were not involved in the original application development.

Reducing Risk

If the earlier application is working well and is fulfilling user and business needs, a common expectation is that a direct upgrade to the new platform involves far less risk — it is simply the identical business logic moved to a new platform.

Legacy code has often been facetiously defined as “code that works.” It has been used, tested, and improved over a long period of time. If it works, there is no need to rewrite it. Writing new code also implies creating new bugs that must be found and fixed. Because much of the original code is automatically upgraded, you do not

need to write as much new code as you would if you rewrote the application from the beginning. This results in less risk of introducing new bugs to the application.

Maximizing Return on Investment

Businesses only approve new projects that, directly or indirectly, will show a return on investment. For this reason, expectations about a return on investment are implicit in any upgrade project.

Traditionally, calculating the ROI is a complex task because there are so many variables to consider. This guide can provide some pointers for calculating the ROI, but many of the variables come from your own business practices and how the upgraded applications are used. In simple terms, the ROI is a combination of the amount of extra money your business makes because of the upgrade, plus the amount of money that the upgrade saves you, minus the cost of executing the upgrade:

ROI = Money Made + Money Saved – Cost of Upgrade.

The money you make because of the upgrade is completely dependent on the applications that you upgrade and how they are used in your business. However, it is often related to the new features, services, and performance that your application offers as a direct result of the upgrade. The cost of the upgrade can be estimated before you start the upgrade; later sections explain this in detail.

You have to analyze your applications yourself and look at how the improvements offered by the upgrade affect the operating and maintenance costs of the applications. Because of the new technologies offered by Visual Studio .NET, there is also a possibility that the upgraded application can reach new markets or be used by a broader audience.

The .NET Framework and Visual Studio offer important improvements for software development and maintenance that reduce the overall cost of building and maintaining your applications. Deployment, configuration, and maintenance of the deployed application are easier and require fewer administration resources. XML-based configuration files, new deployment possibilities, and less invasive deployment activities mean that deployment is less likely to cause damage to or be damaged by installations of other software. Each of these aspects helps to reduce overall costs.

Increased performance, security, and scalability mean that your applications are more responsive and secure. This results in less downtime, more transactions, and more user satisfaction. All of these aspects translate into higher productivity and more effective business processes.

Finally, because Microsoft is going to stop providing support for Visual Basic 6.0 and all earlier versions, it is likely that soon after there will be decreasing numbers of developers who have a good knowledge of Visual Basic 6.0. This will make finding

developers more difficult and maintaining your application more and more expensive. Also, many businesses have made significant investments in training their people and want to know that their skills are current. If they still need to go outside the company for expertise, they also want to know that there is a ready pool of labor that they can choose from. Otherwise, in time, the ROI from an upgrade will come from the fact that it is simply more expensive to not upgrade.

Achieving Functional Equivalence

Functional equivalence is discussed in some detail in Chapter 2 and it is suggested as the necessary primary goal of any upgrade project. In fact, functional equivalence at a minimum is only natural to expect from a project that involves moving your application to a new platform.

For the most part, this is a valid and an achievable goal. However, it is not without its obstacles. Functionality that is provided by components that are not under your control and are not available in .NET may need to be rewritten or replaced by other third-party components or custom in-house components.

If you promise functional equivalence, make sure to confirm that it is achievable by checking the compatibility of all third-party COM components and libraries with .NET or, alternatively, the availability of a .NET version of those components.

Dealing with an End to Official Support for Visual Basic 6.0

As mentioned in Chapter 1, official support for Visual Basic 6.0 from Microsoft will cease completely in March 2008. It is a major objective of many businesses to ensure that their IT department and all their software operate on platforms and toolsets that have manufacturer-provided support.

If you opt for a full upgrade of all your applications, instead of just a partial upgrade as discussed in Chapter 2, you will no longer be dependent on Visual Basic 6.0. In a partial upgrade, you upgrade only certain parts of your applications and use one of the mechanisms available for interoperability between Visual Basic 6.0 and Visual Basic .NET to get them to communicate with the parts that were not upgraded. (For more information, see Chapter 13, "Working with the Windows API.") However, if you take this approach, parts of your application will obviously still require Visual Basic 6.0. If one of your objectives is to follow Microsoft's planned support path, you will eventually need to move all the pieces of applications over to Visual Basic .NET.

Technical Objectives

Any upgrade project will have technical objectives behind it. This section discusses some of the common technical objectives for performing an upgrade.

Adding New Functionality

Upgrading applications to a newer framework such as the Microsoft .NET Framework enables organizations to enhance their systems with cutting edge technology. This allows them to meet ever-expanding user and business needs such as Web enabling, Web services, XML, ADO.NET, and ASP.NET. And because critical applications are no longer dependent on outdated and/or unsupported legacy environments, there is a clear path for longer-term support and evolution.

This is a common expectation and is often a prime reason for moving to .NET in the first place. However, remember that these features do automatically appear. The key word is *enable*; .NET enables you to take advantage of these technologies. After an application is upgraded to .NET, you have all the capabilities and possibilities of .NET available to you, but you still need to plan how to start taking advantage of these technologies and then initiate a development cycle to implement the plan.

Achieving Better Performance and Scalability

Better performance and better scalability are two key improvements that .NET provides. It is common to expect better performance and scalability from your upgraded application.

In some cases, without any extra effort beyond upgrading your code to .NET, you can see dramatic performance improvements in your applications. For example, the white paper titled "Advantages of Migrating from Visual Basic 6.0 to Visual Basic .NET" (available on MSDN) documents developers who upgraded an ASP and Visual Basic 6.0 application to .NET. After the upgrade, they saw the throughput and the number of users supported simultaneously increase threefold without any modifications to the code other than what was needed to get the code running in .NET.

Such performance improvements are not always this automatic. When you start your upgrade project, you should consider the design of your application. For example, you should consider how it does things in Visual Basic 6.0 and how it will do them in Visual Basic .NET. Sometimes you may need to reconsider how things should be done in .NET in order to improve performance and scalability. For example, rewriting your ActiveX Data Objects (ADO) data access code to use ADO.NET will likely show great improvements. The rest of the chapters in this book, particularly Chapter 20, "Common Technology Scenario Advancements," give many pointers to where you can improve the performance and scalability of your applications when you upgrade to Visual Basic .NET.

Accelerating the Development Process

A decision to rewrite code implies building the application entirely from the beginning. In contrast, it is reasonable to expect that an upgrade can benefit from the fact that much of the conversion of your original code is automated. In fact, this results in huge improvements in productivity from upgrading, as compared to rewriting.

The main reason for this huge difference is that a rewrite involves designing, creating, testing, and debugging new algorithms, whereas an upgrade means simply translating algorithms to the new language and ensuring that the code performs identically to the original application.

Consolidating to a Single Framework

One of the common reasons for upgrading your applications is consolidation or framework unification by upgrading several languages and/or frameworks to one language and framework.

One of the biggest problems for IT organizations is supporting multiple applications that run on different frameworks and/or are written in different programming languages. This causes compatibility and integration issues among the applications and can be very costly to manage. Instead, if applications run on a single framework, it would be easier to manage, tune, enhance and deploy them as well as integrate and make them interoperate. The only impediment to migrate applications to run on single framework is the time and effort required to and the only way to overcome this impediment is to automate the migration.

In the context of this guide, if an organization has applications based on the .NET Framework as well as applications based on Visual Basic 6.0 then it makes sense to upgrade the Visual Basic 6.0 applications to Visual Basic .NET. With the help of the Visual Basic Upgrade Wizard, all the Visual Basic 6.0 applications can be made to run on the .NET Framework using the Visual Basic .NET language at a fraction of the time and cost that would be required to rewrite them. This will help your organization support and maintain its applications better than having to support both Visual Basic 6.0 and the .NET Framework.

Ease of Deployment

Also of importance to IT is deploying an application across an entire organization. Applications with sophisticated deployment schemes can be costly to install among multiple computers. DLL versioning conflicts result in lost productivity for users and costly trouble-shooting for IT personnel.

The .NET Framework provides application deployment options that simplify the process of deploying applications. It also provides options to help in protecting against version conflicts, thus ensuring that deploying a new application will not break currently installed applications.

For more information about deployment options, see the "Upgrading Application Setup" section in Chapter 16, "Application Completion."

Gathering Data

As mentioned at the beginning of the chapter, our goal is to determine how best to upgrade your applications to Visual Basic .NET and how much it will cost to do so. With this in mind, this section examines two ways to view your applications and extract information about them to help with your upgrade decisions.

The first way to view the application is based on how it is used. In this case, you will have to look at the environment in which the application executes, who the users are and how they use the application, and what features the users actually use. When looking at your application from this perspective, remember that users in this context can be human end users or other systems that interact with your applications. This technique for analyzing the application is especially useful if you have reason to believe that your application contains a lot of features that were created long ago but are no longer needed.

The second way to view the application is to analyze its content. This includes the features that it offers, the technologies that it uses to provide those features, the code that was written to manipulate the technologies, and the interactions between all these pieces. Therefore, you are interested in the source code that was used to build the application, its size, complexity, and the third-party libraries it references. Also of interest is how the applications are structured, what subsystems the design highlights, what dependencies the applications have on external systems, and the mechanisms used by these subsystems and external systems to interact with each other. The assessment tool included with this guide is a great way to start this kind of analysis.

The following subsections examine the techniques for analyzing your application's usage and how the assessment tool helps catalog the elements of your application.

Assessing Application Usage

A necessary activity in developing an upgrade plan is to perform an assessment of the application's usage. This section examines two techniques for doing this: use case analysis and input/output analysis.

Use Case Analysis

A very common technique for describing the requirements of software applications is through *use cases*. A use case is a document that describes one particular scenario where that system is or will be used. A complete set of use cases amounts to a set of functional requirements for a software application and it can be used as the starting point for more detailed functional specifications and test plans. For this discussion, a

set of use cases can be used to define the set of functionality that needs to be upgraded to Visual Basic .NET. If your application already has use cases, you can use them as a starting point. Just keep in mind that new usage scenarios may have been added since the original use cases were created because use cases are usually created before any application development even begins.

Examples of Use Cases

A use case usually consists of text describing a set of steps that lead to the completion of a specific task. For example, it can describe the steps a user would be expected to execute to process a check received from a client.

A typical use case contains information such as the following:

- **Name**. This identifies the use case.
- **Description**. This describes the use case.
- **Pre-conditions**. These indicate assumptions about what should be true before the steps in the use case are executed.
- **Steps**. These are the steps that are required to execute the use case.
- **Post-conditions**. These indicate the conditions that should exist after successful completion of the steps of the use case.
- **Actors**. This indicates who would normally execute these steps.
- **Variations**. These indicate alternatives to the steps. Finally, post-conditions indicate a way to confirm that the steps were performed correctly.

The actual size of a use case can vary from a few of lines to several pages, depending on the detail needed and the complexity of the task that is being described. At a minimum, a use case should have a name that identifies the task in just a few words and the steps expected to execute the use case. Names should be short but descriptive; for example, "Print a document," "Transfer between current accounts," "Check production level," and "Export report to Excel." The steps should describe the information that must be provided to the application and the information that is expected in return, but it should not add detail where it is not necessary.

There is a lot of documentation available for creating effective use cases, and it is all appropriate to upgrade projects. Some simple things to remember when defining use cases are:

- Use descriptive and precise names for your use cases. Avoid vague terms and software-related terms. Use terminology from the business domain to which the application belongs (such as Account, Client, Purchase Order, or Report) instead of from the world of software and databases (such as classes, objects, SELECT, or query).
- Address atomic tasks with each use case. Avoid creating use cases that involve more than one goal and avoid creating use cases that address a partial process.

- Scope the use case properly. Make sure the scope of the use case is clear. This can be done by including triggers and expected results with each use case. A trigger is some event that usually requires or causes the initiation of the use case. Expected results are the direct consequences of the use case.
- If you are using use cases that were created for the original application requirements, confirm that the use case covers a scenario that is still valid and useful. Some older use cases may describe tasks that are no longer needed or have changed since their original conception.
- Involve the users in the development of these use cases. In the end, it is the user's requirements that are reflected in the use cases, and they should be involved in creating and verifying use cases.

Creating New Use Cases

Ideally, you should already have use cases from when the original application was being designed. If there are no such use cases available, here are some ideas for how to create use cases now:

- Interview current users of the application.
- Run focus groups and brainstorming sessions with stakeholders.
- Gather information through questionnaires and surveys.
- Observe users' daily routines.
- Analyze external systems and their interaction with your applications.

Getting Information from Use Cases

After you have a representative set of use cases, you are in a position to categorize code, components, and projects into those that implement required features and those that are no longer needed. This helps avoid upgrading unnecessary parts of your application, which reduces cost and effort.

Another application for the use cases is as the basis of a set of system tests that can be used to ensure the existence and the quality of required functionality.

Inputs/Outputs Analysis

A classic way to describe computer software is by means of cataloging the full set of data that it receives and the full set of data that it produces. This technique is a simple but effective way of creating a formal specification for the behavior and the usefulness of an application.

This guide does not document the implementation of the pieces of the application, and it documents protocols, formats, the user interface, and other forms of interaction with the application only in relation to the input or output being described. For example, the user interface screen that a user interacts with to add client information to your application is described in terms of its content instead of its layout.

Examples of Inputs

Inputs are anything that is entered into the application by an employee, a client, or a user of any other kind, or an external application. Examples of inputs include:

- Raw data manually entered by a user to a data entry screen of the application, such as customer information.
- Any documents built externally to the application and sent to the application, such as e-mail messages, reports, lists, or raw data.
- Asynchronous messages received from other applications.
- Requests from users over the Internet or intranet.
- Software events for which the application is monitoring including timers, amounts, or email messages.
- Hardware events for which the application is monitoring.

Examples of Outputs

Outputs are anything that the application produces, displays on the screen, prints, sends to other applications, or stores on some storage device such as a hard disk, floppy disks, CDs, tapes, or the network. Examples of outputs include:

- Reports that contain raw or summary information based on existing data and usually in response to some query or as a result of a scheduled event.
- E-mail messages, faxes, or other electronically transported documents.
- Graphs, text, bar charts, graphics, sound, and other representations of data.
- Events, alerts, and other messages that are transparently logged or require immediate attention.

Getting Information from Inputs/Outputs

With a complete set of inputs and outputs, it is often even easier to identify code that is responsible for those inputs and outputs than it is with use cases. Again, this is useful if you believe that you can save time by identifying components, files, or even projects that are no longer used and therefore do not need to be upgraded.

Inputs and outputs can also help with testing because it is usually possible to create mappings from inputs to outputs, and based on this, to build tests that can confirm the correct functionality of the upgraded software.

Application Environment

The environments in which the applications are developed and deployed both play an important role in helping define priorities and avenues of investigation when issues need to be resolved.

For example, if you are not a part of the original development team, it is important to know who is responsible for the original development, architecture, design,

implementation, and, if appropriate, product management. It is essential to have a clear idea of the experience of the programmers who will work on the upgrade because you may need to arrange for training or bring in someone else with more experience in Visual Basic .NET to manage the project.

At the deployment site, you should also know who is responsible for support and administration of the application. Here, too, training may be required because the application technology and deployment characteristics will change.

In each case, you will need to know who makes the decisions so that you can get direction when you need it.

Application Analysis

Determining and understanding the effort required to execute an upgrade project is a complex task that requires the support of different tools. This process involves the division of the entire application into smaller, more manageable pieces that can be independently counted, understood, and estimated. As a consequence, upgrade effort estimation is an analytical process that can be supported at different stages by automated tools.

The original application analysis is an essential part of an upgrade project. It provides a better understanding of the application to be upgraded and produces information necessary for planning and decision making. The assessment tool will generate all that information in addition to information that is useful for the execution of the upgrade, such as the Recommended Upgrade Order report.

The Visual Basic 6.0 Upgrade Assessment Tool works as a measuring instrument for an application upgrade effort. This tool analyzes the application components and the relationships between them from an upgrade perspective, which considers elements, constructs, and features that consume significant resources during an upgrade project.

The main goal of the assessment tool is to analyze Visual Basic 6.0 projects and obtain information useful for planning the upgrade to Visual Basic .NET and for estimating the cost of the upgrade. This tool will generate a group of reports that are used as a basis for further calculations related to task effort and cost. The assessment tool user can specify configuration values that will override the initial estimation inputs; in this way, the assessment tool can be adapted to specific organizations.

The assessment tool analyzes the original application using a Visual Basic 6.0 parser that takes into account all the projects specified by the user. The obtained data is then processed to identify usage patterns and features that are difficult to upgrade. After all these features are classified and counted, a series of reports with detailed and summarized data is generated.

It is important to take into consideration that the assessment tool considers a high percentage of the known upgrade issues that can arise when upgrading a Visual Basic 6.0 application with the upgrade wizard; nonetheless, there are issues that are detected only after the full automated upgrade. The detection of these issues requires complex pattern recognition rules that are available only in the upgrade wizard. These issues can be estimated as a percentage of the rest of the issues, as is done by the assessment tool in the Upgrade Issues report.

The following sections explain the reports produced by the assessment tool and the aspects that are considered in the effort estimation.

Using the Assessment Tool

The assessment tool that accompanies this guide generates two Microsoft Excel® 2003 files. This first file is named MainReport.xls and contains a high level look at the results of the assessment. When you first open the file, you will see a page like the one illustrated in Figure 3.1.

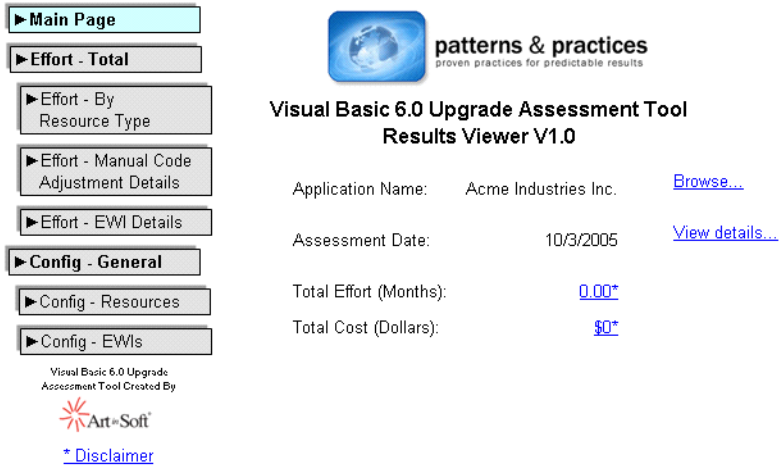

Figure 3.1
MainReport.xls Main Page

From the page illustrated in Figure 3.2, you can access the rest of the report in an ordered fashion.

The first thing to notice is that this report contains a link to the second report, which is in a file named DetailedReport.xls. To allow you to keep changes that you make to the cost and effort configuration between uses of the assessment tool, you can re-link the main report to any other detailed report from any other assessment. All of the application specific data in the main report is linked to the specified detailed report. The following sections discuss the different sheets that exist in these two reports.

In general, the MainReport.xls file contains all the configuration settings, all the estimates of cost, and summaries of the estimates of effort. The DetailedReport.xls file contains detailed reports about the content of the application. The first page of the DetailedReport.xls file is illustrated in Figure 3.2.

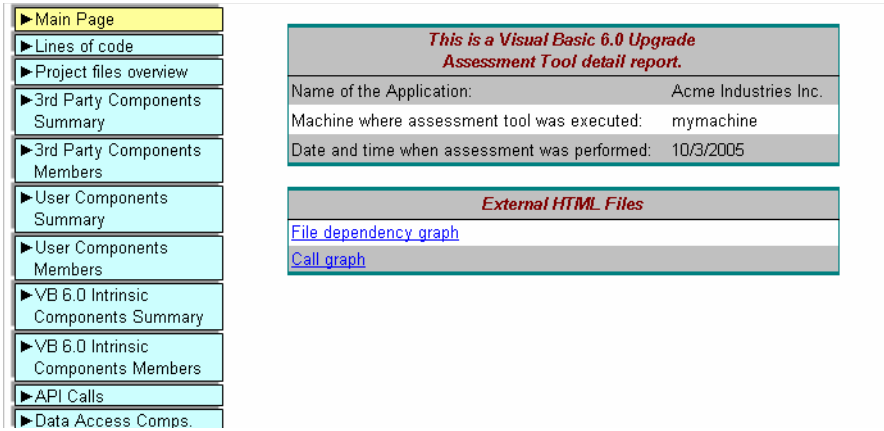

Figure 3.2
DetailedReport.xls Main Page

Current and Target Architecture

The original application architecture and project types are identified by the assessment tool and analyzed to extract all referenced files and components. The dependences between projects are also considered when assessing project groups.

The assessment tool can analyze the following types of Visual Basic 6.0 project types:

- **Standard EXE**. Standard features and functionality are considered when making estimations that are based on average productivity values.
- **Internet Information Services (IIS) application**. The application is analyzed in the standard way.
- **Application that uses distributed components**. The usage of distributed functionality is identified and accumulated to estimate the upgrade cost of this feature.

Visual Basic 6.0 project groups are collections of related projects that can share resources and functionality. The assessment tool is able to process project groups that are combinations of the preceding project types. Shared components are identified and taken into consideration when generating the Recommended Upgrade Order report.

The upgrade path to be followed during the project depends on the relationships between the base components, which are analyzed by the assessment tool to produce an upgrade order suggestion. Another aspect that affects the upgrade path is

the upgrade strategy you choose; it can be a horizontal or vertical strategy, as explained in Chapter 2, "Practices for Successful Upgrades."

Most of the application component relationships will remain in the target application and the application's general architecture according to the application type equivalencies presented in Chapter 4, "Common Application Types." Project type changes or re-architecture tasks are not considered in this analysis for effort estimation because these are deemed to be post-upgrade tasks after the functional equivalence is achieved.

Inventory to Upgrade

The first aspect to be considered in the application analysis is the inventory of resources and components involved in the upgrade process, which includes modules, third-party components, and intrinsic components. Some of these components are user-defined, and their implementation needs to be upgraded In addition to the source code that uses them. For third-party and intrinsic components it is necessary to upgrade the code that accesses the corresponding component; in this case, for assessment purposes, it is relevant to make an inventory of the class members and other elements that are used in the application.

The table illustrated in Figure 3.3 is from the DetailedReport.xls file and shows an example of the Project Files Overview report that is generated by the assessment tool.

Project	Total Files	Classes	Modules	UserControls	PropertyPages	Designers	Forms
LinkerMAC.vbp	139	136	3	0	0	0	0
LinkerUserInt.vbp	111	0	11	0	0	0	100
ConvertUI.vbp	21	1	12	0	0	0	8
LConvertUI.vbp	3	0	0	0	0	0	3
MyLinker.vbp	4	0	0	0	0	1	3
MyNewLinker.vbp	3	0	0	0	0	1	2
Total	281	137	26	0	0	2	116

Figure 3.3
Project Files Overview report

This report presents the projects that were processed by the assessment tool and the quantity of the different types of modules included in each project. This report can be useful to identify projects with different types of functionality; for example, projects that provide user interface elements (with a high percentage of form modules) or projects that implement the business logic functionality (with a high percentage of classes and modules).

User components account for a significant percentage of the upgrade effort; as a consequence, the inventory of user component definitions and instances declared must be taken into consideration when determining the inventory of resources to upgrade. Figure 3.4 illustrates the DetailedReport.xls file and shows an example of the User Components Summary report.

Component	LOC	Count
MaxBusiness.Font		152
MaxBusiness.cHelpManager	367	101
MaxBusiness.asiento		101
MaxBusiness.cMouseWait	17	64
MaxBusiness.DocumentoTipo		43
MaxBusiness.cAsistente	15	41
MaxBusiness.MovimientoStock		15
MaxBusiness.cColGrilla	13	11
MaxBusiness.DiccTablaPlantilla		10
MaxBusiness.EjercicioContable		10
MaxBusiness.Despacho		10
MaxBusiness.Tercero		9
MaxBusiness.ArbolNodo		9
MaxBusiness.fManfiestoCargaCalibres	264	9

Figure 3.4
User Components Summary report

The User Components Summary report considers all user-defined components and the lines of code (LOC) contained in each of them. This report also presents the quantity of instances of each class that are declared in the rest of the user code. The estimation worksheet generated by the assessment tool also includes a detailed report of the user components that provides the member usage for each component.

Third-party and Visual Basic 6.0 intrinsic components are also considered in the application inventory to be upgraded. The upgrade of these components involves the processing of the application source code to identify accesses to each of the component members. All member usages will be analyzed to determine if there is an equivalent member in the target environment; some of these members will not have an equivalent in Visual Basic .NET and will require additional manual work to be upgraded to .NET. Figure 3.5 illustrates the Third Party Component Summary report from the DetailedReport.xls file; it includes all the components used in the application and the quantity of instances that are created for each class.

Component	Count
EditLib.fpText	776
vsOcx6LibCtl.vsElastic	576
Threed.SSCommand	270
Threed.SSCheck	229
EditLib.fpDoubleSingle	206
EditLib.fpDateTime	193
vsOcx6LibCtl.vsIndexTab	132
Threed.SSOption	123
ACTIVESKINLibCtl.SkinForm	119
TrueDBGrid60.TDBGrid	111
GridEX20.GridEX	107
TrueDBGrid60.Column	85

Figure 3.5
Third Party Components Summary report

Visual Basic 6.0 intrinsic components are treated in a similar way to third-party components, with the exception that most of the intrinsic components have an equivalent in the .NET Framework and most of the member accesses require some

type of transformation. The inventory of intrinsic components used in the application is included in the Visual Basic Intrinsic Components Summary report in the estimation worksheet generated by the assessment tool.

User COM objects are also included in the inventory to upgrade; the quantity of occurrences will be considered for effort and cost estimations. It is important to notice that user COM objects will be considered as third-party components when accessed from a project different from that in which it was declared.

In the case of user-defined components, there is an important percentage of upgrade effort dedicated to the processing of the components' internal code, including member declarations and usage of features that have a high upgrade cost. Detection of these features is done by the assessment tool and is included in the estimation worksheet it generates. The detected features include:

- **API calls**. The report includes the function name and library in addition to the data types of the parameters received. One column shows the quantity of usages of the function in the rest of the code.
- **Data access**. The report presents the usage of different data access technologies, including ADO, RDO, and DAO.
- **COM+**. The report presents a summarized and detailed view of the usage of COM+ and Microsoft Transaction Server (MTS) classes.

Using the information presented in these reports, it is possible to identify components of the application that require special attention in the upgrade planning. For example, heavy use of ADO recordsets is an indication that a vertical upgrade strategy should be used for the affected components. For more information about upgrade planning, see Chapter 2, "Practices for Successful Upgrades," and Chapter 4, "Common Application Types."

Source Code Metrics

Typically, the size of an application is defined in terms of the lines of code. Also, the application source code can be classified and counted according to the origin of the code; for example, the assessment tool is able to identify code lines related to visual component declarations, comments, blank lines, and user-written code. The quantity of lines of code has some limitations as an estimate of the complexity of an application from the upgrade perspective. A large application that uses only features supported by the upgrade wizard can have a low upgrade effort while a small application that makes extensive use of unsupported features can have a high upgrade cost. However, source code metrics can still be used to estimate the size and effort necessary to execute some tasks involved in the upgrade process. These tasks have a limited complexity and depend mainly on the size of the application and the quantity of modules and components that form it. Examples of these tasks are the

Application Resource Inventory and the Migration Order Definition; both of these are estimated in the Effort – Total report generated by the assessment tool.

The assessment tool helps to capture source code metrics by producing a report of the number of lines of code in a project. Figure 3.6 shows an example of the lines of code (LOC) report in the DetailedReport.xls file produced by the assessment tool.

Project	Total Lines	Visual Lines	Code Lines	Comment Lines	Blank Lines
LinkerDAC.vbp	270,334	0	179,190	53,192	37,952
LinkerUserInt.vbp	214,773	41,065	98,639	51,485	23,584
ConvertUI.vbp	26,337	758	17,362	3,751	4,466
LConvertUI.vbp	1,705	1,458	111	92	44
MyLinker.vbp	4,472	3,543	839	1	88
MyNewLinker.vbp	3,382	2,598	724	0	59
Total	521,003	49,422	296,865	108,521	66,193

Figure 3.6
Lines of code (LOC) report

Source code lines are classified by the assessment tool as a result of analyzing all the content of the application modules. Each line is parsed and classified to determine how many visual lines, code lines, and comment lines are written in the Visual Basic 6.0 projects processed. Visual lines are lines that were generated by Visual Studio.

Handling Unsupported Features

The upgrade wizard automatically upgrades most of the language constructs and intrinsic components. It also supports third-party components and different types of Visual Basic 6.0 projects. However, there are still features that are not fully supported by the upgrade engine and require some level of manual work to be upgraded to Visual Basic .NET. The Visual Basic Upgrade Wizard Companion, available from ArtinSoft, supports many features that are not supported by the upgrade wizard.

When an application is upgraded, the unsupported features produce different types of errors in the target application. Compilation or run-time errors can be produced and user intervention will be required to fix them; this results in the need for resources to review the code, perform the correction, and execute the appropriate tests.

The assessment tool detects the unsupported features by analyzing the code and searching for code patterns that will produce upgrade issues after the application is upgraded. The estimation worksheet has a **Config – EWI** tab where the cost and effort values associated with each issue can be reviewed and adjusted according to specific needs. The values that are initially inserted in this tab are derived from ArtinSoft's experience in Visual Basic upgrade projects. The configuration values are taken into consideration when generating the **Effort – EWIs** tab in the MainReport.xls report file, as illustrated in Figure 3.7 on the next page.

NAME	MSDN ID	Resource	Occurrences	Hours per Occurrence	Cost per Hour	Total Hours	Total Cost
ParamArray was changed from ByRef to ByVal	1003	DEV	0	0.10	$50	0.00	$0
Statement was removed. Variables were explicitly declared	1005	DEV	0	0.02	$50	0.00	$0
Statement is not supported	1014	DEV	0	0.13	$50	0.00	$0
Declaring a parameter 'As Any' is not supported.	1016	DEV	10	0.05	$50	0.50	$25
Statement was changed	1021	DEV	188	0.07	$50	12.53	$627
DoEvents does not return a value.	1022	DEV	0	0.07	$50	0.00	$0
Object may not be destroyed until it is garbage collected.	1029	ARC	206	0.08	$80	17.17	$1,373
#If #EndIf block was not upgraded because the expression did not evaluate to True or was not evaluated.	1035	ARC	17	0.07	$80	1.13	$91
Statement is not supported	1039	DEV	53	0.03	$50	1.77	$88

Figure 3.7
Upgrade Issues tab

The **Effort – EWIs** tab of the MainReport.xls file generated by the assessment tool reports all the detected issues and the corresponding quantity of occurrences, their complexities, and cost estimation values. It is important to take into consideration that the assessment tool does not detect all possible issues that can be generated by the upgrade wizard; however, most of the issues that have a significant impact on the upgrade effort are considered. As previously stated, the **Config – EWIs** tab includes a column that can be used to adjust the effort value obtained for the upgrade issues to make your estimates more accurate.

Application Dependences

The assessment tool identifies the dependencies between the application components analyzing different aspects of the source code, including:

- **Application references**. Explicit references to user components indicate that there is a dependence relationship between different user projects.
- **Variable declarations**. When a variable is declared using a type that is defined in a different module or project, a dependence relationship between different modules or projects is established.
- **Member accesses**. Member accesses also produce dependences between subroutines, functions, or properties.

The assessment tool identifies dependencies between members of project groups. When a project references another project that forms part of the same project group, the assessment tool identifies the referenced project as a user component and informs the user about the relationship between these projects.

Knowledge about the usage relationships between components is essential for different aspects of the upgrade plan, including upgrade order definition, testing, and debugging. For more information about upgrade planning, see Chapter 5, "The Visual Basic Upgrade Process."

Based on the dependence relationships analysis, the assessment tool generates an **Upgrade Order** report in the DetailedReport.xls file that includes a suggestion of the file upgrade order. An example of this report is illustrated in Figure 3.8.

Project	File
AdminUI	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\frmProductAddition.frm
	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\mdiAdmin.frm
	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\frmProductUpdate.frm
	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\frmCustomerAddition.frm
	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\Resources.bas
	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\frmSupplierAddition.frm
	G:\Depot\Microsoft\VB6Migration\Test\Samples\RichClientSample\AdminUI\frmSupplierUpdate.frm

Figure 3.8
Example Upgrade Order report

The report is divided into sections that correspond to each of the application's projects. The files in each group are ordered according to the dependences between the files in the group. The first file in each group has no dependences on other user-defined files in the same group. As a result, this file can be upgraded and tested independently. The next file in the list can be upgraded and tested based on the previous upgraded file.

Each of these files can be upgraded in a staged way, providing increasing levels of functionality. The assessment tool produces the Upgrade Order report by looking for modules that depend on the minimum quantity of user components, and then a new dependency level is generated based on the components included in the previous level. In this way, each level is built on the functionality provided by previous levels. This report is useful for identifying possible upgrade paths; if a horizontal upgrade strategy is going to be used, the suggested upgrade order can be followed. It is also possible to upgrade modules from different groups at the same time if a vertical upgrade strategy will be executed for some pieces of the application.

Missing Application Elements

After the initial upgrade preparation steps are complete, it is possible that some of the application components are not yet available on the computer where the automated upgrade is going to take place. After these components are identified, it will be necessary to correct the environment before the upgrade process can be restarted. These additional steps will consume time and resources that need to be accounted for.

The assessment tool identifies missing application elements that can be reviewed in the Missing Components and Missing Files reports included in the estimation

worksheet. Missing elements are identified according to the user defined modules and explicit references included in the analyzed projects.

Figure 3.9 illustrates the Missing Components report from the DetailedReport.xls file.

Path	Version	Name
c:\windows\system32\stdole.tlb	2.0.0	OLE Automation
c:\windows\system32\comsvcs.dll	1.0.0	COM+ Services Type Library
c:\windows\system32\dx8vb.dll	1.0.0	DirectX 8 for Visual Basic Type Library
mscomctl.ocx	2.0.0	
mschrt20.ocx	2.0.0	
mscomm32.ocx	1.1.0	
mscal.ocx	7.0.0	
tabctl32.ocx	1.1.0	

Figure 3.9
Missing Components report

The Missing Components report includes the file name and path of the missing components in addition to the corresponding version and the registered name. It is necessary to install these components on the upgrade computer to execute the automated upgrade. The quantity of missing components is considered in the effort estimation (**Effort – By Tasks** tab in the MainReport.xls file produced by the assessment tool) according to the missing elements configuration data.

Figure 3.10 illustrates the Missing Files report from the DetailedReport.xls file.

Name	Path	File Type	Project Name
File 1	C:\MyApp\Forms\File1.frm	Form	Project 1
File 2	C:\MyApp\Modules\File2.frm	Module	Project 1
File 3	C:\MyApp\Forms\File3.frm	Form	Project 1
File 4	C:\MyApp\Controls\File4.frm	User Control	Project 1
File 5	C:\MyApp\Forms\File5.frm	Class	Project 1
File 6	C:\MyApp\Modules\File6.frm	Module	Project 2
File 7	C:\MyApp\Forms\File7.frm	Form	Project 2
File 8	C:\MyApp\Forms\File8.frm	Form	Project 2
File 9	C:\MyApp\Forms\File9.frm	Form	Project 2
File 10	C:\MyApp\Controls\File10.frm	User Control	Project 2

Figure 3.10
Missing Files report

This report includes the file name, path, and module type for each missing file. The project where the missing file is detected is also included. In the same way that the missing components are considered in the estimation effort, missing user files are also considered.

Estimating Effort and Cost

After deciding the best way to upgrade your application, the second goal of application analysis and assessment is determining how much effort is actually needed to upgrade your applications to functionally-equivalent .NET applications. When estimating effort, you need to include all the steps of the upgrade from the initial analysis of the requirements up to the deployment of the application and retraining of developers and users if necessary.

After you have an estimate of effort, you have the most important piece to estimate the cost of the upgrade. However, the cost of the upgrade also depends on other elements. One such element is the cost of the people that do the work. Different parts of the upgrade require people with different profiles. Project administrators, developers, testers, and architects all cost different amounts of money.

Other things to keep in mind are training, hardware, software licenses, and any interruptions to your business or the cost of avoiding those interruptions when you switch to the new version of your application.

The effort estimation reports (**Effort – Total**, **Effort – By Task**, **Effort – By Resource Type**, and **Effort – EWIs**) generated by the assessment tool provide configurable parameters that help fine-tune a calculation of the effort and time required to complete the upgrade of a project. This report is created along with all the application's technology details.

Methodology Overview

It is important to understand the concept of effort in the upgrade context. The effort of each task is expressed in hours that correspond to normalized time, that is, it corresponds with the time that the task would take if it was executed using average resources. If there is a highly experienced programmer in charge of that task, it will be finished in less time. In addition to average resources provided by default, the Effort Estimation report includes parameters such as junior developer, junior tester, senior developer, and senior tester; each with a different cost per hour.

The cost of most tasks is directly derived from the cost of the resources necessary to perform the tasks and the estimated effort required to execute the task. For example, the cost of upgrading a COM component is higher if a senior developer is responsible for the conversion, although this may result in reduced effort. In contrast, a junior developer will cost less than an experienced developer but will require more time to achieve a functionally-equivalent upgrade. For more specific tasks, the cost is obtained from a per-occurrence cost that is related to the each task's degree of difficulty and the kind of resources necessary to perform it. The **Config – General** tab in the MainReport.xls file generated by the assessment tool contains default values for cost and effort for different upgrade tasks. These values can be modified

to match more closely to your own development capabilities and costs. Figure 3.11 illustrates the **Config – General** tab.

Upgrade Task	Effort per occurrence (Hours)	Resource	Description
1000 lines of code	0.25	DEV	Migrate 1000 lines of code assuming no issues reported by the upgrade wizard. This affects the cost only in that large amounts of code increases risks.
Property Page	2.00	DEV	The number of hours and the cost per hour to migrate one property page assuming no issues reported by the upgrade wizard.
Designer	3.00	DEV	The number of hours and the cost per hour to migrate one designer assuming no issues reported by the upgrade wizard.
Form	3.00	DEV	The number of hours and the cost per hour to migrate one Form assuming no issues reported by the upgrade wizard.
3rd-Party Component Declarations	0.30	DEV	The number of hours and cost per hour to migrate each 3rd Party Component used in a variable declaration
3rd-Party Component Member Usage	0.10	DEV	The number of hours and cost per hour to migrate each use of a member of a 3rd Party Component
User Component Definitions	0.00	DEV	The number of hours and the cost per hour to migrate each User Component
User Component Member Usage	0.00	DEV	The number of hours and the cost per hour to migrate each use of a User Component member
VB Intrinsic Component Declarations	0.10	DEV	The number of hours and cost per hour to migrate each VB Intrinsic Component used in a variable declaration

Figure 3.11
Config – General tab

Some tasks directly depend on the number of lines of code that comprise individual components or files. The estimated effort is the number of lines of code multiplied by some constant. This constant is based on work performed on previous upgrade projects involving various different types of applications.

For tasks whose cost is derived from the quantity of occurrences of a feature, the application conversion estimate of effort is calculated by counting up all features' occurrences within the project and adding the cost to upgrade each occurrence and the effort per occurrence in terms of time. This is included on the **Effort – By Task** tab in the MainReport.xls file generated by the assessment tool, with totals reflecting time and cost for all occurrences of each feature. Figure 3.12 illustrates a sample of this tab.

Each EWI has an effort and cost associated with it. All these values are derived from the steps necessary to resolve each issue in other upgrade projects. These values can be configured on the **Effort – EWIs** tab in the MainReport.xls file. An example of this table is shown in Figure 3.13.

This tab allows you to change the default effort and cost associated with each EWI to help you prepare a more accurate estimate of the effort required to address upgrade issues.

Feature	Occurrances	Effort per occurrence (Hours)	Cost per occurrence ($)	Total hours	Total cost	Resource
Non-visual, executable lines of code	294,758	0	0.0125	74	$ 3,684.48	DEV
Property Pages	0	2	100.00	0	$ -	DEV
Designers	2	3	150.00	6	$ 300.00	DEV
Forms	116	3	150.00	348	$ 17,400.00	DEV
Third Party Component Declarations	55	1	25.00	28	$ 1,375.00	DEV
Third Party Component Member Usage	2,173	0	5.00	217	$ 10,865.00	DEV
User Component Declarations	36	2	100.00	72	$ 3,600.00	DEV
User Components Member Usage	2,688	0	5.00	269	$ 13,440.00	DEV
VB Intrinsic Components	18	0	12.50	5	$ 225.00	DEV
VB Intrinsic Components Member	16,556	0	5.00	1,656	$ 82,780.00	DEV
API Calls	27	0	24.00	8	$ 648.00	ARC
ADO Data Access Controls	6	0	5.00	1	$ 30.00	DEV
ADO Data Access Member Usage	250	0	5.00	25	$ 1,250.00	DEV

Figure 3.12
Manual Code Adjustment Effort tab

NAME	MSDN ID	Resource	Effort per occurrence (Minutes)	Effort per occurrence (Hours)	Cost per hour	Functionality
ParamArray was changed from ByRef to ByVal	1003	DEV	6	0.10	50	Use of ByRef modifier.
Statement was removed. Variables were explicitly declared	1005	DEV	1	0.02	50	Use of DefType. Each declaration occurrence should be counted. Use of variables declared by these statements won't be counted.
Statement is not supported	1014	DEV	8	0.13	50	Detects use of unsupported flow control statement On ... GoSub.
Declaring a parameter 'As Any' is not supported.	1016	DEV	3	0.05	50	Use of "As Any" modifier in external declarations.
Statement was changed	1021	DEV	4	0.07	50	Detects use of unsupported member vba.Information.IsMissing.

Figure 3.13
Effort – EWIs tab

Aspects to Be Estimated

The different phases to be estimated relate directly to the steps of the upgrade procedure. In summary, the items to be estimated are:

- **Project kickoff**. This includes the estimated time for the upgrade project kick-off meeting. Typically, it is not a large value. It serves as a reminder that these cost estimates have a well-defined scope. It also reminds you of the many things that you must do before the upgrade project begins, such as training, research, and purchasing software and hardware, which are not included here.
- **Application preparation**. This includes the estimated time for the setup and verification of the development environment of the original application. This includes compilation of the original application to verify that it is complete and source code modifications to accelerate the conversion process later on.
- **Application conversion**. This includes the estimated time for the execution of the upgrade wizard and completing the upgrade with manual changes.
- **Testing and debugging**. This includes the estimated time for the execution of the original test cases and fixing any run-time errors.

- **Project management**. This includes the estimated time for administrative tasks related to scheduling and management.
- **Configuration management**. This includes the estimated time for configuration management of the products obtained throughout the entire upgrade process.

For more information, see Chapter 5, "The Visual Basic Upgrade Process."

The application composition is the inventory to be upgraded; this is reflected on the **Project Files Overview** tab. The inventory of the application is the first aspect to be considered when estimating the upgrade effort. This inventory will provide a general vision of the projects and modules to be upgraded. The content of these modules will be detailed in the Effort Estimation report. The application composition is an important report to plan the rest of the upgrade and set priorities for the different projects and modules. For instance, if there are several user controls that present a dependency among classes, it is wise to set a high priority to those components to have them ready when upgrading the forms and classes that depend on them.

As mentioned in the "Methodology Overview" section, the **Config – EWI** tab includes an estimation of the effort needed to upgrade an issue. The values provided by the assessment tool and that are generated as part of this tab affect the following worksheets:

- Effort – EWIs
- Effort – By Task
- Effort – Total

The **Effort – By Task** tab consolidates different aspects of the effort estimation, including features/code items such as property pages, designers, third-party components, ADO, RDO, COM, COM+, and more. Each element is a summary of more detailed information that can be found in the DetailedReport.xls file.

The tasks included on the **Effort – By Task** tab have a high degree of detail behind them, and the estimation is calculated as a function of the data in the details. The types of resources used to perform the different tasks are also explicitly indicated in this report to give an idea of the level of experience that each task will likely need, and the subsequent cost.

Understanding the Effort – Total Worksheet

A complete summary of the effort and cost estimations can be found on the **Effort – Total** worksheet in the Microsoft Excel workbook that is generated by the assessment tool.

Before looking at the worksheet in detail, be aware that it generates an estimate that you must fully review and customize before you use it in your own project estimations. Although almost every aspect of your application is covered somewhere in

this workbook and each has an associated weight indicating the effort required to upgrade that particular aspect, the weights for some aspects are left at zero. This means that the assessment tool does not consider that all aspects are important to the estimation. The aspects that do have weights for effort and cost have values that may not be appropriate for your organization. Also, remember that every upgrade project is different, and although similar worksheets to this one have been used in real upgrade projects, there are always new things to consider or assumptions built into estimate worksheets that do not apply to every project. It is likely that you will have to make some adjustments before applying this estimation to your own projects. For this reason, the next paragraphs explain how the assessment tool arrives at the cost estimation it produces in the Excel workbook. Every effort has been made to base the estimates on projects that are as close as possible to typical. The formulas and weights in Figure 3.14 are based on experience with real upgrade projects and accurately reflect that experience.

Task	Effort (Hours)	Cost	Resource
Application preparation			
Development environment	4	$0	DEV
Application resource inventory	21	$0	DEV
Compilation verification	8	$0	DEV
Upgrade order definition	10	$0	DEV
Upgrade wizard report review	15	$0	DEV
Total	58	$0	
Application conversion			
Upgrade wizard execution	7	$0	DEV
Manual code adjustment	3604	$0	See details
System integration and smoke test	901	$0	DEV
Administrative Tasks	135	$0	DLE
Total	4647	$0	
Testing and debugging			
Test case creation	465	$0	STE
Test case execution	46	$0	TES
Bug management	186	$0	TES
Regression test	186	$0	TES
Run-time error correction	929	$0	DEV
Administrative Tasks	181	$0	STE
Total	1994	$0	
Project management	335	$0	PM
Configuration management	8	$0	CM
Totals	7,042	$0	

Figure 3.14
Effort –Total tab

The **Effort – Total** worksheet is divided into three sections: "Application preparation," "Application conversion," and "Testing and debugging." Each section will be examined in detail. However, please keep the following notes in mind:

- Most cells in the Effort (Hours) column are references to named cells on the **General Configuration** tab or are functions of named cells from there. If you

make any changes, it is recommended that you make those changes to the General Configuration worksheet instead of to the Effort worksheet.

- Any cell or set of adjacent cells can be given a name and used in formulas. You will see names used as if they were single values (for example, =NameA+NameB) and collections of values (for example, =SUM(NameC)).
- For consistency, the worksheets and formulas are organized so that you will only ever need to change either the General Configuration worksheet or the EWI Configuration worksheet. The General Configuration worksheet is referenced primarily in the Total Effort worksheet, and the EWI Configuration worksheet is referenced mostly in the Upgrade Issues Table worksheet. However, it is okay to change the other worksheets if you want to. A third worksheet that you will likely have to change is the Resource Costs worksheet, which specifies how much it costs for different types of people to work on the upgrade project. These values change from country to country, from business to business, and even from month to month, so you will probably have to review them.

Be careful when you make changes to the worksheets generated by the assessment tool; when it runs again, it will create a new workbook, possibly overwriting all your changes. Always keep a backup of the workbook file before re-running the assessment tool. It is recommended that you only make changes to the MainReport.xls file and keep a backup of it. You can easily change the MainReport file to link to any DetailedReport.xls file. This saves you from having to reconfigure the assessment tool or copy configuration information between reports.

Application Preparation

The application preparation phase is described in detail in Chapter 5, "The Visual Basic Upgrade Process." In the context of this discussion, the definition of application preparation is broadened to include everything from the moment that you are preparing the computers the upgrade will be performed on up to just before the upgrade wizard runs for the last time to generate the initial code that will be used as the source base for the new Visual Basic .NET application.

This does not include the time spent studying alternatives or any of the other activities that are performed before an upgrade project is approved. It is assumed that the decision to upgrade has already been made and that the upgrade project has been approved.

Nor does it include any "code preparation" activities that are described in Chapter 5, such as running the Visual Basic Code Advisor to clean up the code to make it more amenable to upgrade. These activities can help reduce overall upgrade time but including them in these estimations would make them overly complex. We suggest that you use the assessment tool after you do any code preparation. If your code preparation is successful, you should see a reduction in the effort estimated by the assessment tool before and after code preparation is complete.

Most of the tasks in this phase have fixed estimations of effort and cost or have estimations based on the number of lines of code in the application. These are explained here:

- **Development environment**. This includes the setup of the upgrade wizard, which will be used to analyze and upgrade the application. This effort is fixed and configurable by the user of the assessment tool.
- **Application resource inventory**. This includes an initial review of the application and gathering all the files and tools that are needed to compile the application in Visual Basic 6.0. It also includes time for collecting existing information about the applications architecture and design. This value depends on the size of the application.
- **Compilation verification**. The original application is compiled to verify that all the required files are available. This effort is fixed and configurable by the user of the assessment tool.
- **Upgrade order definition**. The upgrade order is determined by the assessment tool reports and priorities that are implied by your own strategic and technical goals. This value also depends on the size of the application.
- **Upgrade wizard report review**. Preliminary test upgrade of the application to test the system resources and detect common errors and practices that are more easily fixed in Visual Basic 6.0. This task also depends on the size of the application.

Application Conversion

The application conversion phase includes all the tasks required to get the *green code* to functional equivalence. Green code is the name for the unmodified code generated by the upgrade wizard that serves as the initial code base for the upgrade.

The tasks in this phase are:

- **Upgrade wizard execution**. This involves execution of the upgrade wizard to obtain the green code. This effort depends on the size of the application.
- **Issue resolution**. This involves manual resolution of all the issues mentioned in the Effort – EWIs worksheet. This sheet is described in greater detail later in this chapter.
- **Code review**. After all the issues are resolved, it is necessary to integrate all the new code, compile it, and debug it.
- **Administrative tasks**. This involves administrative tasks performed during this phase, such as scheduling and resource assignment. It is calculated as a percentage of all the other tasks in this phase.

Testing and Debugging

The last phase includes all the quality assurance tasks that must be preformed to ensure that the application is functionally equivalent to the original application. All the tasks in this phase are calculated as percentages of the total effort in the Application Conversion phase. These tasks are:

- **Test case creation**. This task involves creating new test cases or adapting existing test cases to be used for quality assurance and to verify functional equivalence.
- **Test case execution**. This includes multiple executions of the test cases during testing as the application moves toward functional equivalence.
- **Run-time error correction**. This task includes bug fixes, workarounds, and other resolutions of detected run-time errors.
- **Administrative tasks**. This task includes bug management, test case management, regression testing, and other administrative tasks that are typically associated with testing and debugging.

Understanding the Estimate

As previously stated, estimating development projects is traditionally a very difficult thing to get right. The reports and estimations generated by the assessment tool are based on extensive experience with upgrade projects and take into account as much detail as possible from that experience. Tasks outside the three phases explained earlier are not included.

After the most important tasks are known, you must decide how much time each task will require. The assessment tool estimates are only suggestions based on developer experience. You should review all of the features identified in the report and the amount of time allocated to each one. It may be necessary to modify these estimates if they do not reflect your organization's experience; for example, if you have no one with experience in that area and believe more time is required or if you have already solved the problem before and are confident it can be done more quickly.

Also keep in mind that unforeseen issues can occur. You may be using a third-party library that is not taken into consideration by the assessment tool, or your application may make unconventional use of some Visual Basic 6.0 features or behaviors that was not considered by the assessment tool.

For these reasons, you should consider the numbers generated by the assessment tool as a rough estimate. You can get more accurate results by reviewing all the issues identified by the assessment tool and making sure that the effort assigned to each individual issue makes sense in your organization. Also, be certain to obtain your developers' approval of the estimation because they are the ones that have to deliver in the end.

Understanding the Configuration Settings

This section describes the configuration settings for the assessment tool. Each of these settings can be found in the MainReport.xls file that is produced by the assessment tool.

Config – General

The **Config – General** tab allows you to configure the estimated effort for specific upgrade issues, such as correcting and verifying every 1,000 lines of code. It includes the following columns:

- **Migration Task**. This column includes tasks that must be performed during the upgrade. Some of the tasks refer to specific characteristics of the application (such as lines of code) or types of components used by the application (such as COM objects or ADO).
- **Effort per occurrence (Hours)**. This column defines the time required to upgrade an occurrence of an upgrade task, measured in hours. This value should be configured to reflect the capabilities of the developers in your company. The default value is usually a good average.
- **Cost per hour**. This column identifies the cost per hour of upgrading an occurrence of one of these issues. This is highly variable and depends on the skill level of the person that will be working on each issue and the salary levels for someone of that level.
- **Resource**. This column identifies the recommended resource type that should be assigned to resolving each issue.
- **Description**. This column provides a description for each upgrade task.

Config – By Resources

Resource costs can be found in the **Config – By Resources** tab. They represent an estimation of development resources required to resolve an upgrade issue. Figure 3.15 shows an example **Config – By Resources** tab.

Resource	ID	Cost per hour	Description
Developer	DEV	50	Application developer.
Architect	ARC	80	Senior developer with experience in the design and
Development Leader	DLE	90	Leader of the development team, involved in architecture and
Tester	TES	40	Application tester.
Senior Tester	STE	70	Senior application tester with experience in the original
Project Manager	PM	80	Administrative personnel for scheduling and management.
Configuration	CM	70	Configuration management

Figure 3.15
Config – By Resources tab

The **Config – By Resources** tab includes the following four columns:

- **Resource**. This column names different levels of experience and skill that may be needed during an upgrade.
- **ID**. This column identifies an abbreviation for this type of resource that is used in other tables in the report.
- **Cost per hour**. This column specifies the cost per hour associated with each resource. As mentioned earlier, this is highly variable and should be modified to reflect your own costs.
- **Description**. This column contains a description of each type of resource.

Config – Fixed Tasks

The Config – Fixed Tasks table introduces a set of processes to be accomplished in any upgrade project by the resources listed on the **Config – By Resource** tab. Figure 3.16 shows an example of this tab.

Task	Hours	Description
Project Kick-off	2	Time dedicated to the migration project kick-off meeting.
Development Environment Preparation	4	Setup and verification of the development environment of the original application.
Compilation Verification of the Original Application	8	The original application is compiled in order to verify the setup is OK.
Configuration Management	8	Configuration management of the products obtained during the migration project.

Figure 3.16
Config – Fixed Task tab

The **Config – Fixed Tasks** tab includes the following three columns:

- **Task**. This column lists some common tasks that are a part of every upgrade project.
- **Hours**. This column lists the time required to complete each task. Again, this may need to be changed to reflect you own circumstances.
- **Description**. This column provides a description of each task.

Config – EWIs

The **Config – EWIs** tab displays information about the upgrade estimates for EWIs. It includes the following seven columns:

- **Name**. This column contains the name of each error, warning, and issue.
- **MSDN ID**. This column contains the Microsoft Developer's Network ID of the EWI.
- **Resource**. This column identifies the recommended resource type that should be assigned to resolving each EWI.

- **Effort per occurrence (Minutes)**. This column defines the time required to upgrade an occurrence of an upgrade EWI measured in minutes. This value should be configured to reflect the capabilities of the developers in your company. The default value is usually a good average.
- **Effort per occurrence (Hours)**. This column is the same as the preceding one, except the time is displayed in hours instead of minutes.
- **Cost per hour**. This column identifies the cost associated with the upgrade of each issue by an average developer. This is highly variable and depends on the skill level of the person that will be working on each issue and the salary levels for someone of that level.
- **Functionality**. This column contains a more detailed description of the EWI.

Summary

A successful upgrade project depends on careful assessment and analysis of the project to upgrade. A thorough application of these tasks ensures accurate planning and budgeting. This chapter discussed the information you need to gather to perform these tasks and some of the tools available to help you gather it.

More Information

For more information about improving the performance of your applications, see the white paper titled "Advantages of Migrating from Visual Basic 6.0 to Visual Basic .NET" on MSDN:
http://msdn.microsoft.com/vbasic/productinfo/advantage/default.aspx.

For more information about the Visual Basic Companion from ArtinSoft, go to the ArtinSoft Web site:
http://www.artinsoft.com/Default.aspx.

4

Common Application Types

This chapter can help you to understand how common Visual Basic 6.0 application types are treated before, during, and after an upgrade. The chapter explores the dependencies between an application type and the application's actual architecture from an upgrade perspective. After you have made the decision to upgrade, use this chapter to learn about the similarities and differences between Visual Basic 6.0 and Visual Basic .NET application types, and how to achieve a smooth transition that takes advantage of features available in the upgraded version.

Identifying and Upgrading Application Types

You can learn about your application and the process of upgrading it by comparing it to other applications with similar requirements and architectures. By selecting an application type, you choose a set of capabilities and characteristics that are well suited to the problem that your application solves. Application types are characterized by traits such as audience, intended use, architecture, and capabilities.

In Visual Basic 6.0, there are a variety of application types to choose from. These application types can be divided into two categories: application components and full application executables.

The Visual Basic Upgrade Wizard supports different Visual Basic 6.0 application types, including standard executables (.exes), ActiveX components, and Internet Information Services (IIS) applications. Depending on the original project type, the upgrade wizard generates a .NET project whose output is a Windows-based application (.exe) or a class library (a dynamic link library, or DLL).

To get the most out of the services and features in the .NET platform, you should evaluate the architecture and type of your *upgraded* application. For example, you can transform a highly coupled or monolithic desktop application into an application that includes several independent components in well-defined tiers and takes advantage of technologies provided by Visual Basic .NET. This kind of improvement is discussed in detail in Chapter 17, "Introduction to Application Advancement."

Determining Application Type and Functional Equivalency

When you upgrade your application, your first goal is to achieve functional equivalence between the legacy and upgraded versions. The equivalency between Visual Basic 6.0 and Visual Basic .NET application architecture and project type is shown in the Table 4.1.

Table 4.1: Application Type Equivalency Table

Application Architecture	Visual Basic 6.0 Project Type	Visual Basic .NET Project Type
Desktop application	Standard executable	Windows-based application
Web application	Internet Information Service (IIS) application	ASP. NET Web application
Distributed application	Application that uses distributed component object model (DCOM), Microsoft Transaction Services (MTS), or COM+ services	Application that uses components of System Runtime Remoting and System Enterprise Services .NET Framework namespaces

This chapter provides a high-level summary of the upgrade process for each application architecture and type.

Determining Component and Project Types

Visual Basic applications are built with different components that provide the basic functionality and services that the applications require. Some components can be used in different application architectures; for example, an out-of-process reusable component library can be used in a desktop application or in a distributed application.

Table 4.2 shows how component types correspond to Visual Basic project types.

Table 4.2: Components and Corresponding Project Types

Component Type	Visual Basic 6.0 Project Type	Visual Basic .NET Project Type
Reusable library (out-of-process)	ActiveX .exe	No direct equivalent: upgrade to Windows-based application or class library project.
Reusable library (in-process)	ActiveX DLL	Class library
Visual application component	ActiveX control	Windows control library
Visual application component to be hosted in an Internet application	ActiveX document	No direct equivalent. Re-implement using .NET Frame work components such as UserControls and XML Web forms.

For a summary of how to upgrade these component types, see the "Application Components" section later in this chapter.

Desktop and Web Applications

Desktop and Web applications are two basic frames that include all of the components and source code that make up an application. When you upgrade a desktop or Web application, you should consider the components and their interconnections, the structure of each component, and the overall application architecture.

This section first explains upgrade considerations that are common to both desktop and Web applications, and then concentrates on specific aspects of each application type.

A desktop application can require considerable effort to upgrade to the .NET Framework. One of the difficulties in upgrading an application is because Visual Basic has been completely redesigned for the .NET Framework, and functions and controls do not correspond exactly to their counterparts in Visual Basic 6.0.[1]

Architecture Considerations

As explained in Chapter 2, "Practices for Successful Upgrades," you may need to approach the upgrade by using one of the following two approaches:

- **Vertical upgrade**. This approach requires that you isolate and replace a piece of your application through all *n* tiers.
- **Horizontal upgrade**. This approach requires that you replace an entire tier of your application.

To assure functional equivalence independently of your chosen approach, you must have comprehensive unit and system test suites for the entire application, as well as for individual upgrade modules, components, or tiers. You may find it convenient to upgrade portions of the application that you can test in the early stages of the process. This approach allows you to have different developers or development teams work in parallel, and it helps you to correct issues early in your project. However, as explained in the forthcoming sections, it is not always feasible to independently upgrade and test pieces of an application.

For more information about testing and debugging an upgraded application, see "Testing and Debugging the Upgraded Application" in Chapter 5, "The Visual Basic Upgrade Process," and Chapter 21, "Testing Upgraded Applications."

Single-Tier Applications

Single-tier (or monolithic) applications are applications in which most or all of the code for a given feature is contained in a single source code file. In Visual Basic, this usually means that all of the functional logic is in the control event handler subroutines. All necessary components and functionality are included in a group of forms, making them the conceptual units for the application features. In such applications, development tends to be driven by the user interface. If new features are needed in the application, new components are built into existing forms. For example, when a single-tier application requires database access, then a component for data access is plugged into the form, and all methods that need to access the database use the newly integrated component.

Single-tier applications do not have separate functional layers for data access, business logic, or the user interface. All necessary services and components are integrated in the same development unit (usually a form). This kind of application is highly coupled because all components are closely tied together, usually in the same file or procedure, with many shared structures and connection points. A single-tier application also has low cohesion, which means that the mechanisms that constitute different conceptual entities are so closely tied together that conceptual barriers are obscured. This makes modifying functionality difficult because making a change can affect more than just the intended functional behavior.

When you compile an application of this type, you generate a single executable file that contains all of the necessary information for user-defined forms and modules. To deploy such an application, you must install the executable file and any related third-party components on each of user computer.

The remainder of this section describes issues that you should consider when you upgrade a single-tier application.

Although the example uses a single multiple-document interface (MDI) child form, the concepts presented also apply to solutions composed of several forms and modules and to Web applications that are highly coupled.

The form in this example (which could just as easily be a Web page) allows a user to have a general overview of the telephone calls received at an IT help desk. After the user locates a record, he or she can request more information about it. The form contains a data-bound grid that displays records in a table with telephone call data. The grid uses an ActiveX Data Object (ADO) data control as its data source: the data control provides the communication with the system's database. Because the form displays the full list of calls, the control has been made invisible. Figure 4.1 provides an overview of the form components and architecture.

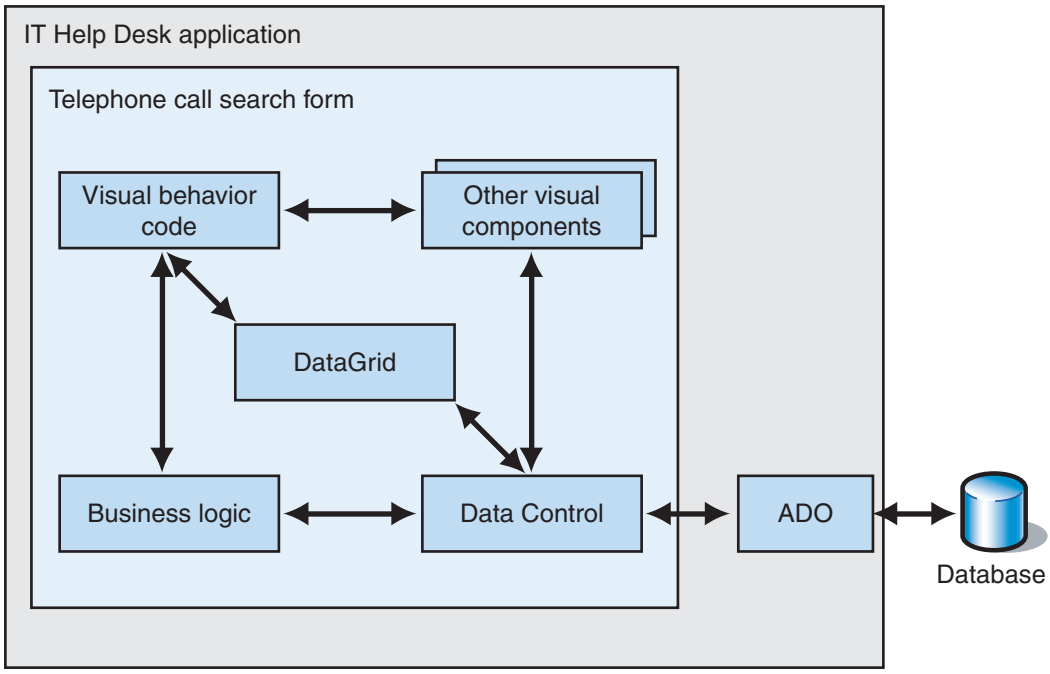

Figure 4.1
Typical monolithic form components

The outer box represents the single tier application, which contains the telephone calls search form. This form includes components for data access, business logic, and visual components, represented in the diagram by the smaller boxes. The arrows in the diagram show the dependences between components. The visual behavior code manages the rest of components, The **DataGrid** control uses an ADO data control to display the information extracted from the database. The business logic interprets the telephone numbers if a user requests details about a call. Although the boxes related to business logic and data access are shown as independent entities, the corresponding code could be intermixed with the rest of the form's code. Note also that all of the functionality is enclosed in the telephone call search form and cannot be used by any application modules outside this form.

Because the complete functionality of this example is encapsulated in a single form, it must all be upgraded and potentially adjusted before the form can be compiled, executed, and tested. For example, the business logic included in the form cannot be upgraded and tested independently of the rest of the code unless the developer rearchitects the original form. The monolithic structure of the application makes an incremental upgrade and test of isolated functionality extremely difficult. The form and all its elements must be upgraded as a single functional unit that constitutes the entire application. In this sense, the upgrade strategy to be followed for a monolithic application is a complete upgrade strategy.

Redesigning Your Single-Tier Application

When you upgrade a single-tier application, your best strategy is to redesign the application and separate its functionality into distinct parts. This approach provides modularity and functional cohesion, making it easier to modify pieces of the application, add or remove functionality, or make functionality available to other applications.

Using a vertical strategy to upgrade parts of your application to the new architecture provides an effective test bed for the new design. The .NET Framework provides architectural support that makes it much easier to build on an upgraded base. (Note that using a vertical strategy also applies to two and three tier applications.)

After you use a vertical approach to upgrade part of the application, you can use interoperability to integrate some of the managed code and the existing unmanaged code. Be careful when you choose which portion of the application to upgrade, because the choice may affect development costs. For example, the cost to integrate a managed form and a group of unmanaged business classes, which can be done through interoperability, is low compared to the costs associated with integrating managed and unmanaged forms that need to interact. This last kind of integration could require increases development costs because of different event models and a user-defined communication and marshaling scheme between managed and unmanaged forms code.

After you have integrated a subset of the application functionality, testing is necessary to ensure that the new and old pieces of the application work together, share data, and provide a seamless experience to the end user or client developer. After successful testing, you will have a basic infrastructure, and you can continue with the vertical upgrade of the application components.

In most cases, you should restructure an application after you have executed an automated upgrade and achieved functional equivalence. This approach allows you to take advantage of the improved Visual Basic .NET features. However, depending on the features supported by the Visual Basic Upgrade Wizard or other automated upgrade tools, you may need to make some structure and component changes before you can automatically upgrade your Visual Basic 6.0 source code. You may need to reorganize the code and replace unsupported classes with supported ones so that the upgrade wizard can apply all the necessary mappings and transformations. For example, data access object (DAO) and remote data object (RDO) components can be slightly reorganized and changed to ADO, which has better support in the Visual Basic Upgrade Wizard and other automated upgrade tools. For more information about upgrade wizard unsupported functionality or about data access component upgrades, see Chapters 7 – 12.

The equivalent .NET application architecture for the upgraded application is a Visual Basic .NET Windows-based application project whose startup object corresponds to the original Visual Basic 6.0 object. (The Visual Basic Upgrade Wizard

generates a NET solution file that contains only this project.) When the wizard upgrades a typical single-tier application, it replaces all library references with references to the common equivalents in the .NET Framework base class library. References to the **System** namespace provide basic functionality for the Form class and XML capabilities. The **Microsoft.VisualBasic** assembly provides functions that are similar to the functions available in the Visual Basic 6.0 VBA library. Each class library that was referenced in the original application will have a corresponding reference in the Visual Basic .NET project. ActiveX control (.ocx) components will have an additional reference that begins with the prefix *Ax* and corresponds to a Windows Forms control wrapper. These references are generated by the **aximp** utility. For example, a reference to the component **MSComctlLib** for common controls results in a new reference to **AxMSComctlLib**. This additional reference provides support for the native members and behavior that all Windows Forms controls support.

The upgraded application will have the same structure as the original. All of the original relationships between application components are in the upgraded version. The Visual Basic Upgrade Wizard automatically generates interoperability wrappers for all ActiveX components that require it. For example, in the telephone calls example form described earlier, a wrapper is generated for the **DataGrid** and ADO components. For more information about interoperability, see the "Application Components" section later in this chapter.

It is important to note that the remaining components are upgraded to native .NET Framework classes, and most of the source code is converted to Visual Basic .NET. You can manually upgrade the wrapped ActiveX components to native .NET components after applying the automatic tools. For example, you can upgrade ADO to ADO.NET and upgrade the **DataGrid** control to the .NET Framework **DataGrid** or to a third-party data grid. However, manual adjustments require more time and effort to achieve.

Two-Tier Applications

A two-tier application is composed of a first tier — the user interface tier — that contains a user interface and business logic, and a second tier — the data tier — that contains all of the data access components. The user interacts with the interface tier, and uses it to perform all application-specific functions. The data tier contains all data sources and manages them so that they can provide all necessary data services to the user interface tier.

The user interface of a Visual Basic 6.0 application can contain forms, Web pages, or both. Additionally, a Visual Basic 6.0 application can have a front end that can be accessed programmatically. In addition, enterprise applications can have rich forms clients as well as thin Web clients that can be accessed remotely.

Figure 4.2 shows a two-tier application form based on the telephone list and filter window explained in the previous section. In this example, the data access components are no longer part of the form module. Instead, they are defined independently, which allows for modularity and code reuse. This architecture also offers more upgrade alternatives than the single-tier architecture. The separation of tiers permits each tier to be upgraded and tested independently. This gives more flexibility to the upgrade plan.

A two-tier application can be upgraded horizontally or vertically. If you select a horizontal strategy, you can update an entire tier and leave the remaining tier unchanged. The upgraded tier can use interoperability features to work with the tier that was not upgraded. You can then upgrade the remaining tier and integrate it into the target application as time and budget permits. Alternatively, you can use a vertical upgrade strategy as explained in the previous section.

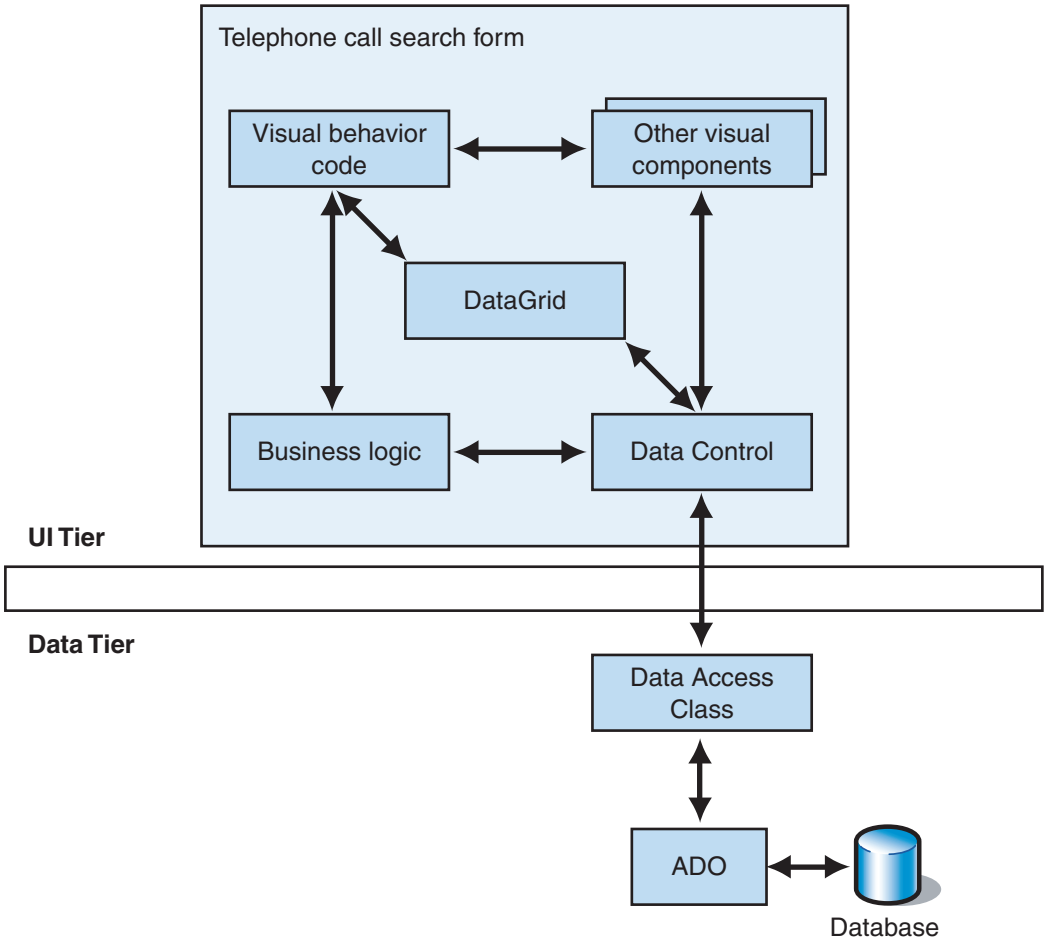

Figure 4.2
Typical two-tier application architecture

When you upgrade a two-tier application, the upgrade wizard produces a Visual Basic .NET project that contains files and references similar to those produced for a single-tier application. The main difference is that a data access class is included in the project files. This class is independent of the other classes and provides services that can be used by any other class. This class contains some wrapped classes that correspond to low-level data access classes (ADO) to provide basic services. If you manually upgrade ADO to ADO.NET, you must also change this data access class. However, if your application was designed in a modular way (with low coupling), you need to make only minimal changes to the rest of the application because internal changes in the user-defined data access class will not affect its users.

Three-Tier Applications

A three-tier application maximizes component cohesion and minimizes tier coupling. This architecture uses three different tiers for the different aspects of the application: a user interface tier, a business logic tier, and a data tier. The difference between a three-tier and a two-tier application is the addition of a business logic tier. The business logic tier contains application rules and algorithms; therefore, the user interface tier is only responsible for managing the interaction between the user and the application. The business logic tier can be segmented into additional logical tiers to produce a multi-tier architecture. The components of the business logic tier can also have a close interaction with the data tier. Figure 4.3 on the next page shows a typical three-tier application.

Because of the further separation of tiers, an upgrade strategy for a three-tier application can be even more flexible than an upgrade strategy for a two-tier application. As with two-tier applications, a three-tier application can be upgraded in a horizontal or vertical way. By using a horizontal strategy, you can upgrade all or part of a tier and leave the other tiers unchanged. The upgraded code can use interoperability features to access the unchanged code. Then, you can gradually upgrade the other tiers and integrate them into the application.

Another important feature of a three-tier architecture is that it allows you to take advantage of specialized skills that your developers may have, and have them work in parallel. For example, one developer or development team could work on modular user interface components while other developers upgrade and adjust the components in the business logic and data access tiers.

When you upgrade a three-tier application, the resulting application will also have three tiers. The upgrade wizard generates ActiveX wrappers for components used in the UI tier. It is probable that you will need to perform additional UI testing because some of the UI components require wrappers, custom upgrades, or reimplementation. However, you can reduce the associated risks by upgrading the UI tier first and connecting it to the other tiers by using interoperability, while you test the UI tier early in your project. This approach allows you to upgrade, test, and integrate the

other tiers in parallel. For additional advice about which tier to upgrade first, see Chapter 2, "Practices for Successful Upgrades."

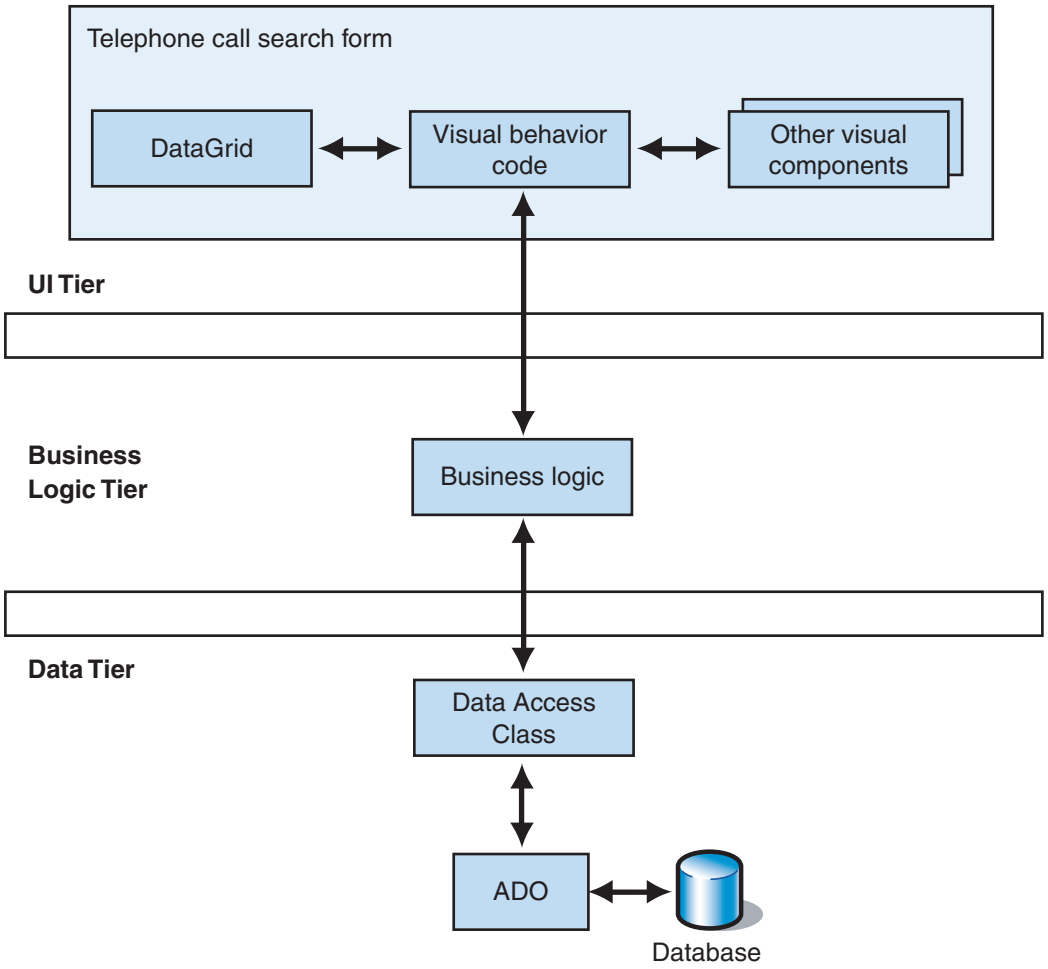

Figure 4.3
Typical three-tier application architecture

Most of the effort to deploy an upgraded three-tier application is spent on setting up the system servers. You must set up application and data servers to allow the clients to access the application. Client setup is simpler because most of the components are installed on the servers, and client computers use an Internet browser to access the application.

Desktop Applications

To create a Visual Basic 6.0 desktop application, you use the standard executable project template. By default, this template starts with an empty form that becomes one of the primary interfaces. The Visual Basic 6.0 integrated development environment (IDE) allows you to include supplementary components and references to take advantage of existing functionality.

A typical desktop application that uses a highly coupled architecture consists of a fat client, possible middle-tier COM components with business logic, a data access tier, and a connection to a data store. In this type of application, the presentation and business logic reside on the client.

In general, a desktop application had the following characteristics:

- **Intended usage**. The application has many visual cues or aids and provides a complex interaction with the user.
- **Architecture**. The application architecture is flexible: for example, it can be a single tier, fully coupled application or a completely modular application with well-defined business logic and data access tiers.
- **Necessary capabilities**. The application requires user interface components, basic services, and data access functionality. Multithreading capability may be required, but synchronized process distribution is rarely needed unless the application uses the distributed component object model (DCOM) or COM+.

The Visual Basic 6.0 to Visual Basic .NET Upgrade Wizard automatically upgrades desktop applications to be Visual Basic .NET Windows-based applications. The wizard does not change the general organization of the code. Therefore, the original application tiers remain intact. However, there are architecture-related issues that you should consider when you upgrade a desktop application. These issues include the following:

- **Language issues**. You may need to perform additional manual upgrading to resolve language issues, such as external DLL function declarations or thread safety. This work can include creating or changing classes so that they provide basic API services or synchronization among objects. You can upgrade external declarations to .NET Framework classes and new interface classes. In addition, you may need to add references. Note that Visual Basic .NET components (including **UserControls**) are not inherently thread-safe. Multiple threads are allowed to access a component simultaneously. As a result, code that is thread safe in Visual Basic 6.0 can be unsafe when it is upgraded to Visual Basic .NET. This change will affect you if you are running your component in a multithreaded environment such as Internet Explorer, Internet Information Services, or COM+.[2] In this case, you must include new synchronization mechanisms to ensure thread safety.

- **Forms issues**. Dynamic data exchange (DDE) and drag-and-drop functionality can cause upgrade issues. Neither Visual Basic .NET nor the .NET Framework support DDE. Because Visual Basic drag-and-drop is considered an outdated feature, the upgrade wizard does not upgrade drag-and-drop – related code. Instead, it preserves the Visual Basic 6.0 drag-and-drop code in the upgraded Visual Basic .NET project. The code will not compile; therefore, you must either delete it and remove the drag-and-drop functionality or change the code to use OLE drag and drop. For more information about these issues, see Chapter 9, "Upgrading Visual Basic 6.0 Forms Features."
- **ActiveX control upgrade**. The upgrade wizard supports most ActiveX components. The upgrade of these components takes advantage of the interoperability that can be built around the original component. However, if an application control is not supported, you must reimplement and re-architect some application functionality, particularly in the user interface. In applications that make extensive use of complex third-party controls, you should perform limited upgrade tests with a simplified user interface. This will help you to determine how many controls are fully or partially supported.

> **For Visual Basic 2005:**
>
> The Visual Studio 2005 version of the Visual Basic Upgrade Wizard will upgrade more ActiveX controls to .NET native controls rather than wrapped controls. The additional supported controls (which are contained in the Microsoft Windows Common Controls library) include: **ToolBar**, **StatusBar**, **ProgressBar**, **TreeView**, **ListView** and **ImageList**.

To deploy an upgraded .NET desktop application, you must do the following:

- Install the application executable and library files on the client computer.
- Use the .NET Framework Assembly Registration Utility (**regasm**) to register the.NET Framework library assemblies that are shared by different applications.
- If the application contains ActiveX components, register these components on each client computer.
- Distribute all the corresponding ActiveX wrapper assemblies.

Web Applications

In most cases, you would use the following technologies to build a Web site or Web application with Visual Basic 6.0:

- **Active Server Pages (.asp)**. This is a server-side scripting run-time environment used to create interactive Web server applications. An .asp file can include HTML sections, script commands, and COM components that execute on the Web server and render a Web page for the client.
- **Internet Information Server (IIS) projects**. These are applications composed of **WebClasses** that run on the Web server and respond to HTTP requests from

clients. **WebClasses** can handle state information for the application. For more information about upgrading an IIS project, see Appendix C, "Introduction to Upgrading ASP," and Chapter 10, "Upgrading Web Applications."

The functionally equivalent server-side technology in Visual Studio .NET is ASP.NET. This is not simply a new version of ASP. It has been entirely redesigned based on the .NET Framework. This section highlights the main considerations that you should take into account when you upgrade ASP code to ASP.NET.

Figure 4.4 shows the general architecture of a Visual Basic 6.0 Web application. A client Web browser sends a request to IIS, and IIS responds by rendering active server pages. The ASP files access COM components that contain part of the application's business logic and data access layers. These components run in the Web server computer or in an application server. The larger, background boxes represent platform servers that provide functionality to different conceptual tiers; for example, the Web server box includes business logic functionality and part of the user interface functionality because it has to provide presentation elements as well executable code in ASP files. The ASP files include application logic and presentation elements in a single module; that is why the active server pages box crosses the barrier between the UI tier and the business logic tier.

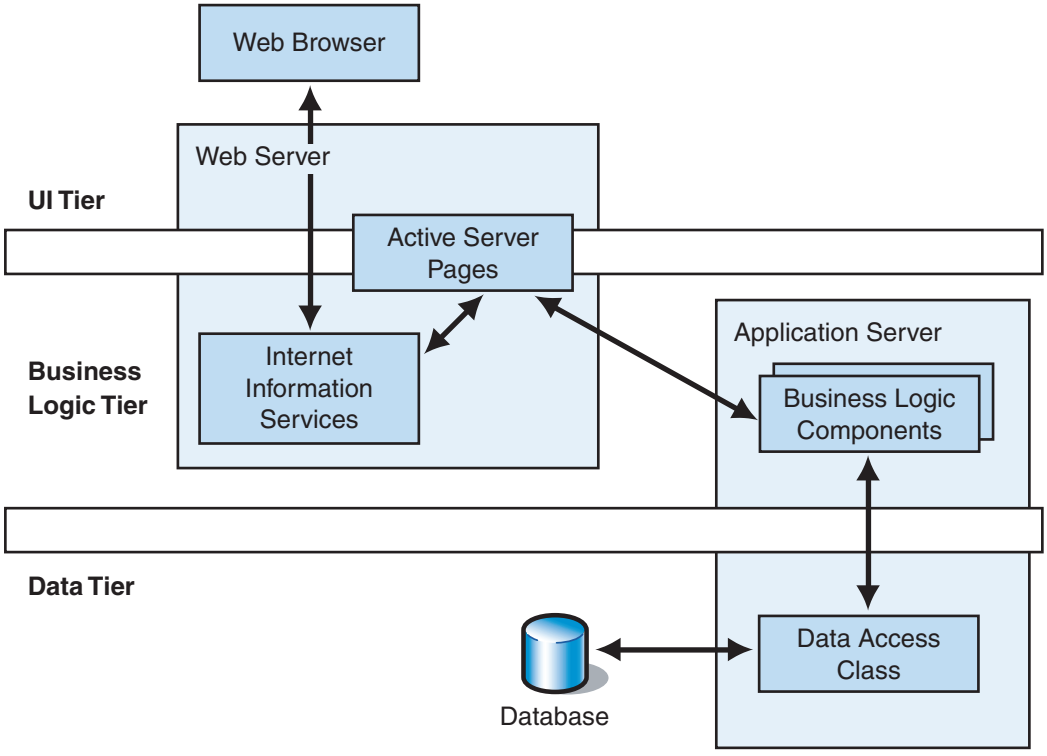

Figure 4.4
Visual Basic 6.0 Web application architecture

Figure 4.5 shows the architecture of an ASP.NET Web application. If you compare Figure 4.4 to Figure 4.5, you can see the underlying differences between the two technologies, and where you need to focus your upgrade efforts.

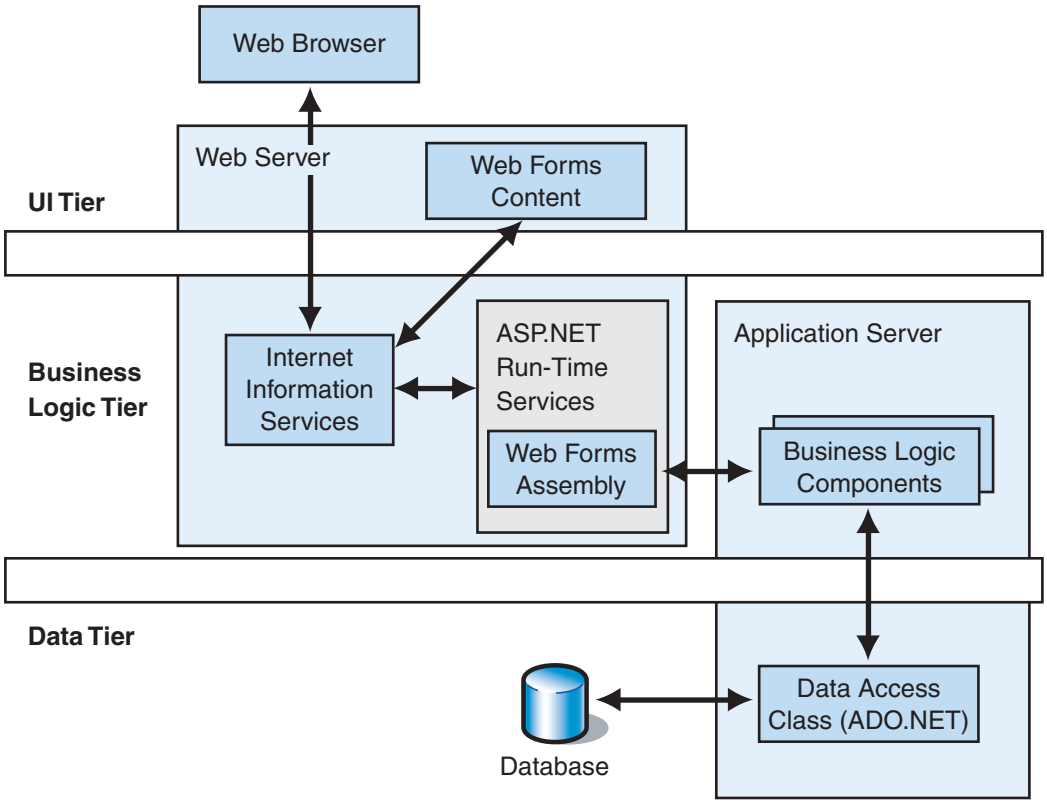

Figure 4.5
ASP.NET Web application architecture

In a transition from ASP to ASP.NET, most of the architecture changes occur in the components to be deployed on the Web server. The Web document model is completely changed. In ASP.NET, Web pages are built as Web forms, which have separate files for document content and document presentation logic and behavior. Therefore, the business logic and user interface aspects of Web pages are separate. In this model, the Web form content box is included in the UI tier, and the Web form executable code (Web form assembly) is included in the business logic tier. The ASP.NET run-time component provides additional services, including Web security, caching, and other performance features.

Coexistence

ASP and ASP.NET can coexist on the same Web server; however, they are executed as different, non-communicating processes.

A Web application within a site can contain both ASP.NET pages and ASP pages. This offers some advantages if you need to move a large, functionally disjointed and rapidly changing site to ASP.NET one piece at a time. You can upgrade different sections of the application that do not depend on each other to ASP.NET, and they can coexist with the rest of the application. If you are transitioning to ASP.NET as a long-term strategy, you should use this opportunity to make as many architectural and design improvements as you can. For more information about upgrading to ASP, see "Migrating to ASP.NET: Key Considerations" on MSDN.

Because the same Web server can access both ASP and ASP.NET pages, you do not need to upgrade your existing ASP pages to ASP.NET – compatible pages. However, there are many advantages to doing so. Some of the biggest advantages include the following:

- **Increased performance**. Independent tests have shown that ASP.NET applications can handle two to three times the requests per second as classic ASP applications. For more details about performance improvements, see Chapter 1, "Introduction."
- **Increased stability**. The ASP.NET runtime closely monitors and manages processes. If a process malfunctions (for example, if it leaks or deadlocks), ASP.NET can create a new process to replace it. This helps to keep your application available to handle requests.
- **Increased developer productivity**. Features such as server controls and event handling in ASP.NET can help you to build applications more rapidly and with fewer lines of code. It is also easier to separate code from HTML content.

For more information about converting to ASP.NET, see "Converting ASP to ASP.NET" on MSDN.

The ASP to ASP.NET Migration Assistant

Microsoft provides a free tool for automatically converting ASP pages to ASP.NET. The tool does not automatically convert all of the ASP features, but it can simplify an upgrade project by automating some of the steps. To download the migration assistant, see "ASP to ASP.NET Migration Assistant" in the Microsoft ASP.NET Developer Center on MSDN.

If you are upgrading an enterprise application that uses ASP, you may need to use two different upgrade assistants. If the application has a business logic tier, user-defined classes in the data access tier, and additional components that run on an application server, you should use the Visual Basic Upgrade Wizard. After you

successfully upgrade these foundation components, you can convert the ASP code by using the ASP to ASP.NET Migration Assistant.

For a detailed discussion of the ASP to ASP.NET Migration Assistant, see Appendix C, "Introduction to Upgrading ASP."

Porting the ASP Application to ASP.NET

When you upgrade an ASP application to ASP.NET, you must decide how much time you want to spend incorporating the features of ASP.NET into the existing ASP application. You can approach this work in one of two ways:

- You can run the migration assistant on the ASP pages, and then apply manual changes.
- You can re-architect the application and use many of the features of .NET, including ASP.NET Web controls, ADO.NET, and the .NET Framework classes.

Although the latter option will make your ASP.NET pages more readable, maintainable, and feature-rich, it will also require more time and effort to complete the upgrade.

Key Considerations

When you upgrade existing ASP pages to ASP.NET pages, you should also be aware of the following:

- **Core API changes**. The core ASP APIs have a few intrinsic objects (**Request**, **Response**, **Server**, and so on) and their associated methods. With the exception of a few simple changes, these APIs continue to function correctly under ASP.NET.
- **Structural changes**. Structural changes are those that affect the layout and coding style of ASP pages. You need to be aware of several of these to ensure that your code will work in ASP.NET.
- **Visual Basic language changes**. There are some changes between VB Script and Visual Basic .NET script that you should prepare for before you upgrade your application.
- **COM-related changes**. COM has not been changed at all. However, you need to understand how COM objects behave when you use them with ASP.NET.
- **Application configuration changes**. In ASP, all Web application configuration information is stored in the system registry and the IIS metabase, while in ASP.NET each applications has it own Web.config file.
- **State management issues**. If your application uses the **Session** or **Application** intrinsic object to store state information, you can continue to use these in ASP.NET without any problems. As an added benefit, ASP.NET provides additional options for your state storage location.

- **Base class libraries**. The base class libraries (BCL) provide a set of fundamental building blocks that you can use in any application you develop, whether your application is an ASP.NET application, a Windows Forms application, or a Web service. You can significantly improve your application by using the BCL.

For information about each of these concerns, see Appendix C, "Introduction to Upgrading ASP" and "Migrating to ASP.NET: Key Considerations" on MSDN.

The deployment of an upgraded Web application requires the installation of a client and one or more server computers. All necessary client-side ActiveX components must be installed on the client. The server must be set up with the complete set of server controls, component libraries, and data providers.

Application Components

Microsoft developed the component object model (COM) to allow interaction between applications and to provide a platform for code reuse.

You can create the following types of COM components with Visual Basic 6.0:

- **ActiveX code libraries**. You can compile these components as COM executable programs or as DLLs. These libraries include classes that you use by creating instances in the client application. In Visual Basic 6.0, the project templates you use to create these COM components are referred to as ActiveX executables and ActiveX DLLs.
- **ActiveX controls**. These controls are standard interface elements that allow you to rapidly assemble reusable forms and dialog boxes.
- **ActiveX documents**. ActiveX documents are COM components that must be hosted and activated within a document container. This technology permits generic shell applications, such as Internet Explorer, to host different types of documents.

This section provides general information that you can use when you upgrade any type of supported application component.

Native DLLs and Assemblies

In Visual Basic 6.0, application components are compiled as DLLs that contain executable code to be used by client applications. These libraries are deployed as files that have .dll or .exe file name extensions and require the library information to be registered in the system registry before client applications can access the library elements.

An assembly is the primary building block of a .NET Framework – based application. It is a component library that is built, versioned, and deployed as a single implementation unit. Every assembly contains a manifest that describes that assembly. The manifest contains library metadata that maintains the following information:

- The assembly name, version, culture, and digital signature (if the assembly is to be shared across applications).
- The names of the files that the assembly is built with.
- The names of resources used in the assembly, including information about which resources are exported from the assembly.
- Compile-time dependencies on other assemblies.
- Permissions required for the assembly.

The .NET common language runtime (CLR) uses this information to resolve references, enforce version binding policy, and validate the integrity of loaded assemblies. By using the assembly manifest, the CLR eliminates the need to register .NET components in the system registry before a client application can use them, and therefore simplifies application deployment and reduces versioning problems. Most end users and developers are familiar with versioning and deployment issues that can occur in component-based systems. Many of these issues are related to the registry entries that are necessary to activate a COM class. The use of assemblies in the .NET Framework solves a high percentage of these deployment problems. Assemblies facilitate zero-impact application installation, and they simplify uninstalling and replicating applications.

Versioning Problems

Currently two versioning problems occur with Win32 applications:

- Versioning rules cannot be expressed for pieces of an application and enforced by the operating system. The current approach relies on backward compatibility, which is often difficult to guarantee.
- There is no way to maintain consistency between sets of components that were built together and the set that is present at run time.

These versioning problems combine to create DLL conflicts, where installing one application can inadvertently break an existing application because a certain software component or DLL was installed that was not fully backward compatible with a previous version. After this problem occurs, there is no support in the system for diagnosing and fixing the problem.

The Assembly Solution

To solve versioning problems, as well as other problems that lead to DLL conflicts, the .NET CLR provides assemblies as a means to achieve the following goals:

- Enable developers to specify version rules between different software components.
- Provide the infrastructure to enforce versioning rules.
- Provide the infrastructure to allow multiple versions of a component to be run simultaneously (this is called *side-by-side execution*).

Interoperability Between .NET and COM

When you upgrade application components from Visual Basic 6.0 to Visual Basic .NET, you should take into consideration the interoperability capabilities of the .NET CLR. For example, you can upgrade a third-party component through different mechanisms, including interoperability wrappers, upgrading to .NET Framework components, and upgrading to third-party components designed for Visual Basic .NET. You should select the appropriate option for your application by considering the effort required, the expected results, the available resources, your future application development plans, and other factors.

Interoperability wrappers allow you to access COM components from a .NET programming language. (COM clients cannot access .NET assemblies directly.) A Visual Basic 6.0 application can access a combination of converted components and wrapped components. For example, you can create interoperability wrappers for your COM components test them in Visual Basic .NET. If the wrapped component works as expected, it can be accessed through the wrapper or you can upgrade it to .NET.

► **To create a COM wrapper**

1. Use the Regsvr32 tool (Regsvr32.exe) to register the original COM component on the target computer.
2. Use the Type Library Importer utility (Tlbimp.exe) to create an interoperability wrapper.
3. If the component is an ActiveX control, generate a Windows Forms ActiveX control importer by using the Windows ActiveX Control Importer utility (Aximp.exe).

For more information about creating COM wrappers and for example code, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET."

Figure 4.6 on the next page shows how interoperability is set up and how a .NET client uses it.

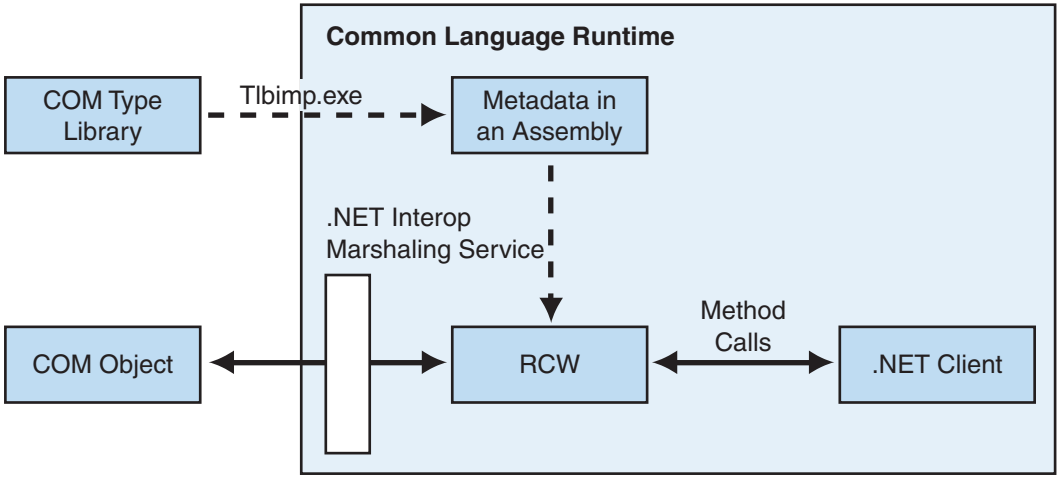

Figure 4.6
Accessing a COM component through interoperability

In Figure 4.6, the Type Library Importer utility (Tlbimp.exe) analyzes the COM library, and converts the type definitions to equivalent definitions in an assembly that contains metadata which corresponds to the original COM library. The CLR. creates a runtime callable wrapper (RCW) for each COM object. The main function of the RCW is to marshal the calls between the .NET client and the COM object. This marshaling includes the conversion of data types so that the two technologies can communicate. The .NET client can access the COM library by using the RCW as an intermediary.

The interoperability wrapper created is a .NET assembly that wraps the COM component. The assembly must be registered in the global assembly cache if it will be shared by different applications. There are several reasons why you might want to put an assembly in the global assembly cache:

- **The global assembly cache provides a central location for shared assemblies**. If an assembly will be used by multiple applications, it should be put in the global assembly cache.
- **The global assembly cache can improve file security**. Administrators often protect the WINNT directory by using an access control list (ACL) to control write and execute access. Because the global assembly cache is installed in the WINNT directory, it inherits that directory's ACL.
- **The global assembly cache allows side-by-side versioning**. The global assembly cache can maintain multiple copies of assemblies with the same name, but different version information.
- **The global assembly cache is the primary CLR search location**. The common language runtime checks the global assembly cache for an assembly that matches the assembly request before probing or using the code base information in a configuration file.

For more information about wrappers, see "Customizing Standard Wrappers" in the *.NET Framework Developer's Guide* on MSDN.

Despite the advantages, you should share assemblies by installing them in the global assembly cache only when necessary. As a general guideline, keep assembly dependencies private and place assemblies in the application directory unless sharing an assembly is explicitly required. In addition, you do not have to install assemblies in the global assembly cache to make them accessible to COM Interop or unmanaged code.

Note that there are situations where you should not install an assembly into the global assembly cache. For example, if you place one of the assemblies in an application in the global assembly cache, you can no longer replicate or install the application by using XCOPY installation. In this case, you must place all of the application's assemblies into the global assembly cache. For more information about working with assemblies, see "Working with Assemblies and the Global Assembly Cache" in the *.NET Framework Developer's Guide* on MSDN.

There are two ways to install an assembly in the global assembly cache:

- **Use Microsoft Windows Installer 2.0**. This is the recommended and the most common way to add assemblies to the global assembly cache.
- **Use the Global Assembly Cache tool (Gacutil.exe)**. This approach is recommended for development purposes. However, you should not use this approach to add production assemblies to the global assembly cache.

Reusable Libraries

Visual Basic 6.0 components can run in any one of three places:

- In the same address space as the client (in-process).
- On the same computer (out-of-process).
- On a remote computer (out-of-process).

An in-process component can be implemented as a DLL or as an ActiveX control. Each client application that uses the component starts a new instance of the component.

Out-of-process components are implemented as executable files and run in their own process space on a local or remote computer. Communication between the client and an out-of-process component requires that parameters and return values be marshaled across process boundaries. A single instance of an out-of-process component can service many clients, share global data, and provide insulation between client applications. This kind of component can be implemented by means of DCOM or COM+. For information about upgrading this type of component, see the "Distributed Applications" section later in this chapter.

The rest of this section explains how to upgrade of ActiveX code components. These components correspond to the Visual Basic 6.0 ActiveX DLL project template.

In most cases, code components have no user interface. Instead, they consist of libraries of classes that allow client applications to instantiate objects based on those classes. In earlier documentation, code components were referred to as OLE Automation servers.

The equivalent .NET project for an ActiveX DLL is a .NET class library. The Visual Basic Upgrade Wizard automatically upgrades most of the code inside the original ActiveX DLL project and generates the corresponding Visual Basic .NET project type. After an automated upgrade, you will have language issues that you must correct manually. For an explanation on unsupported language features, see Chapters 7 – 11.

Visual Basic .NET class libraries contain reusable classes or components that can be shared with other projects. This project type is considered to be windowless and will not contain a Windows Form class; this restriction may be an obstacle for you if you used the Visual Basic 6.0 code component library to provide standard libraries of modal and modeless dialogs. However, the Visual Basic Upgrade Wizard upgrades form objects and compiles them inside the component. If it is necessary to allow other applications to use these form classes, you must change the corresponding access level from **Friend** to **Public**.

To further improve your application, you should move all visual classes to another component. This will separate the presentation and business tiers that exist in a single component and produce a more modular architecture.

ActiveX Controls

An ActiveX control is a COM component with user interface elements. ActiveX controls were previously known as OLE controls or OCX controls. These controls can be used in the same way as any of the standard built-in controls, and provide an extension to the Visual Basic 6.0 toolbox. ActiveX controls created in Visual Basic can be used in different container applications, including Visual Basic applications, Microsoft Office documents, and Web pages accessed through a Web browser like Microsoft Internet Explorer.

An application can provide ActiveX controls with other Visual Basic 6.0 components, or it can access and use third-party ActiveX controls. If the ActiveX controls are components in your application, you can use the Visual Basic Upgrade Wizard to upgrade the source code for the control. If the ActiveX control source code is not available, you can replace the control with other upgraded components or you can use interoperability to access the control, as described previously in this chapter. The remainder of this section explains how to upgrade user-defined ActiveX controls.

In Visual Basic 6.0, an ActiveX control is always composed of a **UserControl** object and any controls (known as *constituent controls*) that are placed on it. The equivalent .NET component for an ActiveX control is a .NET **UserControl**. When you use the Visual Basic Upgrade Wizard to upgrade an ActiveX control, a class library project is created inside the target solution. The project contains a **UserControl** module that encloses a class declaration which is equivalent to the original Visual Basic 6.0 **UserControl**. The new class inherits from **System.Windows.Forms.UserControl**, which in turn inherits from **ContainerControl** and contains all of the standard positioning and mnemonic-handling code that is necessary in a base for creating new controls.

The wizard automatically upgrades aspects of the Visual Basic 6.0 **UserControls** to .NET **UserControls**, as follows:

- **Constituent visual elements**. The wizard updates basic elements of the **UserControl** and all corresponding design time properties.
- **Internal logic**. The wizard updates code that controls the component behavior and the interaction with clients.
- **Event handlers**. The wizard converts event handler declarations included in component clients in accordance with the new .NET **UserControl** events. Note that only those component clients upgraded during the same upgrade process as the original **UserControl** are automatically updated to accommodate changes caused by the upgrade of the **UserControl**. If other user applications use a compiled version of the ActiveX control, their access will be upgraded as access to a wrapped ActiveX control. You can improve this last upgrade by including all necessary components in the same upgrade project so that the wizard can process them at the same time.

The Visual Basic Upgrade Wizard does not support Visual Basic 6.0 project groups. If ActiveX controls are included in a Visual Basic 6.0 project group, you should use the following strategy to upgrade the project package:

1. Obtain a list of the dependencies between the projects that compose the application.
2. Determine which projects do not depend on other projects, and upgrade them as individual components.
3. For each project that has a dependency on another project, replicate all needed objects into the dependent project. This will allow the dependent project to process all of the transformations that occurred in other project objects. Upgrade the dependent projects.
4. When all projects are upgraded, eliminate redundant objects and set the proper dependencies between projects. These should be the same as in the original project group.

When you upgrade a user-defined ActiveX control to Visual Basic .NET, you can include it in the control toolbox, just as you would any pre-defined control. User-defined ActiveX controls appear in the **User Control Tab** of the toolbox, while the predefined controls appear in the **General** tab of the toolbox. You can place instances of the upgraded control on .NET Windows Forms, and you can set the control's design time properties by using the .NET form editor.

You can use property pages to define a custom interface for setting properties of an ActiveX control; this is an extension of the **Properties** window. However, the Visual Basic Upgrade Wizard does not support this feature, and you must reimplement the property page user interface, synchronize the property values and the state of the control, and associate the control properties and dialog box fields. For more information, see Chapter 9, "Upgrading Visual Basic 6.0 Forms Features."

ActiveX Controls Embedded in Web Pages

You can insert ActiveX controls in Web pages by using the **<Object>** HTML tag. This tag receives a **ClassId** parameter that corresponds to the ActiveX component identifier. The identifier is required so that the component can be registered on the client computer and instantiated and displayed correctly by the Web browser.

You can achieve functional equivalence for an upgraded Web application by leaving embedded controls inside **Object** tags. If the original ASP application included server-side components, you can wrap these components by using the interoperability feature.

Although you can access embedded controls programmatically, you may discover that the controls affect the quality of the application you are upgrading, as follows:

- You must use additional languages — such as VBScript, JScript®, JavaScript, and DHTML — to achieve dynamic behavior.
- Your need more steps to deploy the application, and additional files need to be downloaded.
- The component will have fewer resources — such as application servers and database resources — available to it.

ASP.NET provides a different model for Web development, which includes control types that are specifically designed to be used in a Web environment. These control types include:

- **HTML server controls**. These are HTML elements exposed to the server so you can program them. HTML server controls expose an object model that maps very closely to the HTML elements that they render.
- **Web server controls**. These are controls with more built-in features than HTML server controls. Web server controls include form-type controls, such as buttons and text boxes, and special-purpose controls, such as a calendar. Web server

controls are more abstract than HTML server controls in that their object model does not necessarily reflect HTML syntax.

- **Validation controls**. These are controls that incorporate logic to allow you to test a user's input. You can attach these controls to an input control to test what the user enters for that input control. Validation controls are provided to allow you to check for a required field, to test against a specific value or pattern of characters, to verify that a value lies within a range, and so on.
- **User controls**. These are controls that you create as Web Forms pages. You can insert Web Forms user controls in other Web Forms pages, which is an easy way to create menus, toolbars, and other reusable elements.

These control types provide reimplementation alternatives for ActiveX controls embedded in Web pages. Depending on the functionality of the original ActiveX control, you may find it convenient and appropriate to implement it as a Web control. For example, if an application has an ActiveX control that communicates with different servers to obtain and process data, it is generally more efficient to create a new Web server control that executes the communication and heavy processing on the server and renders visual elements to the Web client.

The ASP to ASP.NET Migration Assistant tool treats ActiveX controls embedded in Web pages as HTML tags and keeps them in the target ASP.NET files. Therefore, these elements of the Web page are not automatically transformed. You must manually complete any further upgrades.

You should wait until after you complete the initial (automatic) upgrade before you re-implement any components. After you reach functional equivalence, you can complete any re-implement as part of the upgraded application advancement. For more information about this process, see Chapter 17, "Introduction to Application Advancement" or "Introduction to ASP.NET Server Controls" on MSDN.

ActiveX Documents

ActiveX documents are COM components that must be hosted and activated within a document container. They provide the application functionality; they also provide the ability to persist and distribute copies of the data intrinsic to the application. These components can be used on HTML pages or as alternatives to HTML pages, and they can be deployed so that users can navigate transparently between ActiveX documents and other pages in the application or Web site.

The .NET Framework does not have a component type equivalent to ActiveX documents. Therefore, you must redesign and/or re-implement ActiveX documents by using a .NET Framework – based component.

The following list presents important ActiveX document features that you should consider when you upgrade to Visual Basic .NET:[3]

- **Automatic downloading of components over the Internet**. You can create a link to the ActiveX document so that the browser can find and download all components that are needed to run the component.
- **Hyperlinking of objects**. In a hyperlink-aware container, you can use the properties and methods of the Visual Basic **Hyperlink** object to jump to an URL or navigate through the history list.
- **Interaction with the container window**. You can use ActiveX documents to access additional pieces of the container window; for example, the component menus can be merged with the browser menus.
- **Storing of data**. When you deploy an ActiveX document in Internet Explorer, you can store data through the **PropertyBag** object.

You can use the upgrade wizard to partially upgrade ActiveX documents if you move the ActiveX document source code to another module type that the upgrade wizard supports. For example, you can copy the source code and the controls contained in the ActiveX document into a new Visual Basic 6.0 **UserControl**. A considerable portion of the **UserControl** source code will be automatically upgraded; however, all unsupported features in the original ActiveX document source code will not be upgraded and must be re-implemented.

You can re-implement ActiveX document components by using Visual Basic .NET **UserControls** or XML Web forms, which can be hosted in different types of containers. You will lose some of the original functionality unless you redesign and implement it in the new platform. You can implement most of this functionality by using the services provided by the .NET Framework, such as serialization interfaces and the ability to run at the server or at the client depending on your chosen design.

Distributed Applications

A distributed application is one in which some of the application components are executed on remote computers and there is interaction between local and remote components. To have an effective interaction between these components, you must use numerous basic services, such as communication protocols, security, and resource locator services. A distributed application offers important advantages for the enterprise environment, including improved scalability, reliability and failure tolerance.

The following sections describe some of the things you should consider when you upgrade components of distributed applications.

DCOM Applications

The distributed component object model (DCOM) extends the component object model (COM) to support communication between objects on different computers on

a local area network (LAN), a wide area network (WAN), or even the Internet. With DCOM, your application can be distributed to any location that makes sense to your users and to the application.

DCOM is an extension of COM; therefore, you can use your existing COM-based applications, components, tools, and knowledge when you move to DCOM. DCOM handles low-level details of network protocols.

Remoting is considered to be the .NET equivalent of DCOM. It is a framework designed to simplify communication between objects. Remoting supports communication between objects that exist in different application domains and have different contexts, regardless of whether or not the objects are on the same computer or even the same application domain.

The remoting framework is built into the common language runtime and can be used to build sophisticated distributed applications. Some of the features provided by .NET remoting are:[4]

- **Proxy objects**. .NET remoting creates a proxy object when a client activates a remote object. The proxy object acts as a representative of the remote object. All calls made to the proxy are forwarded to the correct remote object instance.
- **Object passing**. The remoting framework provides mechanisms for passing objects between applications. These mechanisms handle the marshaling details.
- **Activation models**. .NET remoting provides more than one way to activate remote objects.
- **Stateless and stateful objects**. .NET remoting provides many ways to achieve state management and to manage stateless objects.
- **Channels and serialization**. .NET remoting provides a transport mechanism called .NET channel services to transport messages between applications and application domains. .NET provides serialization (it can transport objects across byte streams) and uses formatters to encode and decode these byte streams.
- **Lease-based lifetime**. .NET remoting allows the application to control the length of time that a client can remain connected to a remote object by using a *lease*. When the lease expires, the object is automatically disconnected. .NET remoting also provides methods to extend a lease when needed.
- **Hosting objects in IIS**. .NET remoting objects can be hosted in any .NET-based executable or managed service, but they can also be hosted in IIS. This allows remoting objects to be exposed as Web services.

You can upgrade a Visual Basic 6.0 application that uses distributed component technologies to Visual Basic .NET in two different ways:

- By using new COM classes
- By using .NET remoting

The following sections describe the advantages and disadvantages of each approach.

Using COM to Upgrade a Distributed Application

For the first option, you must create additional COM classes to serve as proxies that communicate with the DCOM infrastructure. Managed code can access these new classes by using COM interoperability. This alternative has the following advantages:

- **It provides an easy and fast upgrade implementation**. The new COM wrapper classes only need to provide a communication link between the DCOM infrastructure and Visual Basic .NET.
- **It provides upgrade path flexibility**. Because this solution does not need extensive resources, you can use multiple stages to implement it if your application is complex.
- **It allows early testing of component interaction**. You can schedule early testing of these components in parallel with other upgrade tasks.

This is a mixed solution, where managed code interacts with unmanaged components and operating system services. The main disadvantages of this approach are:

- **Additional communication requirements**. The new intermediary classes and interoperability wrappers consume more communication and processing resources.
- **Adoption of new technologies is limited**. Because of the dependency on legacy technologies, adoption of new features and services offered by the .NET Framework are limited.
- **Reduced maintainability**. The new classes increase the complexity of the application.

Using .NET Remoting to Upgrade a Distributed Application

Another upgrade option is reimplement the application and use .NET remoting services. You can upgrade the application logic by using the Visual Basic Upgrade Wizard. You will then need to use .NET remoting services to re-implement the code that uses DCOM. To enable remoting, the distributed objects must derive from the **MarshalByRefObject** class. Note that this is a built-in feature of COM+ classes, which inherit from **ServicedComponent**, which in turn inherits from the **MarshalByRefObject** class.

MTS and COM+ Applications

After you create and deploy a business logic COM component in a single computer environment, you may need to separate the component from its original run-time environment and place it in a different administrative and execution platform. In this way, you can achieve a multi-tier architecture for your application.

There are several reasons why you may need to perform for this type of separation, including scalability, manageability, accountability, and performance. Microsoft provides two main technologies to create this type of architecture: Microsoft Transaction Server (MTS) and COM+.

MTS provides transaction services, which facilitate the transition from single-user to multi-user architectures and provide application infrastructure and administrative support for building scalable and robust enterprise applications. Although transaction management is an integral part of many applications, MTS also provides useful services for applications that do not use transactions at all.

Components that encapsulate application logic can run under the control of MTS and are invoked by presentation services from a variety of clients, including:[5]

- Traditional applications that were developed in Visual Basic or other languages that support COM.
- Web browsers.
- Active Server Pages scripts that run within IIS.

You can use Visual Basic 6.0 with MTS to build *n*-tier applications. To do this, you develop COM DLL components that run on middle-tiered servers under the control of MTS. When clients call these COM DLLs, Windows automatically routes the requests to MTS.

Other services provided by MTS that are relevant to upgrade efforts include:

- Component transactions.
- Object brokering.
- Resource pooling.
- Just-in-time activation.
- Administration.

Upgrade Strategy

One upgrade strategy that you can use is to provide proxy COM classes that can be accessed through interoperability, as explained in the "DCOM Applications" section earlier in this chapter. However, this approach may not be adequate in all situations, and you may need to re-implement MTS and COM+ services.

The .NET Framework contains classes that offer similar functionality or provide an interface to mechanisms used by COM+. These classes are included in the **System.EnterpriseServices** namespace, which provides an infrastructure for enterprise applications that allows .NET objects to access COM+ services. These classes make .NET Framework objects useful for enterprise applications.

The core services employed by MTS and COM+ applications are encapsulated in the COM+ services type library (COMSVCS.DLL).

Note: Most of the upgrade effort for components discussed in this section will be automatically performed by the Visual Studio 2005 version of the Visual Basic Upgrade Wizard.[6]

General Considerations for COM+ Visual Basic 6.0 Projects

You should understand the information in this section so that you can correctly prepare and set up the .NET application that will be the target of a COM+ application.

The **System.EnterpriseServices** namespace contains all of the classes necessary to interact with COM+. To enable COM+, you must include a reference to this namespace in the target Visual Basic .NET project.

You need to use the Strong Name tool (Sn.exe, which is included with the .NET Framework) to generate a key file, and then add this file and an assembly attributes file that includes the **AssemblyKeyFile** attribute to the project file. Give the project file the same name as the original Visual Basic 6.0 project, and store it in the upgrade output folder.

You should include documentation or utilities to help users register assemblies with .NET Services Installation Tool (Regsvcs.exe) so they can deploy the upgraded components. You can use the registration helper API (**System.EnterpriseServices.RegistrationHelper.InstallAssembly**) for this purpose.

Table 4.3 shows the components and services that provide the core MTS and COM+ functionality and their corresponding equivalents in the .NET Framework. For detailed information about upgrading each component, see Chapter 15, "Upgrading MTS and COM+ Applications."

Note: Each .NET component or namespace shown in the table should be prefixed with **System.EnterpriseServices**; for example, the full name for **CompensatingResourceManager** is **System.EnterpriseServices.CompensatingResourceManager**.

Table 4.3: MTS and COM+ Components .NET Equivalents

COM+ Component/Service	.NET Equivalent
Compensating resource manager	Components in the **CompensatingResourceManager** namespace
Object pooling	**ServicedComponent**
Application security	**SharedPropertyManager**, **SharedPropertyGroup**, **SharedProperty**
Object constructor strings	**ServicedComponent**
Transactions	The functionality can be achieved with the class **ServicedComponent** and an appropriate mapping for the **MTSTransactionMode** property, as specified with a **TransactionOption** enumeration value.

Unsupported Functionality

The COM+ services type library is used with Visual Basic and other languages capable of building COM applications. Some members included in this library cannot be used in Visual Basic 6.0 because they were specifically created for use with Visual C++. An example of such a class is the **CServiceConfig** class. These classes may have .NET Framework mappings, but they are unsupported by Visual Basic 6.0.

Similarly, Visual Basic 6.0 supports members that do not have semantic equivalents in the .NET Framework. These unsupported components include functionality in the following areas: security, context management, events management, resource locator, service configuration, and log management.

For more information about COM+ upgrades and unsupported functionality, see Chapter 15, "Upgrading MTS and COM+ Applications."

Summary

Understanding the possible application types that you may need to upgrade will help you to better understand how each type should be upgraded. Each type has features that are easy to upgrade and features that are hard to upgrade. Being aware of potential problem areas before you begin can lessen the impact those problem areas will have on your upgrade process.

With an understanding of the different categories of applications, you can now begin learning the general upgrade processes and practices that are applicable to all upgrade projects. The next chapter will give you the details you need to understand the Visual Basic 6.0 to Visual Basic .NET upgrade process.

More Information

For more information about upgrading to ASP see "Migrating to ASP.NET: Key Considerations" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/aspnetmigrissues.asp.

For a free tool to automatically convert ASP pages to ASP.NET, see "ASP to ASP.NET Migration Assistant" in the Microsoft ASP.NET Developer Center on MSDN:
http://msdn.microsoft.com/asp.net/migration/aspmig/aspmigasst/default.aspx.

For more information about wrappers, see "Customizing Standard Wrappers" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcustomizingstandardwrappers.asp.

For more information about working with assemblies, see "Working with Assemblies and the Global Assembly Cache" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconworkingwithassembliesglobalassemblycache.asp.

For more information about ASP.NET server controls, see "Introduction to ASP.NET Server Controls" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconintroductiontowebformscontrols.asp.

Footnotes

1. Upgrading Microsoft Visual Basic 6.0 to Visual Basic .NET e-Book. Chapter 3: Upgrading options \ Selecting projects to upgrade
2. Upgrading Microsoft Visual Basic 6.0 to Visual Basic .NET e-Book. Chapter 11: Resolving issues with language \ Making Your Code Thread-Safe
3. Desktop Applications with Microsoft Visual Basic 6.0, MCSD Training Kit.
4. Upgrading Microsoft Visual Basic 6.0 to Visual Basic .NET e-Book. Chapter 21: Upgrading Distributed Applications \ Remoting
5. Web Applications with Microsoft Visual InterDev 6.0, MCSD Training Kit.
6. Visual Basic Upgrade Wizard Whidbey version specification, COM+ Services Type Library Migration

5

The Visual Basic Upgrade Process

This chapter provides an in-depth explanation of the technical aspects of the Microsoft Visual Basic 6.0 to Visual Basic .NET upgrade process. It explains each step of the process, from initial preparation to final deployment of the application. It also describes how to use the Visual Basic Upgrade Wizard to automate much of the upgrade. It also discusses tasks you will have to manually perform and provides general recommendations to achieve a successful upgrade.

Procedure Overview

The steps involved in an upgrade project can be divided in the following groups:

1. Application preparation
2. Application upgrade
3. Application testing and debugging

These task groups require specialized skills and experience. The tasks included in each group can be executed with a certain level of parallelism, depending on the availability and specialization of resources needed. The task groups do not define a linear order of execution; there may be groups of tasks that are simultaneously executed for different parts of the application.

The upgrade procedure, including inputs and outputs where applicable, is summarized here:

1. **Application preparation**. This includes:
   a. Development environment preparation.

      Input: Software development installers, third party component installers

      Output: A fully configured application development environment that corresponds to the application to be upgraded
   b. Upgrade tool preparation.

Input: Upgrade wizard installer, installers for the analysis tools

Output: A fully configured system where the original application can run

c. Obtaining the application resource inventory.

Input: Available information about the specifications and design of the original application

Output: Catalog of documentation useful for upgrade purposes

d. Compilation verification.

Input: The original Visual Basic 6.0 application in a correctly configured system

Output: System where the original application can be compiled, debugged, and executed

e. Project upgrade order definition.

Input: Original source code

Output: An analysis of the application component dependencies based on analysis tools that can be used to plan the upgrade order of the different components

f. Reviewing the upgrade wizard report.

Input: Upgrade wizard report for a partial test upgrade of the original application.

Output: Detection of computing resource problems and estimation of upgrade effort.

2. **Application upgrade**. This includes:

a. Execution of the Visual Basic Upgrade Wizard.

Input: Original Visual Basic 6.0 source code and upgrade system

Output: Initial upgraded application code base in Visual Basic .NET; this code base will likely contain upgrade issues that will later need to be addressed

b. Verifying the progress of the upgrade.

This is a control step that verifies the correct execution of the upgrade wizard.

c. Upgrade wizard execution error correction.

This is a corrective step that can be applied when the upgrade wizard execution experiences problems.

d. Completing the upgrade with manual changes.

Input: Initial upgraded application code base in Visual Basic .NET

Output: An upgraded Visual Basic .NET application that can be compiled

3. **Testing and debugging the upgraded application**. This includes:
   a. Original test cases execution.
      Input: An upgraded Visual Basic .NET application that can be compiled
      Output: List of broken test cases and run-time bugs detected in the application
   b. Fixing run-time errors.
      Input: An upgraded Visual Basic .NET application that can be compiled
      Output: An upgraded Visual Basic .NET application that can be correctly run

The remainder of this chapter details each of these steps.

Application Preparation

The Visual Basic Upgrade Wizard is a tool that can automatically perform complex tasks and help ease the upgrade process. The results obtained with this tool are affected by the preparation applied to the original source code and how much of this application is amenable for automated upgrade to Visual Basic .NET. This section provides guidelines for the preparation of your application and the processing environment to maximize the work done by this tool and have a successful upgrade.

Appropriate preparation of your application will save a considerable amount of work in different areas, including management, development, and testing. Preparation for upgrade processes is especially important because it involves many aspects over a long period of time. Errors in the initial stages of the process can easily propagate and consume an extended amount of work. Additionally, a difficulty discovered after the automated upgrade may require a part of the application to be upgraded again, possibly producing delays in the other stages of the upgrade.

It is recommended that you begin by first preparing the upgrade development environment. It is likely that the existing development environment is already properly set up for building the original source code, so this preparation is relatively inexpensive. Even when problems are identified, it is typically a matter of reinstalling missing components, and correcting these types of errors should not be problematic. Investing the time to correctly prepare the development environment and correcting issues like missing components before you begin will save considerable time and effort later in the upgrade process.

After the upgrade development environment is prepared, work can be concentrated on preparing the source code itself. This includes examining the source code, verifying the compilation, and identifying common upgrade issues.

As with any complex procedure, it is beneficial to document all the important steps in an upgrade procedure log, including the products of each upgrade step, component dependency relationships, and upgrade order. This document will provide you

with a clear view of the process, allowing you to plan and measure the effort required for the future steps of the upgrade process. It will also be useful as a base for future upgrade projects. Figure 5.1 illustrates a sample upgrade procedure log.

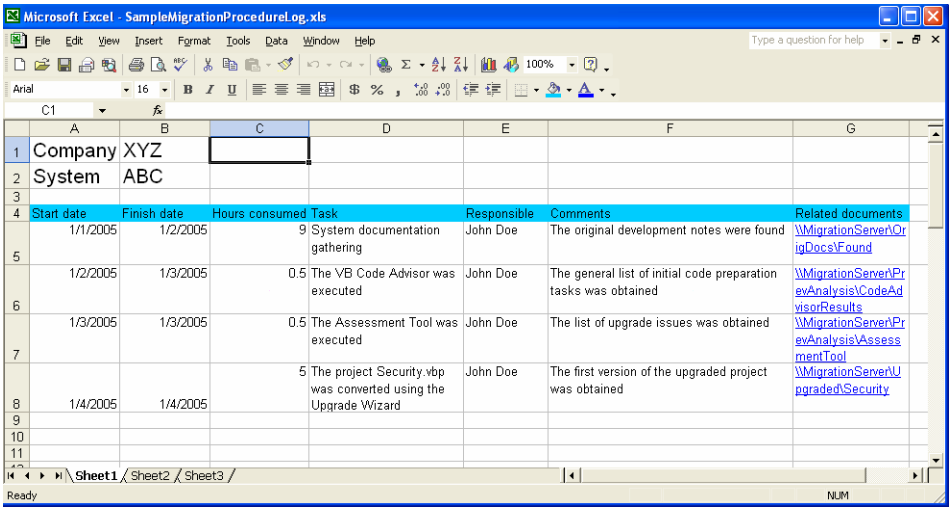

Figure 5.1
Sample upgrade procedure log

The preparation steps are detailed in the following sections.

Development Environment Preparation

Careful preparation of your development environment before beginning an upgrade project will help to ensure success. It is recommended that you execute the upgrade process on a computer with the same development environment that was used to create the original application. This will facilitate the analysis of the original application and will allow for the execution of the initial preparation tests.

Typically, computers used for application development have physical resources that are extremely useful, if not essential, for the execution of the upgrade wizard.

There are two main environment aspect types that affect the upgrade process. The first aspect type, system resources, affects the speed of the upgrade process. The second environment aspect type, external dependencies, affects the normal execution and termination of the upgrade wizard. It will produce exceptions whenever necessary components are not present on the computer where the upgrade process is executed.

System Resources

The upgrade wizard loads all global declarations of the application into memory and keeps them there throughout the upgrade process; this is necessary to resolve the types of members that are accessed from other modules. When a file is being upgraded, it is loaded into memory, and all necessary transformations are applied to it. Upon completion, the result is written to disk and the information about the module's local declarations is deleted from the upgrade wizard data structures.

The upgrade wizard is not an I/O intensive application; however, it makes demanding usage of memory and processor resources for the storage of language structures and the execution of transformation rules. A Visual Basic 6.0 application with 20,000 lines of code will have a peak memory usage between 160 MB to 200 MB. The amount of memory available for the upgrade process will dramatically affect its speed. The minimum memory size recommended for a computer on which the upgrade wizard will be applied is 512 MB. The amount of memory should grow according to the size and complexity of the application. Visual Basic 6.0 applications with more than 100,000 lines of code will require a minimum of 1 GB of memory to be upgraded. If the size of the application is millions of lines of code, a computer with 2 to 3 GB of memory should be used. It may also be convenient to separate the application to upgrade it in more manageable pieces.

With respect to the CPU recommendation, an Intel Pentium 4 class processor or equivalent can process an average size application in a range of hours, assuming the system has enough RAM.

The system's hard drive should have enough free space to store at least three times the size of the original application. This is necessary to generate all temporary files and the final result of the upgrade.

External Dependencies

Typical Visual Basic 6.0 applications use third-party components. Some applications explicitly reference external DLLs through the project settings while others dynamically reference and create objects based on the components currently installed.

The upgrade of these applications requires special attention to be put on the installation and availability of these external components. All third-party components referenced by the application to be upgraded must be installed and available. If a component that is explicitly referenced in the project is not available in the system, the command-line version of the upgrade tool, Visual Basic Upgrade Tool, will display the following message and stop the execution:

> Exception occurred: Could not load referenced component:
>
> *NameOfTheComponent*.ocx (8.0.0)
>
> You need to install this component before you upgrade the project.

> It is recommended you install Visual Basic 6.0, with all referenced components, and ensure the application compiles and runs before upgrading.

Similarly, if the application is upgraded through the use of the wizard version of the upgrade tool (available in Visual Studio .NET), the upgrade wizard will display the following message and stop the execution:

> Upgrade failed: Exception occurred: Could not load referenced component: *NameOfTheComponent*.ocx (2.0.0)
>
> You need to install this component before you upgrade the project.
>
> It is recommended you install Visual Basic 6.0, with all referenced components, and ensure the application compiles and runs before upgrading.

Additionally, the following information will be shown as part of the project upgrade log.

```
    <Issue
        Type = "Global Error"
        Number = "4002"
    >Could not load referenced component: NameOfTheComponent.ocx (2.0.0) You
need to install this component before you upgrade the project.It is recommended
you install VB6.0, with all referenced components, and ensure the application
compiles and runs before upgrading.</Issue>
```

When installing trial versions of third-party components, some of them will display a license information dialog box each time the component is instantiated. This dialog box will suspend the execution of the host application, such as the upgrade wizard. If trial versions are installed for some of the components, it is recommended that you pay attention to the messages displayed by the upgrade wizard until the preprocessing phase starts so you can respond to all dialog boxes that are displayed by trial version components. After the preprocessing phase starts, the upgrade wizard can be left unattended and will continue the upgrade until the upgraded code base is obtained.

As part of the application preparation, it is also important to make limited tests on the upgrade of third-party components that have a complex user interface. Reduced size applications can be written to instantiate and access the core functionality of these components. After these applications are written, the upgrade wizard can be used; then the code it produces can be evaluated. Special attention should be dedicated to design and run time visualization and behavior of the upgraded components. In this preliminary component test phase, the interaction between the component, the interoperability wrapper, and Visual Basic .NET components will be verified. For more information about interoperability, see Chapter 14, "Interop Between Visual Basic 6 and Visual Basic .NET." It may be useful to identify and isolate problematic elements that will negatively affect the automated upgrade of

the full application. Most of the reasons for these problems are non-standard interfaces or implementations of some components. If one of these components produces run-time errors during the execution of the upgrade wizard, one possible solution is to remove the component and upgrade the rest of the application. The functionality provided by this component can then be implemented again in Visual Basic .NET.

Upgrade Wizard Preparation

Preparation for an upgrade process involves the following three areas that consider different aspects of the original application:

- **Preparation of the upgrade wizard and external aspects of the application**. This area includes the setup of all necessary tools for development and upgrade. For more information, see the "Development Environment Preparation" and "Third-Party Components" sections earlier in this chapter.
- **Evaluation of the application**. In this area of preparation, the application is evaluated to assess its composition and the possible difficulties that may arise when the upgrade process has been applied. The effort and cost estimation will be generated before the actual upgrade. This aspect of preparation is assisted by various tools, including the Visual Basic 6.0 Assessment Tool. For more information about assessment and effort estimations, see Chapter 3, "Assessment and Analysis."
- **Adjustment of the original application**. In this area of preparation, adjustments are made to the original application to facilitate the upgrade process. These changes can be made to increase the automatic work done by the upgrade wizard and to reduce the manual completion tasks that are necessary after the automatic process. These changes are based on the initial assessment and general advice that can be obtained from different sources such as documentation and advisor tools. For more information about application adjustment, see the "Visual Basic 6.0 Assessment Tool" and "Removing Unused Components" sections later in this chapter.

The preparation of internal aspects of the application is essential because it will directly affect the quality of the obtained product and the amount of manual effort required after the automated upgrade. Software tools for the preparation of internal aspects of the application have proven to be extremely useful to assess the size and complexity of the application to be upgraded and to assist the developer in the changes that need to be made. The union of these data provides the foundation for accurate upgrade effort estimations.

The tools that will be discussed in this section provide different types of information that can be used for code assessment, effort estimations, and the adjustment of the original code. These tools extract the application information directly from the source code by parsing the different application modules. Each has a different

analysis focus and provides complementary views of the application and the process of upgrading it. The tools are listed here:

- The Visual Basic 6.0 Upgrade Assessment Tool generates an extensive inventory of the application; it pays special attention to the features that are difficult to upgrade.
- The Visual Basic 6.0 Code Advisor recognizes features in the original application that can be modified to have a smoother transition to Visual Basic .NET.

Although the tools for upgrade preparation make an exhaustive review of the code, the time required to complete the process is a fraction of the full upgrade process. This is mainly because the tools analyze the code fragments and identify certain patterns, but the tools do not apply transformations. These preparation tools are provided as a way to quickly estimate and improve the results that will be obtained after the upgrade process is applied.

Visual Basic 6.0 Upgrade Assessment Tool

The Visual Basic 6.0 Upgrade Assessment Tool analyzes Visual Basic 6.0 source code and generates extensive information about an application. For more information about the types of information gathered by the tool, see Chapter 3, "Assessment and Analysis."

The tool can analyze multiple project files at the same time. This allows a user to select them individually, by groups, or by searching all .vbp files in a directory and its subdirectories. The tool has a weights file that can be used to assign a complexity value to each upgrade issue and generate customizable effort estimation data.

The assessment tool generates different types of reports with application statistics and upgrade issues that can be used to estimate the upgrade effort. Most, but not all, of the issues that require manual adjustments are identified. Among the statistics generated, there are data related to control usage, accesses to class members, and library occurrences.

An additional source of information that is useful for estimation tasks is the Upgrade Issues table. This report contains data regarding the usage of functionality that has proven to be difficult to upgrade. Each issue contained in the table has an associated complexity level that is a measure of the upgrade difficulty and the manual work required. Some of the issues that can be detected and included in this table are COM+ functionality, Visual Basic 6.0 performance features for games, and ActiveX documents.

For more information about the assessment tool, see Chapter 3, "Assessment and Analysis."

Visual Basic 6.0 Code Advisor

The Visual Basic 6.0 Code Advisor is a Visual Basic 6.0 add-in that can be used to review the source code of an application. This tool helps the developer to ensure that the application meets predetermined coding standards. The coding standards are based on best practices developed by Microsoft to produce robust and easy-to-maintain code.

The issues identified by this tool are also an alert sign to the developer that is upgrading an application to Visual Basic .NET. These issues can be corrected in the original source code to have a smoother transition. The code advisor tool will not find every upgrade issue, but it is designed to help locate the most common issues and speed up the upgrade process. The issues detected by this tool can be extended using regular expressions.

The following list shows a sample of the predetermined rules that are detected by this tool:

- **Late binding of variant object**. An object is late-bound when it is assigned to a variable declared to be of type **Object**. Objects of this type can hold references to any object. When the Visual Basic Upgrade Wizard upgrades a variable whose type is **Object**, it cannot identify the specific type of the variable, and none of the accesses to the corresponding class members are upgraded. This problem can be fixed by using specific types in the declaration of all variables. This is enforced using the directive **Option Strict On**. The Visual Basic 6.0 Code Advisor detects undeclared variables and suggests the developer use **Option Strict On**.
- **Missing option explicit**. Leaving this option undeclared allows the developer to use undeclared variables in the source code. Undeclared variables default to type **Variant**, which can hold a value of any type. Such variables result in the same upgrade problems as mentioned in the preceding bullet: the upgrade wizard cannot determine the correct property to use when your code uses the default property of an object declared as **Variant**.
- **Soft binding of Form or Control**. Variables declared as **Form** or **Control** can cause problems when upgrading. In Visual Basic 6.0, these generic classes can be used with properties and methods that are defined for particular forms and controls. **Form** and **Control** variables are supported in Visual Basic .NET, but accessing properties that are specific to a particular type of form or control is not supported.
- **Soft binding using ActiveForm and ActiveControl**. This usage pattern has the same upgrade problems as stated in the preceding bullet.
- **String functions that return Variant**. Visual Basic 6.0 provides a group of string functions that have two versions, one that returns a **Variant**, and another that returns a **String** (for example, **Left** and **Left$**). **Variants** can contain null values; assigning a null value to a variable that expects a string may cause a run-time

error that is difficult to debug. In Visual Basic .NET, only the **String** version of these functions is supported.

- **OLE control not upgraded**. There is no equivalent for the **OLE Container** control in Visual Basic .NET. When you attempt to upgrade a form that contains an **OLE Container** control, any code for the **OLE Container** control will not be upgraded. Instead, the control is replaced by a **Label** control to highlight the fact that the **OLE Container** cannot be automatically upgraded.
- **No Line control in Visual Basic .NET**. The **Line** control has no direct equivalent in Visual Basic .NET. When you attempt to upgrade a form that contains a horizontal or vertical **Line** control, it will be replaced by a **Label** control. Non-vertical and non-horizontal lines are not upgraded at all and must be manually replaced using graphics function calls in Visual Basic .NET.
- **Property/Method/Event not upgraded**. The referenced Visual Basic 6.0 property, method, or event has no direct equivalent in Visual Basic .NET. When the application is upgraded, the corresponding line of code will be copied unmodified to the target code. This will cause a compilation error in Visual Basic .NET.

For more information about the Visual Basic 6.0 Code Advisor, see "Visual Basic 6.0 Code Advisor" in the Microsoft Visual Basic Developer Center on MSDN.

Removing Unused Components

During the lifetime of the original application, it is likely to be modified several times, potentially changing the dependences between the different components. It is also possible that some part of a component's code is no longer used by the rest of the application.

For clarity and manageability, components or code fragments that are no longer used in the application should be removed. From the upgrade point of view, it is also important to remove unnecessary application elements before starting the process. This reduces the upgrade issues that will be generated after the automated upgrade and also decreases the quantity of code that needs to be upgraded.

The assessment tool can be used to identify unused components. The file dependence graph reports the relationships between application files. Files that are never used can be identified by reviewing this report. Third-party components and external DLL function references should also be examined to identify additional unused elements that can be removed.

Obtaining the Application Resource Inventory

The upgrade of an application from one language to another involves many tasks with different levels of complexity and requirements. The transition from a programming language and a development environment to another language that is hosted in a different environment is an important part of the upgrade, but there are

additional tasks that have to be done to take full advantage of the new resources and continue improving the application along its new life cycle in the new platform.

The initial source code inventory made with the assessment tool is essential for the upgrade effort estimation. When upgrading earlier versions of applications, it is also important to have an inventory of resources such as documentation, design diagrams, test cases, and specifications. These resources are helpful in the final stages of the upgrade procedure. The original test cases are the basis for testing the upgraded application. It is important to evaluation design diagrams and similar documentation to make decisions about the target architecture. This information also helps you to have a high level view of the application and identify parts or functionality that require special attention during the upgrade. Database and general design documentation can also be useful when adjusting the upgraded application. For example, **Variant** variables that are used to access database fields can be re-declared with more specific types using field data type information.

Compilation Verification

The upgrade wizard requires source code to be syntactically correct. This can be verified by compiling the original application before the execution of the upgrade wizard. All compilation errors produced in the original application must be corrected to provide an appropriate input to the upgrade wizard. The compilation of the original application is also useful to verify that all the components and user-defined source code referenced by the application are available in the system. In the case of project groups, it is important to confirm that the entire group and individual projects can be correctly compiled. The upgrade of projects groups is explained in detail in the next section.

When you try to open a project that references an unavailable component, the following error message appears:

> Errors during load. Refer to 'C:*ReferenceSample**Form1*.log' for details

You can find the cause of this error message by opening the specified file. A sample error line might look like the following:

> Line 13: Class MSComctlLib.TreeView of control TreeView1 was not a loaded control class.

To fix this compilation error and allow the upgrade wizard to upgrade this application, it is necessary to install the component indicated in the error description. In this example, it would be necessary to install the **MSComctlLib.TreeView** component.

It is also recommended that you perform a test run of the original application on the computer where the upgrade will take place. This test run verifies that all components are available and that the application is producing the expected results. It is important to take into consideration that all the run-time errors that exist in the

original application are likely to remain in the target application. Knowing the errors contained in the original application provides a basis for testing and improving the target application. For more information, see the "Testing and Debugging the Upgraded Application" section later in this chapter.

Project Upgrade Order Definition

The upgrade order of the files contained in a Visual Basic 6.0 application is determined by the file order specified in the corresponding application project file. This order also determines which result files are written first in the target application directory. This information can be useful when testing and isolating problems in the upgrade of a specific application. A Visual Basic 6.0 application can be composed of different projects that are independently accessed and managed. The projects can also be organized in a project group file. For information about upgrading an application built using a project group file, see the "Project Group Upgrade" section later in this chapter.

A project can contain source code files that correspond to the different tiers of the application. These tiers can also be distributed in different projects. For upgrade purposes, it is important to clearly understand the correspondence between the application tiers and the project elements that compose it. The project and component upgrade order is determined by the association between components and application tiers and the component dependencies. Another factor to be considered when establishing the processing order is the chosen upgrade strategy. A vertical or horizontal upgrade strategy can be selected based on the criteria explained in Chapter 1, "Introduction," and Chapter 4, "Common Application Types."

The initial approach recommended for the upgrade of an application is to process the core application components first. These components provide services and functionality to other elements of the application. An alternative approach is to leave these basic components for the final stages of the upgrade. The other components can access them using interoperability techniques to test intermediate upgrade results.

Determine All Dependencies

When one application component uses another component, the first component is said to be dependant on the second. There are different types of usage that require different ways to identify such dependencies. Usage can appear in the form of access to a class member, invocation of a method, declaration of a variable with a particular type or the inclusion of an embedded control. Dependencies also exist at the project level and can be established through project references. The dependency relationship is transitive, which means that if a component A is dependant on a component B and B is dependant on a component C, then A is also dependant on C. The terminology for this is that B is directly dependant on C and A is indirectly dependant on C,

because there is an immediate dependency between B and C and an intermediate dependency between A and C.

As previously mentioned, dependencies can be identified based on how one component references or accesses another. Dependencies can be easily identified for an application with few components. However, as the number of components and their interactions increase, obtaining a complete view of the component dependencies becomes more difficult. In most cases, considering the dependencies at the project level will be sufficient for a successful upgrade; however, if there are complex and difficult-to-upgrade components inside the projects, dependency considerations should also be applied to them.

The knowledge about the dependencies between components and projects is essential for the upgrade engineer. This information will be used in conjunction with other key considerations such as strategic priorities, business value, and available resources to focus the upgrade efforts and maximize the results obtained.

The assessment tool generates a file dependency graph that is a useful tool to identify components that have no dependencies. Figure 5.2 illustrates this report.

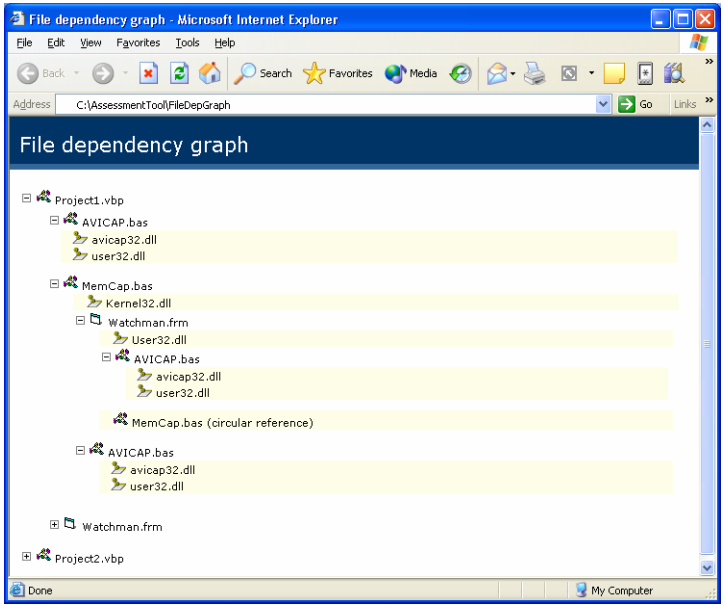

Figure 5.2
A sample file dependency graph

This report can be used to identify independent components. In this example, the first project included in the report, Project1.vbp, includes three files: AVICAP.bas, MemCap.bas, and Watchman.frm. These files have different dependency relationships identified in the report. The files Watchman.frm and AVICAP.bas are included

in the folder labeled MemCap.bas, which means that MemCap.bas depends on the other files. Independent user-defined files depend only on third-party components or system components, such as user32.dll. In this example report, AVICAP.bas is an independent component because it does not depend on other user-defined files.

The first decision to make regarding component upgrade is whether the independent components are going to be upgraded to Visual Basic .NET or are they going to be accessed through interoperability. This decision should be based on a cost-benefit analysis, as discussed in Chapter 2, "Practices for Successful Upgrades." If the components are not difficult to upgrade and they can benefit from the new .NET features to produce a better target architecture, the components should be upgraded. However, if the component is complex, or the benefit of upgrade is not justified by the cost, it may be best to leave the component in Visual Basic 6.0 and use it in the new application through interoperability techniques.

After the independent components are processed, you must treat the components that are directly dependant on the components just upgraded in the same way. This procedure needs to be repeated for each level of dependant components until the entire application is processed. This method is a high-level view of the application upgrade processing order; it can be further refined considering aspects such as the testing phase and process efficiency and risks: these are discussed later in this chapter.

If dependency relationships are followed in a systematic way, testing the application in a progressive way, that is, with an increasing quantity of upgraded components, will be facilitated. Different testing milestones can be defined in the upgrade plan according to the dependency graph.

There is an additional consideration that needs to be taken into account when defining the upgrade plan and component upgrade order: striking a balance between the pieces of the application that are upgraded and the testing that is applied to the different components.

One possible strategy is to upgrade a group of the application components and perform unit testing of the corresponding application functionality at the end of the upgrade. After the entire application is upgraded, system testing can begin. One advantage of this approach is that there is a greater degree of parallelism; several components can be upgraded simultaneously with almost no coordination. The main disadvantage is the difficulty in isolating problems during the testing phase. This strategy is recommended for the upgrade of applications with low or mid complexity. Upgrade complexity is determined by the language features and the size of the application to be upgraded. For more information, see Chapter 3, "Assessment and Analysis."

Another possible strategy is to upgrade the components by strictly following the order of the dependency graph; this implies testing the application functionality in an incremental fashion after each component is upgraded. One of the advantages of this strategy is that problems can be easily isolated. However, an important disadvantage is that additional planning and coordination will be necessary during the project execution. Furthermore, less parallelism is possible in this approach. This strategy is recommended for the upgrade of applications with high complexity.

As stated in the introductory text of the "Application Preparation" section earlier in this chapter, it is important to document all important decisions in an upgrade project log. This document should include the project upgrade order and the information that was used to determine it.

Reviewing the Upgrade Wizard Report

As part of the preparation stage, it is important to perform a partial test upgrade of the application to determine or detect each of the following:

- **Upgrade process speed**. The time consumed by the upgrade of a part of the application can provide an idea of the time required to upgrade the entire application. This information is useful for the upgrade procedure planning.
- **System resources**. The available system resources, especially the amount of system memory, can be evaluated using a part of the application. The consumption of system resources is affected by the application complexity; if system resources show signs of excessive usage, such as extreme memory swapping (referred to as *thrashing*), the resources should be increased.
- **Setup problems**. Common setup problems, such as unavailable third-party components or out-of-date source code, can be detected in a partial test upgrade.
- **Other common errors**. Source code issues can be identified and corrected before the full upgrade of the application.

The upgrade wizard report generated during the test upgrade can be examined to get an idea of the amount of manual work that will be necessary after the automated process. It will provide a vision to plan for the upgrade and prepare the necessary resources. This report can also be filtered and analyzed as described in Chapter 3, "Assessment and Analysis."

The upgrade wizard produces a report that contains errors, warnings, and issues that are found during the project upgrade. When a project is upgraded, the upgrade wizard upgrades most of the code to Visual Basic .NET, but some elements cannot be automatically transformed. These language elements will require manual modifications after the automated process has finished. The following code example shows one of the most common scenarios that will produce a warning when upgraded using the upgrade wizard.

```
Sub SetObjLabel(obj As Variant)
    obj.Caption = "Initial text"
End Sub
Private Sub Form_Load()
    SetObjLabel Label1
End Sub
```

In this code example, one of the **Form**'s controls is sent to a subroutine that sets some initial values. When applied, the upgrade wizard produces the following code.

```
Sub SetObjLabel(ByRef obj As Object)
    'UPGRADE_WARNING: Couldn't resolve default property of object obj.Caption.
    obj.Caption = "Initial text"
End Sub
Private Sub Form1_Load(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    SetObjLabel(Label1)
End Sub
```

As this shows, much of the code is automatically upgraded to Visual Basic .NET. However, the line in bold font represents upgrade issues identified by the upgrade wizard. The comment indicates what the issue is, and the code line is code that cannot be automatically upgraded by the upgrade wizard. The code is copied unchanged from the original code base to the upgraded code.

The problem results from late binding of the variable object to a **Label** object. As stated earlier in the chapter, late binding cannot be resolved by the upgrade wizard; it cannot determine the specific object type that is being accessed when the **Caption** property is referenced. As a result, it cannot find an equivalent expression for that property. If this code is built and executed, a run-time exception will be generated. In most cases, the solution for this problem is to use a more specific data type for the affected variable and replace the accessed members with the equivalent .NET members. This is demonstrated in the following Visual Basic .NET code example.

```
Sub SetObjLabel(ByRef obj as System.Windows.Forms.Label)
    obj.Text = "Initial text"
End Sub
```

Note that the UPGRADE_ISSUE comment can be removed after the correction is made.

The subroutine **SetObjLabel** has a specific purpose that has been accidentally changed with the correction to the code. In the original application, **SetObjLabel** can receive any kind of control that has a **Caption** property. In the new version, the subroutine can only receive **Label** controls. This is an important consideration that can easily be overlooked. An alternative correction that will maintain the behavior of

the original code is to change the type of the parameter to **System.Windows.Forms.Control**, which has a **Text** property and is the ancestor of all .NET controls. This will restore the possibility of accepting any control, as shown here.

```
Sub SetObjLabel(ByRef obj as System.Windows.Forms.Control)
    obj.Text = "Initial text"
End Sub
```

Note: The ArtinSoft Visual Basic Upgrade Wizard Companion has an advanced mechanism for automatic type deduction of undeclared variables. This mechanism performs a static analysis of the source code based on the usage pattern of most of the Visual Basic 6.0 language elements and the information they provide regarding the most probable data type of the involved symbols. The relationships between symbols are also analyzed to obtain a precise conclusion about the data type of variables, parameters, or other language elements that are late bound. Using the upgrade wizard companion, the previous code example would have been automatically upgraded to the code example shown here.

```
Sub SetObjLabel(ByRef obj as System.Windows.Forms.Label)
    obj.Text = "Initial text"
End Sub
```

Depending on the additional invocations of **SetObjLabel**, the object parameter would have been declared as **System.Windows.Forms.Control** or other control type.

For more information about the upgrade wizard companion and additional upgrade services, go to the ArtinSoft Web site.

The following list presents the most common upgrade issues and how they can be solved after the automated upgrade has taken place:

- **Property <object>.<property> was not upgraded**. This warning is generated when a property cannot be automatically upgraded. To correct this problem, it is necessary to use a workaround that replaces the original functionality. It may require the reimplementation of certain application functionality.
- **<objecttype> object <object> was not upgraded**. This occurs when objects of unsupported classes are declared and used. Additional development work may be necessary to fix this problem. For more information, see Chapters 7 through 10.
- **Use of Null/IsNull detected Null is not supported in Visual Basic .NET**. **Null** is upgraded to **System.DBNull.Value** and **IsNull** is changed to **IsDBNull**. This value and function do not have the same behavior as in Visual Basic 6.0. Also, **Null** propagation is no longer supported in Visual Basic .NET. The application logic needs to be reviewed to change the usage of **Null**.

- **<functionname> has a new behavior**. Some functions have different behaviors in Visual Basic .NET. This requires a detailed review of the code to ensure that the application logic has not been altered.
- **<object> event <variable>.<event> was not upgraded**. Some events cannot be upgraded to Visual Basic .NET. The corresponding methods must be called if necessary.
- **Could not resolve default property of object '<objectname>'**. This is one of the most common issues you will encounter. It occurs whenever a late-bound variable is set to a value. This is the warning discussed in the previous code example.
- **Could not load referenced component <reference>**. This message is related to the setup of the application that is being upgraded. It will be included whenever a reference, such as an ActiveX control or class library, cannot be found. To correct this problem, verify that all necessary components are installed on the computer. This issue can be avoided by installing Visual Basic 6.0, with all referenced components, and ensuring the application compiles and runs before beginning the upgrade.

For a description of errors, warnings, and issues, see the left pane of "Visual Basic 6.0 Upgrading Reference" on MSDN.

> **For Visual Basic 2005:**
>
> The upgrade wizard included in Visual Studio 2005 has significant differences from the previous version. These differences include additional support for more Visual Basic 6.0 features and new upgrade messages among others. The changes in the upgrade wizard result in fewer errors, warnings and issues (EWIs) (since more features are now supported) and reorganization of the distribution of effort required to address identified EWIs.

Application Upgrade

The Visual Basic Upgrade Wizard is the primary tool for application upgrade. This section describes the execution options that are available and how to get the most out of the wizard.

The automated application upgrade is the middle stage in the complete upgrade process. Before the upgrade, it is necessary to prepare and set up all necessary components. Usually, some manual adjustments are necessary after the automated upgrade to reach a compileable application with functional equivalence. The enhancement of the target application can begin after that.

There are different versions of the upgrade wizard that include different features to further automate the upgrade result. Most of the upgrade features discussed in this book apply to the Visual Studio .NET (also known as Visual Studio .NET 2003)

version of the upgrade wizard. The version of the upgrade wizard included in Visual Studio 2005 has improved performance and reliability, as well as additional upgrade capabilities, including support for additional ActiveX controls and improved support for the upgrade of COM+ functionality. The ArtinSoft Visual Basic Upgrade Wizard Companion as additional upgrade mechanisms that support context aware upgrade of enumerations, code improvement capabilities, automatic type deduction for **Variant** and **Object** variables, and other advancements. For more information about this upgrade tool and additional upgrade tools and services, go to the ArtinSoft Web site.

This section focuses on the Visual Studio .NET upgrade wizard, but when it is applicable, the features of other versions of the upgrade wizard will be highlighted. For more detailed information about the upgrade wizard and the automated upgrade process, see Chapter 6, "Understanding the Visual Basic Upgrade Wizard."

Execution of the Visual Basic Upgrade Wizard

The upgrade wizard can be used in two ways: it can be used as either a command line tool or as a wizard available through the Visual Studio .NET IDE. The wizard version is automatically invoked whenever you attempt to open a Visual Basic 6.0 project file in Visual Studio .NET. Both of these approaches are discussed in this section.

When performing an upgrade, it is recommended that you install Visual Basic 6.0 and Visual Basic .NET on the computer on which you are conducting the upgrade; this allows you to test the original and target application and installs basic components that are required by the upgrade wizard.

To facilitate the discussion, a sample Visual Basic 6.0 project will be used as an example for the step of applying the upgrade wizard. The sample project is named SumApp. When the application runs, the result of the addition of the two input fields is displayed in a box after the user clicks the command button. The application verifies that the user has entered appropriate numeric values in the boxes before it executes the addition.

Accessing the Upgrade Wizard from Visual Studio .NET

This section documents upgrading the sample application using the upgrade wizard in the Visual Studio .NET IDE. The following steps must be executed to upgrade the project:

1. Open Visual Studio .NET.
2. On the **File** menu, click **Open**, and then click **Project**. This displays the **Open Project** dialog box. Navigate to the location of the Visual Basic 6.0 project file, and then click the desired project file. For our example, navigate to the location in which you saved the SumApp project, click **SumApp.vbp**, and then click **Open**.

3. Read the welcome page, and then click **Next**.
4. On the **Choose a project type** page, the appropriate option is selected by default. Click **Next**.

   Note: This page sets the options for the upgrade. There are two available choices to select the type of application that will be generated for the target project. EXE projects upgrade to projects that produce an executable file, and DLL/custom control projects upgrade to projects that produce DLLs. For standard EXE and DLL projects, the upgrade type is determined automatically. For other types, such as EXE server projects that can act like a hybrid of an EXE and a DLL, these projects can be upgraded to either Visual Basic .NET EXE or DLL projects. For this example, the sample application is a standard EXE, so the **EXE** option is automatically selected, and the **DLL/custom projects upgrade** option is unavailable.

5. On the **Specify a Location for Your New Project** page, the wizard suggests the target directory to be called *projectname*.NET, where *projectname* is your original project's name. In our example, the suggested name is SumApp.NET. Click **Next**.

   Note: If the destination directory already exists and contains files, a warning is displayed that indicates that the output directory is not empty. The wizard will ask you to choose another directory. If he directory does not already exist, a Warning displays that asks for confirmation to create the directory. Click **Yes** to create the directory or **No** to specify a different target directory. For our example, the SumApp.NET folder should not yet exist, so click **Yes** to create the target directory and proceed to the next page of the wizard.

6. The **Ready to Upgrade** page of the wizard informs you that it is ready to perform the upgrade. Click **Next** to start the upgrade.

The next page of the wizard displays information about the upgrade process status. A **Status** box shows the task the upgrade wizard is currently executing and a progress bar indicates an approximation of the time left for the task. The upgrade can take anywhere from a minute (for a small project such as the sample application) to several hours (for a project with hundreds of forms and classes). When the processing is complete, the new upgraded project opens in Visual Studio .NET. Figure 5.3 illustrates a sample.

The upgraded project has a form module, **Form1**, that corresponds to the Visual Basic 6.0 form module in the original project. In Solution Explorer (in the top right corner of the IDE illustrated Figure 5.3), double-click **Form1.vb** to open the upgraded form. The new form should appear similar to the original. Figure 5.4 illustrates this.

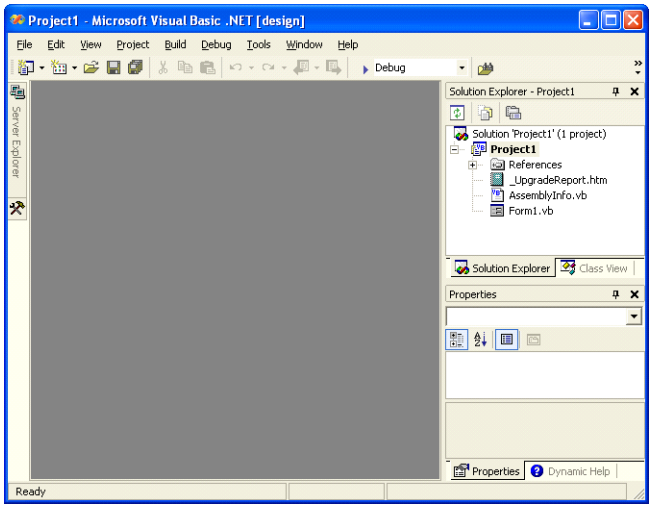

Figure 5.3
The sample application after applying the Visual Basic Upgrade Wizard

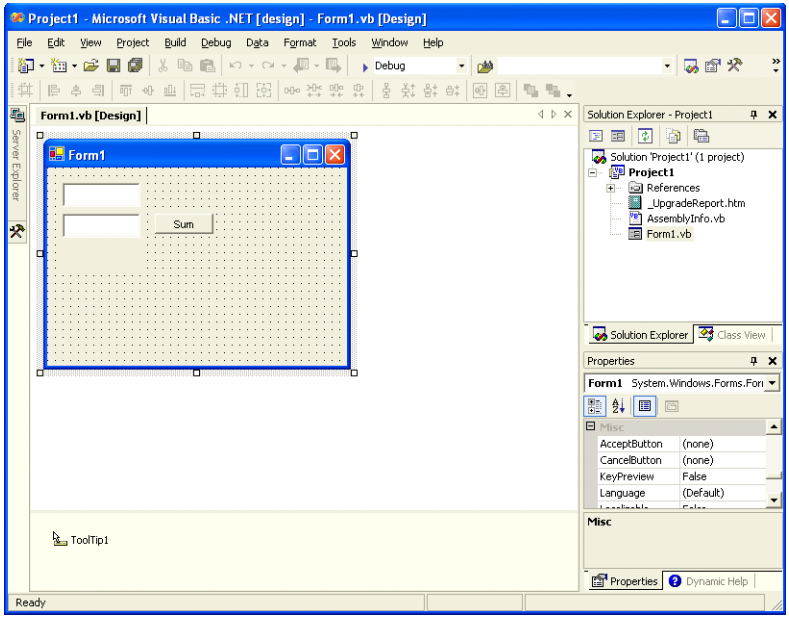

Figure 5.4
The upgraded form

To run the upgraded application, press F5. The first time an upgraded project is built or run, you will be prompted to save the solution file. The .NET solution corresponds to the root of the tree displayed in Solution Explorer. A solution file is similar to a group file in Visual Basic 6.0. The upgraded project is contained in this solution file. Accept the suggested name for the solution, and then press **Enter** to save it. The project is then automatically built and executed. **Form1** is displayed and waits for user input. Figure 5.5 illustrates the result obtained for sample input data.

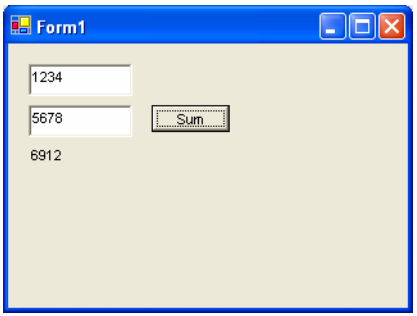

Figure 5.5
Execution of the upgraded application

Accessing the Upgrade Tool from the Command Line

The automated upgrade tool can also be applied from the command-line using the VBUpgrade.exe upgrade program. This program uses the same upgrade engine as the upgrade wizard accessed in Visual Studio .NET. Assuming the default installation directory is used, this program can be found in the following path:

drive:\Program Files\Microsoft Visual Studio .NET *version*\Vb7\VBUpgrade\

In this path, *drive* indicates the drive letter on which the installation resides and *version* indicates the Visual Studio .NET version. For example, this guide was written for the Visual Studio .NET 2003 IDE, so the full path would be the following:

drive:\Program Files\Microsoft Visual Studio .NET 2003\Vb7\VBUpgrade\

If you installed the upgrade tool to a location other than the default directory, or if you are otherwise unsure about where the program is located, you can use Windows Explorer to locate it. The remaining text for this example assumes you have the directory of this utility in your command search path.

The command-line upgrade tool produces the same results as the upgrade wizard. To upgrade a project, the project file name and the destination directory need to be indicated as arguments to the command. If the destination directory does not exist, the upgrade tool automatically creates it. For example, the following statement upgrades our sample project, stored in C:\SumApp\SumApp.vbp to the C:\SumApp\SumApp.NET directory.

```
vbupgrade c:\SumApp\SumApp.vbp /Out c:\SumApp\SumApp.NET
```

Note that you should replace the directory names with the appropriate locations.

The command-line version of the upgrade tool does not have wizard pages. Furthermore, this tool does not delete any files in the destination directory. If the destination directory already exists and contains files, the command-line tool stops running and returns an error.

It is recommended to use the command-line tool when the available memory on the upgrade system is limited. Using this tool saves the system resources needed by Visual Studio .NET when the upgrade tool runs. Depending on the system configuration, Visual Studio .NET consumes approximately 15 MB of memory.

Command Line Options

The command-line version of the upgrade tool has many options that you can specify when performing an upgrade. You can obtain a complete list of the available options for this tool by entering the following command at the command prompt: vbupgrade /?

This command displays a welcome message followed by the available option switches and their meanings.

```
Microsoft (R) Visual Basic.NET Upgrade Tool Version 7.00.9238.0
Copyright (C) Microsoft Corp 2000-2001.
Portions copyright ArtinSoft S.A.
All rights reserved.
Usage: VBUpgrade <filename> [/Out <directory>] [/NoLog • /LogFile <filename>] [/
Verbose] [/GenerateInterfaces]
/?                    Display this message
/Out                    Target directory (default is ".\OutDir")
/Verbose                Outputs status and results
/NoLog                  Don't write a log file
/LogFile                Log file name (default is
                        "<ProjectFileName>.log")
/GenerateInterfaces    Generates interfaces for public classes
```

The tool has a verbose mode that allows a user to identify exactly what the tool is currently doing. This is useful as a way to determine the upgrade progress of large applications. If you use this tool while upgrading the sample application, the tool returns the following.

```
C:\Program Files\Microsoft Visual Studio .NET 2003\Vb7\VBUpgrade>vbupgrade
C:\SumApp.vbp /Verbose
Microsoft (R) Visual Basic.NET Upgrade Tool Version 7.00.9238.0
Copyright (C) Microsoft Corp 2000-2001.
Portions copyright ArtinSoft S.A.
All rights reserved.

Initializing...
Parsing Form1.frm...
```

```
Loading User Type Library Form1
Pre-processing Form1...
Upgrading Form1...
Upgrading Form1.Form1
Upgrading Form1.Command1
Upgrading Form1.Text2
Upgrading Form1.Text1
Upgrading Form1.Label1
Upgrading Form1.Command1_Click
Writing Form1.vb...
Writing Form1.resx...
Writing project file Project1.vbproj...
Writing project file Project1.vbproj.user...
Writing project file _UpgradeReport.htm...
Writing AssemblyInfo.vb...
Writing project file Project1.vbproj...
Writing project file Project1.vbproj.user...
Writing project file _UpgradeReport.htm...
```

Note that the preceding example assumes the default installation location for Visual Studio .NET. In this sample output, the user can see the different stages that are involved in the upgrade of a Visual Basic 6.0 project. It is important to note that some of the output lines may take some time to appear, sometimes several minutes, while the tool is executing. The state of the upgrade process can be checked using the Windows Task Manager, as discussed in the next section.

Project Group Upgrade

With Visual Basic 6.0, developers can group different projects into a single project group file. When a group file is compiled, all projects specified in the group are compiled together. Furthermore, the debugger can navigate through the code in all projects. This functionality is useful when there are project dependencies in the application. The upgrade wizard does not support project groups, primarily because of the size of the context information that must be simultaneously managed for all the projects.

In Visual Basic .NET, the equivalent for a project group is a solution. A solution can contain projects written in different languages with references between them. The projects are compiled and debugged together just as in a Visual Basic 6.0 project group.

Testing an upgraded application with moderate complexity to high complexity requires at least the same effort as testing the original application. The testing of some of the application functionality can be performed in parallel with the upgrade of other components. These parallel tasks require coordination and the definition of a project upgrade order accordingly. For more information, see the “Testing and Debugging the Upgraded Application” section later in this chapter.

Depending on the strategy used for intermediate result testing, the following project upgrade orders might be followed:

- First upgrade the projects that have no dependencies. If the core functionality is upgraded first, testing can be done at different levels, starting from the core components and including more functionality in each iteration. This approach provides isolation of problems and can be used for upgrade projects with high complexity and that do not require early demonstrations of upgraded functionality.
- First upgrade the projects on which no other projects depend. The components at the top of the dependency hierarchy can be upgraded first and if necessary the application functionality can be tested using interoperability for the remaining components. This approach can produce early demonstrations of upgraded functionality, but it may require additional effort for the creation of interoperability wrappers.

A mixed approach can also be used to obtain a combination of benefits.

The following example shows how to upgrade a project group using the first approach explained. The sample project group has two projects; one of them generates a DLL that contains a class used to perform addition operations. The second project is a front-end program that displays a form that asks the user for input and uses the services provided by the DLL. When the project group is compiled and the generated program is executed, a form displays that asks the user for numerical values to be added. If an incorrect value is entered, an error is raised in the DLL and the form displays the error description. The sample application can be found on the companion CD.

To upgrade this project group when the priority is placed on the core components, the following steps must be performed:

1. Open the Group1.vbg file group in Visual Basic 6.0.
2. Using the upgrade wizard, upgrade all projects that have no dependencies. Each project must be processed individually, as follows:
   a. In Visual Basic 6.0, observe the references associated with Project1. In the projects window, select **Project1**. By default, this is at the upper-right corner of the IDE. In the **Project** menu select **References**. The **References - Project1.vbp** dialog box displays.
   b. The **References – Project1.vbp** dialog box shows that Project1 depends on Utilities. The Utilities project has no further dependencies, so it should be upgraded first. For large applications, this kind of dependency analysis can be done with the Assessment Tool File dependency graph, as explained in the "Determine All Dependencies" section earlier in this chapter. Upgrade the Utilities project using the upgrade wizard, as explained in the "Accessing the Upgrade Wizard from Visual Studio .NET" section.

3. After Visual Basic .NET is opened, the dependant project can be upgraded and added to the solution. On the **File** menu, point to **Add Project**, and then click **Existing Project**. This opens the **Add Existing Project** dialog box. Click **Project1.vbp**, and then click **Open** to start the upgrade. Accept the default upgrade options, as described in the "Accessing the Upgrade Wizard from Visual Studio .NET" section.
4. Replace the newly upgraded project references with the corresponding .NET project. To do so, apply the following steps:
   a. In Solution Explorer, open the Project1 project folder. Open the **References** folder. Right-click **Utilities**, and then click **Remove**. This removes the reference to the old version of Utilities.
   b. To add a new reference to the new Visual Basic .NET version of the Utilities project, right-click the References folder, and then click **Add Reference**. The **Add Reference** dialog box displays. To view the available projects, click the **Projects** tab. To add the reference, double-click **Utilities**, and then click **OK**.

Note: Now that the reference to the original Visual Basic 6.0 version of Utilities has been replaced by the newly upgraded version, there may be differences in its interfaces and data types. These differences will produce compilation or run-time errors in the places where this component is accessed. When Project1 was upgraded in step 3, the upgrade wizard considered the reference to the original version of Utilities and it was treated as an external third-party component. An interoperability wrapper was generated and no transformations were applied to the member declarations. Because of this, the usages of this component inside Project1 will remain the same. The differences can arise when the Utilities library is upgraded. The upgrade wizard will transform some of its interface declarations and therefore the places where the component is used must be updated. These changes include event parameters and upgraded data types.

---

5. Review the accesses to Utilities inside Project1, and updated them if necessary. For our example, the Utilities interface has not changed during the upgrade and there are no modifications needed in Project1. However, with more complex interfaces and data types, additional work will likely be necessary to make the application compile.
6. The solution is now ready to be compiled, run, and tested. Set Project1 as the startup project by right-clicking it in Solution Explorer, and then clicking **Set as StartUp Project**.
7. Now that these two projects are upgraded, you can apply the corresponding tests. After this subset of the full application functionality is tested and corrected, a new level of functionality can be upgraded and tested on a stable base. For our current example, there are no further projects in the group to upgrade, so the process is complete.

Alternatively, this project can be upgraded with initial emphasis on those projects that no other projects in the group are dependent on. To apply this strategy, the following steps must be performed:

1. Open Visual Studio .NET.
2. Upgrade one of the projects that no other projects in the group depend on. For this sample project group, no project depends on Project1, so it will be upgraded first. Follow the procedure in the "Accessing the Upgrade Wizard from Visual Studio .NET" section.
3. Next, upgrade a project that provides services to the project upgraded in step 1. For this example, the only project that is used by Project1 is Utilities. In Visual Basic .NET, click **Add Project** on the **File** menu, and then click **Existing Project**. This opens the **Add Existing Project** dialog box. Click **Utilities**, and then click **Open**. Accept the default upgrade options.
4. Project1 still references the Visual Basic 6.0 version of Utilities. This reference must be deleted and replaced with a reference to the upgraded Visual Basic .NET version of this project. To remove the reference, expand the Project1 references node and remove the reference to Utilities by right-clicking it and then clicking **Remove**.
5. Add a reference to the new Utilities project: Right-click **Project1 References**, and then click **Add Reference**. In the **Add Reference** dialog box, click the **Projects** tab, double-click **Utilities**, and then click **OK**.
6. See step 5 in the previous procedure for additional considerations when replacing the **Utilities** reference from the Visual Basic 6.0 version of the component to the Visual Basic .NET version. The same issues are possible despite a change in the order of component upgrade.
7. The solution is ready to be compiled, run, and tested. Note that after step 2, Project1 may be tested using the non-upgraded version of Utilities through interoperability.

Although the example used to illustrate these two procedures was relatively small, keep in mind that either of these upgrade approaches can be applied to project groups composed of many projects. The approach you choose to apply should be based on careful analysis of the components and their dependencies. A project group containing many components with no dependencies will likely benefit from the application of the first upgrade approach presented. Alternatively, a project group with many components that have no other components dependent on them may benefit from the second approach.

Verifying the Progress of the Upgrade

When the upgrade tool is executed from the command line, two processes are launched:

- **VBUpgrade.exe**. This is the front end of the tool for the command line mode.
- **VBUD.exe**. This is the upgrade engine.

The same upgrade engine is launched when the upgrade wizard is accessed in Visual Studio .NET, but in this case, the only process that will be launched is VBUD.exe.

The Windows Task Manager displays the two upgrade processes launched by the command-line version of the upgrade tool.

The upgrade wizard for the 2003 version of Visual Studio .NET processes source code files one by one and stores each result as a temporary file. After all files are processed in that way, the final Visual Basic .NET source code is written to the output directory. This is the final result delivered to the user. If there was a problem during the upgrade of a file and the upgrade tool has to terminate execution, any previous temporary results will be lost and no final Visual Basic .NET results will be obtained. To avoid this situation, the Visual Basic 2005 version of the upgrade wizard and the Visual Basic Upgrade Wizard Companion from ArtinSoft produce the final upgrade results on a file-by-file basis. After a file is processed, the final source code is written and the target project and solution files are updated. This allows the user to use the results, even if a fatal error occurred during the upgrade of another file.

Fixing Problems with the Upgrade Wizard Execution

This section includes some common problems that may arise when running the upgrade wizard. In general, the upgrade process can be left unattended after all the required information is provided. However, large applications may take several hours; while it is upgrading, different situations that require user attention may arise. The following list presents the most common issues that might occur:

- **Memory swapping**. When the computer that is used for the upgrade of an application does not have sufficient memory resources for the processing of a large application, the operating system tries to provide additional memory to the upgrade process by swapping memory to disk. The swapping back and forth between memory and disk may consume more processing power than the other tasks. In this situation, the system is said to be in a "thrashing" state. In this state, the upgrade process can take a significant amount of additional time to finish. When this occurs, it is recommended that you stop the automated upgrade, increase the system memory, and begin the automated upgrade again. Figure 5.16 illustrates a system that is upgrading a huge application and is in a thrashing state. Notice that all the processor time is consumed by the kernel and almost all the memory is used.

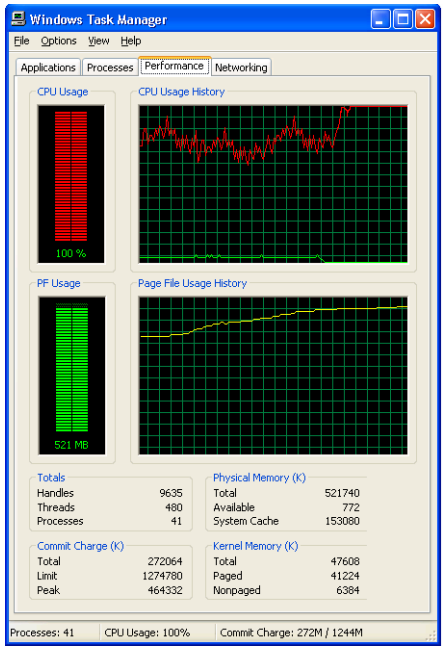

Figure 5.6
The Windows Task Manager Performance display in a typical "thrashing" scenario

- **Incorrect references**. When an application has more than one project to be upgraded, you may choose to upgrade one project at a time. You can make some source code reorganization according to upgrade priority or other factors. If a reference to a user project or a third-party component is accidentally changed to another reference during this reorganization, the upgrade wizard will not be able to obtain all the necessary information about members and data types. This will generate warnings about default property extraction in all places where members of these components are used, because none of these members will be automatically upgraded by the upgrade wizard. If this happens, the corresponding reference and accesses to the elements contained in it will have to be reviewed.
- **Trial version of third-party components**. The upgrade wizard loads all third-party components referenced in the project. Some trial version components display a license dialog box and wait for user input whenever the components is instantiated. This will cause the upgrade process to stop until the dialog box is closed. You should pay attention to these dialog boxes to allow the upgrade process to continue.
- **Upgrade wizard exceptions**. It is an unusual scenario, but when a file is being processed by the upgrade wizard, it may be possible for the tool to encounter a fatal problem. For example, third-party components with a nonstandard interface or other conditions will produce an exception. In this case, the upgrade tool will terminate execution. The first step to solve this problem is to isolate the file and

source code that is producing the exception. After this is done, the source code can be commented. After the problematic code is removed, the upgrade process can be started again. You may want to report these types of exceptions to newsgroups so that other developers can avoid them. Some suggested newsgroups are microsoft.public.dotnet.languages.vb.upgrade and microsoft.public.dotnet.languages.visualbasic.migration.

Being aware of and planning ahead for known exceptions can help you to have a smoother upgrade.

Completing the Upgrade with Manual Changes

After the upgrade wizard is applied to a Visual Basic 6.0 project resulting in Visual Basic .NET code, it is often be necessary to make adjustments to the upgraded code to make it run in the new environment. This section discusses some of the changes you may need to make.

Reaching a Buildable State

The first step in this process is to make the application compile in Visual Studio .NET. Before the application is compiled, you should review the project references to ensure all necessary references are present and correct. In Solution Explorer, open the References folder in the corresponding project and verify that all references are correctly included. If there is a warning icon next to a reference, the most probable reason is that there was a problem generating an interoperability wrapper or including it in the project. The reference must be removed and then manually added again to remove the warning state.

After the project references are verified, you can attempt to build the new version of the application. On the **Build** menu, click **Build Solution**.

The Visual Studio .NET IDE allows developers to identify and navigate through all the compilation errors that were found. Moreover, the Task List shows all problems that were detected; you can double-click on any one of the lines to go to the specific line of code that caused the problem. The problems can be fixed using the information provided in the Task List item and applying the techniques shown in Chapters 7 through 11.

After the application compiles successfully, the design time visualization of the application forms and controls can be verified. Design-time code can be checked by opening each form and verifying that all controls are correctly displayed. If a control is not visible, verify the corresponding reference and look for unsupported features.

After the general verification is complete, the testing and debugging phase can begin.

The following section presents some of the most common issues generated during the automated portion of the upgrade. These issues will be reported by the upgrade wizard and will produce compilation errors in Visual Basic .NET.

Common Compilation Errors

This section presents some of the most common compilation errors caused by unsupported features of Visual Basic 6.0. The upgrade wizard will not upgrade these issues because there is no equivalent functionality in Visual Basic .NET or there is implicit information in the original source code that is explicitly required. For each issue, code examples in original Visual Basic 6.0 and the auto-generated Visual Basic .NET are provided, in addition to a discussion about the modifications needed to correct the problem. The objective of this section is to illustrate the general procedure and techniques necessary to correct problems that can arise after an application has been upgraded to Visual Basic .NET. For information about specific unsupported features, see Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features."

By far, the most common upgrade problem reported by the upgrade wizard in typical Visual Basic 6.0 applications is the default property extraction problem caused by lack of information about the data type of a variable. Depending on the usage of the variable, this problem can cause compilation or run-time errors. For example, the following comment will be generated by the upgrade wizard when this problem arises:

```
'UPGRADE_WARNING: Couldn't resolve default property of object MyObj.MyProperty.
```

To correct this problem, apply the techniques presented in the "Reviewing the Upgrade Wizard Report" section earlier in this chapter.

The following sections present additional upgrade issues that are detected by the upgrade wizard and require user intervention to be corrected.

Problems with As Any Parameter Type

When external DLL functions are accessed in Visual Basic 6.0, it is often necessary to include **As Any** parameters in the corresponding function parameter declaration. This data type is no longer supported in Visual Basic .NET. The problem caused by this change is demonstrated through the following example Visual Basic 6.0 code.

```
Private Declare Function SystemParametersInfo Lib "user32" _
    Alias "SystemParametersInfoA" (ByVal uAction As Integer, ByVal uParam As _
    Integer, ByRef lpvParam As Any, ByVal fuWinIni As Integer) As Integer

Private Function GetKeyBoardDelay() As Integer
    Const SPI_GETKEYBOARDDELAY As Integer = 22
    Dim delayVal As Integer
    Call SystemParametersInfo(SPI_GETKEYBOARDDELAY, 0, delayVal, 0)
    GetKeyBoardDelay = delayVal
End Function
```

Running the upgrade wizard results in the following.

```
'UPGRADE_ISSUE: Declaring a parameter 'As Any' is not supported.
Private Declare Function SystemParametersInfo Lib "user32"  _
    Alias "SystemParametersInfoA"(ByVal uAction As Short, ByVal uParam As Short, _
    ByRef lpvParam As Any, ByVal fuWinIni As Short) As Short

Private Function GetKeyBoardDelay() As Short
    Const SPI_GETKEYBOARDDELAY As Short = 22
    Dim delayVal As Short
    Call SystemParametersInfo(SPI_GETKEYBOARDDELAY, 0, delayVal, 0)
    GetKeyBoardDelay = delayVal
End Function
```

The latter code will produce a compilation error in Visual Basic .NET caused by the usage of an unknown **As Any** data type. To correct this problem, it is necessary to investigate the specification of the external function **SystemParametersInfo**. This function gets the value of system parameters based on the **uAction** parameter and stores the result in the **lpvParam** parameter. The **lpvParam** parameter data type must be changed to a specific data type according to the usage of the function in the source code. In this case, the system parameter to be obtained is the keyboard auto-repeat delay, and the expected result is a short type. To fix this problem, change the **As Any** data type to **As Short**.

For more information about correcting this error, see Chapter 13, "Working with the Windows API."

Changes to Properties of Commonly Used Objects

The following source code compiles and runs perfectly in Visual Basic 6.0; however, when it is upgraded to Visual Basic .NET, a compilation error result because of an unsupported property, as shown here.

```
' List1 is a list box that is embedded in the current Form
List1.AddItem "Hello"
List1.ListIndex = List1.NewIndex
```

After the upgrade wizard is applied to the code, the following code results.

```
' List1 is a list box that is embedded in the current Form
List1.Items.Add("Hello")
' UPGRADE_ISSUE: list box property List1.NewIndex was not upgraded.
List1.SelectedIndex = List1.NewIndex
```

This code results in a compilation error in Visual Basic .NET because the **NewIndex** property is used to retrieve the index of the item most recently added to a **ListBox** control. However, the value of **NewIndex** cannot always be determined in the upgraded code. This error can be corrected by saving the return value of the **Add** method, which is the index of the newly added item. This is demonstrated in the following code example, with changes highlighted in bold.

```
' List1 is a ListBox that is embedded in the current Form
Dim NewIndex As Integer
NewIndex = List1.Items.Add("Hello")
List1.SelectedIndex = NewIndex
```

Discontinued Support for GoSub

The **GoSub** directive available in Visual Basic 6.0 can be used to branch a program's execution to a subroutine or label. The following code example demonstrates the use of **On…GoSub** in Visual Basic 6.0.

```
Function Decide(v As Integer) As String
    On v GoSub case1, case2, case3
case1:
    Decide = "case1"
    Exit Function
case2:
    Decide = "case2"
    Exit Function
case3:
    Decide = "case3"
    Exit Function
End Function
```

Applying the upgrade wizard to this code results in the following.

```
Function Decide(ByRef v As Short) As String
    ' UPGRADE_ISSUE: On...Gosub statement is not supported.
    On v GoSub case1, case2, case3
case1:
    Decide = "case1"
    Exit Function
case2:
    Decide = "case2"
    Exit Function
case3:
    Decide = "case3"
    Exit Function
End Function
```

The **On … GoSub** construct is no longer supported in Visual Basic .NET. The upgraded code produced by the upgrade wizard causes a compilation error. However, the code can corrected by removing the problematic construct, as demonstrated here.

```
Function Decide(ByRef v As Short) As String
    If v = 1 Then
        Decide = "case1"
    ElseIf v = 2 Then
        Decide = "case2"
    ElseIf v = 3 Then
        Decide = "case3"
    End If
End Function
```

Changes in Multiple Document Interface Form (MDI) ActiveForm

In Visual Basic 6.0, an **MDIForm** can refer to its active child form using the **ActiveForm** property. Controls in this form can be accessed as properties. In Visual Basic .NET, **ActiveForm** is a generic type that has no controls.

The following sample source code works fine in Visual Basic 6.0.

```
' It is assumed that MDIForm1 has a child form called Form1. Form1 is currently
' displayed and has a Label control called Label1.
Dim controlStr As String
controlStr = MDIForm1.ActiveForm.Label1.Caption
```

When the upgrade wizard is applied, the following output is produced.

```
Dim controlStr As String
' UPGRADE_ISSUE: Control Label1 could not be resolved because it was within the
' generic namespace ActiveMDIChild.
controlStr = MDIForm1.DefInstance.ActiveMDIChild.Label1.Caption
```

As noted in the **UPGRADE_ISSUE** comment, the **Label1** control can no longer be resolved in the upgraded code because of the change in **ActiveForm**. To correct this problem, it is necessary to make an explicit type cast with the corresponding **Form** class and manually upgrade the accessed control member. This is demonstrated in the following code example.

```
Dim controlStr As String
controlStr = CType(MDIForm1.DefInstance.ActiveMdiChild, Form1).Label1.Text
```

The issues presented in this section have shown the most common manual changes you will have to apply to code produced by the upgrade wizard. However, this is not a comprehensive list. For information about other upgrade issues and the manual corrections you will have to apply to fix them, see Chapters 7 through 13.

Testing and Debugging the Upgraded Application

The objective of the testing and debugging phase is to identify and correct problems that appear when running the application. This section explores testing and debugging of your upgraded application.

Upgrade Report Issues

The following types of issues might be included in the upgrade report and will need to be addressed:

- **Upgrade issues**. These indicate unsupported class members or language elements. For each item identified, the functionality needs to be reviewed, and some of the affected expressions will require manual upgrade to similar .NET elements. Each upgrade should be tested in a limited scenario.
- **Upgrade ToDos**. These identify points in the source code that require user intervention. The instructions included in these items should be followed. Testing of the resultant code verifies that corrections were properly applied.
- **Upgrade warnings**. These items identify source code that will compile but may generate a run-time error. This code needs to be thoroughly tested to identify if and how run-time errors occur. Additional changes to prevent the occurrence of the run-time errors may be necessary.
- **Design errors**. These problems arise when unsupported design-time properties are used. Members whose behaviors have changed are also included here. The functionality that these members provide can be redesigned or changed by similar .NET elements. The code needs to be tested to determine whether the behavior differences have affected the core application functionality.
- **Global warnings**. Global issues such as the lack of deterministic object lifetime because of garbage collection are included here. Most of these warnings are caused by setup issues that can be easily fixed and tested. Other warnings require more work to be fixed, such as manual upgrade in the case of source code lines that are not recognized by the upgrade wizard.
- **Upgrade notes**. These are generated when the source code has been significantly altered or the resultant code has some behavior differences from the original Visual Basic 6.0 version. The code needs to be tested to determine whether the behavior differences have affected the core application functionality.

Unit and system testing should be performed for all the upgraded application functionality. During this phase, the original application specifications and test cases are essential. It may also be necessary to adjust or add test cases for the Visual Basic .NET version of the application.

When upgrading large applications, it may be necessary to test some parts of the application while other parts are still being upgraded. In this situation, the following

points regarding the integration between the upgrade and testing tasks should be taken into consideration:

- Because of component or project upgrade order issues, there may be components that require testing but depend on components that have not yet been upgraded. The original components can be accessed through interoperability to allow the advancement of the testing tasks for those components that have been upgraded. Using this approach allows work to be continued in parallel, and less coordination is required between the upgrade and testing phases because test interoperability wrappers can be independently generated as needed. The main disadvantages are that problems can be difficult to isolate in the integrated application and extra work is required to create the interoperability wrappers that will not be necessary in the final product.
- Testing can be performed in the order established by the component dependency graph. In each test iteration, a level of the application functionality will be verified. When a component is tested, all the components it depends on are also tested, so it is necessary for all such components to have their upgrade completed. The most basic components should be upgraded first to allow the testing of the more complex components. The advantages of this approach are that the work is done in a progressive way and it is easy to isolate and correct all detected problems. The main disadvantages are that more planning and coordination is required and the degree of parallelism is diminished.

A hybrid testing strategy can also be applied. Basic components can be upgraded and tested as indicated in the second strategy while some application functionality can be upgraded using the first strategy. This would allow you to have early upgrade results that can be demonstrated to end users, while the deep core upgrade is still in progress. In the end, both workflows converge in one intermediate point where the basic components are upgraded and can provide all necessary services to the components in the higher levels of functionality. The main disadvantage of this hybrid approach is the difficulty in identifying fragments of the application functionality that are appropriate for the each upgrade strategy and that do not require redundant work.

Fixing Run-Time Errors

After an application is upgraded, the most difficult errors to detect are those that can only be identified at run time. A high percentage of potential run-time errors are detected by the upgrade wizard. However, there may be problems that will be detected only after the upgraded application runs and is thoroughly tested. The original application test cases and design documentation is particularly important to identify such errors.

The remainder of this section presents some common upgrade issues that are detected by the upgrade wizard and that produce run-time errors. These issues will need to be manually corrected.

Changes to User-Defined Types

User-defined types can be defined in Visual Basic 6.0 using the **Type** keyword. All the members included in a **Type** block are automatically initialized by the Visual Basic 6.0 runtime. The following code example illustrates creating a user-defined type.

```
Const LF_FACESIZE = 32
Private Type MyLogFont
    fHeight As Integer
    fWidth As Integer
    FaceName(LF_FACESIZE) As Byte
End Type

Sub TestFont()
    Dim a As Byte
    Dim f As MyLogFont
    a = f.FaceName(1)
End Sub
```

Notice how the **FaceName** field is declared and initialized with a fixed size.

The equivalent construct in Visual Basic .NET is the **Structure**. Unlike the **Type** construct, the members of a **Structure** need to be explicitly initialized using the **Initialize** method. When the upgrade wizard is applied to the preceding code, it translates the type to a structure and includes the **Initialize** method for you. The following code shows the result produced by the upgrade wizard.

```
Const LF_FACESIZE As Short = 32
Private Structure MyLogFont
    Dim fHeight As Short
    Dim fWidth As Short
    <VBFixedArray(LF_FACESIZE)> Dim FaceName() As Byte
    ' UPGRADE_TODO: "Initialize" must be called to initialize instances of this
    ' structure.
    Public Sub Initialize()
        ReDim FaceName(LF_FACESIZE)
    End Sub
End Structure

Sub TestFont()
    Dim a As Byte
    ' UPGRADE_WARNING: Arrays in structure f may need to be
    ' initialized before they can be used.
    Dim f As MyLogFont
    a = f.FaceName(1)
End Sub
```

This upgraded code will compile without problems. However, when the application is executed and **TestFont** is invoked, a run-time error will occur because the **Initialize** method has not been invoked and the array members of the structure have not been initialized. To correct this problem, you can apply the following modification to **TestFont** to invoke the **Initialize** method.

```
Sub TestFont()
    Dim a As Byte
    Dim f As MyLogFont
    f.Initialize()
    a = f.FaceName(1)
End Sub
```

After the **Initialize** method is invoked, the array members of the type are initialized and the run-time error no longer occurs.

Changes to Null and IsNull

Null and **IsNull** have different behaviors in Visual Basic .NET than in Visual Basic 6.0. The upgrade wizard detects the usage of **Null** and **IsNull**. **Null** is upgraded to the closest value available in Visual Basic .NET, which is **System.DBNull.Value**. However, Visual Basic .NET does not support propagating the **Null** value. This will result in behavior differences that produce run-time errors. This is illustrated in the following code example.

```
Sub TestNull()
    dbVal = Null
    res = 5 * dbVal
    If IsNull(res) Then
        ' Perform some action
    End If
End Sub
```

Note that **dbVal** is usually obtained by accessing a database through a query. To simplify this example, the **Null** value is directly assigned to this variable.

When this code is upgraded using the upgrade wizard, the following result is obtained.

```
' UPGRADE_ISSUE: IsNull function is not supported.
Sub TestNull()
    Dim res As Object
    Dim dbVal As Object
    ' UPGRADE_WARNING: Use of Null/IsNull() detected.
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object dbVal.
    dbVal = System.DBNull.Value
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object dbVal.
```

```
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object res.
    res = 5 * dbVal
    ' UPGRADE_WARNING: Use of Null/IsNull() detected.
    If IsDbNull(res) Then
        ' Perform some action
    End If
End Sub
```

Note the number of UPGRADE_WARNING comments inserted into the code. This code will compile without problems, but it will result in run-time errors when executed.

To correct this problem, it is necessary to manually adjust the **Null** value propagation and adjust the conditions that rely on the null checking functions. The necessary corrections are shown in bold the following code example.

```
Sub TestNull()
    Dim res As Object
    Dim dbVal As Object
    dbVal = System.DBNull.Value
    If Not IsDBNull(dbVal) Then
        res = 5 * dbVal
    End If
    If IsDBNull(dbVal) Or IsDBNull(res) Then
         ' Perform some action
    End If
End Sub
```

Changes to Array Indexing

In Visual Basic 6.0, it is possible to create arrays whose lower index bound is a value other than 0. In Visual Basic .NET, all arrays must have a lower index bound of 0.

The upgrade wizard ensures all arrays have a lower bound of 0. However, if the source code relies on the calculated size of arrays, the behavior of the array code in the upgraded version will likely have different behavior than the original code. The following code example illustrates this situation.

```
Function GetArraySize(a As Variant) As Long
    Size = UBound(a) - LBound(a) + 1
    GetArraySize = Size
End Function
Sub testArraySize()
    Dim a(5 To 10)
    If GetArraySize(a) > 7 Then
        ' Perform some action
    End If
End Sub
```

After the code is upgraded with the upgrade wizard, the following result is obtained.

```
Function GetArraySize(ByRef a As Object) As Integer
    Dim aSize As Object
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object size.
    aSize = UBound(a) - LBound(a) + 1
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object size.
    GetArraySize = aSize
End Function
Sub testArraySize()
    ' UPGRADE_WARNING: Lower bound of array was changed from 5 to 0.
    Dim a(10) As Object
    Dim res As Integer
    If GetArraySize(a) > 7 Then
        ' Perform some action
    End If
End Sub
```

In the original version of the code, the statements in the **If** block will not be executed because the size of the array is only 6. However, after the automated upgrade, the resulting array in the upgraded code has a lager size of 11, which will trigger the execution of the **If** block. This will most likely be an undesired behavior.

Fixing this specific problem requires a detailed review of the original and upgraded code to conserve the original array size without affecting the application behavior. In this example, the upgraded array should have 6 elements, indexed from 0 to 5. All the code that depends on the array size will be corrected with that change. Code that depends on the specific position of elements will require review because the index range has changed. The following code example demonstrates this adjustment to array bounds.

```
Function GetArraySize(ByRef a As Object) As Integer
    Dim aSize As Object
    aSize = UBound(a) - LBound(a) + 1
    GetArraySize = aSize
End Function
Sub testArraySize()
    Dim a(5) As Object
    Dim res As Integer
    If GetArraySize(a) > 7 Then
        'Perform some action
    End If
End Sub
```

For more strategies that you can apply to deal with changes in array bounds, see Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features."

Changes to the Activate Event

It is not uncommon to include code to recalculate or refresh the state of a **Form** in the Visual Basic 6.0 **Activate** event. In Visual Basic 6.0, this event is raised only when switching between forms in the application. In Visual Basic .NET, the equivalent **Activated** event is raised in these situations, but it is also raised when switching from other applications. This will produce run-time differences between the two versions. The following Visual Basic 6.0 source code illustrates the situation.

```
Private Sub Form_Activate()
    ' Refreshing code
End Sub
```

After upgrading the code with the upgrade wizard, the following code is produced.

```
' UPGRADE_WARNING: Form event Form1.Activate has a new behavior.
Private Sub Form1_Activated(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles MyBase.Activated
    ' Refreshing code
End Sub
```

The problem is that the **Activated** event is raised more frequently than the Visual Basic **Activate** event. **Activated** is raised whenever the focus is switched from other applications. This results in different behavior than the original code.

To correct this difference, it will be necessary to differentiate activations coming from other applications and activations coming from other forms in the same application. This can be done by using a global variable that stores the last application form that was activated. This variable can be implemented as a **Public Shared** member in a new application class. When the focus is changed to another application and then returns to the same form, the global variable will still have the same value. The **Activated** event code can be skipped based on that value. The following code example demonstrates this solution.

```
Friend Class MyAppGlobals
    Public Shared AppActiveForm = ""
End Class

Private Sub Form1_Activated(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles MyBase.Activated
    If Not MyAppGlobals.AppActiveForm = MyBase.Name Then
        ' Refreshing code
        MyAppGlobals.AppActiveForm = MyBase.Name
    End If
End Sub
```

Changes to ComboBox Control Events

This example shows another situation in which behavior differences exist between the original and the upgraded version of an application. These differences may produce run-time errors.

In Visual Basic 6.0, the **ComboBox** control has a **Change** event that is raised when the user writes text into the control. In Visual Basic .NET, the corresponding event, **TextChanged**, is raised in the same situation. However, this event is also raised when the corresponding form is initialized, before it is even displayed. This may cause errors at run time because the **TextChanged** event will be raised form loads and the **ComboBox** has not yet been initialized.

The following Visual Basic 6.0 code example demonstrates this problem.

```
Dim myClass1 As Class1

Private Sub Combo1_Change()
    myClass1.hello
End Sub

Private Sub Form_Load()
    Set myClass1 = New Class1
End Sub
```

The auto-upgraded code is shown here.

```
Dim myClass1 As Class1

' UPGRADE_WARNING: Event Combo1.TextChanged may raise when form is
' initialized.
' UPGRADE_WARNING: ComboBox event Combo1.Change was upgraded to
' Combo1.TextChanged which has a new behavior.
Private Sub Combo1_TextChanged(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles Combo1.TextChanged
    myClass1.hello()
End Sub

Private Sub Form1_Load(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    myClass1 = New Class1
End Sub
```

Now the **Combo1_TextChanged** event handler method will be executed whenever the corresponding form is initialized. This will produce a run-time error because **myClass1** is not yet initialized; this happens in the **Form1_Load** event handler. The solution to this problem requires the creation of a new attribute in the corresponding **Form** class. This attribute will indicate whether the form is being initialized. This attribute must be set at the beginning and at the end of the **InitializeComponent**

method. The following code extract, which should be contained in the **Form1** class, demonstrates the correction.

```
Friend Class Form1
    Inherits System.Windows.Forms.Form

    Private initializing As Boolean
    Dim myClass1 As Class1

    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        initializing = True
        ' The original code included in InitializeComponent must be copied
        ' here
        initializing = False
    End Sub

    Private Sub Combo1_TextChanged(ByVal eventSender As System.Object, _
                ByVal eventArgs As System.EventArgs) Handles Combo1.TextChanged
        If Not initializing Then
            myClass1.hello()
        End If
    End Sub

    Private Sub Form1_Load(ByVal eventSender As System.Object, _
                ByVal eventArgs As System.EventArgs) Handles MyBase.Load
        myClass1 = New Class1
    End Sub
End Class
```

Summary

A successful and efficient upgrade is best achieved by applying proven upgrade procedures. Careful pre-upgrade preparation will save both time and effort during later stages of the upgrade process. Understanding common upgrade issues and identifying such potential issues before you begin can make solving them smoother. Finally, taking advantage of tools that help automate as much of the upgrade as possible can save time and effort. This chapter has demonstrated the steps of an upgrade strategy that can be applied to ensure your applications are upgraded successfully in the most efficient way possible.

More Information

For more information about the Visual Basic 6.0 Code Advisor, see "Visual Basic 6.0 Code Advisor" in the Microsoft Visual Basic Developer Center on MSDN: *http://msdn.microsoft.com/vbasic/downloads/codeadvisor/default.aspx.*

For more information about the Visual Basic Upgrade Wizard Companion, upgrade tool, and additional upgrade services, go to the ArtinSoft Web site: *http://www.artinsoft.com/.*

For a description of errors, warnings, and issues, see the left pane of "Visual Basic 6.0 Upgrading Reference" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vboriVisualBasic60CompatibilityReference.asp.*

6

Understanding the Visual Basic Upgrade Wizard

The Visual Basic Upgrade Wizard is an automated source code upgrade tool that upgrades a project written in Visual Basic 6.0 to a series of files written in Visual Basic .NET. It gives you the ability to keep older applications in use and to take full advantage of the .NET Framework.

The upgrade tool works very much like a software developer who upgrades an application manually: It takes into account the different aspects of software creation and transformation, including program organization, statements, expressions, symbols, data types, literal values and variables. Using this data, the upgrade wizard can generate Visual Basic .NET source code that conserves the algorithms and general style of the original source code. The upgraded code can then be understood and improved by the human developer.

Using the Upgrade Tool

Before discussing the Visual Basic Upgrade Wizard, it is important to first look at the Visual Basic Upgrade Tool. The upgrade tool upgrades Visual Basic 6.0 projects to Visual Basic .NET. The upgrade tool creates a new project, copies each file from the original project into the new project, and modifies the files as necessary. Finally, it generates a report detailing what was done and what you need to do to finish the upgrade.

The upgrade tool is available in two forms: the upgrade wizard that is integrated into the Visual Studio .NET development environment and a command-line tool that is available with the .NET Framework. To invoke the upgrade wizard, you open a Visual Basic 6.0 project file (.vbp) in Visual Studio .NET. Doing so will automatically

start the upgrade wizard and it will take you through the upgrade process step-by-step. When Visual Studio .NET is not available, or if you prefer working from the command-line, you can invoke the command-line version of the upgrade tool.

For complete details about how to run both the wizard version and command-line version of the upgrade tool, see "Execution of the Visual Basic Upgrade Wizard" in Chapter 5, "The Visual Basic Upgrade Process."

Note: The Visual Basic Upgrade Wizard is the version of the tool that is typically used in upgrade projects. Throughout this chapter, the term *upgrade wizard* is used to refer to the upgrade tool. However, the content of this chapter applies equally to both the upgrade wizard and the command-line version of the tool.

Tasks Performed by the Upgrade Wizard

When you use the upgrade wizard on a Visual Basic 6.0 project, the upgrade wizard creates a new .NET project that contains all of the upgraded files from the original Visual Basic 6.0 project. Table 6.1 shows Visual Basic 6.0 projects with their .NET equivalents, or alternatives if there are not any .NET equivalents:

Table 6.1: Visual Basic .NET Project Equivalents for Visual Basic 6.0 Project Types

Visual Basic 6.0	Visual Basic .NET
Standard EXE	Windows Application
ActiveX DLL	Class Library
ActiveX EXE	Class Library
ActiveX Control	Windows Control Library
ActiveX Document	No direct equivalent. Visual Basic .NET can interoperate with ActiveX Documents.
DHTML Application	No direct equivalent. Use ASP.NET Web Application.
IIS Application (WebClass)	No direct equivalent. Use ASP.NET Web Application.

Note: For more information about the ASP.NET Web application, see Chapter 4, "Common Application Types", Chapter 20, "Common Technology Scenario Advancements," and Appendix C, "Introduction to Upgrading ASP."

Code Modification

When the upgrade wizard upgrades your project, it modifies the code in the following ways.

- **Language changes**. The upgrade wizard upgrades Visual Basic 6.0 code to use the new keywords and language structures introduced in Visual Basic .NET. Readability, logic structure, and code ownership are preserved, and supporting run-time functions are added to achieve functionality.
- **Core functions**. References to Visual Basic 6.0 core functions are upgraded to equivalent classes and methods in the .NET Framework, when possible. When there are no equivalent .NET functions, or the upgrade wizard cannot determine the function to use, the wizard adds a note to the code file so you can fix it manually.
- **Forms**. Visual Basic 6.0 forms and controls are upgraded to .NET Windows Form Controls equivalents. For more information about Windows Form Controls, see "Forms" in the "Visual Basic 6.0 Objects" section later in this chapter.
- **Web classes**. Visual Basic 6.0 Web classes are upgraded to .NET Web Forms. For more information about .NET Web Forms, see "Web Classes" in the "Visual Basic 6.0 Language Elements" section later in this chapter.
- **Data**. Data and connection objects are upgraded to Visual Basic .NET runtime callable wrappers (RCW) corresponding to data and connection objects. Upgrading these components to ADO.NET requires additional work.
- **User controls**. Visual Basic 6.0 ActiveX user control components are upgraded to .NET user controls. For more information about .NET user controls, see "User Controls" in the "Visual Basic 6.0 Objects" section later in this chapter.

Redundant features, such as DDE, and features that rely on the Visual Basic 6.0 runtime, such as **ScaleMode**, are not upgraded but are reported as warnings and errors.

Reference Checking

When the upgrade wizard upgrades a project to Visual Basic .NET, it inspects the code for references to other files in the project, and to any external libraries, components, COM objects, and files. When the upgrade wizard renames project files — usually by changing file name extensions to conform to Visual Basic .NET conventions — it automatically updates any references to changed files with the new file names and locations. The upgrade wizard also automatically upgrades any resource files and binary resources for use with the Visual Basic .NET project.

The upgrade wizard instantiates any external components referenced in project files to obtain information about them so it can generate type casts, identify member mappings, and otherwise improve the quality of the upgraded code. For the upgrade wizard to successfully do this, any external components referenced by files in your project must be correctly installed on the computer that you will use to perform the upgrade.

The ArtinSoft Visual Basic Upgrade Wizard Companion

An enhanced version of the upgrade wizard named the ArtinSoft Visual Basic Upgrade Wizard Companion can also parse database schema definition files to obtain additional information about field data types and produce improved Visual Basic .NET code. This and other enhancements can help you further shorten your upgrade process by automating more of it. The ArtinSoft Visual Basic Upgrade Wizard Companion is sold and supported by ArtinSoft. Its price is determined by the size of the application source code. For more information, go to the ArtinSoft Web site.

The Upgrade Report

When the upgrade wizard upgrades your project, it generates an upgrade report. You can use this report to help you make the manual changes that your project requires. Figure 6.1 shows an example of the upgrade report.

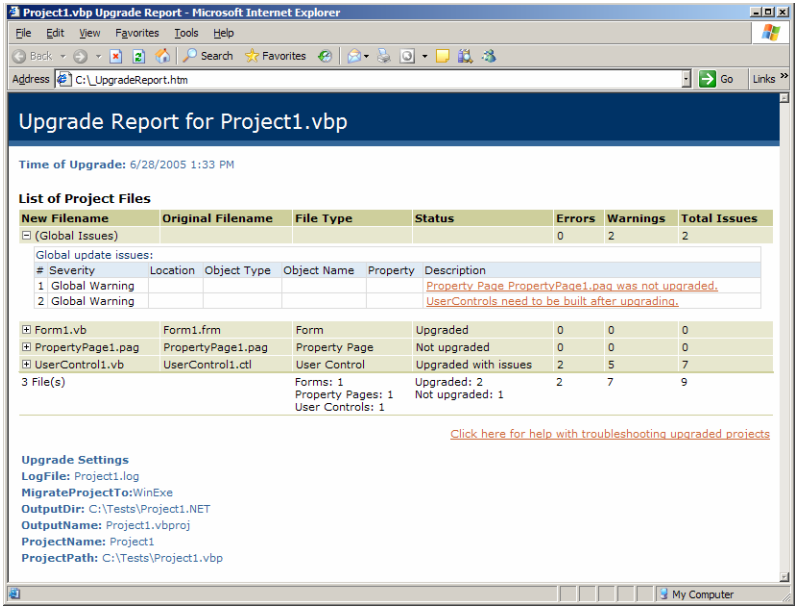

Figure 6.1
A sample upgrade report

The upgrade report contains information about the upgrade process, as well as a list of issues uncovered during the upgrade that you need to address before your project can be compiled. The upgrade report is in HTML format; you can view it within the Visual Studio .NET development environment by double-clicking the _UpgradeReport.htm file in Solution Explorer, or you can view it in an external Web browser by choosing **Browse With** from the **File** menu.

The top portion of the report contains a list of global issues followed by a listing for each file that was upgraded. By expanding the section for each file, you can view a detailed list of issues that will need to be addressed. Each issue includes information about the severity of the issue, the location of the code that must be fixed, and a description of the issue. The last portion of the report contains general information about the upgrade, including the settings that were used during the upgrade and the location of the new project files.

In addition to the report, the upgrade wizard inserts comments into your upgraded code alerting you to statements that will need to be changed. These comments are displayed as TO DO tasks in the new Visual Studio .NET Task List window, so you can easily see what changes are required. Double-clicking a task in the task list will immediately take you to the relevant line of code. Each task and item in the upgrade report is associated with an online Help topic that gives further information on why the code needs to be changed, and guidance on what you need to do to resolve the issue.

Supported Elements

This section details how the upgrade wizard upgrades Visual Basic 6.0 language structures, native libraries, forms, and ActiveX objects to Visual Basic .NET.

Visual Basic 6.0 Language Elements

The upgrade wizard upgrades different Visual Basic 6.0 elements to Visual Basic .NET, as discussed in the next sections.

Declarations

The upgrade wizard automatically upgrades variable declarations to their equivalent.NET Framework types, following the Visual Basic .NET structure and taking into account the Visual Basic 6.0 declaration guidelines.

For example, in this Visual Basic 6.0 code, `a` and `b` are implicitly declared with the **Variant** data type, and `c` is declared explicitly as a 16-bit integer.

```
Dim a, b, c As Integer
```

In Visual Basic .NET, variables must be declared explicitly, and there is no **Variant** data type. The upgrade wizard changes the code to declare `a` and `b` using the *object* data type, which is the .NET Framework's closest equivalent to **Variant**, and declares `c` as *short*, which is what Visual Basic .NET calls its 16-bit integer data type.

```
Dim a, b As Object
Dim c As Short
```

Routines

Parameters can be passed to a procedure or function in one of two ways: by reference (**ByRef**) or by value (**ByVal**). When a parameter is passed to a subprocedure by reference, changes made to the parameter's value affect the actual variable or object passed by the calling procedure. When a parameter is passed by value, the subprocedure uses a copy of the original variable or object, and any changes the subprocedure makes to it do not affect the original.

By default, Visual Basic 6.0 uses the **ByRef** method when the parameter's data type is an intrinsic one such as **Integer**, **Long**, **Boolean**, **String**, and so on. When a parameter uses a non-intrinsic data type, Visual Basic 6.0 passes it by value by default.

By comparison, Visual Basic .NET passes *all* parameters by value unless otherwise specified. If a parameter keyword is specified in the signature a Visual Basic 6.0 routine, the upgrade wizard will generate the Visual Basic .NET routine accordingly. If **ByVal** or **ByRef** are not specified for a parameter in a Visual Basic 6.0 routine, the upgrade wizard will specify **ByRef** for the parameter.

The following Visual Basic 6.0 code uses both explicit and implicit methods to pass parameters.

```
Public Sub ByValTestSub(ByVal p1 As String)
    p1 = "Hello World"
End Sub

Public Function TestFunc(p1 As Integer, p2 As Integer) As Integer
    TestFunc = p1 + p2
End Function
```

The upgrade wizard explicitly passes the parameters by reference to the **TestFunc** function, so it will operate the same way in Visual Basic .NET as it did in Visual Basic 6.0.

```
Public Sub ByValTestSub(ByVal p1 As String)
    p1 = "Hello World"
End Sub

Public Function TestFunc(ByRef p1 As Short, ByRef p2 As Short) As Short
    TestFunc = p1 + p2
End Function
```

When a parameter is declared as **Optional** in Visual Basic 6.0, you do not have to specify a default value for the parameter. For example, this Visual Basic 6.0 subroutine can take an optional **String** parameter, **p1**.

```
Public Sub OpTestSub(Optional p1 As String)
End Sub
```

In Visual Basic .NET, you must specify a default value when declaring an optional parameter. The subroutine uses the default value for the variable when the calling procedure does not supply the parameter. The upgrade wizard automatically adds default values for all optional parameters that do not already have them.

```
Public Sub OpTestSub(Optional ByRef p1 As String = "")
End Sub
```

For numeric data types, the upgrade wizard uses a default value of 0. For strings, it uses an empty string (“”). For objects, it uses **Nothing** as a default value.

Properties

In Visual Basic 6.0, properties are defined using the **Property Get, Property Let,** and **Property Set** statements. Visual Basic .NET replaces these statements with a new property declaration syntax that uses **Get** and **Set** *accessors* to provide access to the property.

This Visual Basic 6.0 code uses **Property Get**, **Property Let**, and **Property Set** statements to expose a property named **Text**.

```
Private mText As String
Private mObj As Object

'Text properties
Public Property Get Text() As String
    Text = mText
End Property
Public Property Let Text(ByVal Value As String)
    mText = Value
End Property

'Object properties
Public Property Get Obj() as Object
    Obj = mObj
End Property
Public Property Set Obj(ByVal Value As Object)
    Set mObj = Value
End Property
```

The upgrade wizard combines these statements into a single property declaration for each property. In this particular example, two new property declarations are created, as shown here.

```
Private mText As String
Private mObj As Object

Public Property Text() As String
    Get
        Return mText
```

```
    End Get
    Set(ByVal Value As String)
        mText = Value
    End Set
End Property

Public Property Obj() As Object
    Get
        Obj = mObj
    End Get
    Set(ByVal Value As Object)
        mObj = Value
    End Set
End Property
```

In Visual Basic 6.0, you can create a read-only property by omitting the **Property Let** and **Property Set** statements. For example, to make the Text and Obj properties in the preceding example read-only, you can remove the Property Set and Property Let code, as shown here.

```
Private mText As String
Private mObj As Object
Private mText_w As String
Private mObj_w As Object

Public Property Get Text() As String
    Text = mText
End Property

Public Property Get Obj() As Object
    Set Obj = mObj
End Property

Public Property Set TextW(value As String)
    mText_w = value
End Property

Public Property Set ObjW(value As Object)
    Set mObj_w = value
End Property
```

In Visual Basic .NET, read-only properties are declared using the **ReadOnly** modifier. When you upgrade a Visual Basic 6.0 read-only property to Visual Basic .NET, the upgrade wizard will add the **ReadOnly** modifier to the property declaration. An analogous declaration is made for write-only properties, where the upgrade wizard uses a **WriteOnly** modifier. The resulting code for this example is shown here.

```
Private mText As String
Private mObj As Object
Private mText_w As String
Private mObj_w As Object
```

```
Public ReadOnly Property Text() As String
    Get
        Return mText
    End Get
End Property
Public ReadOnly Property Obj() As Object
    Get
        Obj = mObj
    End Get
End Property
Public WriteOnly Property TextW() As String
    Set(ByVal Value As String)
        mText_w = Value
    End Set
End Property
Public WriteOnly Property ObjW() As Object
    Set(ByVal Value As Object)
        mObj_w = Value
    End Set
End Property
```

Event Handlers

Events in Visual Basic .NET are built on top of delegates, so they are handled very differently from events in Visual Basic 6.0.

The upgrade wizard automatically handles the details surrounding events by generating signatures for the methods according to the rules of the .NET Framework, and by adding the necessary Visual Basic keywords to comply with the Visual Basic .NET events infrastructure.

For example, consider a Visual Basic 6.0 class, **Class1** that defines an **OnChange** event with a **Text** property as follows.

```
Private mText As String

Public Event OnChange(ByVal Text As String)

Private Sub Change(ByVal Text As String)
    RaiseEvent OnChange(Text)
End Sub

Public Property Get Text() As String
    Text = mText
End Property

Public Property Let Text(ByVal Value As String)
    mText = Value
    Call Change(Value)
End Property
```

A Visual Basic 6.0 client application consumes the event. The application contains a **TextBox** control and a **Button** control, and creates a **Class1** object called **Instance**. The application contains a form load event (**Form_Load)**, a button click event (**Command1_Click**), and a **Class1** change event (**Instance_OnChange**). When the button is clicked, the text property of the **Class1** is modified, that in turn raises the **Instance_OnChange** event.

```
Public WithEvents Instance As Class1

Private Sub Command1_Click()
    Instance.Text = Text1.Text
End Sub
Private Sub Form_Load()
    Set Instance = New Class1
End Sub
Private Sub Instance_OnChange(ByVal Text As String)
   Call MsgBox("Changed to: " + Text)
End Sub
```

When **Class1** is upgraded to Visual Basic .NET with the upgrade wizard, it produces the following code.

```
Option Strict Off
Option Explicit On
Friend Class Class1
    Private mText As String
    Public Event OnChange(ByVal Text As String)

    Private Sub Change(ByVal Text As String)
        RaiseEvent OnChange(Text)
    End Sub

    Public Property Text() As String
        Get
            Return mText
        End Get
        Set(ByVal Value As String)
            mText = Value
            Call Change(Value)
        End Set
    End Property
End Class
```

The upgrade wizard upgrades the client as follows.

```
Option Strict Off
Option Explicit On
Friend Class Form1
    Inherits System.Windows.Forms.Form
#Region "Windows Form Designer generated code "
    …
    Public WithEvents Text1 As System.Windows.Forms.TextBox
    Public WithEvents Command1 As System.Windows.Forms.Button
    …
#End Region
#Region "Upgrade Support "
    …
#End Region

    Public WithEvents Instance As Class1

    Private Sub Command1_Click(ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) Handles Command1.Click
        Instance.Text = Text1.Text
    End Sub

    Private Sub Form1_Load(ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) Handles MyBase.Load
        Instance = New Class1
    End Sub

    ' UPGRADE_NOTE: Text was upgraded to Text_Renamed. …
    Private Sub Instance_OnChange(ByVal Text_Renamed As String) Handles _
        Instance.OnChange
        Call MsgBox("Changed to: " & Text_Renamed)
    End Sub
End Class
```

In both cases, the upgrade wizard has upgraded the Visual Basic 6.0 code to Visual Basic .NET, and no additional modifications are required.

An important change related to events is in parameter passing. In Visual Basic 6.0, the event subroutine contains the name and type of each parameter. In Visual Basic .NET, the parameters are bundled up in an **EventArgs** object and a reference to this object is passed to the event handling subroutine. Also, the event subroutine will receive a reference to the object that initiated the event.

To illustrate the difference in parameter passing for event handling subroutines, consider a Visual Basic 6.0 form that contains a **ListBox** control named **List1**. If **List1** is set up with the check box style, you can have an event handler deal with a checked item. This is shown in the following code example.

```
Private Sub List1_ItemCheck(Item As Integer)
   MsgBox "You checked item: " & Item
End Sub
```

The upgrade wizard can be applied to the code. This results in an upgraded version of the code with the new event handler format. The following code example shows the results.

```
' UPGRADE_WARNING: ListBox event List1.ItemCheck has a new behavior.
Private Sub List1_ItemCheck(ByVal eventSender As System.Object, _
                 ByVal eventArgs As System.Windows.Forms.ItemCheckEventArgs) _
                 Handles List1.ItemCheck
   MsgBox("You checked item " & eventArgs.Index)
End Sub
```

Notice how the item that is checked is passed directly as a parameter in the original code. In contrast, it is passed as a member of the **ItemCheckEventArgs** object **eventArgs** in the upgraded version. Note, too, that the original version required only the selected item as a parameter, where the upgraded version requires the event arguments object, as well as a reference to the object that initiated the event.

Control Structures

Some of the Visual Basic .NET control structures are slightly different from the control structures in Visual Basic 6.0. For example, the **Wend** keyword, that ends **While** loops in Visual Basic 6.0, is replaced by **End While** in Visual Basic .NET. The upgrade wizard automatically upgrades Visual Basic 6.0 flow statements and preserves their functionality.

The following Visual Basic 6.0 code demonstrates the **If…Then, For…Next,** and **While…Wend** control structures.

```
Dim i As Integer
Dim bol As Boolean
Dim value As Integer

bol = True

If bol = True Then
    i = 0
Else
    i = -1
End If
For i = 0 To 10
    value = value + i
Next i

i = 0
While i < 10
    i = i + 1
Wend
```

The upgrade wizard upgrades the control structures to Visual Basic .NET as follows.

```
Dim i As Short
Dim bol As Boolean
Dim value As Short
bol = True

If bol = True Then
    i = 0
Else
    i = -1
End If

For i = 0 To 10
    value = value + i
Next i

i = 0
While i < 10
    i = i + 1
End While
```

Modules and Classes

Modules and classes are fully supported by the upgrade wizard. In Visual Studio 6.0, module and class files have distinct file name extensions (.bas for modules, .cls for classes), and are organized in specific folders within the Project folder.

In Visual Basic .NET, modules and class files both have the same file name extension, .vb, and are distinguished from one another by the structure within each file.

The upgrade wizard automatically handles file name upgrades. For each element, the upgrade wizard will compose a name based on a specific structure. The following code example shows how the upgrade wizard handles a name upgrade for a module.

```
Module MODULENAME
End Module
```

The following is for classes.

```
Friend Class Class1
End Class
```

Interfaces

In Visual Basic 6.0, common classes can be treated as interfaces. Consider the following Visual Basic 6.0 class, **Class1**.

```
Class1:
Public Sub Test()
    MsgBox "Hello from class 1"
End Sub
```

Class2 uses the **implements** keyword to implement **Class1**.

```
Class2:
Implements Class1
Public Sub Class1_Test()
    MsgBox "Hello from class 2"
End Sub

Public Sub Test()
    Dim c As Class1
    Set c = New Class1
    c.Test
End Sub
```

When the upgrade wizard upgrades **Class1** to Visual Basic .NET, it also creates an interface with a similar name as shown here.

```
Interface _Class1
    Sub Test()
End Interface
Friend Class Class1
    Implements _Class1
    Public Sub Test() Implements _Class1.Test
        MsgBox("Hello from class 1")
    End Sub
End Class
```

The upgrade wizard generates both a class and an interface to replicate the behavior of Visual Basic 6.0. In this case, an object can be used both as an interface and as a regular class. **Class2** that implements **Class1** and uses it as a regular class is upgraded as follows.

```
Friend Class Class2
    Implements _Class1
    Public Sub Class1_Test() Implements _Class1.Test
        MsgBox("Hello from class 2")
    End Sub
    Public Sub Test()
        Dim c As _Class1
        c = New Class1
        c.Test()
    End Sub
End Class
```

Web Classes

In Visual Basic 6.0, WebClass projects (also known as IIS application projects) are used to create Web applications based on Active Server Pages (ASP) technology. In Visual Basic .NET, ASP.NET Web application projects are used to create Web applications based on the newer ASP.NET technology.

Visual Basic 6.0 WebClass projects can be upgraded to ASP.NET Web Application projects using the upgrade wizard. The process for upgrading WebClass projects is essentially the same as for any other project type; however, there are some related issues that you should consider.

There are two prerequisites for upgrading a WebClass project:

- Internet Information Services (IIS) must be installed and running on the upgrade computer.
- You must have administrative privileges that allow you to install applications.

When upgrading WebClass projects with the upgrade wizard, the following scenarios take place:

- When a WebClass project is upgraded, the upgrade wizard will use a new project name by default, *projectname*.NET, where *projectname* is the name of the Visual Basic 6.0 project. This name is then used when naming the virtual directory. This virtual directory will also be configured as an application in IIS; and the application-friendly name will be the project name.
- If the virtual directory name already exists on the http://localhost server (which is typical if you repeat the application of the upgrade wizard on the same project in order to resolve upgrade issues), the upgrade wizard will append a number to the names of the virtual directory and the project to ensure uniqueness.
- When a Visual Basic 6.0 WebClass project is upgraded to Visual Basic.NET, the .asp file for the project is upgraded to an .aspx file. However, any references to the .asp file within an HTML template file are not automatically changed to .aspx references. This is because a template file might contain references to other .asp files that were not part of the WebClass project.
- The upgrade wizard only copies HTML template files to the new project directory. Any other .html files or image files are not copied to the new directory.
- When the upgrade wizard adds HTML files to a Visual Basic .NET project, it adds them as content files by default. When a WebClass project is upgraded, HTML files are added as embedded resources. If you add HTML files to the project after it is upgraded, you must set their **Build Action** property to **Embedded Resource** to make them visible to the application.
- The scope of **Function** and **Sub** procedures in your Visual Basic 6.0 code (for example, **ProcessTags** or **Respond**) will be changed from **Private** to **Public** to allow the WebClass Compatibility runtime to execute them.

- Declarations will be added to your project for the WebClass and for each of the WebItems and Templates in the WebClass project.
- A **Page_Load** event procedure will be added to the project, creating a **WebClass** object as well as **WebItem** object for each of the WebItems and Templates associated with the Visual Basic 6.0 WebClass project.
- In the **Page_Load** event procedure, you will see a call to the WebClass Compatibility runtime, **WebClass.ProcessEvents**. This allows the runtime to render the WebItem specified in the request URL. This code is the only new code added to your upgraded project, and is only added to emulate the underlying behavior of the Visual Basic 6.0 WebClass runtime.

Note: There are a number of behavioral differences for properties, methods, and events between ASP and ASP.NET objects. For more information on behavioral differences, see Chapter 4, "Common Application Types," and Appendix C, "Introduction to Upgrading ASP."

Late Binding

Both Visual Basic 6.0 and Visual Basic .NET support late-bound objects. This is the practice of declaring a variable as the **Object** data type and assigning it to an instance of a class at run time. However, during the upgrade process, late-bound objects can introduce problems when default properties are resolved, or in cases where the underlying object model has changed and properties, methods, and events need to be upgraded.

For example, consider a Visual Basic 6.0 form called **Form1** with a label called **Label1**. The following Visual Basic 6.0 code example sets the caption of the label to "SomeText".

```
Dim o As Object
Set o = Me.Label1
o.Caption = "SomeText"
```

In Visual Basic .NET, which uses the .NET Framework's Windows Forms classes, the **Caption** property of a label control is now called **Text**. When the upgrade wizard upgrades your code to Visual Basic .NET, it upgrades the **Caption** property to **Text** on all of your Label objects. However, because late-bound objects are type-less, the upgrade wizard cannot detect how you intend to use them, or if any properties should be translated. In such cases, you will need to change the code yourself after it is upgraded.

There are two ways to correctly upgrade late-bound objects:

- You can rewrite the original Visual Basic 6.0 code using early-bound objects, so that the upgrade wizard will upgrade it automatically, as shown here.

```
Dim o As Label
Set o = Me.Label1
o.Caption = "SomeText"
```

  As a rule, you should declare variables using the appropriate object type instead of the general **Object** data type whenever possible.

- You can modify the Visual Basic .NET code obtained after the upgrade process to eliminate unnecessary late-bound variables and manually upgrade the affected class members, as shown here.

```
Dim o As System.Windows.Forms.Label
Set o = Me.Label1
o.Text = "SomeText"
```

Note: You can use the ArtinSoft Visual Basic Upgrade Wizard Companion as an alternative solution to the late-bound variable problem. The ArtinSoft Visual Basic Upgrade Wizard Companion Type Inference feature deduces the most appropriate data types for variables, parameters, and returns values, and avoids using generic data types such as **Object**. When the companion tool encounters an **Object** variable, it declares the variable using the appropriate type so that you do not have to make manual modifications. For more information about the ArtinSoft Visual Basic Upgrade Wizard Companion, go to the ArtinSoft Web site.

Type Casts

The upgrade wizard upgrades type casting statements using the appropriate type restrictions. For example, this Visual Basic 6.0 code converts a 16-bit integer to a string, and then back to an integer.

```
Dim i As Integer
Dim s As String

i = 1
s = CStr(i)
i = CInt(s)
```

The Visual Basic 6.0 **Integer** data type corresponds to the Visual Basic .NET **Short** data type. The upgrade wizard uses the appropriate casting statement when it creates the Visual Basic .NET code, as shown here.

```
Dim i As Short
Dim s As String

i = 1
s = CStr(i)
i = CShort(s)
```

Enums

Enumerations, or enums, are a special type of value type. An enum has a name and a set of fields that define values for a primitive data type.

The upgrade wizard automatically upgrades enumerated types and their references to the format used by Visual Basic .NET. For example, the following Visual Basic .NET code declares and uses an enumerated type.

```
Public Enum MyEnum
    FirstValue
    SecondValue
    ThirdValue
End Enum
Public Sub TestEnum()
    Dim e As MyEnum
    e = SecondValue
End Sub
```

The upgrade wizard automatically modifies the code for Visual Basic .NET as follows.

```
Public Enum MyEnum
    FirstValue
    SecondValue
    ThirdValue
End Enum

Public Sub TestEnum()
    Dim e As MyEnum
    e = MyEnum.SecondValue
End Sub
```

User-Defined Types

In Visual Basic 6.0, you can use user-defined types (UDTs) to create groups of data items of different types. In Visual Basic .NET, UDTs are called structures. A structure associates one or more members with each other and with the structure itself. When you declare a structure, it becomes a composite data type, and you can declare variables of that type.

UDTs and the Upgrade Wizard

The upgrade wizard automatically upgrades UDTs to Visual Basic .NET structures. For example, the following Visual Basic 6.0 code defines and uses a UDT.

```
Private Type MyType
    x As Integer
    y As Integer
    name As String
End Type
Public Sub TestUDT()
    Dim udt As MyType
    udt.name = "Joe"
    udt.x = 5
    udt.y = 10
End Sub
```

The Visual Basic .NET code produced by the upgrade wizard uses the **Structure** keyword to redefine the UDT as a structure.

```
Private Structure MyType
    Dim x As Short
    Dim y As Short
    Dim name As String
End Structure
Public Sub TestUDT()
    Dim udt As MyType
    udt.name = "Joe"
    udt.x = 5
    udt.y = 10
End Sub
```

Upgrading UDTs with Fixed-Length Fields

The previous example shows a UDT that contains native data type fields. The size of these fields is determined by the programming language and the way it is compiled. The fact that there are equivalent .NET data types for each of the original fields allows the automatic upgrade to be performed; this is not always the case.

It is typical to have UDTs that contain fixed-length fields that are composed of basic data elements, such as arrays or fixed-length strings. This type of UDT is commonly used when accessing legacy flat files, as discussed in Chapter 11, "Upgrading String and File Operations," in the section, "Accessing Fixed-Length Records Using UDTs."

The following example shows a Visual Basic 6.0 UDT that contains fixed-length fields.

```
Type employee
    name As String * 15
    last_name As String * 20
    department As String * 6
    phone_ext(3) As Long
    salary As Integer
End Type
```

The upgrade wizard upgrades the code as follows.

```
Structure employee
    <VBFixedString(15),System.Runtime.InteropServices.MarshalAs( _
    System.Runtime.InteropServices.UnmanagedType.ByValTStr, _
    SizeConst:=15)> Public name As String

    <VBFixedString(20),System.Runtime.InteropServices.MarshalAs( _
    System.Runtime.InteropServices.UnmanagedType.ByValTStr, _
    SizeConst:=20)> Public last_name As String

    <VBFixedString(6),System.Runtime.InteropServices.MarshalAs( _
    System.Runtime.InteropServices.UnmanagedType.ByValTStr, _
    SizeConst:=6)> Public department As String

    <VBFixedArray(3)> Dim phone_ext() As Integer
    Dim salary As Short

    ' UPGRADE_TODO: "Initialize" must be called to initialize instances of
    ' this structure. Click for more:
    ' 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="vbup1026"'
    Public Sub Initialize()
        ReDim phone_ext(3)
    End Sub
End Structure
```

The fields whose data type is a fixed-length string cannot be directly declared in the Visual Basic .NET structure because fixed-length strings are not a native type in

Visual Basic .NET and cannot be included in structures. One possible solution that can be used in variable declarations is to use the **VB6.FixedLengthString** compatibility class, but because it is an object it also cannot be included in structures.

To solve the problem for structures, a programmer can indicate that a field in a structure must be treated as a fixed-length element for file I/O purposes. This functionality corresponds to the .NET marshaling capabilities and is specified using the **VBFixedString** or **VBFixedArray** attributes; the argument passed to these attributes represents the size of the field. In this manner, an equivalent Visual Basic .NET structure can be obtained. The preceding Visual Basic .NET code example demonstrates the use of this marshaling attribute.

Note: The technique shown here is applicable only when dealing with fixed-length strings and arrays in file I/O scenarios. In interop situations, you will have to use the **MarshalAs** attribute to properly deal with fixed-length fields. For more information about the **MarshalAs** attribute, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET."

Visual Basic 6.0 Native Libraries (VB, VBA, VBRUN)

The Visual Basic 6.0 native libraries — which provide much of the language's core functionality — have been replaced by the .NET Framework itself. The .NET Framework provides the .NET languages with core functionality through the **System** namespace and other namespaces. The upgrade wizard automatically replaces calls to the Visual Basic 6.0 native libraries with calls to the equivalent .NET Framework objects and classes.

For example, the following Visual Basic 6.0 code uses functions from the VB, VBA, and VBRUN libraries, as indicated.

```
Public Sub TestLibraries()
    Dim s As String                    ' Library: VBA
    s = CStr("2005")                   ' Library: VBA

    Dim c As ColorConstants            ' Library: VBRUN
    c = vbBlue                         ' Library: VBRUN

    MsgBox App.EXEName                 ' Library: VB
End Sub
```

The Visual Basic .NET code covered by the upgrade wizard instead uses objects and methods from the .NET Framework.

```
Public Sub TestLibraries()
Dim s As String                       ' Library: VBA
s = CStr("2005")                      ' Library: VBA

    Dim c As System.Drawing.Color     ' Library: VBRUN
```

```
    c = System.Drawing.Color.Blue      ' Library: VBRUN

    MsgBox(VB6.GetExeName())           ' Library: VB
End Sub
```

Visual Basic 6.0 Objects

This section explains how the upgrade wizard upgrades forms and visual components. It also summarizes the similarities and differences between Visual Basic 6.0 graphical controls and their .NET equivalents.

Forms

The upgrade wizard upgrades Visual Basic 6.0 forms (files with the file name extension .frm) to Windows Forms, the .NET Framework's form development system, which uses classes in the **System.Windows.Forms.Form** namespace:

```
Friend Class Form1
    Inherits System.Windows.Forms.Form
    …
End Class
```

Upgraded form files are saved with the file name extension .vb, like other Visual Basic .NET code files. Each Visual Basic 6.0 form's properties, methods, and events (PME) is upgraded to an equivalent **System.Windows.Forms.Form** PMEs. For example, this Visual Basic 6.0 code changes two visual properties of the current form when a button is clicked.

```
Private Sub Command1_Click()
    Me.Caption = "My Form"
    Me.BackColor = ColorConstants.vbBlue
End Sub
```

The upgrade wizard automatically upgrades the code and modifies the default visual appearance of the form to the Visual Basic .NET default.

```
Option Strict Off
Option Explicit On
Friend Class Form1
    Inherits System.Windows.Forms.Form
#Region "Windows Form Designer generated code "
    …
    ' Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer
    Public WithEvents Command1 As System.Windows.Forms.Button

    ' NOTE: The following procedure is required by the Windows Form Designer
    ' It can be modified using the Windows Form Designer.
    ' Do not modify it using the code editor.
```

```
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        Me.components = New System.ComponentModel.Container()
        Me.Command1 = New System.Windows.Forms.Button
        Me.Text = "Form1"
        Me.ClientSize = New System.Drawing.Size(548, 295)
        Me.Location = New System.Drawing.Point(4, 20)
        Me.StartPosition = _
            System.Windows.Forms.FormStartPosition.WindowsDefaultLocation
        Me.Font = New System.Drawing.Font("Arial", 8!, _
            System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, _
            CType(0, Byte))
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.BackColor = System.Drawing.SystemColors.Control
        Me.FormBorderStyle = System.Windows.Forms.FormBorderStyle.Sizable
        Me.ControlBox = True
        Me.Enabled = True
        Me.KeyPreview = False
        Me.MaximizeBox = True
        Me.MinimizeBox = True
        Me.Cursor = System.Windows.Forms.Cursors.Default
        Me.RightToLeft = System.Windows.Forms.RightToLeft.No
        Me.ShowInTaskbar = True
        Me.HelpButton = False
        Me.WindowState = System.Windows.Forms.FormWindowState.Normal
        Me.Name = "Form1"
…
    End Sub
#End Region
#Region "Upgrade Support "
    …
#End Region

    Private Sub Command1_Click(ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) Handles Command1.Click
        Me.Text = "My Form"
        Me.BackColor = System.Drawing.Color.Blue
    End Sub
End Class
```

For Visual Basic 2005:

The upgrade wizard in Visual Studio 2005 supports additional properties, methods, and events with respect to the previous version. This additional support is based in part on the new features available in Visual Basic 2005; for example, the **BackgroundImageLayout** property allows for a more precise upgrade of the Visual Basic 6.0 **Form.Picture** property, which was originally upgraded to **Form.BackgroundImage** and produced a different behavior message.

Resources

Visual Basic 6.0 supports resource files that have the file name extension .res. Each Visual Basic 6.0 project can have one resource file.

Resource files in Visual Basic .NET have the file name extension .resx. They consist of XML entries that specify objects and strings inside XML tags.

When you use the upgrade wizard to upgrade a Visual Basic 6.0 project that has an associated resource file, the wizard automatically upgrades the Visual Basic 6.0 .res file to a Visual Basic .NET .resx file.

Form Resources

Visual Basic 6.0 form binary files (files with the file name extension .frx) contain necessary resources for the main program form, such as images and icons.

In Visual Basic .NET, binary resources are encoded into .resx files. Forms can retrieve resources automatically from associated resource files.

The upgrade wizard upgrades .frx files to .resx files associated with a Form class.

Source Code Reorganization

In Visual Basic 6.0, every form (including MDIForms) has a pre-declared Form ID, or instance variable, with the same name as the Form class. Declaring a Form ID is similar to using Visual Basic 6.0's **New** keyword: if a line of code references a form and its value is **Nothing**, a new instance of the form is automatically created.

```
' Create an instance if it has not already been done, and show it.
Form1.Show
Form1.BackColor = vbBlue
…
set Form1 = Nothing
' Shows a new instance.
Form1.Show
```

The upgrade wizard simulates Visual Basic 6.0's behavior for forms and MDIForms by creating a public property called **DefInstance**. For the previous Visual Basic 6.0 code example, the upgrade wizard produces the following Visual Basic .NET code.

```
' Create an instance if it has not already been done, and show it.
Form1.DefInstance.Show
Form1.DefInstance.BackColor = vbBlue
…
Form1.DefInstance = Nothing
' Shows a new instance.
Form1.DefInstance.Show
```

For Visual Basic 2005:

Visual Basic 2005 supports the use of the Form class name as a pre-declared instance. This change is supported by the Visual Studio 2005 version of the upgrade wizard to produce the following result used in the previous Visual Basic 6.0 code example:

```
' Create an instance if it has not already been done, and show it.
Form1.Show
Form1.BackColor = vbBlue
…
Form1 = Nothing
' Shows a new instance.
Form1.Show
```

In addition, the upgrade wizard automatically adds the following code to each Form and MDIForm in the project to support the creation of a new instance of a Form.

```
#Region " Upgrade Support "
Private Shared m_vb6FormDefInstance As Form1
Private Shared m_InitializingDefInstance As Boolean
Public Shared Property DefInstance As Form1
    Get
        If m_vb6FormDefInstance Is Nothing OrElse m_vb6FormDefInstance.IsDisposed
_
            Then
            m_InitializingDefInstance = True
            m_vb6FormDefInstance = New Form1
            m_InitializingDefInstance = False
        End If
        DefInstance = m_vb6FormDefInstance
    End Get
    Set
        m_vb6FormDefInstance = Value
    End Set
End Property
#End Region
```

Note: The "Form1" type should be replaced with the form class names.

The following is the Visual Basic .NET code required to instantiate a SDI form.

```
Public Sub New()
    MyBase.New

        If m_vb6FormDefInstance Is Nothing Then
            If m_InitializingDefInstance Then
                m_vb6FormDefInstance = Me
            Else
```

```
                Try
                    ' For the start-up form, the first instance created is the
                    ' default instance
                    If System.Reflection.Assembly. _
                        GetExecutingAssembly.EntryPoint.DeclaringType _
                        Is Me.GetType Then
                        m_vb6FormDefInstance = Me
                    End If
                Catch
                End Try
            End If
        End If

    ' This call is required by the Windows Form Designer.
    InitializeComponent

    ' If this is an MDI child, code to set the MDIParent comes here.
    ' If this is an MDI child, and the MDIForm in the project has
    ' AuthoShowChildren=True, code comes here to automatically show the form.
End Sub
```

If the Form is an MDI Form, the code in **Sub New** is slightly different.

```
Public Sub New()
    MyBase.New
        If m_vb6FormDefInstance Is Nothing Then
            ' Allow only a single instance of this MDI form to be created.
            m_vb6FormDefInstance = Me
        End If

    ' This call is required by the Windows Form Designer.
    InitializeComponent
    MDIForm_Initialize_Renamed()      * this call only if required
End Sub
```

The upgrade wizard also moves all visual components and forms initializations to the **InitializeComponent** method. For example, in a Visual Basic 6.0 form the form's caption is stored in the .frm file.

```
Begin VB.Form Form1
    Caption         =   "Form1"
```

The upgrade wizard moves this code to the **InitializeComponent** method.

```
Friend Class Form1
    Inherits System.Windows.Forms.Form
#Region "Windows Form Designer generated code "
…
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        …
        Me.Text = "Form1"
```

```
        …
        End Sub
#End Region
…
End Class
```

> **For Visual Basic 2005:**
>
> Windows Forms will support **Partial Types;** where user code is stored in one file and **InItializeComponents** and any non-user code will be stored in another file. This allows for splitting the Forms file format into two separate files. The upgrade wizard will upgrade Forms to use partial types. For more information about partial types, see "What's New with the Visual Basic Upgrade Wizard in Visual Basic 2005" on MSDN.

Measurement Unit Conversion

In Visual Basic 6.0, the **ScaleMode** property can be used to change the coordinate system for a form or **PictureBox** control from the default scale of twips. Visual Basic .NET does not support multiple coordinate systems; all measurements must be expressed in pixels. When you upgrade a project to Visual Basic .NET, the upgrade wizard automatically converts the design-time coordinates of visual components from twips to pixels. Code that sets the **ScaleMode** property at run time will cause a compilation error and must be manually modified.

Note: The upgrade wizard assumes that the design-time setting of the **ScaleMode** property is **Twip**; if this is not the case, the upgrade will be incorrect and must be fixed.

The upgrade wizard also uses upgraded methods in the **Microsoft.VisualBasic.Compatibility.VB6** namespace to upgrade run-time modifications of a control's position. For example, this Visual Basic 6.0 method changes the **Height** property of a button.

```
Public Sub x()
    Me.Command1.Height = 5
End Sub
```

The upgrade wizard upgrades this code to Visual Basic .NET as follows.

```
Public Sub x()
        Me.Command2.Height = VB6.TwipsToPixelsY(5)
End Sub
```

As with its upgrade of design-time coordinates, the upgrade wizard assumes that the design-time setting of the **ScaleMode** property is **Twip**. If it is not, you will need to manually fix the method.

Visual Basic 6.0 Native Controls

The .NET Framework provides equivalents for most of the Visual Basic 6.0 standard user interface controls; however, some Visual Basic 6.0 controls have no .NET equivalents, and others have been renamed. Table 6.2 lists the standard controls from the Visual Basic 6.0 Toolbox with their equivalents in the .NET Framework.

Table 6.2: Visual Basic 6.0 Standard Controls and Their .NET Framework Equivalents

Visual Basic 6.0 Standard Controls	Upgrade Wizard Output
PictureBox	System.Windows.Forms.PictureBox. Note that if the PictureBox control is used as a container for other controls, the upgrade wizard upgrades it to System.Windows.Forms.Panel.
Label	System.Windows.Forms.Label
TextBox	System.Windows.Forms.TextBox
Frame	System.Windows.Forms.GroupBox
CommandButton	System.Windows.Forms.Button
CheckBox	System.Windows.Forms.CheckBox
OptionButton	System.Windows.Forms.RadioButton
ListBox	System.Windows.Forms.ListBox
ComboBox	System.Windows.Forms.ComboBox
HScrollBar	System.Windows.Forms.HScrollBar
VscrollBar	System.Windows.Forms.VScrollBar
Timer	System.Windows.Forms.Timer
DriveListBox	Microsoft.VisualBasic.Compatibility.VB6.DriveListBox
DirListBox	Microsoft.VisualBasic.Compatibility.VB6.DirListBox
FileListBox	Microsoft.VisualBasic.Compatibility.VB6.FileListBox
Shape	Not applicable. You use a class in the .NET Common Language Runtime (CLR) to draw shapes.
Line	Not applicable. You use a class in the .NET CLR to draw a line.
Image	System.Windows.Forms.PictureBox
Data	Not applicable. Visual Basic .NET handles data binding differently from Visual Basic 6.0.
OLE	Not applicable.
ImageList	System.Windows.Forms.PictureBox

The upgrade wizard upgrades any **Shape**, **Line**, **Data**, and **OLE** controls in your forms to **System.Windows.Forms.Label** controls, and sets each label's **BackColor** property to **Red**.

The upgrade wizard automatically upgrades the functionality of supported visual components to Visual Basic .NET. For example, consider a Visual Basic 6.0 project that contains a **CommandButton** with a click event, a **Label**, a **TextBox** and a **ComboBox**. When the button is clicked, the other controls are modified as follows.

```
Private Sub Command1_Click()
    Me.Text1.Text = "Hello World"
    Me.Combo1.AddItem ("Item 1")
    Me.Label1.Caption = "Good bye"
End Sub
```

The upgrade wizard upgrades the component to Visual Basic .NET, and changes the methods used to modify the controls accordingly.

```
Option Strict Off
Option Explicit On
Friend Class Form1
    Inherits System.Windows.Forms.Form
#Region "Windows Form Designer generated code "
…
    Public WithEvents Combo1 As System.Windows.Forms.ComboBox
    Public WithEvents Command1 As System.Windows.Forms.Button
    Public WithEvents Text1 As System.Windows.Forms.TextBox
    Public WithEvents Label1 As System.Windows.Forms.Label
…
#End Region
#Region "Upgrade Support "
    …
#End Region

    Private Sub Command1_Click(ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) Handles Command1.Click
        Me.Text1.Text = "Hello World"
        Me.Combo1.Items.Add(("Item 1"))
        Me.Label1.Text = "Good bye"
    End Sub
End Class
```

Control Arrays

A control array is a group of controls that share the same name, type, and event procedures. Elements of the same control array have their own property settings.

The upgrade wizard relays on functions and objects in the **Microsoft.VisualBasic.Compatibility.VB6** namespace to upgrade control arrays.

Note: Classes in the **Microsoft.VisualBasic.Compatibility** namespace and its child namespaces are provided for use by the Visual Basic 6.0 to Visual Basic .NET upgrade tools. In most cases, these classes duplicate functionality that can be achieved using other parts of the .NET Framework. They are only necessary in cases where the Visual Basic 6.0 code model differs significantly from the .NET implementation.

Table 6.3 lists the control arrays in the **Microsoft.VisualBasic.Compatibility.VB6** namespace, which are used to upgrade Visual Basic 6.0 control arrays.

Table 6.3: Control Arrays Available in the Microsoft.VisualBasic.Compatibility.VB6 Namespace

Object	Description
BaseControlArray	Parent class for Visual Basic 6.0 control array emulation.
BaseOcxArray	Parent class for emulated arrays of **ActiveX** controls.
ButtonArray	Emulates a Visual Basic 6.0 control array of **CommandButton** controls.
CheckBoxArray	Emulates a Visual Basic 6.0 control array of **CheckBox** controls.
CheckedListBoxArray	Emulates a Visual Basic 6.0 control array of **ListBox** controls with the **Style** property set to **Checked**.
ComboBoxArray	Emulates a Visual Basic 6.0 control array of **ComboBox** controls.
DirListBoxArray	Emulates a Visual Basic 6.0 control array of **DirListBox** controls.
DriveListBoxArray	Emulates a Visual Basic 6.0 control array of **DriveListBox** controls.
FileListBoxArray	Emulates a Visual Basic 6.0 control array of **FileListBox** controls.
GroupBoxArray	Emulates a Visual Basic 6.0 control array of **Frame** controls.
HScrollBarArray	Emulates a Visual Basic 6.0 control array of **HScrollBar** controls.
LabelArray	Emulates a Visual Basic 6.0 control array of **Label** controls.
ListBoxArray	Emulates a Visual Basic 6.0 control array of **ListBox** controls.
MenuItemArray	Emulates a Visual Basic 6.0 control array of **Menu** controls.
PanelArray	Emulates a Visual Basic 6.0 control array of **PictureBox** controls that contain child controls.
PictureBoxArray	Emulates a Visual Basic 6.0 control array of **PictureBox** controls.
RadioButtonArray	Emulates a Visual Basic 6.0 control array of **OptionButton** controls.
TabControlArray	Emulates a Visual Basic 6.0 control array of **TabStrip** controls.
TextBoxArray	Emulates a Visual Basic 6.0 control array of **TextBox** controls.
TimerArray	Emulates a Visual Basic 6.0 control array of **Timer** controls.
VscrollBarArray	Emulates a Visual Basic 6.0 control array of **VScrollBar** controls.

Data Environments

Visual Basic 6.0 data environment modules have the file name extension .dsr. They contain information used to create ADO connections and commands to access data.

Data environments, their connections, and commands are upgraded to Visual Basic .NET using classes in the **Microsoft.VisualBasic.Compatibility.VB6.BaseDataEnvironment**, **ADODB.Connection** and **ADODB.Command** namespaces, respectively. The upgrade wizard converts Active Designer (.dsr) files to .vb files. Each .vb file contains a class that inherits from **Microsoft.VisualBasic.Compatibility.VB6.BaseDataEnvironment**.

For more information about data access, see "Upgrading Data Environment" in Chapter 12, "Upgrading Data Access."

ActiveX

This section describes how the upgrade wizard upgrades Visual Basic 6.0 applications that use ActiveX components and controls to Visual Basic .NET.

Upgrading ActiveX Components

When the upgrade wizard encounters ActiveX components in a Visual Basic 6.0 application, it creates two types of wrappers around each component:

- **A runtime callable wrapper (RCW)**. The .NET Framework's Type Library Importer (Tlbimp.exe) creates an RCW for the ActiveX component and all dependent assemblies.
- **A Windows Forms wrapper (WFW)**. A WFW "merges" the ActiveX component's native properties, methods, and events with those of **System.Windows.Forms.AxHost**. This is a class that inherits from **System.Windows.Forms.Control** and exposes ActiveX controls to the .NET Framework as fully functional Windows Forms controls.

The upgrade wizard adds these wrappers to the project as DLLs. For example, if you add the ActiveX: Microsoft Calendar Control (MSCAL.OCX) to a Visual Basic 6.0 project and use the upgrade wizard to upgrade the project to Visual Basic .NET, the upgrade wizard will create two files: AxInterop.MSACAL.dll (the WFW) and Interop.MSACAL.dll (the RCW).

The upgrade wizard also automatically upgrades references to the ActiveX component and its properties, methods, and events, and adds any design-time initializations of the component's properties to the **InitializeComponent** method.

This Visual Basic .NET code example represents the output of the upgrade wizard after upgrading a Visual Basic 6.0 application with one form containing the Microsoft Calendar Control (named "Calendar1" here).

```
Option Strict Off
Option Explicit On
Friend Class Form1
    Inherits System.Windows.Forms.Form
#Region "Windows Form Designer generated code "
    …

'Required by the Windows Form Designer
    …
    Public WithEvents Calendar1 As AxMSACAL.AxCalendar
    ' NOTE: The following procedure is required by the Windows Form Designer
    ' It can be modified using the Windows Form Designer.
    ' Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub InitializeComponent()
        …
        Me.Calendar1 = New AxMSACAL.AxCalendar
        CType(Me.Calendar1, System.ComponentModel.ISupportInitialize).BeginInit()
        …
        Calendar1.OcxState = CType(resources.GetObject("Calendar1.OcxState"), _
            System.Windows.Forms.AxHost.State)
        Me.Calendar1.Size = New System.Drawing.Size(273, 177)
        Me.Calendar1.Location = New System.Drawing.Point(56, 8)
        Me.Calendar1.TabIndex = 0
        Me.Calendar1.Name = "Calendar1"
        Me.Controls.Add(Calendar1)
        CType(Me.Calendar1, System.ComponentModel.ISupportInitialize).EndInit()
    End Sub
#End Region
…
End Class
```

ActiveX Controls Upgraded to .NET Intrinsic Controls

The current version of the upgrade wizard automatically upgrades the Microsoft Tabbed Dialog Control 6.0 ActiveX component to the .NET Framework control **System.Windows.Forms.TabControl**.

> **For Visual Basic 2005:**
>
> The Visual Basic 2005 version of the upgrade wizard will have the ability to upgrade additional ActiveX controls to their .NET Framework equivalents.

User Controls

Visual Basic 6.0 uses user controls that are also known as ActiveX Control projects, to create ActiveX controls. After user controls are compiled, they can be hosted in any container that supports ActiveX, including Visual Basic 6.0 forms and Internet Explorer.

In Visual Basic .NET, Windows Class Library projects are used to create reusable classes and components that can be shared with other projects and hosted in Windows Forms applications. The Windows Class Library template replaces the ActiveX DLL project template in Visual Basic 6.0.

Note: Unlike Visual Basic 6.0 User Controls, Windows Class Library controls can not be hosted in Internet Explorer.

The upgrade wizard upgrades Visual Basic 6.0 user controls to Windows Class Libraries. If the entire project is an ActiveX DLL, the upgrade is automatic. If the project is a standard Visual Basic project that contains User Controls, however the upgrade wizard will not create a separate project for these components. After upgrading the application, you should manually create a Class Library project and move the components from the upgraded application to the new project.

In Visual Basic 6.0, the **ReadProperties** and **WriteProperties** events are used to retrieve or save a **UserControl**'s values to a **PropertyBag** object. In Visual Basic .NET, the **PropertyBag** object is no longer supported, and the **ReadProperties** and **WriteProperties** events no longer exist. Visual Basic .NET does not support property pages (.pag and .pax files) that contain serialized data for controls, including **PropertyBag** information. Like any other unsupported project item, the original .pag file (and .pax file, as well, if it exists), will be copied to the destination directory and added to the project with a build action of **None**. For more information on the upgrade of **Property Pages** see "Property Pages" in Chapter 9, "Upgrading Visual Basic 6.0 Forms Features."

The upgrade wizard automatically upgrades the remaining functionality of Visual Basic 6.0 user controls, using **System.Windows.Forms.UserControl** as the base class. Expressions, flow control statements, properties, events, procedures, and functions are upgraded following the same rules that apply to the Visual Basic 6.0 application upgrades.

For more information about components, see Chapter 4, "Common Application Types," Chapter 14 "Interop Between Visual Basic 6.0 and Visual Basic .NET," and Chapter 15, "Upgrading MTS and COM+ Applications."

Summary

To ensure an efficient upgrade of your application, you should apply tools that automate as much of the upgrade as possible.

While several tools are available, the most accessible is part of the Visual Studio .NET IDE: the Visual Basic Upgrade Wizard. The upgrade wizard is capable of automatically performing much of the upgrade of your code. It also helps to identify

portions of code that cannot be automatically upgraded and provides references to the MSDN website for additional help with the issues that it reports.

To make the best use of the upgrade wizard, it is helpful to understand the Visual Basic 6.0 features that it supports and automatically upgrades and those that it does not. This chapter highlights the features that the upgrade wizard can upgrade and explains how it performs these upgrades. With this information, you can efficiently prepare your Visual Basic 6.0 application for use with the upgrade wizard. You will also understand what you need to do after the upgrade wizard had done its job.

Although the upgrade wizard is a powerful tool, there are some features that cannot be automatically upgraded. The following chapters of this guide will provide you with the techniques you need to facilitate features that require manual attention.

More Information

For more information about the ArtinSoft Visual Basic Upgrade Wizard Companion, go to the ArtinSoft Web site:
http://www.artinsoft.com/.

For more information about partial types in Visual Basic 2005, see “What’s New with the Visual Basic Upgrade Wizard in Visual Basic 2005" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/VBUpgrade.asp.

7

Upgrading Commonly-Used Visual Basic 6.0 Objects

The Basic programming language has been around for a long time in a variety of forms — Microsoft GW-BASIC, Microsoft QuickBasic, Microsoft Visual Basic, and Microsoft Visual Basic for Applications (VBA), to name just a few varieties. Each product provides its own brand of the Basic programming language, supporting or not supporting certain types and language constructs. Each product has also made an attempt to clean up or simplify the language.

Microsoft Visual Basic .NET offers another version of the Basic language, adding its own types and language constructs and omitting others. As a result, when you upgrade your applications from Visual Basic 6.0 to Visual Basic .NET, you may find some issues in the upgraded applications. Some of these issues can be remedied with a one-line fix, whereas others require you to recode the problem area or even redesign it.

Different programmers will provide different solutions for the same problem; any particular programming problem can be solved in many different ways. For each of the problems in this chapter, a sample solution that you can use in your own code is provided. However, the Microsoft .NET Framework often provides several different alternatives that achieve the same effect. You may find that another approach is more suitable for the particular problem you are trying to solve. Where possible, alternative solutions are presented along with references to sources where you can learn more about them.

Typical Visual Basic 6.0 applications are built with standard objects available in the language. Several of these objects have become obsolete in Visual Basic .NET and require special handling to upgrade. This chapter describes the most commonly used Visual Basic 6.0 objects that are no longer supported in Visual Basic .NET and how to replace them with Visual Basic .NET technologies.

For Visual Basic 2005:

The upgrade wizard included in Visual Studio .NET 2003 converts third-party components using ActiveX wrappers. The upgrade wizard included with Visual Studio .NET 2005 supports the most common third-party component libraries and upgrades the classes and controls to intrinsic .NET components, eliminating the need of ActiveX wrappers.

The new libraries and the main components supported by the Visual Basic Upgrade Wizard 2005 are:

- Microsoft Common Dialog Control 6.0: CommonDialog
- Microsoft Internet Controls: WebBrowse
- Microsoft Masked Edit Control 6.0: MaskEdBox
- Microsoft Rich Textbox Control 6.0: RichTextBox
- Microsoft Windows Common Controls 6.0:
  - TreeView
  - StatusBar
  - ProgressBar
  - ImageList
  - ListView
  - ToolBar
- COM+ Services Type Library: Most of the classes are supported.

Additionally, the Visual Basic Upgrade Wizard Companion (also known as the Visual Basic Companion) includes support with different levels of automation for the following components and technologies:

- Upgrade of ADO to ADO.NET
- Recordset (ADOR)
- Microsoft Windows Common Controls-2 6.0
- Microsoft Windows Common Controls 5.0
- Microsoft FlexGrid Control 6.0
- Microsoft XML, version 2.6
- ActiveX Threed controls
  - SSPanel
  - SSSplitter
  - SSTab
- ComponentOne VSFlexGrid 7.0 (OLEDB and Light)

To obtain the Visual Basic Upgrade Wizard Companion, go to the ArtinSoft Web site.

Upgrading the App Object

The **App** object in Visual Basic 6.0 is a global object that is used to set or retrieve information about the application. There is no direct equivalent for the **App** object in Visual Basic .NET; however, most of the properties can be mapped to equivalent properties in the Microsoft .NET Framework. For example, the **App** object's version information properties are replaced by assembly attributes in Visual Basic .NET.

In Visual Basic 6.0, you can set the version information by using the **Project Properties** dialog box. To reach this dialog box, click on the **Project** menu and select ***Project* Properties** (where ***Project*** is the name of your project). To set the version information, click the **Make** tab and fill in the appropriate values in the **Version Number** area.

In Visual Basic .NET you should use the assembly attributes, which are set by editing the **AssemblyInfo** file of your converted project.

For example, suppose you have a Visual Basic 6.0 form with one **PictureBox** control and one **Label** control on it. Assume all properties specifying the version of the project are set. During form loading, the **App** object is used to retrieve the information about the application version as major, minor, and revision numbers and the path of the executable file. The following code example demonstrates this.

```
Private Sub Form_Load()
    If App.PrevInstance = True Then
        MsgBox ("The application is already running!")
        End
    End If

    ' Show the application's version number on the form label.
    lblVersion.Caption = "Version " & App.Major & "." & App.Minor & _
                         "." & App.Revision

    ' Displaying an image file that is stored in the application's
    ' folder by retrieving the path to the executing application.
    Picture1.Picture = LoadPicture(App.Path & "\Logo.jpg")
End Sub
```

Using the Visual Basic Upgrade Wizard on the preceding code yields an upgrade issue that must be manually addressed. Because there is no direct equivalent for the **App** object in Visual Basic .NET, some parts of the code that reference it cannot be translated by the upgrade wizard. In this case, the **Revision** property will not be translated by the upgrade wizard, and the code using **App.Revision** remains unchanged. Instead, the upgrade wizard inserts a comment highlighting the issue in the code at the point in which the issue appears. The result of applying the upgrade wizard to this code is shown on the next page.

```
' UPGRADE_ISSUE: App property App.Revision was not upgraded.
lblVersion.Text = "Version " & _
    System.Diagnostics.FileVersionInfo.GetVersionInfo( _
        System.Reflection.Assembly.GetExecutingAssembly.Location).FileMajorPart &
_
        "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
        System.Reflection.Assembly.GetExecutingAssembly.Location).FileMinorPart &
_
       "." & App.Revision
```

Note: This code example has been modified from that produced by the upgrade wizard to make it easier to read. The comment and code are each generated on a single line, whereas this code example has been broken up across multiple lines. This does not affect the compilation of the code in any way.

The code generated by the upgrade wizard is unable to translate the **App.Revision** property reference, and as a result, the code will not compile. There are two ways to fix the error. The first is to simply remove the properties that cannot be upgraded from the code. Using this approach, **App.Revision** would be removed from the example, and the resulting code would appear as follows.

```
Private Sub Form1_Load(ByVal eventSender As System.Object, _
                       ByVal eventArgs As System.EventArgs) _
                       Handles MyBase.Load
    If (UBound(Diagnostics.Process.GetProcessesByName( _
        Diagnostics.Process.GetCurrentProcess.ProcessName)) > 0) Then
         MsgBox("The application is already running!")
         End
    End If

    lblVersion.Text = "Version " & _
        System.Diagnostics.FileVersionInfo.GetVersionInfo( _
        System.Reflection.Assembly.GetExecutingAssembly.Location _
        ).FileMajorPart & _
        "." & System.Diagnostics.FileVersionInfo.GetVersionInfo( _
        System.Reflection.Assembly.GetExecutingAssembly.Location _
        ).FileMinorPart

    ' Displaying an image file that is stored in the
    ' application's folder by retrieving the path to the
    ' executing application.
    Picture1.Image = System.Drawing.Image.FromFile(VB6.GetPath & "\Logo.jpg")
End Sub
```

This approach allows the application to be built, but it will probably result in a loss of functionality. A second approach is to search for components in the Microsoft .NET Framework that can replace the desired functionality. In the preceding example, the problem with the **App.Revision** property can be fixed using the **ProductVersion** property of the **Application** object, which returns the *entire* version number of the application in 0.0.0.0 format. Here is the replacement code.

```
lblVersion.Text = "Version " & Application.ProductVersion
```

Note that because the **ProductVersion** property returns the entire version number for the application, some portions of the code that the upgrade wizard did translate have also been removed.

If you want to access the revision value as a single integer, the mechanism for doing this is provided in Table 7.1.

For a comprehensive list of **App** object properties and methods that have equivalent replacements in Visual Basic .NET, see "App Object Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN. Table 7.1 summarizes the **App** object properties and methods, and provides Visual Basic .NET equivalents you can use to replace them.

Table 7.1: Visual Basic .NET Equivalents for App Object Properties and Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Comments	System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).Comments
CompanyName	(System.Reflection.Assembly.GetExecutingAssembly.Location).CompanyName
EXEName	VB6.GetEXEName()
FileDescription	System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileDescription
HelpFile	See the "Upgrading Integrated Help" section of Chapter 16, "Application Completion," for information about Visual Basic .NET equivalents for the **HelpFile** property.
HInstance	VB6.GetHInstance()
LegalCopyright	System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).LegalCopyright
LegalTrademarks	System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).LegalTrademarks
LogEvent method **LogMode** **LogPath**	Logging in Visual Basic .NET is handled by event logs. You can replace functionality by using the class **System.Diagnostics.EventLog**.
Major	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileMajorPart** **Note**: The format for version numbers is different in Visual Basic .NET.
Minor	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileMinorPart** **Note**: The format for version numbers is different in Visual Basic .NET.

continued

Visual Basic 6.0	Visual Basic .NET Equivalent
NonModalAllowed	This was a read-only property related to ActiveX DLL files. The .NET common language runtime automatically manages this behavior, so this property can be removed.
OleRequestPendingMsgText **OleRequestPendingMsgTitle** **OleRequestPendingTimeout** **OleServerBusyMsgText** **OleServerBusyMsgTitle** **OleServerBusyRaiseError** **OleServerBusyTimeout**	These properties relate to OLE automation, which is not supported by Visual Basic .NET. For alternatives, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET."
Path	VB6.GetPath
PrevInstance	UBound(Diagnostics.Process.GetProcessesByName (Diagnostics.Process.- GetCurrentProcess.ProcessName)) > 0
ProductName	System.Diagnostics.FileVersionInfo.GetVersionInfo(System. Reflection.Assembly.GetExecutingAssembly.Location). ProductName
RetainedProject	Visual Basic .NET does not have the ability to retain a project in memory, so this property must be removed or replaced by the **False** literal.
Revision	If this property is used in combination with **Major** and **Minor**, all references can be replaced by **Application.ProductVersion**.If this property is used alone, the number can be replaced by **System.Diagnostics.FileVersionInfo.GetVersionInfo(System. Reflection.Assembly.GetExecutingAssembly.Location). FileBuildPart** These calls provide the product version information that is stored in the binary executable file.To get the revision number of an assembly file use **System.Reflection.AssemblyName.GetAssemblyName(System. Reflection.Assembly.GetExecutingAssembly.Location).Version. Revision**. **Note:** The format for version numbers is different in Visual Basic .NET.
StartLogging method	Logging in to Visual Basic .NET is handled by way of event logs. You can replace functionality by using the class **System.Diagnostics.EventLog**.
StartMode	In Visual Basic 6.0, this property was used to start an application as an ActiveX component. ActiveX component creation is not supported in Visual Studio .NET. To simulate this behavior, you can add an additional optional parameter to the constructor of the main form, and use 0 as default value when the application is loaded directly by the user.

Visual Basic 6.0	Visual Basic .NET Equivalent
TaskVisible	To simulate this behavior, you can use **System.Windows.Forms.Form.ShowInTaskBar**, or you can change your project to a Windows service or a console application that will not show up in the task list.
ThreadID	The threading model is different in Visual Basic .NET, but this property can be replaced by the method **AppDomain.GetCurrentThreadId()**.
Title	**System.Reflection.Assembly.GetExecutingAssembly.GetName.Name** There is no mapping for **Set**. In Visual Basic .NET, the name that is displayed in the Microsoft Windows Task List is the **Text** property of the form.
UnattendedApp	References to this property should be removed. For unattended applications in Visual Basic .NET, choose a Console Application project.

For Visual Basic 2005:

Visual Studio 2005 includes the new **My** namespace that provides a full-featured set of members and services designed to make the most common programming scenarios the easiest to implement. Additional support for the Visual Basic 6.0 **App** object can be obtained through the **My.Application** namespace. The upgrade wizard included with this version of Visual Studio .NET will automatically upgrade all applicable **App** members, including **CompanyName**, **EXEName**, **FileDescription**, **LegalCopyright**, **LegalTrademarks**, **LogEvent**, **LogPath**, **Major**, and **Minor** among others.

Additionally, the **My** namespace exposes functionality to handle computer and resource objects and functions. The corresponding classes are included in **My.Computer** and **My.Resources**.

For example, one of the conversions that the Visual Basic Upgrade Wizard 2005 will perform is shown here.

The following is the original Visual Basic 6.0 code.

```
…
labelText = LoadResString(1)
…
```

The following is the code obtained with the Visual Basic Upgrade Wizard 2003.

```
…
labelText = VB6.LoadResString(1)
…
```

The following is the code obtained with the Visual Basic Upgrade Wizard 2005.

```
…
labelText = My.Resources.str1
…
```

Upgrading the Screen Object

Similar to the **App** object, the Visual Basic 6.0 **Screen** object provides properties and methods for retrieving, and in some cases setting, global application properties. Using the **Screen** object, you can set the **MousePointer** property to display an hourglass pointer when a modal form is visible or retrieve the active control or active form of the application. Although there is not a direct equivalent for the **Screen** object in Visual Basic .NET, most of the properties can be mapped to equivalent properties in the .NET Framework.

Consider the following Visual Basic 6.0 example in which the **Screen** object is used to change the mouse pointer, maximize the active form, and change the text in a specified **Textbox** control when that control is active.

```
Screen.MousePointer = vbHourglass
Screen.ActiveForm.WindowState = vbMaximized
If Screen.ActiveControl.Name = "Command1" Then
    Screen.ActiveControl.Caption = "Maximized"
End If
Screen.MousePointer = vbDefault
```

When the upgrade wizard is applied, some properties of the **Screen** code are left unchanged and marked with upgrade warnings, as shown here.

```
' UPGRADE_WARNING: Screen property Screen.MousePointer
' has a new behavior.
System.Windows.Forms.Cursor.Current = System.Windows.Forms.Cursors.WaitCursor
System.Windows.Forms.Form.ActiveForm.WindowState =
System.Windows.Forms.FormWindowState.Maximized
' UPGRADE_ISSUE: Control Name could not be resolved because
' it was within the generic namespace ActiveControl.
If VB6.GetActiveControl().Name = "txtState" Then
    ' UPGRADE_ISSUE: Control Caption could not be resolved because
    ' it was within the generic namespace ActiveControl.
    VB6.GetActiveControl().Text = "Maximized"
End If
' UPGRADE_WARNING: Screen property Screen.MousePointer has a
' new behavior.
System.Windows.Forms.Cursor.Current = System.Windows.Forms.Cursors.Default
```

Despite the number of issues identified, modifying the upgraded code to correct the issues is not difficult. Some of the identified warnings can be removed and some instructions can be replaced by properties of Microsoft .NET Framework. For example, the **VB6.GetActiveControl** reference can be replaced with **Me.ActiveControl**. This property is defined in the **System.Windows.Forms.ContainerControl** class, which is the ancestor of the .NET **Form** class. The property can be accessed through **Me** or

System.Windows.Forms.Form.ActiveForm, which return a **Form** object. Here is the modified code.

```
System.Windows.Forms.Cursor.Current = System.Windows.Forms.Cursors.WaitCursor
System.Windows.Forms.Form.ActiveForm.WindowState = _
    System.Windows.Forms.FormWindowState.Maximized

If Me.ActiveControl.Name = "txtState" Then
    Me.ActiveControl.Text = "Maximized"
End If
System.Windows.Forms.Cursor.Current = System.Windows.Forms.Cursors.Default
```

For more information about the changes to the **Screen** object, see "Screen Object Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN.

Table 7.2 is a comprehensive list of the properties of **Screen** object and their equivalent features in Visual Basic .NET.

Table 7.2: Visual Basic .NET Equivalents for Screen Object Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
ActiveControl	System.Windows.Forms.Form.ActiveForm.ActiveControl
ActiveForm	System.Windows.Forms.Form.ActiveForm
FontCount	System.Drawing.FontFamily.Families.Length
Fonts	System.Drawing.FontFamily.Families
Height	System.Windows.Forms.Screen.PrimaryScreen.Bounds.Height

MouseIcon	This feature must be upgraded manually by declaring a **Cursor** object and initializing it with the **MouseIcon**'s path.
	This is demonstrated in the following sample code:
	
	`' Visual Basic 6.0Screen.MouseIcon = _`
	`  LoadPicture("C:\WINDOWS\Cursors\3dwns.cur")`
	
	`' Visual Basic .NET`
	`Me.Cursor = New Cursor("C:\WINDOWS\Cursors\3dwns.cur")`
	
	For more information, see the section "Dealing with Changes to the MousePointers Property" in Chapter 9, "Upgrading Visual Basic 6.0 Forms Features."

MousePointer	System.Windows.Forms.Form.ActiveForm.Cursor.Current
TwipsPerPixelX	VB6.TwipsPerPixelX
TwipsPerPixelY	VB6.TwipsPerPixelY
Width	**System.Windows.Forms.Screen.PrimaryScreen.Bounds.Width**

Upgrading the Printer Object

In Visual Basic 6.0, the **Printer** object enables applications to communicate with a system printer. The **Printer** object contains functions for printing text, lines, and images. However, the printing model in the .NET Framework is quite different from that in Visual Basic 6.0. The **Printer** object has become obsolete. Fortunately, the functionality of the **Printer** object can be achieved with the classes that are provided in the **System.Drawing.Printing** namespace, particularly the **PrintDocument** class.

There are two main options to upgrade the Visual Basic 6.0 printing functionality. The first option is to replace the **Printer** object members with equivalent functionality provided by the **PrintDocument** class. Typically, this requires reimplementation of the functionality; this will be demonstrated in the next code example. The second option is to create your own **Printer** class in Visual Basic .NET based on the .NET **PrintDocument** class. This approach will allow you to mask the .NET **PrintDocument** class functionality with the names provided in the Visual Basic 6.0 **Printer** object. This second option will also be demonstrated later in this section.

Applying the first approach will require reimplementation of the printing code. The following Visual Basic 6.0 code example prints four lines of text with different font settings:

```
Private Sub Form_Load()
    With Printer
        Printer.Print "Normal Line"
        .FontBold = True
        Printer.Print "Bold Line"
        .FontItalic = True
        Printer.Print "Bold and Italic Line"
        .FontBold = False
        .FontItalic = False
        Printer.Print "Second Normal Line"
        .EndDoc
    End With
End Sub
```

This code can be reimplemented in Visual Basic .NET using methods of the **PrintDocument** class, as shown here.

```
Dim WithEvents prn As New Printing.PrintDocument

Private Sub Form1_Load(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    prn.Print()
End Sub

Private Sub prn_PrintPage(ByVal sender As Object, _
ByVal e As System.Drawing.Printing.PrintPageEventArgs) Handles prn.PrintPage
    Dim currFont As Font
```

```
    Dim yPos As Integer
    yPos = 1
    With e.Graphics
        currFont = New Font("Arial", 10, FontStyle.Regular)
        .DrawString("Normal Line", currFont, Brushes.Black, 1, yPos)
        yPos = yPos + currFont.GetHeight

        currFont = New Font("Arial", 10, FontStyle.Bold)
        .DrawString("Bold Line", currFont, Brushes.Black, 1, yPos)
        yPos = yPos + currFont.GetHeight

        currFont = New Font("Arial", 10, FontStyle.Bold Or FontStyle.Italic)
        .DrawString("Bold and Italic Line", currFont, Brushes.Black, 1, yPos)
        yPos = yPos + currFont.GetHeight

        currFont = New Font("Arial", 10, FontStyle.Regular)
        .DrawString("Second Normal Line", currFont, Brushes.Black, 1, yPos)
    End With
End Sub
```

Notice how the structure of the code has changed and how the quantity of drawing methods has increased. The .NET Framework provides more control and power over the drawing operations, but from the upgrade point of view, it has a learning curve, and the manual reimplementation of all the printing functionality can consume extensive resources.

For more information about the equivalent functionality for the **Printer** object members, see Tables 7.3 and 7.4 at the end of this section. For further guidance on printing from Visual Basic .NET with the **PrintDocument** component, see "Printing with the PrintDocument Component" in on MSDN.

The second upgrade approach involves the creation of your own **Printer** class. This allows you to consolidate several drawing methods and collections of graphical objects (such as **Circle** or **Line**) and to store the coordinates that will be used to print these objects when the **Print** method or the **EndDoc** method is called. When the **Print** method of the **PrintDocument** class is called, the **PrintPage** event is raised. This event can be used to signal the application to draw all the objects and text stored in the collections of your **PrinterClass**. The following Visual Basic .NET code provides a small demonstration of how this **Printer** class can be developed.

```
Imports Microsoft.VisualBasic.Compatibility
' Imports Microsoft.VisualBasic.
Public Class PrinterClass
    Public Enum FillStyleConstants As Short
        vbFSSolid = 0
        vbFSTransparent = 1
    End Enum
    ' Scale mode Constants
    Public Enum ScaleModeConstants As Short
        vbTwips = 1
```

```
    vbPixels = 3
End Enum
Private Structure LineInfo
    Dim pen As System.Drawing.Pen
    Dim p1, p2 As Drawing.Point
End Structure
Private Structure RectangleInfo
    Dim pen As System.Drawing.Pen
    Dim rec As Drawing.Rectangle
    Dim FillStyle As FillStyleConstants
    Dim FillColor As System.Drawing.Color
End Structure
Private Structure PageInfo
    Public Lines() As LineInfo
    Public Circles() As RectangleInfo
End Structure
Private Pages() As PageInfo
Private PageIndex = 0
Private WithEvents InnerPrinter As New System.Drawing.Printing.PrintDocument
Private objPen As New System.Drawing.Pen(System.Drawing.Brushes.Black)
Private intCurrentX As Double = 20
Private intCurrentY As Double = 20
Public ScaleMode As ScaleModeConstants = ScaleModeConstants.vbTwips
Public FillColor As Drawing.Color
Public FillStyle As FillStyleConstants = FillStyleConstants.vbFSTransparent

Public Sub New()
    ReDim Pages(0)
    Pages(0) = New PageInfo
End Sub

' Terminates a print operation sent to the Printer object,
' releasing the document to the print device or spooler.
Public Sub EndDoc()
    Dim j As Integer
    For j = 0 To Pages.Length - 1
        PageIndex = j
        InnerPrinter.Print()
    Next
End Sub
Public Sub Circle(ByVal P As Drawing.Point, ByVal radius As Double, _
                  Optional ByVal [Step] As Boolean = False)
    Circle(P, radius, objPen.Color, [Step])
End Sub
Public Sub Circle(ByVal P As Drawing.Point, ByVal radius As Double, _
                ByVal Color As Drawing.Color, _
                Optional ByVal [Step] As Boolean = False)

    Dim diameter As Double

    If [Step] = True Then
        P.X = P.X + CurrentX
        P.Y = P.Y + CurrentY
    End If
```

```
        diameter = radius * 2

        ' Moves the CurrentX and CurrentY properties
        CurrentX = P.X + radius
        CurrentY = P.Y + radius

        P.X = ConvertToPixelsX(P.X)
        P.Y = ConvertToPixelsY(P.Y)
        diameter = ConvertToPixelsX(diameter)

        If IsNothing(Pages(PageIndex).Circles) Then
            ReDim Preserve Pages(PageIndex).Circles(0)
        Else
            ReDim Preserve
Pages(PageIndex).Circles(Pages(PageIndex).Circles.Length)
        End If

        Pages(PageIndex).Circles(Pages(PageIndex).Circles.Length - 1).rec = _
            New Drawing.Rectangle(P.X, P.Y, diameter, diameter)
        Pages(PageIndex).Circles(Pages(PageIndex).Circles.Length - 1).pen = objPen
        Pages(PageIndex).Circles(Pages(PageIndex).Circles.Length - 1).FillColor =
_
            FillColor
        Pages(PageIndex).Circles(Pages(PageIndex).Circles.Length - 1).FillStyle =
_
            FillStyle
    End Sub
    Private Sub Printer_PrintPage(ByVal sender As Object, _
                ByVal e As System.Drawing.Printing.PrintPageEventArgs) _
                Handles InnerPrinter.PrintPage
        Dim i As Integer
        ' Draw all circles
        If IsNothing(Pages(PageIndex).Circles) = False Then
            For i = 0 To Pages(PageIndex).Circles.Length - 1
                Dim x, y As Integer
                e.Graphics.DrawEllipse(Pages(PageIndex).Circles(i).pen, _
                    Pages(PageIndex).Circles(i).rec)
                If Pages(PageIndex).Circles(i).FillStyle = _
                         FillStyleConstants.vbFSSolid Then
                    e.Graphics.FillEllipse( _
                        New Drawing.SolidBrush( _
                        Pages(PageIndex).Circles(i).FillColor), _
                        Pages(PageIndex).Circles(i).rec)
                End If
            Next
        End If
    End Sub
    Public Property CurrentX() As Double
        Get
            Return ConvertToPixelsX(intCurrentX)
        End Get
        Set(ByVal Value As Double)
            intCurrentX = ConvertToPixelsX(Value)
        End Set
```

```
    End Property
    Public Property CurrentY() As Double
        Get
            Return ConvertToPixelsY(intCurrentY)
        End Get
        Set(ByVal Value As Double)
            intCurrentY = ConvertToPixelsY(Value)
        End Set
    End Property
    Public Function ConvertToPixelsX(ByVal num As Double) As Double
        Return IIf(ScaleMode = ScaleModeConstants.vbTwips, _
                   VB6.TwipsToPixelsX(num), num)
    End Function

    Public Function ConvertToPixelsY(ByVal num As Double) As Double
        Return IIf(ScaleMode = ScaleModeConstants.vbTwips, _
                   VB6.TwipsToPixelsY(num), num)
    End Function
End Class
```

Tables 7.3 and 7.4 list Visual Basic 6.0 **Printer** object properties and methods and their Visual Basic .NET equivalents. Where there are no direct equivalents, links are provided to additional information.

The following example shows you how to convert Visual Basic 6.0 code that uses a printer object to Visual Basic .NET code that uses your own printer class as mentioned earlier.

```
Printer.FillColor = vbBlue
Printer.FillStyle = vbSolid
Printer.Circle (3000, 3000), 1500, vbRed
Printer.EndDoc
```

The preceding Visual Basic 6.0 code will print a circle filled with blue color and it will have a red border. To convert this code to Visual Basic .NET, you can use the printer class mentioned earlier and perform the following steps.

First, you have to add this printer class to the .NET project. After this, you have to add a reference to the **Micosoft.VisualBasic.Compatibility** assembly.

The next step is to create a Visual Basic .NET module and add the following code.

```
Module Module1
    Public Printer As PrinterClass = New PrinterClass
End Module
```

Finally, you need to perform some small changes to the original Visual Basic 6.0 code to make it consistent with the new definition of your own printer class. Here is the resulting Visual Basic .NET code.

```
Printer.FillColor = Color.Blue
Printer.FillStyle = PrinterClass.FillStyleConstants.vbFSSolid
Printer.Circle(New Point(3000, 3000), 1500, Color.Red)
Printer.EndDoc()
```

Table 7.3: Visual Basic .NET Equivalents for Printer Object Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
ColorMode	PrintDocument.PrinterSettings.DefaultPageSettings.Color
Copies	PrintDocument.PrinterSettings.Copies
CurrentX	No equivalent. Replaced by location and dimension arguments of various methods of the **Graphics** class.This property could be simulated in your printer class with a double variable.
CurrentY	No equivalent. Replaced by location and dimension arguments of various methods of the **Graphics** class.This property could be simulated in your printer class with a double variable.
DeviceName	PrintDocument.PrinterSettings.PrinterName
DrawMode	No equivalent. For details, see "Graphics Changes in Visual Basic .NET" on MSDN.
DrawStyle	System.Drawing.Drawing2D.DashStyle
DrawWidth	System.Drawing.Pen.Width
DriverName	No equivalent. No longer needed; printer drivers are managed by Windows.
Duplex	System.Drawing.Printing.Duplex
FillColor	No equivalent. For details, see "Graphics Changes in Visual Basic .NET" on MSDN.This property could be simulated in your printer class with a **System.Drawing.Color** variable.
FillStyle	This functionality can be obtained using a **Drawing.SolidBrush** to fill the **Draws** objects.
Font	This property could be simulated in your printer class with a **System.Drawing.Font** class.
FontBold	To get the value, use: **System.Drawing.Font.Bold**.To set the value, use: **VB6.FontChangeBold**.
FontCount	System.Drawing.FontFamily.GetFamilies().Length
FontItalic	To get the value, use: **System.Drawing.Font.Italic**.To set the value, use: **VB6.FontChangeItalic**.
FontName	To get the value, use: **System.Drawing.Font.Name**.To set the value, use: **VB6.FontChangeName**.
Fonts	System.Drawing.FontFamily.GetFamilies
FontSize	To get the value, use: **System.Drawing.Font.Size**.To set the value, use: **VB6.FontChangeSize**.
FontStrikethru	To get the value, use: **System.Drawing.Font.Strikeout**.To set the value, use: **VB6.FontChangeStrikeout**.

continued

Visual Basic 6.0	Visual Basic .NET Equivalent
FontTransparent	No equivalent. For details, see "Font Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN.
FontUnderline	To get the value, use: **System.Drawing.Font.Underline**.To set the value, use: **VB6.FontChangeUnderline**.
ForeColor	System.Drawing.Pen.Color
hDC	PrintDocument.PrinterSettings.GetHdevmode.ToInt32
Height	PrintDocument.DefaultPageSettings.PaperSize.Height
Orientation	PrintDocument.DefaultPageSettings.Landscape
Page	No direct equivalent. The current page number is not tracked; however, you can easily do this by setting a variable in the **BeginPrint** event and incrementing it in the **PrintPage** event.
PaperBin	PrintDocument.PrinterSettings.PaperSources
PaperSize	PrintDocument.DefaultPageSettings.PaperSize
Port	No longer necessary. The **PrintPreviewDialog** control automatically sets port information.
PrintQuality	PrintDocument.DefaultPageSettings.PrinterResolution
RightToLeft	No longer necessary. The direction of printing is controlled by the localization settings in Windows.
ScaleHeight	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN.
ScaleLeft	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" on MSDN.
ScaleMode	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" on MSDN.
ScaleTop	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" on MSDN.
ScaleWidth	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" on MSDN.
TrackDefault	No direct equivalent. The **IsDefaultPrinter** property of the **PrinterSettings** class can be used to determine whether a printer is the default, but printing is no longer halted if the default printer changes.
TwipsPerPixelX	No longer necessary. Measurements in Visual Basic .NET are always in pixels.
TwipsPerPixelY	No longer necessary. Measurements in Visual Basic .NET are always in pixels.
Width	PrintDocument.DefaultPageSettings.PaperSize.Height
Zoom	No longer necessary. If the printer has zoom capabilities, settings are automatically exposed in the **Print** dialog box.

Table 7.4: Visual Basic .NET equivalents for Printer object methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Circle	PrintPageEvents.Graphics.DrawEllipse
EndDoc	PrintDocument.Print
KillDoc	PrintEventArgs.Cancel
Line	PrintPageEvents.Graphics.DrawLine
NewPage	This method needs to be simulated with an array of pages and call the method **Print** of the class **PrintDocument** for each page.
PaintPicture	PrintPageEvents.Graphics.DrawImage
Print	PrintPageEvents.Graphics.DrawString
PSet	PrintPageEvents.Graphics.DrawLine
Scale	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN.
ScaleX	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" on MSDN.
ScaleY	No equivalent. For details, see "Coordinate System Changes in Visual Basic .NET" on MSDN.
TextHeight	System.Drawing.Graphics.MeasureString
TextWidth	**System.Drawing.Graphics.MeasureString**

Upgrading the Printers Collection

Visual Basic 6.0 provides the **Printers** collection, which is used to return information about available printers on a system. For example, it is possible to determine the printer layout for a particular printer. Visual Basic .NET does not support the **Printers** collection. The printing model has changed, and as a result, references to the **Printers** collection require manual adjustment when upgrading to Visual Basic .NET.

Consider the following Visual Basic 6.0 example, which uses the **Printers** collection to find the first available printer with its page orientation set to **Portrait**. If such a printer is found in the collection, a message box showing the printer device name is displayed. The code for the example follows.

```
Dim prn As Printer
For Each prn In Printers
   If prn.Orientation = vbPRORPortrait Then
      MsgBox prn.DeviceName
      ' Stop looking for a printer.
      Exit For
   End If
Next
```

Because the **Printers** collection is not supported in Visual Basic .NET, this code cannot be automatically upgraded with the upgrade wizard. Applying the upgrade wizard results in the code being marked with upgrade issues, such as the one demonstrated here.

```
' UPGRADE_ISSUE: Printers collection was not upgraded.
For Each prn In Printers
```

The only way to achieve a similar effect in Visual Basic .NET is to implement your own **Printers** collection, using various classes the Microsoft .NET Framework provides. For example, you can create a new Visual Basic .NET module and class like the following.

```
Module PrintersModule
   Public Printers As New PrintersCollection
End Module

Public Class PrintersCollection : Implements IEnumerable
   Private printer As System.Drawing.Printing.PrinterSettings

   Public ReadOnly Property Count() As Integer
       Get
            Return System.Drawing.Printing.PrinterSettings.InstalledPrinters.Count
       End Get
   End Property

   Public ReadOnly Property Item(ByVal Index As Integer) _
                               As System.Drawing.Printing.PrinterSettings
       Get
            printer = New System.Drawing.Printing.PrinterSettings
            printer.PrinterName = _
               System.Drawing.Printing.PrinterSettings.InstalledPrinters.Item(Index)
            Return printer
       End Get
   End Property
   Overridable Function GetEnumerator() As IEnumerator _
                                    Implements  IEnumerable.GetEnumerator
      Dim Count As Integer = _
           System.Drawing.Printing.PrinterSettings.InstalledPrinters.Count
       Dim printersArray(Count) As System.Drawing.Printing.PrinterSettings
      Dim i As Integer
      For i = 0 To Count - 1
           printersArray(i) = New System.Drawing.Printing.PrinterSettings
           printersArray(i).PrinterName = _
                System.Drawing.Printing.PrinterSettings.InstalledPrinters.Item(i)
      Next
       Return printersArray.GetEnumerator()
   End Function
End Class
```

After the preceding module and class are created and added to the upgrade solution, the original code can be upgraded with only a few minor tweaks. These are shown here (changes are highlighted in **bold**).

```
Dim prn As System.Drawing.Printing.PrinterSettings
For Each prn In Printers
   If prn.DefaultPageSettings.Landscape = False Then
      MsgBox(prn.PrinterName)
      Exit For
   End If
Next
```

For more information about replacing the **Printers** collection functionality in Visual Basic .NET, see "Printers Collection Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN.

Upgrading the Forms Collection

The **Forms** collection in Visual Basic 6.0 is a collection of all loaded forms in the project. The most common uses of the **Forms** collection are to determine whether a form is loaded, to iterate through the loaded forms, and to unload a form by name. Visual Basic .NET does not support the **Forms** collection, but as with the other objects discussed in this chapter, it does provide similar functionality.

The following Visual Basic 6.0 example shows how to iterate through the **Forms** collection, determine whether a form called **frmSecurity** is open, and if so, disable the buttons **btnInsert**, **btnUpdate**, and **btnDelete**.

```
Public Sub checkForm()
    Dim privateForm As Form
    Dim tempControl As Control
    For Each privateForm In Forms
        If privateForm.Name = "frmSecurity" Then
            For Each tempControl In privateForm.Controls
                If tempControl.Name = "btnInsert" Or _
                   tempControl.Name = "btnUpdate" Or _
                   tempControl.Name = "btnDelete" Then
                    tempControl.Enabled = False
                End If
            Next
            MsgBox "For security reasons, this form can not be shown"
        End If
    Next
End Sub
```

The **Forms** collection cannot be automatically upgraded with the upgrade wizard because Visual Basic .NET does not support the **Forms** collection. As with the previous examples, attempting to use the upgrade wizard with code like the example will generate upgrade issue comments like the following.

```
' UPGRADE_ISSUE: Forms collection was not upgraded.
For Each privateForm In Forms
```

The same approach for solving the **Printers** collection issues can also be applied to the **Forms** collection issues. Create a module and class masking the Visual Basic .NET functionality, as shown earlier in the **Printers** example. The code might look like the following.

```
Module FormsCollection
   Public Forms As New FormsCollectionClass()
End Module

Class FormsCollectionClass : Implements IEnumerable
   Private collec As New Collection()
   Sub Add(ByVal tempForm As Form)
      collec.Add(tempForm)
   End Sub
   Sub Remove(ByVal tempForm As Form)
      Dim itemCount As Integer
      For itemCount = 1 To collec.Count
         If tempForm Is collec.Item(itemCount) Then
            collec.Remove(itemCount)
            Exit For
         End If
      Next
   End Sub
   ReadOnly Property Item(ByVal index) As Form
      Get
         Return collec.Item(index)
      End Get
   End Property
   Overridable Function GetEnumerator() As _
      IEnumerator Implements IEnumerable.GetEnumerator
      Return collec.GetEnumerator
   End Function
End Class
```

The **FormsCollectionClass** will store all the loaded forms in a collection. Note that you need to make sure that each form is added to the collection when it is created and removed from the collection after it is released. For example, in the **New** event for **frmSecurity**, you can add it to the collection by adding the following line of code.

```
Forms.Add(Me)
```

Similarly, in the **Disposed** event, you can remove **frmSecurity** from the collection by adding the following line of code.

```
Forms.Remove(Me)
```

The **Add** and **Remove** lines must be added to the **New** and **Disposed** events for every form in the solution.

After the new module and class are added to the upgrade solution, the original **checkForm** subroutine code can be used without requiring any changes at all.

For Visual Basic 2005:

This solution will be obsolete in the new Visual Basic 2005, because the **Forms** collection is implemented in the **My.Application.OpenForms** class. This will make it possible for an automatic upgrade to be performed, with no need to create wrapper classes or modules with global members as shown in this example.

Upgraded **Forms** collection code will look very similar to the original code, as shown here.

```
Public Sub checkForm()
   Dim privateForm As System.Windows.Forms.Form
   Dim tempControl As System.Windows.Forms.Control
   For Each privateForm In My.Application.OpenForms
      If privateForm.Name = "frmSecurity" Then
         For Each tempControl In privateForm.Controls
            If tempControl.Name = "btnInsert" Or _
               tempControl.Name = "btnUpdate" Or tempControl.Name = "btnDelete"
Then
               tempControl.Enabled = False
            End If
         Next tempControl
         MsgBox("For security this form can not be showed")
      End If
   Next privateForm
End Sub
```

Upgrading the Clipboard Object

Visual Basic 6.0 has a **Clipboard** object that allows you to store and retrieve text and graphics to and from the Clipboard. In Visual Basic .NET, you manipulate the **Clipboard** using the **System.Windows.Forms.Clipboard** namespace. The new **Clipboard** classes are more flexible than those in Visual Basic 6.0 in that they allow you to set and retrieve data in a particular format and query the contents of the **Clipboard** object to see what formats are supported. The new flexibility comes at a cost; Visual Basic 6.0 **Clipboard** object code cannot be automatically upgraded.

However, you will find it straightforward to implement the same functionality in Visual Basic .NET.

For example, suppose you have a Visual Basic 6.0 form with one **TextBox** control on it named **Text1**. The following code sets and retrieves the text "Hello" to and from the clipboard, using the **vbCFText** constant to indicate that only plaintext format is applied.

```
Clipboard.Clear
Clipboard.SetText "Hello", vbCFText
If Clipboard.GetFormat(vbCFText) Then
   Text1.Text = Clipboard.GetText(vbCFText)
End If
```

Using the upgrade wizard to upgrade this code causes the **Clipboard** object code to be left unchanged but marked with upgrade warnings, as shown here.

```
' UPGRADE_ISSUE: Clipboard method Clipboard.Clear was not upgraded.
```

The upgraded code causes compile errors in Visual Basic .NET, because the **Clipboard** object functions are not supported in Visual Basic .NET. Correcting the issues requires deleting the **Clipboard** object code that was not upgraded and replacing it with Visual Basic .NET functionality. The new Visual Basic .NET version of the example is shown here.

```
Dim datobj As New System.Windows.Forms.DataObject

datobj.SetData("")
datobj.SetData System.Windows.Forms.DataFormats.Text, "hello"
System.Windows.Forms.Clipboard.SetDataObject(datobj)

If System.Windows.Forms.Clipboard.GetDataObject.GetDataPresent( _
   System.Windows.Forms.DataFormats.Text) Then
   Text1.Text = System.Windows.Forms.Clipboard.GetDataObject.GetData( _
   System.Windows.Forms.DataFormats.Text)
End If
```

The remaining **Clipboard** object functions and the equivalent functionality in Visual Basic .NET are shown in Table 7.5. The **DataFormat** parameter should be replaced with the appropriate format, such as **DataFormats.Text** or **DataFormats.Rtf**.

Rewriting the **Clipboard** code with Visual Basic .NET functionality requires you to know the content's format when it is moved to the Clipboard, such as plaintext as used in this example. Table 7.6 lists the **Clipboard** class's constants and each equivalent feature in Visual Basic .NET.

Table 7.5: Visual Basic .NET Clipboard Methods Equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
Clear method	Clipboard.SetDataObject("")
Clipboard.GetData	Clipboard.GetDataObject.GetData(*DataFormat*, True)
GetFormat	Clipboard.GetDataObject.GetDataPresent(*DataFormat*)
GetText	Clipboard.GetDataObject.GetData(*DataFormat*, True)
SetData	Dim datobj As New System.Windows.Forms.DataObjectdatobj.SetData(*DataFormat*, *data*)System.Windows.Forms.Clipboard.SetDataObject(*datobj*)
SetText	**Dim datobj As New System.Windows.Forms.DataObjectdatobj.SetData(*DataFormat*, *data*)System.Windows.Forms.Clipboard.SetDataObject(*datobj*)**

Table 7.6: Visual Basic .NET Clipboard Constant Equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
vbCFBitmap	System.Windows.Forms.DataFormats.Bitmap
vbCFDIB	System.Windows.Forms.DataFormats.DIB
vbCFEMetafile	System.Windows.Forms.DataFormats.EnhancedMetafile
vbCFFiles	System.Windows.Forms.DataFormats.FileDrop
vbCFLink	No equivalent. For more information about the **Clipboard** object constant **vbCFLink**, see "Dynamic Data Exchange Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN.
vbCFMetafile	System.Windows.Forms.DataFormats.MetafilePict
vbCFPalette	System.Windows.Forms.DataFormats.Palette
vbCFRTF	System.Windows.Forms.DataFormats.Rtf
vbCFText	**System.Windows.Forms.DataFormats.Text**

For Visual Basic 2005:

My.Computer is one of the top level groupings under the **My** namespace provided with Visual Studio .NET 2005. **My.Computer** provides access to instances of the most commonly used objects of the .NET Framework that relate to items in the computer running the application. One of these items is the Clipboard. The Visual Basic Upgrade Wizard 2005 will automatically upgrade the Visual Basic 6.0 Clipboard members to **My.Computer.Clipboard** members.

Upgrading the Licenses Collection

Visual Basic 6.0 allows you to dynamically load ActiveX controls when an application is running. Some ActiveX controls require the use of a license for them to be used within a program. The **Licenses** collection of an application contains the license information for all loaded ActiveX controls and allows you to add and remove each license as needed. The licensing model in Visual Basic .NET is completely different, so upgrading applications that use the **Licenses** collection requires some effort.

The following code example demonstrates a typical use of the **Licenses** collection in Visual Basic 6.0, where the license for the control **Customers.ListCustomer** is checked before this component becomes visible. If the license is not set, a message is displayed.

```
On Error GoTo NoLicense
    Dim customer As VBControlExtender
    Licenses.Add "Customers.ListCustomer"
    Set customer = Me.Controls.Add("Customers.ListCustomer", "ListCustomer")
    customer.Visible = True
    Exit Sub

NoLicense:
    'make sure the error is about the license for this control.
    If Err.Number = 731 Then
      MsgBox "You need a License to dynamically load this ActiveX
Customers.ListCustomer control"
    End If
```

However, Visual Basic .NET compiles the license directly into the executable. That means that the control must already be present in the project before it can be dynamically added at run time. Thus, license management cannot be automatically upgraded, and an alternative strategy must be applied in Visual Basic .NET.

One way to do this is to add a dummy form to the project and put all ActiveX controls that will be dynamically added to this form. After you create this dummy form, you can dynamically add the ActiveX control to any form in your project. The resulting Visual Basic .NET code might look like the following.

```
On Error GoTo NoLicense
    Dim customer As System.Windows.Forms.AxHost
    customer = New Customers.ListCustomer
    Me.Controls.Add("Customers.ListCustomer", "ListCustomer")
    customer.Visible = True
    Exit Sub

NoLicense:
    ' Make sure the error is about the license for this control.
    If Err.Number = 731 Then
        MsgBox("You need a License to dynamically load this ActiveX
```

```
Customers.ListCustomer control")
    End If
```

Note that the license management has been removed from the code. It is no longer necessary.

Upgrading the Controls Collection

To add and remove controls at run time, Visual Basic 6.0 provides the **Controls** collection. The **Controls** collection is a dynamic collection of controls on a form or container. Through this collection, it is possible to add and remove controls on a form (or container) at run time.

For example, consider the following code that adds a **Label** dynamically to a form at run time.

```
Dim labelControl As Control
Set labelControl = Me.Controls.Add("VB.Label", "labelControl")
labelControl.Visible = True
```

This code adds a **Label** to the form and makes it visible. It is also easy in Visual Basic 6.0 to dynamically remove a control. The following line of code removes the **Label** that was added in the preceding code example.

```
Me.Controls.Remove "labelControl"
```

One of the major disadvantages of the Visual Basic 6.0 model is that the **Controls** collection does not support Microsoft IntelliSense® technology. You have to remember the methods, parameters, and **ProgID** of the control to add.

The upgrade wizard does not automatically upgrade Visual Basic 6.0 code using the **Controls** collection **Add** method because of changes in the behavior of **Add** between Visual Basic 6.0 and Visual Basic .NET. Fortunately, Visual Basic .NET does provide an alternative to achieve the same functionality. It is relatively easy to add and remove intrinsic controls in Visual Basic .NET. The following example demonstrates how to dynamically add a **Label** to a form at run time in Visual Basic .NET.

```
Dim c As Control
c = New Label()
c.Name = "labelControl"
Me.Controls.Add(c)
```

In Windows Forms, the form model used in Visual Basic .NET, controls are indexed by number instead of name. To remove a control by name, you have to iterate through the **Me.Controls** collection, find the desired control, and remove it. The following code shows how to remove the newly added control by name in Visual Basic .NET.

```
Dim tempControl As Control
For Each tempControl In Me.Controls
   If tempControl.Name = "labelControl" Then
      Me.Controls.Remove(tempControl)
      Exit For
   End If
Next
```

For Visual Basic 2005:

In the new Visual Basic 2005, the **Controls** collection will provide methods that allow removing controls in the same manner as Visual Basic 6.0, so you will not have to iterate through the **Controls** collection. For example, it will be possible to replace the preceding iteration example with a single line of code.

```
Me.Controls.RemoveByKey(tempControl)
```

The **Item** indexed property allows the user to access one control in the collection by specifying the numeric index of the control. In Visual Basic 2005, it is also possible to access one control by indicating the name of the control (as a string).

Visual Studio 2005 also adds other new methods to the **ControlCollection** class that help achieve some of the functionality that Visual Basic 6.0 programmers expected from the **ControlArray**. **ControlCollection** is the type of the **Controls** property of the **Control** class. This class now contains a **find()** method that allows you to search a control and all its child controls recursively for a control with a specific name and returns an array with all the matching controls. The other new methods are **RemoveByKey()**, **ContainsKey()**, **IndexOfKey()**, and a new string parameter for the **Item** property. All these new methods provide ways to manipulate the **ControlCollection** using the names of the controls instead of their index. For more information, see "Control.ControlCollection Class (System.Windows.Forms)" in the .NET Framework Class Library on MSDN.

For more information about the new **Controls** collection available in Visual Basic 2005, see "Controls Collection for Visual Basic 6.0 Users" in Visual Basic Concepts on MSDN. Please note that at the time of this writing, the content contained in this documentation refers to a pre-release version of Visual Basic 2005 and is subject to change.

Dynamically adding ActiveX controls at run time requires a bit more work. Visual Basic .NET creates wrappers for ActiveX controls. These wrappers must exist before a control can be added. In addition, most ActiveX controls have design-time licenses that must be present before the control can be created on the form. Visual Basic .NET compiles the license into the executable. These factors mean that the control must already be present in the project before it can be dynamically added at run time. One way to do this is to add a dummy form to the project and put all ActiveX controls that will be dynamically added to this form. After you create this dummy form, you can dynamically add the ActiveX control to any form in your project. The following code shows how to dynamically add a **Windows Common Controls TreeView** control to a form (assuming the project already has a dummy form with a **TreeView** added).

```
Dim activexControl As New AxMSComctlLib.AxTreeView()
activexControl.Name = "TreeViewUsuers"
Me.Controls.Add(activexControl)
```

After the control is added, it can be dynamically removed in a manner similar to removing an intrinsic control. Only adding requires additional work.

It is important to consider that there is a behavior difference between the Visual Basic 6.0 and Visual Basic .NET **Controls** collection. In Visual Basic 6.0, this collection contains all the controls that have been added to a particular **Form**, including controls that were added to other container controls such as **PictureBox** or **Frame** controls. In Visual Basic .NET, the **Controls** collection includes only the controls that were directly added to the control that owns the collection; for example, if a **Form** contains a **Panel** control that contains a **Button** control named **button1**, **button1** will not be contained in the **Controls** collection that belongs to the **Form**; instead of that, **button1** will be contained in the panel's **Controls** collection. As a consequence, iterating through the elements in a form's **Controls** collection will not consider all the controls contained in the form, as it did in Visual Basic 6.0. For more information about the **Controls** collection behavior differences, see "Getting Back Your Visual Basic 6.0 Goodies" on MSDN.

The following code example presents a typical **For Each** loop that will perform an operation on all the controls in a Visual Basic 6.0 form.

```
Function ListOfControlNames() As String
    Dim res As String
    For Each c In Me.Controls
        res = res & c.Name & " "
    Next
    ListOfControlNames = res
End Function
```

If the form has the layout illustrated in Figure 7.1, the output of the previous function will be "Frame1 Command1."

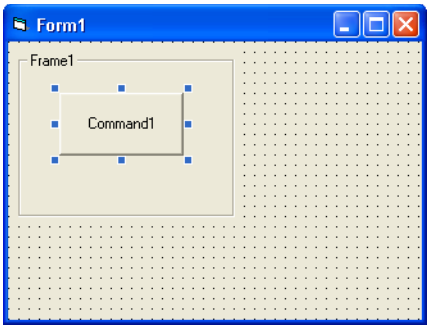

Figure 7.1
Form with nested controls

After the **ListOfControlNames** function is upgraded, the following code will be the result.

```
Function ListOfControlNames() As String
    Dim c As System.Windows.Forms.Control
    Dim res As String
    For Each c In Me.Controls
        res = res & c.Name & " "
    Next c
    ListOfControlNames = res
End Function
```

In Visual Basic .NET, the output of the function will be "Frame1." There is a clear difference with the result obtained in Visual Basic 6.0. Correcting this difference would require the definition of two new functions that recursively obtain the controls contained in Form1. The two new functions are shown here.

```
Function MyControls() As ArrayList
    Dim res As New ArrayList
    GetAllControls(Me, res)
    Return res
End Function
Function GetAllControls(ByVal c As Control, ByVal res As ArrayList)
    Dim curControl As Control
    For Each curControl In c.Controls
        res.Add(curControl)
        GetAllControls(curControl, res)
    Next
End Function
```

The **For Each** loop in the **ListOfControlNames** function must be changed to use the **MyControls** function instead of **Me.Controls**. It is important to notice that the order in which the controls are processed may change because of the recursive way in which the controls are obtained, if your code depends on the order in which the controls are obtained, it will require adjustments to be functionally equivalent to the Visual Basic 6.0 code.

Summary

Changes in Visual Basic .NET have made several objects from previous versions of the language obsolete. The compatibility library is provided to help minimize the effort in upgrading some of these objects. For objects that are not in the compatibility library, Visual Basic .NET almost always offers the same functionality through new objects and functions. This chapter described the most common Visual Basic 6.0 objects that are now obsolete, and how to replace these features in Visual Basic .NET.

Changes to objects are not the only changes that have taken place between Visual Basic 6.0 and Visual Basic .NET. There are other language features that have also changed or become obsolete. The next chapter discusses how to upgrade some of the more commonly-used language features that cannot be automatically upgraded by the upgrade wizard.

More Information

Robinson, Ed, Robert Ian Oliver, and Michael Bond. *Upgrading Microsoft Visual Basic 6.0 To Microsoft Visual Basic .NET*, Redmond: Microsoft Press, 2001, ISBN: 073561587X. Also available on MSDN: *http://msdn.microsoft.com/vbrun/staythepath/additionalresources/upgradingvb6/.*

For information about obtaining the Visual Basic Upgrade Wizard Companion and additional upgrade tools and services, see the ArtinSoft Web site: *http://www.artinsoft.com/.*

For a comprehensive list of **App** object properties and methods that have equivalent replacements in Visual Basic .NET, see "App Object Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vxconChangesToAppObjectInVisualBasicNET.asp.*

For more information about the changes to the **Screen** object, see "Screen Object Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vxconChangesToScreenObjectInVisualBasicNET.asp.*

For more information about printing from Visual Basic .NET with the **PrintDocument** component, see "Printing with the PrintDocument Component" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconprintingwithprintdocumentcontrol.asp.

For information about details, see "Graphics Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbcongraphicschangesinvisualbasicnet.asp.

For more information about the **Controls** collection behavior differences, see "Getting Back Your Visual Basic 6.0 Goodies" on MSDN:
http://msdn.microsoft.com/vbasic/using/columns/adventures/default.aspx?pull=/library/en-us/dnadvnet/html/vbnet05132003.asp.

For information about finding an equivalent replacement for the Visual Basic 6.0 **FontTransparent** property, see "Font Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconfontchangesinvisualbasic60.asp.

For information about finding an equivalent replacement for the Visual Basic 6.0 **ScaleHeight** property, see "Coordinate System Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconcoordinatesystemchangesinvisualbasicnet.asp.

For more information about replacing the **Printers** collection functionality in Visual Basic .NET, see "Printers Collection Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vxconprinterscollectionchangesinvisualbasicnet.asp.

For more information about Visual Studio 2005 methods to manipulate the **ControlCollection** using the names of the controls instead of their index, see "Control.ControlCollection Class (System.Windows.Forms)" in the .NET Framework Class Library on MSDN:
http://msdn2.microsoft.com/library/f4w0yshb(en-us,vs.80).aspx.

For more information about the new **Controls** collection available in Visual Basic 2005, see "Controls Collection for Visual Basic 6.0 Users" on MSDN:
http://msdn2.microsoft.com/library/7e4daa9c(en-us,vs.80).aspx.

For more information about the **Clipboard** object constant **vbCFLink**, see "Dynamic Data Exchange Changes in Visual Basic .NET" in *Visual Basic Concepts* on MSDN:
http://msdn.microsoft.com/library/en-us/vbcon/html/vbcondynamicdataexchangechangesinvisualbasicnet.asp.

8

Upgrading Commonly-Used Visual Basic 6.0 Language Features

In addition to commonly used objects, other features of the Microsoft Visual Basic 6.0 language have become obsolete and are not directly supported in Visual Basic .NET. However, in most cases Visual Basic .NET has equivalent functionality for unsupported features. This chapter describes the unsupported language features and the alternative functionality you can use to achieve the same effect in Visual Basic .NET.

As you read this chapter, please take particular note of the need for strong quality assurance. This is especially true for all code that is upgraded with the help of the Visual Basic Upgrade Wizard. In some cases, the wizard will successfully create Visual Basic .NET source code files that will compile without error. However, some of the upgraded code may have different behavior than the original code or may generate run-time exceptions that will result in program failures if the problem areas are not fixed. Each discussion will note any such problems you should check for, when appropriate.

Resolving Issues with Default Properties

Default properties in Visual Basic 6.0 allow you to simplify your code by specifying a property that is set or retrieved by using only the object name. Thus, a commonly used property for an object can be referenced in code using only the object name instead of the fully qualified *object.property* name. However, default properties are no longer supported in the Visual Basic .NET.

When using the upgrade wizard to upgrade code with default properties, the wizard will modify code by inserting the property name wherever a default property is used. This happens only if the wizard is able to determine which property is the

default property for each object reference. If the upgrade wizard cannot determine the property being referenced as a default property, as is the case with late-bound objects, the wizard will leave your code unchanged but insert a warning comment indicating the issue to be resolved.

There are at least two ways to handle this situation. The first is to modify your original code base to remove any use of default properties and explicitly refer to all properties by their full names. The second option is to modify the upgraded code by manually correcting the default property issues identified by the wizard. The second option is discussed here.

The following Visual Basic 6.0 code example demonstrates the differences in upgrade results when default properties are accessed using both late-bound and early-bound objects.

```
Private Sub CopyButton_Click()
   EarlyBoundCopy Text1, Text2
   LateBoundCopy Text1, Text3
End Sub

' This method's parameters are explicitly defined as TextBoxes.
Private Sub EarlyBoundCopy(sourceCtrl As TextBox, destCtrl As TextBox)
   destCtrl.Text = sourceCtrl
End Sub

' This method's parameters are variants (the default type).
Private Sub LateBoundCopy(sourceCtrl, destCtrl)
   destCtrl.Text = sourceCtrl
End Sub
```

When the upgrade wizard is applied to the preceding code, it will insert two runtime warnings in your code, as shown here. The bold items are upgrade warning comments inserted by the upgrade wizard to indicate problems it encountered while attempting to resolve default properties.

```
Private Sub CopyButton_Click(ByVal eventSender As System.Object, _
   ByVal eventArgs As System.EventArgs) Handles CopyButton.Click
   EarlyBoundCopy(Text1, Text2)
   LateBoundCopy(Text1, Text3)
End Sub

' This method's parameters are explicitly defined as TextBoxes.
Private Sub EarlyBoundCopy(ByRef sourceCtrl As  _
   System.Windows.Forms.TextBox, ByRef destCtrl As _
   System.Windows.Forms.TextBox)
   destCtrl.Text = sourceCtrl.Text
End Sub

' This method's parameters are variants (the default type).
Private Sub LateBoundCopy(ByRef sourceCtrl As Object, _
```

```
   ByRef destCtrl As Object)
   ' UPGRADE_WARNING: Couldn't resolve default property of object
   ' destCtrl.Text.
   ' UPGRADE_WARNING: Couldn't resolve default property of object
   ' sourceCtrl.
   destCtrl.Text = sourceCtrl
End Sub
```

The code produced by the upgrade wizard will compile successfully, but it will result in a run-time exception when **LateBoundCopy** is invoked. This is because default properties are handled differently in Visual Basic .NET; they require parameters. Because default properties are now supported in a different way, the runtime will attempt to cast **sourceCtrl** as a string, resulting in an **InvalidCastException** (because it is not a string but a **TextBox**). For more information about parameterized default properties in Visual Basic .NET, see "Default Properties Changes in Visual Basic" in *Visual Basic Language Concepts* on MSDN.

Note: Carefully examine all UPGRADE_WARNING comments relating to default properties generated by the upgrade wizard. Failure to correct the issues in source code will likely result in unhandled run-time exceptions, causing the program to fail.

Modifying the code generated by the upgrade wizard requires manually appending the correct property to the object name. In this code example, the appropriate property of **sourceCtrl** would be the **Text** property.

```
Private Sub LateBoundCopy(ByRef sourceCtrl As Object, _
   ByRef destCtrl As Object)
   destCtrl.Text = sourceCtrl.Text
End Sub
```

Modifying the upgraded code in this fashion is exactly what the upgrade wizard does for early-bound objects. In such situations, the wizard can identify the appropriate default property and add it to the upgraded code for you. Whenever possible, it is best to use early binding in code.

Resolving Issues with Custom Collection Classes

Collection classes allow you to group objects, typically to allow for better organization and processing of related objects. Chapter 7 addressed some of the standard collection classes available in Visual Basic 6.0, such as the **Controls** and **Printers** collections. However, it is also possible to build your own custom collections. Often, this is done by using the Visual Basic 6.0 Class Builder.

Upgrading custom collections created with the class builder can be accomplished with the upgrade wizard. However, the number of manual adjustments that will

have to be made will depend on any modifications made to the collection code beyond what was generated by the class builder.

For collections with little or no custom code, the upgrade wizard will do most of the upgrade work for you, inserting only a **ToDo** comment to flag the need to review the resulting code. With simple modifications, the final result will behave as the original collection.

Consider the following Visual Basic 6.0 collection example.

```
' Local variable to hold collection
Private myCollec As Collection

Public Sub Add(obj As Object)
    mCol.Add obj
End Sub

Public Sub Insert(obj As Object, Key As Variant)
    mCol.Add obj, Key
End Sub

Public Property Get Item(Index As Variant) As Object
    ' Used when referencing an element in the collection
    ' vntIndexKey contains either the Index or Key to the collection,
    ' this is why it is declared as a Variant
    ' Syntax: Set myVar = x.Item(xyz) or Set myVar = x.Item(5)
  Set Item = myCollec(Index)
End Property

Public Property Get Count() As Long
    ' Used when retrieving the number of elements in the
    ' collection. Syntax: Debug.Print x.Count
    Count = myCollec.Count
End Property


Public Sub Remove(vntIndexKey As Variant)
    ' Used when removing an element from the collection
    ' vntIndexKey contains either the Index or Key, which is why
    ' it is declared as a Variant
    ' Syntax: x.Remove(xyz)


    myCollec.Remove vntIndexKey
End Sub


Public Property Get NewEnum() As IUnknown
    ' This property allows you to enumerate
    ' this collection with the For...Each syntax
    Set NewEnum = myCollec.[_NewEnum]
End Property
```

```
Private Sub Class_Initialize()
    ' Creates the collection when this class is created
    Set myCollec = New Collection
End Sub


Private Sub Class_Terminate()
    ' Destroys collection when this class is terminated
    Set myCollec = Nothing
End Sub
```

When the upgrade wizard is applied to the preceding collection code, it will produce Visual Basic .NET code to achieve the same behavior. However, the **NewEnum** property of the collection will be commented out and replaced with a new method: **GetEnumerator**. The upgraded code as produced by the upgrade wizard is shown here.

```
Friend Class MyCollection
    Implements System.Collections.IEnumerable
    ' Local variable to hold collection
    Private myCollec As Collection

    Public Sub Add(ByRef obj As Object)
        Dim mCol As Object
             ' UPGRADE_WARNING: Could not resolve default property
             ' of object mCol.Add.
        mCol.Add(obj)
    End Sub

    Public Sub Insert(ByRef obj As Object, ByRef Key As Object)
        Dim mCol As Object
             ' UPGRADE_WARNING: Could not resolve default property
             ' of object mCol.Add.
        mCol.Add(obj, Key)
    End Sub

    Public ReadOnly Property Item(ByVal Index As Object) As Object
        Get
            ' Used when referencing an element in the collection
                ' vntIndexKey contains either the Index or Key to the
            ' collection, this is why it is declared as a Variant
            ' Syntax: Set abc = x.Item(xyz) or Set abc = x.Item(5)
            Item = myCollec.Item(Index)
        End Get
    End Property

    Public ReadOnly Property Count() As Integer
        Get
            ' Used when retrieving the number of elements in the
            ' collection. Syntax: Debug.Print x.Count
            Count = myCollec.Count()
        End Get
```

```
    End Property


    ' UPGRADE_NOTE: NewEnum property was commented out.
    ' Public ReadOnly Property NewEnum() As stdole.IUnknown
        ' Get
            ' This property allows you to enumerate
            ' This collection with the For...Each syntax
            ' NewEnum = myCollec._NewEnum
        ' End Get
    ' End Property

    Public Function GetEnumerator() As System.Collections.IEnumerator _
                        Implements System.Collections.IEnumerable.GetEnumerator
                ' UPGRADE_TODO: Uncomment and change the following line to
                ' return the collection enumerator.
        ' GetEnumerator = myCollec.GetEnumerator
    End Function


    Public Sub Remove(ByRef vntIndexKey As Object)
        ' Used when removing an element from the collection
        ' vntIndexKey contains either the Index or Key, which is why
        ' it is declared as a Variant
        ' Syntax: x.Remove(xyz)


        myCollec.Remove(vntIndexKey)
    End Sub


    ' UPGRADE_NOTE: Class_Initialize was upgraded to Class_Initialize_Renamed.
    Private Sub Class_Initialize_Renamed()
        ' Creates the collection when this class is created
        myCollec = New Collection
    End Sub
    Public Sub New()
        MyBase.New()
        Class_Initialize_Renamed()
    End Sub


    ' UPGRADE_NOTE: Class_Terminate was upgraded to Class_Terminate_Renamed.
    Private Sub Class_Terminate_Renamed()
        ' Destroys collection when this class is terminated
        ' UPGRADE_NOTE: Object myCollec may not be destroyed until it is
              ' garbage collected.
        myCollec = Nothing
    End Sub
    Protected Overrides Sub Finalize()
        Class_Terminate_Renamed()
        MyBase.Finalize()
    End Sub
End Class
```

Replacing the **NewEnum** property with the **GetEnumerator** method may require additional work. To ensure that this replacement is reviewed, the upgrade wizard includes an **UPGRADE_TODO** comment in the **GetEnumerator** code and comments out the single line of code that actually returns the enumerator. The result is that you are forced to evaluate the code to determine if returning the enumerator member of the underlying collection is sufficient, or if more custom code is required because of customization in your original collection. In the vast majority of cases, removing comments from this single line of code is all that is required. By forcing you to review it, the upgrade wizard ensures that you have made a conscious decision that the result it produced will have the same behavior as your original Visual Basic 6.0 collection.

Dealing with Changes to Commonly-Used Functions and Objects

Visual Basic .NET provides a compatibility library for many Visual Basic 6.0 functions and objects, to ease upgrade while maintaining the functional behavior of your application. Whenever appropriate, the upgrade wizard will substitute a function in your original code with the Visual Basic .NET equivalent available in the compatibility library. The upgrade wizard will also add a reference to the Visual Basic 6.0 compatibility library namespace (**Microsoft.VisualBasic.Compatibility**) to your solution whenever any such substitutions are made.

Note: Throughout this chapter, the term *compatibility library* is used as a shortened name for the Visual Basic 6.0 compatibility library (**Microsoft.VisualBasic.Compatibility**).

The compatibility library will minimize the amount of effort required to upgrade. This library attempts to duplicate the functional behavior of the original Visual Basic 6.0 functions, but the functionality upgraded to members of that library should be considered as the initial attempt to upgrade that will leverage the application toward a more extensive assimilation to Visual Basic .NET. Some of the functionality in the compatibility library is also available in the Microsoft .NET Framework, allowing the possibility of different upgrade options that you should evaluate according to the available time and resources. In general, upgrading functionality to the .NET Framework instead of using the compatibility library will generate an application with an improved design and usage of references because the quantity of redundant components will be reduced.

As an example, consider the following Visual Basic 6.0 code example.

```
Private Sub cmdUpdate_Click()
    Dim currentTime As Date

    currentTime = Now
    lblDate.Caption = Format(currentTime, "dddd, mmm d yyyy")
    lblTime.Caption = Format(currentTime, "hh:mm:ss AMPM")
End Sub
```

The example code updates the labels on a form with the current date and time whenever a button is clicked. It uses the **Format** function to format the date and time.

Applying the upgrade wizard to this code will result in a reference to the compatibility library's version of the **Format** function, as shown here. The items in bold identify the changes made by the upgrade wizard in order to use the compatibility library.

```
Private Sub cmdUpdate_Click(ByVal eventSender As System.Object, _
                            ByVal eventArgs As System.EventArgs) _
                            Handles cmdUpdate.Click
    Dim currentTime As Date

    currentTime = Now
    lblDate.Text = VB6.Format(currentTime, "dddd, mmm d yyyy")
    lblTime.Text = VB6.Format(currentTime, "hh:mm:ss AMPM")
End Sub
```

This code will compile successfully and exhibit the same behavior as the original code.

An alternative to referencing the compatibility library is to find pure Visual Basic .NET equivalents to achieve the same behavior. In the provided example, an alternative to the compatibility library is to use the Visual Basic .NET version of the **Format** function, provided in the **String** class. The modified Visual Basic .NET code is shown here. The items in bold are the changes that are needed in order to remove the dependence on the compatibility library.

```
Private Sub cmdUpdate_Click(ByVal eventSender As System.Object, _
                            ByVal eventArgs As System.EventArgs) _
                            Handles cmdUpdate.Click
    Dim currentTime As Date

    currentTime = Now
    lblDate.Text = String.Format("{0:dddd, mmm d yyyy}", currentTime)
    lblTime.Text = String.Format("{0:hh:mm:ss tt}", currentTime)
End Sub
```

The revised version of this function has the same behavior as the original function, but it is no longer dependent on the compatibility library.

Keep in mind that there may be more than one Visual Basic .NET equivalent for a function in the compatibility library. For example, an alternative way to rewrite the example is to replace **VB6.Format** with the **ToString** method of the **DateTime** class, which the **currentTime** object belongs to. The following example code shows the modification. Again, the items in bold are the changes that would be applied to remove dependence on the compatibility library and replace the String.Format.

```
Private Sub cmdUpdate_Click(ByVal eventSender As System.Object, _
                            ByVal eventArgs As System.EventArgs) _
                            Handles cmdUpdate.Click
    Dim currentTime As Date

    currentTime = Now
    lblDate.Text = currentTime.ToString("dddd, mmm d yyyy")
    lblTime.Text = currentTime.ToString("hh:mm:ss tt")
End Sub
```

The compatibility library contains replacements for many commonly used Visual Basic 6.0 functions and objects. For a complete listing of the functions and objects, see "VisualBasic.Compatibility Namespace Reference" in Visual Basic .NET Help. For more information about using **DateTime.ToString** and **String.Format**, see Visual Basic .NET Help for the **DateTime** and **String** classes, respectively.

Dealing with Changes to TypeOf

In Visual Basic 6.0, the **TypeOf** clause can check an object's type at run time. This clause is used in a decision statement (that is, **If**) to select a course of action based on the type of an object. The clause evaluates to **True** when the object is of the specified type; otherwise, it evaluates to **False**.

Visual Basic .NET supports the **TypeOf** keyword, but its behavior has changed. There are two major changes:

1. In Visual Basic 6.0, user-defined types are considered object types for use in the **TypeOf** clause. In Visual Basic .NET, user-defined types (otherwise known as structures) are no longer considered object types and therefore cannot be used in a **TypeOf** clause.
2. In Visual Basic 6.0, inheritance is only possible by implementing interfaces. In Visual Basic .NET, true inheritance is supported; this allows classes to inherit from other classes. This impact affects the **TypeOf** clause because it will base the answer on the type of the underlying object that a variable refers to, not the type of the variable. For information about inheritance in Visual Basic .NET, see the "Taking Advantage of Object-Oriented Features" section of Chapter 17, "Introduction to Application Advancement."

To illustrate these points, the following example uses of the **TypeOf** clause in a Visual Basic 6.0 application.

```
Private Type MyType
   Data as Integer
End Type

' This example uses TypeOf with a user-defined type.
Sub CheckMyType()
```

```
   Dim demo as MyType
   If TypeOf demo Is MyType Then
      MsgBox "demo is of type MyType"
   End If
End Sub

' This example uses TypeOf with Visual Basic type.
Sub ControlProcessor(MyControl As Control)
   Dim message as String
   If TypeOf MyControl Is CommandButton Then
      Message = "A button was clicked."
   ElseIf TypeOf MyControl Is CheckBox Then
      Message = "A checkbox was changed."
   ElseIf TypeOf MyControl Is TextBox Then
      Message = "A label was clicked."
   End If
   MsgBox Message
End Sub
```

The upgrade wizard can be applied to the preceding code, and an upgraded version will be produced. However, several UPGRADE_WARNING comments will be included in the code, and in the case of the user-defined type, the code will not compile. The code generated by the upgrade wizard is shown here.

```
Private Structure MyType
   Dim a As Short
End Structure

Sub CheckMyType()
   Dim demo As MyType
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   If TypeOf demo Is MyType Then
      MsgBox "demo is of type MyType"
   End If
End Sub

Sub ControlProcessor(ByRef MyControl As System.Windows.Forms.Control)
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   If TypeOf MyControl Is System.Windows.Forms.Button Then
      ' UPGRADE_WARNING: TypeName has a new behavior.
      System.Diagnostics.Debug.WriteLine("You passed in a " & TypeName(MyControl))
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   ElseIf TypeOf MyControl Is System.Windows.Forms.CheckBox Then
      ' UPGRADE_WARNING: TypeName has a new behavior.
      System.Diagnostics.Debug.WriteLine("You passed in a " & TypeName(MyControl))
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   ElseIf TypeOf MyControl Is System.Windows.Forms.TextBox Then
      ' UPGRADE_WARNING: TypeName has a new behavior.
      System.Diagnostics.Debug.WriteLine("You passed in a " & TypeName(MyControl))
   End If
End Sub
```

The issue with the user-defined type requires modification because the **TypeOf** operator cannot be used with user-defined types in Visual Basic .NET. Thus, even though the upgrade wizard left the **TypeOf** clause in the **If** statement when testing against **MyType**, this code will produce a compile-time error. Modifying the code requires removing the **TypeOf** keyword, creating a new object of the target type, and using the **GetType** method to compare the types of the test object and the target object. The modifications are shown here.

```
Private Sub cmdUserDefined_Click(ByVal eventSender As System.Object, _
                                 ByVal eventArgs As System.EventArgs) _
                                 Handles cmdUserDefined.Click
    Dim demo As MyType
    Dim test As MyType
    If demo.GetType Is test.GetType Then
       MsgBox("MyType")
    End If
End Sub
```

This strategy can be used with any user-defined type, if necessary.

The other issues are warnings that result from inheritance in Visual Basic .NET. The use of **TypeOf** in this second context will compile and run, but the code should still be carefully reviewed. Because of inheritance, **TypeOf** can return **True** if the object's type matches an ancestor class of the object. For example, the introduction of a test for type **Object** in the preceding **If** construct may trigger an unexpected result. This is demonstrated here.

```
Sub ControlProcessor(ByRef MyControl As System.Windows.Forms.Control)
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   If TypeOf MyControl Is System.Windows.Forms.Button Then
      ' UPGRADE_WARNING: TypeName has a new behavior.
      System.Diagnostics.Debug.WriteLine("You passed in a " & TypeName(MyControl))
   ElseIf TypeOf MyControl Is Object Then
      System.Diagnostics.Debug.WriteLine("You passed in an Object")
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   ElseIf TypeOf MyControl Is System.Windows.Forms.CheckBox Then
      ' UPGRADE_WARNING: TypeName has a new behavior.
      System.Diagnostics.Debug.WriteLine("You passed in a " & TypeName(MyControl))
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   ElseIf TypeOf MyControl Is System.Windows.Forms.TextBox Then
      ' UPGRADE_WARNING: TypeName has a new behavior.
      System.Diagnostics.Debug.WriteLine("You passed in a " & TypeName(MyControl))
   End If
End Sub
```

The new **ElseIf** branch in the preceding example will cause the "Object" message to display for any control other than a **Button** because all controls inherit from the **Object** class. Thus, without careful review of the upgraded code, an unexpected branch may execute when the **TypeOf** clause is used.

Upgrading References to Visual Basic 6.0 Enum Values

Upgrading code that refers to any predefined Visual Basic 6.0 enumerator values requires special care. The upgrade wizard may replace the enumerator reference with a new object in Visual Basic .NET that represents the same value, but it is not a simple enumerator value. The examples in this section demonstrate this.

If you define a constant in Visual Basic 6.0 to represent a property on a Windows form or control, you may find that the code does not compile after you upgrade the project. Consider the following Visual Basic 6.0 code example.

```
Const myGreenColor = vbGreen
```

This simple statement defines a constant value to be used to represent the color green. It is assigned to the value in the Visual Basic 6.0 predefined enumerator value **vbGreen**. Applying the upgrade wizard will result in the following Visual Basic .NET code.

```
' UPGRADE_NOTE: myGreenColor was changed from a Constant to a Variable.
Dim myGreenColor As System.Drawing.Color = System.Drawing.Color.Lime
```

The first thing to note is that the declaration has changed from a constant declaration to a variable declaration. Why was this change made? The change was made because the upgrade wizard replaced the reference to the Visual Basic .NET representation of the color green. However, **System.Drawing.Color.Lime** is not a constant value; it is a non-constant expression. Instead of being a fixed numeric value, it is a property that returns a **System.Drawing.Color** object.

You can view the declaration for **System.Drawing.Color.Lime** within the upgraded Visual Basic .NET project by right-clicking **Lime** in the code window and then clicking **Go To Definition**. The **Object Browser** will display the definition of **System.Drawing.Color.Lime** as follows.

```
Public Shared ReadOnly Property Lime() As System.Drawing.Color
```

The impact of this change is that in some cases, you may encounter a compiler error when upgrading code that references Visual Basic 6.0 predefined enumerator values. For example, suppose that in your original code you created an enumerator named **RGBColors** containing values for red, green, and blue. You use the enumerator values in code to combine red, blue, black, and yellow to create a custom background for your labels, as shown here.

```
Enum RGBColors
   myDarkRose = vbRed + vbBlue
   myWhite = vbBlue + vbBlack + vbYellow
End Enum
```

```
Private Sub frmCollection_Load()
   Label1.BackColor = RGBColors.myDarkRose
   Label2.BackColor = RGBColors.myWhite
End Sub
```

Applying the upgrade wizard to this code results in the following.

```
Enum RGBColors
    myDarkRose = System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Red) _
                 +
System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Blue)
    myWhite = System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Blue) _
              + System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Black)
_
              + System.Drawing.ColorTranslator.ToOle(System.Drawing.Color.Yellow)
End Enum

Private Sub frmCollection_Load(ByVal eventSender As System.Object, _
                    ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    Label1.BackColor = _
                  System.Drawing.ColorTranslator.FromOle(RGBColors.myDarkRose)
    Label2.BackColor = System.Drawing.ColorTranslator.FromOle(RGBColors.myWhite)
End Sub
```

Because the declarations for the color objects **myDarkRose** and **myWhite** are initialized with values that are not constants, literals, or enumerations, each line results in the compiler error "Constant expression is required."

There are a two ways to fix these issues. First, you can define your own constant values and replace the use of the Visual Basic .NET property for each enumerator with your constants. Alternatively, you can use nonconstant values directly. Each of these alternatives is described in the sections that follow.

Defining Your Own Constant Values

The easiest way to fix your code is to use constant values in place of the **Red**, **Blue**, **Black**, and **Yellow** objects generated by the upgrade wizard. For example, you can declare constants with the values of the Visual Basic 6.0 constants **vbRed**, **vbBalck,** **vbWhite**, and **vbBlue,** so that you can initialize the enumerator members with these new constants. The following code example demonstrates this approach.

```
Const RedBlue As Integer = &HFFs + &HFF0000
Const BlueBlackYellow As Single = &HFF0000 + &H0s + &HFFFFs

Enum RGBColors
    myDarkRose = RedBlue
    myWhite = BlueBlackYellow
End Enum

Private Sub frmCollection_Load(ByVal eventSender As System.Object, _
```

```
                    ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    Label1.BackColor = _
                System.Drawing.ColorTranslator.FromOle(RGBColors.myDarkRose)
    Label2.BackColor = System.Drawing.ColorTranslator.FromOle(RGBColors.myWhite)
End Sub
```

Using the Non-Constant Values

Another way to solve the problem is to replace any instances in which a member of **RGBColors** is used with the matching **System.Drawing.Color**. Taking this approach may require additional research and work because of the differences between constant values and nonconstant properties. For example, the preceding code example computes the value of the background color by performing addition on two enumerator values. If you instead use the **System.Drawing.Color** properties **Red** and **Blue**, you cannot directly compute the background color using this approach because you cannot add two objects together. However, if you can use a method or a function that will give you a meaningful numeric value for the object, you can use the numeric values in your calculation. Using a numeric-to-object conversion function allows you to turn the result of the numeric calculation back into an object representing the calculated value. However, this is possible only if such conversion functions exist.

In the case of color objects, you can obtain a meaningful numeric color value by using the **ToOle** method of the **System.Drawing.ColorTranslator** class. The **ToOle** method takes a color object, such as **System.Drawing.Color.Red**, and obtains an RGB color value for it. For example, **ToOle(System.Drawing.Color.Red)** will return the RGB value 255, or hexadecimal value FF; it is no coincidence that the **vbRed** constant in Visual Basic 6.0 has the same value. After you calculate the new color, you can convert the result back to a color object by using the **ColorTranslator.FromOle** method.

The following code example shows how to modify the example to use the **System.Drawing.Color** properties to achieve the same result as the original code.

```
' Include the Imports statement at the top of the Form file.
Imports System.Drawing
…
Private Sub frmCollection_Load(ByVal eventSender As System.Object, _
                    ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    Label1.BackColor = ColorTranslator.FromOle(ColorTranslator.ToOle(Color.Red) +
_
                        ColorTranslator.ToOle(Color.Blue))
    Label2.BackColor = ColorTranslator.FromOle(ColorTranslator.ToOle(Color.Blue) +
_
                        ColorTranslator.ToOle(Color.Black) +  _
                        ColorTranslator.ToOle(Color.Yellow))
End Sub
```

It is up to you whether you want to define your own constants or use the nonconstant values directly in your code. If, in your original code, you are using the constant values in combination with other constant values, it would make sense to define your own constant values and use the constants in your code. On the other hand, if you are making a simple assignment of a constant value to a property, it would make more sense to use the nonconstant value directly.

Dealing with Changes to Arrays

In Visual Basic 6.0, you can specify the starting index for arrays using the **Option Base** declaration or by specifying the index range in the array declaration. In Visual Basic .NET, all arrays have a starting index of 0. As a result, the **Option Base 1** declaration is not supported, nor is specifying an array index range.

Consider the following Visual Basic 6.0 array declarations.

```
Option Base 1
Dim VariantArray(5) As Variant '5 elements (1-5)

' Specify the array index range in the array declaration.
Dim StringArray(1 To 100) As String    '100 elements
Dim LongArray(15 To 25) As Long        '11 elements
Dim IntArray(-10 To 10) As Integer     '21 elements
```

None of the preceding declarations are supported in Visual Basic .NET. All array declarations must have a starting index of 0. Thus, code in which array declarations have a non-zero starting index must be adjusted. Applying the upgrade wizard to code with such declarations will result in UPGRADE_WARNING comments inserted in the upgraded code wherever the declarations appear. Furthermore, some declarations may be changed and others left unchanged depending on the starting index in the original code.

The following code example shows the result of applying the upgrade wizard to the example declarations in the preceding example.

```
' UPGRADE_WARNING: Lower bound of array VariantArray was changed from 1 to 0.
Dim VariantArray(5) As Object '6 elements (0-5)

' Specify the array index range in the array declaration.
' UPGRADE_WARNING: Lower bound of array StringArray was changed from 1 to 0.
Dim StringArray(100) As String '101 elements
' UPGRADE_WARNING: Lower bound of array LongArray was changed from 15 to 0.
Dim LongArray(25) As Integer '26 elements
' UPGRADE_ISSUE: Declaration type not supported: Array with lower
' bound less than zero.
Dim IntArray(-10 To 10) As Short 'Invalid array declaration
```

Note: The comments in the preceding upgraded code have been altered to reflect the changes in the number of elements each array contains after the upgrade. The actual comments would remain the same as in the original Visual Basic 6.0 code.

Each of the issues will be examined in turn.

For code that uses **Option Base 1**, the upgrade wizard will issue the following UPGRADE_WARNING message.

```
' UPGRADE_WARNING: Lower bound of array VariantArray was
' changed from 1 to 0.
```

This UPGRADE_WARNING is added for each array declaration that does not specify a lower bound. The impact of the change in array bounds is that **VariantArray** now has six elements, indexed from 0 to 5. The same will be true for any array in the original code that did not specify a starting index; the upgraded versions will all begin at index 0 and end at the original ending index.

For array declarations that specify an index range in which a positive lower index is specified, a similar change is made. The upgrade wizard removes the lower index from the declaration, leaving only the ending index. Thus, the upgraded **StringArray** and **LongArray** declarations have been modified so that only the ending index is kept. The impact is that the number of elements in the upgraded arrays is increased by a number equivalent to the original starting index. For example, **StringArray**, which was declared with a starting index of 1, now begins at index 0 and has only one extra element. **LongArray**, which originally had a starting index of 15, now starts at index 0 resulting in 15 additional array elements in the upgraded array.

In each of these examples, the upgraded declarations will compile and run. However, care must be taken if the original code uses dynamic references to the lower bound of the array because the upgraded array will have more elements than the original. Consider the following Visual Basic 6.0 loop that fills **LongArray** with the item counter.

```
Dim i As Integer
Dim Index As Integer
i = 1
For Index = LBound(LongArray) To UBound(LongArray)
   LongArray(Index) = i
   i = i + 1
Next
```

The upgraded code will remain unchanged by the upgrade wizard. However, the behavior is now different because the original code filled only 11 elements (indexed 15 to 25), whereas the upgraded version now fills 26 elements (indexed 0 to 25).

Visual Basic 6.0 arrays declared with a negative lower bound are left unchanged by the upgrade wizard. However, because specifying an array index range in the array declaration is no longer supported, any such declarations in the upgraded code will generate compile time errors. This forces you to manually change any such array declarations to use a 0 lower bound. Furthermore, you may need to change any code that used the array to remove any negative index references.

Consider the fourth array declaration in the examples provided earlier. The array declaration for **IntArray** contains a starting index of -10. The upgrade wizard leaves this declaration unchanged, and as a result the upgraded code will not compile. You must manually change the array declaration in the new code so that it contains the same number of elements as the original code. In this example, an array with indexes -10 through 10 would have 21 elements. The new declaration should then be as shown here.

```
Dim IntArray(20) As Integer
```

In this new declaration, you now have an array 21 elements long, indexed from 0 to 20.

You would also need to check for any references to negative array indexes. Suppose, for example, that your original code had the following loop with hard-coded indexes to process the contents of **IntArray.**

```
Dim i As Integer
For i = -10 To 10
   IntArray(i) = i
Next
```

Adjusting the array declaration will allow the code to compile, but any attempt to execute the preceding loop will result in a run-time exception, **System.IndexOutOfRangeException**, because -10 is outside the range of array indexes for this array. The changes you make will depend on your needs. For example, if you needed to fill this array with the values -10 through 10, you could adjust the code to offset the array index by 10. This is demonstrated here.

```
Dim i As Integer
For i = -10 To 10
   IntArray(i + 10) = i
Next
```

Keep in mind that such code replacement must be performed anywhere array indexes are used to ensure that negative indexes are never used.

As an alternative, you may choose to use the new collections that are provided by the .NET Framework instead of arrays. These include **ArrayList**, **SortedList**, **Queue**, **Stack**, and **HashTable**. Some of these collections allow access to elements using

either a key or an index. Thus, you can assign your original index structure as key values for the items in the collection and continue to refer to these items with these keys instead of the new zero-based indexes. For example, you can replace the array declaration for **IntArray** with a **SortedList** collection and then store your items with your original indexes as keys. This is demonstrated here.

```
Dim IntList As New Collections.SortedList(20)
Dim i As Integer
For i = -10 To 10
   IntList.Add(i,i)
Next
```

For more information about the new collections available in Visual Basic .NET, see "System.Collections Namespace" in the *.NET Framework Class Library* on MSDN.

Legacy Visual Basic Language Features

Visual Basic 6.0 maintains code compatibility for many language features going all the way back to the original version of Visual Basic. These features include elements such as **DefInt**, **VarPtr**, and **GoSub…Return**. These features and others are deprecated in Visual Basic 6.0.

In Visual Basic .NET, some of these features have been eliminated because they lead to confusing and cryptic coding styles and are not appropriate for the kind of large-scale distributed application development that is becoming dominant in today's business environment.

The decision to remove these features was intended to move the language forward in the most expeditious, and ultimately beneficial, way possible. The following Visual Basic 6.0 language features are upgraded by the upgrade wizard but are not recommended for use in your applications: **DefInt**, **DefStr**, **DefObj**, **DefDbl**, **DefLng**, **DefBool**, **DefCur**, **DefSng**, **DefDec**, **DefByte**, **DefDate**, **DefVar**, **GoTo**, **Imp**, and **Eqv**. For code clarity and maintainability, it is recommended that use of these features should be avoided. **Def*Type*** statements will be automatically upgraded by the upgrade tool and comments will be generated in the target code to indicate that **Def*Type*** statements were removed and explicit declarations of the affected variables were added. The upgrade wizard will handle this situation whenever possible, but it is recommended that you remove deprecated features that are not supported by the Visual Basic .NET language from your original source code prior to upgrade.

The following features are not upgraded by the upgrade wizard, and the developer must eliminate them: **GoSub…Then**, **LSet**, **VarPtr**, **ObjPtr**, **StrPtr**, and **Null** and **Empty**. Table 8.1 lists some obsolete features and suggested alternatives to apply when upgrading.

Table 8.1: Obsolete Keywords, Functions, and Statements in Visual Basic .NET

Visual Basic 6.0	Visual Basic .NET alternative
Array the array.	The { } can be used in the declaration of the array, to assign values to
As Any	Visual Basic .NET supports overloading declares so that explicit data types must be used.
Calendar	Available in the **System.Globalization** namespace.
Currency	Replaced by the **Decimal** data type. The **ToString("C")** method can be used to display **Decimal** values as currency according to the current culture settings.
Date	Still supported as a data type (mapping to **System.DateTime**) but is no longer a function returning a 4-byte date value. Use the **Today** property of the **System.DateTime** structure to return the day in the 8-byte CLR format.
Debug.Assert, Debug.Print	Replaced by methods in the **System.Diagnostics** namespace.
Def<*Type*>	No longer supported. All variables must be explicitly declared.
DoEvents	Replaced by a method in the **System.Windows.Forms.Application** class.
Empty, Null, IsEmpty, IsNull	All are handled by **Nothing** keyword. Statements to check for these conditions have also been removed. Database nulls can be tested for using the **IsDBNull** function or against the **System.DBNull.Value** property.
Eqv, Imp	Use the equals (=) operator in conjunction with **Not** and **Or**. For example, replace **A Imp B** with **(Not A) Or B**.
GoSub	No longer supported. Remove all in original code before upgrading.
IsMissing	In Visual Basic .NET, all optional parameters must have default values.
IsObject	Replaced by **IsReference**.
Let, Set	Because parameterless default properties are no longer supported, these statements have been removed.
LSet, RSet	Replaced by **PadRight** and **PadLeft** methods of the **System.String** class, although still supported. However, note that **LSet** can no longer be used to copy user-defined types (UDTs). The equivalent to UDTs in Visual Basic .NET are structures; these cannot be copied directly but the members can be copied one by one to achieve the same result.For more information about using UDTs to access file records, see the section titled "Accessing Fixed-Length Records Using User Defined Types" in Chapter 11, "Upgrading String and File Operations," and the "User-Defined Types" section in Chapter 6, "Understanding the Visual Basic Upgrade Wizard."

continued

Visual Basic 6.0	Visual Basic .NET alternative
PSet, Scale	No longer supported. Equivalent functionality exists in **System.Drawing**.
Option Private Module	Any module can be marked as private using the **Private** keyword.
String Functions	Visual Basic functions that returned strings by appending a dollar sign ($) have now been replaced by overloaded methods. Their counterparts that return **Variant**s have been replaced with overloaded methods that return **Object**s.
Time	Replaced by the **TimeOfDay** property of the **System.DateTime** structure. Note that the **Date** data type is no longer represented as **Double**; instead, it is represented as a **DateTime** structure.
Type	Replaced by the **Structure** statement.
VarType	Although it still exists, it should be replaced by the **GetTypeCode** method of the specific data types.
Variant	Replaced by **Object** as the universal data type.
Wend	Replaced by **End While**.

Upgrading Add-ins

You can use the Visual Basic extensibility object model to ease development through add-ins. Add-ins are tools that you create programmatically using objects and collections in the extensibility model to customize and extend the Visual Basic integrated development environment (IDE). In a sense, add-ins "snap on" to the Visual Basic IDE. Regardless of how they are created, the primary goal of an add-in is to enable you to automate something in the development environment that is difficult, tedious, or time-consuming to accomplish manually. Add-ins are automation tools that save time and labor for the Visual Basic programming environment.

Although add-ins are still supported in Visual Studio .NET, there are differences that require making changes to your project when you upgrade them from Visual Basic 6.0. This section will discuss some of the adjustments you will have to make, using example add-in code for illustration.

The sample add-in we will work with is simple, but it has enough features to demonstrate many of the basic modifications you will have to make in a typical add-in upgrade. Our add-in project is named **LabAddIn** and contains a **Class Module** named **Connect** and a resources file with the same name as the project. This resources file contains a bitmap associated with ID 101. After the add-in is registered on the computer, whenever Visual Studio 6.0 is loaded, the add-in shows a button in the Visual Basic Standard toolbar. Clicking this button inserts the **Option Explicit** directive at the beginning of the current source code document, by way of the

clipboard object. For more information about creating and registering Visual Basic 6.0 add-ins, see "Creating a Basic Add-In" on MSDN.

The code for the **Connect Class Module** of the add-in is shown here.

```
Implements IDTExtensibility

Dim MenuCommandBar As Office.CommandBarControl
Public WithEvents MenuHandler As CommandBarEvents
Dim CurrentVBInstance As VBIDE.VBE
Dim LastButton As Long
Private Sub IDTExtensibility_OnAddInsUpdate(custom() As Variant)
' Required procedure for add-in
End Sub
Private Sub IDTExtensibility_OnConnection(ByVal VBInst As Object, _
            ByVal ConnectMode As VBIDE.vbext_ConnectMode, _
            ByVal AddInInst As VBIDE.AddIn, custom() As Variant)

    Set CurrentVBInstance = VBInst
    CurrentVBInstance.CommandBars("Standard").Visible = True
    LastButton = CurrentVBInstance.CommandBars("Standard").Controls.Count
    Set MenuCommandBar = CurrentVBInstance.CommandBars("Standard").Controls.Add(1,
_
                                                        , , LastButton)
    Clipboard.SetData LoadResPicture(101, vbResBitmap)
    MenuCommandBar.PasteFace
    MenuCommandBar.ToolTipText = "Lab AddIn"
    Set Me.MenuHandler = CurrentVBInstance.Events.CommandBarEvents(MenuCommandBar)
End Sub

Private Sub IDTExtensibility_OnDisconnection(ByVal RemoveMode As _
            VBIDE.vbext_DisconnectMode, custom() As Variant)
    CurrentVBInstance.CommandBars("Standard").Controls.Item(LastButton).Delete
End Sub

Private Sub IDTExtensibility_OnStartupComplete(custom() As Variant)
    ' Required procedure for add-in
End Sub

Private Sub MenuHandler_Click(ByVal CommandBarControl As Object, _
            handled As Boolean, CancelDefault As Boolean)
    Screen.MousePointer = vbHourglass
    On Error GoTo ErrorHandler
    Call CurrentVBInstance.ActiveCodePane.CodeModule.InsertLines(1, _
         "Option Explicit" + Chr(13))
    Screen.MousePointer = vbDefault
    Exit Sub
ErrorHandler:
    MsgBox Err.Description, vbCritical
    Screen.MousePointer = vbDefault
End Sub
```

When the upgrade wizard is applied to this code, the result is the following.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("Connect_NET.Connect")> _
Public Class Connect
    Implements VBIDE.IDTExtensibility

    Dim MenuCommandBar As Microsoft.Office.Core.CommandBarControl
    Public WithEvents MenuHandler As VBIDE.CommandBarEvents
    Dim CurrentVBInstance As VBIDE.VBE
    Dim LastButton As Integer
    Private Sub IDTExtensibility_OnAddInsUpdate(ByRef custom As System.Array) _
                Implements VBIDE.IDTExtensibility.OnAddInsUpdate
        ' Required procedure for add-in
    End Sub
    Private Sub IDTExtensibility_OnConnection(ByVal VBInst As Object, _
                                    ByVal ConnectMode As VBIDE.vbext_ConnectMode,
_
                                    ByVal AddInInst As VBIDE.AddIn, _
                                    ByRef custom As System.Array) _
                                    Implements VBIDE.IDTExtensibility.OnConnection

        CurrentVBInstance = VBInst
        CurrentVBInstance.CommandBars("Standard").Visible = True
     LastButton = CurrentVBInstance.CommandBars("Standard").Controls.Count
     MenuCommandBar = CurrentVBInstance.CommandBars("Standard").Controls.Add( _
                          1,  ,  , LastButton)
        ' UPGRADE_ISSUE: Clipboard method Clipboard.SetData was not upgraded.
        Clipboard.SetData(VB6.LoadResPicture(101, VB6.LoadResConstants.ResBitmap))

        ' UPGRADE_WARNING: Could not resolve default property of object
        ' MenuCommandBar.PasteFace.
        MenuCommandBar.PasteFace()
        MenuCommandBar.ToolTipText = "Lab AddIn"
        Me.MenuHandler = CurrentVBInstance.Events.CommandBarEvents(MenuCommandBar)
    End Sub

    Private Sub IDTExtensibility_OnDisconnection( _
             ByVal RemoveMode As VBIDE.vbext_DisconnectMode, _
             ByRef custom As System.Array) _
             Implements VBIDE.IDTExtensibility.OnDisconnection
        ' UPGRADE_WARNING: Could not resolve default property of object
        ' CurrentVBInstance.CommandBars().Controls.Item().Delete.
       CurrentVBInstance.CommandBars("Standard").Controls.Item(LastButton).Delete()
    End Sub

    Private Sub IDTExtensibility_OnStartupComplete(ByRef custom As System.Array) _
                              Implements VBIDE.IDTExtensibility.OnStartupComplete
        'Required Procedure for add-in
    End Sub
```

```
    Private Sub MenuHandler_Click(ByVal CommandBarControl As Object, _
                                  ByRef handled As Boolean, _
                                  ByRef CancelDefault As Boolean) _
                                  Handles MenuHandler.Click

        ' UPGRADE_WARNING: Screen property Screen.MousePointer has a new behavior.
        System.Windows.Forms.Cursor.Current =  _
           System.Windows.Forms.Cursors.WaitCursor
        On Error GoTo ErrorHandler
        Call CurrentVBInstance.ActiveCodePane.CodeModule.InsertLines(1, _
                        "Option Explicit" & Chr(13))
        ' UPGRADE_WARNING: Screen property Screen.MousePointer has a new behavior.
        System.Windows.Forms.Cursor.Current = System.Windows.Forms.Cursors.Default
    Exit Sub
ErrorHandler:
        MsgBox(Err.Description, MsgBoxStyle.Critical)
        ' UPGRADE_WARNING: Screen property Screen.MousePointer has a new behavior.
        System.Windows.Forms.Cursor.Current = System.Windows.Forms.Cursors.Default
    End Sub
End Class
```

A compilation error related to this code appears in the Task List. The error is a result of changes to the **Clipboard** object as explained earlier in this chapter. You can correct this error by changing the offending instruction, as shown here.

```
' UPGRADE_ISSUE: Clipboard method Clipboard.SetData was not upgraded.
Clipboard.SetData(VB6.LoadResPicture(101, VB6.LoadResConstants.ResBitmap))
```

This statement can be replaced with the following code to deal with the changes to the Clipboard in Visual Basic .NET.

```
Dim d As New System.Windows.Forms.DataObject(VB6.LoadResPicture (101, _
            VB6.LoadResConstants.ResBitmap))
System.Windows.Forms.Clipboard.SetDataObject(d)
```

In addition to correcting the Clipboard code, you must also correct references to the **VBIDE** and **Microsoft.Office.Core** libraries, which enable access to Visual Basic IDE features. These libraries have been replaced by equivalent ones in .NET. To use the new versions of these libraries, you must first delete these references. Next, add the references to the libraries **EnvDTE**, **Extensibility**, and **Office**. These libraries enable the extensibility functionality of Visual Studio .NET and allow access to the current instance of the IDE. You can access the **Add Reference** dialog box by selecting the **Add Reference** item on the **Project** menu.

Making these changes to the references will cause new compilation errors in your code resulting from calls to functions in the VBIDE library. The next step in the upgrade is to correct these errors by replacing the calls with the equivalent versions in the libraries you just referenced. For example, an error occurs with the following line of code.

```
Implements VBIDE.IDTExtensibility
```

You can correct this error by replacing it with the following two lines of Visual Basic .NET code.

```
Implements Extensibility.IDTExtensibility2
Implements EnvDTE.IDTCommandTarget
```

This change substitutes the extensibility interface of the VBIDE library, and it allows you to modify aspects of the Visual Studio .NET IDE's graphical interface.

These changes are still not enough to complete the upgrade. The next step is to import the **EnvDTE**, **Extensibility**, and **System.Runtime.InteropServices** libraries. Add the following lines of code before the declaration of the **Connect** class to import these libraries.

```
Imports EnvDTE
Imports Extensibility
Imports System.Runtime.InteropServices
```

Table 8.2 lists code replacements that must be made in **OnAddInsUpdate**, **OnConnection**, **OnDisconnection**, and **OnStartUpComplete** events.

Table 8.2: Replacements for Code Events

Original code	Replacement
VBIDE.IDTExtensibility	Extensibility.IDTExtensibility2
VBIDE.vbext_ConnectMode	Extensibility.ext_ConnectMode
VBIDE.AddIn	Object
VBEIDE.vbext_DisconnectMode	Extensibility.ext_ConnectMode

The resulting code with changes applied is shown here.

```
Private Sub IDTExtensibility_OnAddInsUpdate(ByRef custom As System.Array) _
            Implements Extensibility.IDTExtensibility2.OnAddInsUpdate

Private Sub IDTExtensibility_OnConnection(ByVal VBInst As Object, _
            ByVal ConnectMode As Extensibility.ext_ConnectMode, _
            ByVal addInInst As Object, ByRef custom As System.Array) _
            Implements Extensibility.IDTExtensibility2.OnConnection

Private Sub IDTExtensibility_OnDisconnection( _
            ByVal RemoveMode As Extensibility.ext_DisconnectMode, _
            ByRef custom As System.Array) Implements _
            Extensibility.IDTExtensibility2.OnDisconnection

Private Sub IDTExtensibility_OnStartupComplete(ByRef custom As System.Array) _
            Implements Extensibility.IDTExtensibility2.OnStartupComplete
```

Your upgraded add-in must include an **OnBeginShutdown** event handler. The following code example can be used.

```
Public Sub IDTExtensibility_OnBeginShutdown(ByRef custom As System.Array) _
           Implements Extensibility.IDTExtensibility2. OnBeginShutdown
   ' Required by add-in
End Sub
```

The following code handles the connect command that will be sent to the plug-in when it is used in the Visual Studio .NET IDE. This will allow a connection with the plug-in and will unfold the corresponding controls on the user interface.

```
Public Sub Exec(ByVal cmdName As String, _
           ByVal executeOption As vsCommandExecOption, ByRef varIn As Object, _
           ByRef varOut As Object, ByRef handled As Boolean) _
           Implements IDTCommandTarget.Exec
    handled = False
    If (executeOption = vsCommandExecOption.vsCommandExecOptionDoDefault) Then
        If cmdName = "LabAddIn.Connect" Then
            handled = True
            Exit Sub
        End If
     End If
End Sub

Public Sub QueryStatus(ByVal cmdName As String, _
           ByVal neededText As vsCommandStatusTextWanted, _
           ByRef statusOption As vsCommandStatus, _
           ByRef commandText As Object) _
           Implements IDTCommandTarget.QueryStatus
    If neededText = _
           EnvDTE.vsCommandStatusTextWanted.vsCommandStatusTextWantedNone Then
        If cmdName = "LabAddIn.Connect" Then
            statusOption = CType(vsCommandStatus.vsCommandStatusEnabled + _
                                 vsCommandStatus.vsCommandStatusSupported, _
                                 vsCommandStatus)
        Else
            statusOption = vsCommandStatus.vsCommandStatusUnsupported
        End If
    End If
End Sub
```

Your new code still has two compilation errors. These can be solved by referencing the **EnvDTE** library instead of the eliminated VBIDE, as shown here.

```
Public WithEvents MenuHandler As CommandBarEvents
Dim CurrentVBInstance As DTE
```

Next, you must change the **ProgId** class attribute to use the correct name of add-in project, the automatic upgrade process sets **Connect_NET.Connect** in this attribute, using the following instruction of the class declaration.

```
<System.Runtime.InteropServices.ProgId("Connect_NET.Connect")>
```

The attribute should be changed by the correct name: in this example, this means changing the name to **LabAddIn.Connect** and the previous will be changed to the following.

```
<System.Runtime.InteropServices.ProgId("LabAddIn.Connect")>
```

Finally, you must change the following line in the **MenuHandler.Click** event.

```
Call CurrentVBInstance.ActiveCodePane.CodeModule.InsertLines(1, _
                            "Option Explicit" & Chr(13))
```

Use the following instructions to replace this previous line of code.

```
Dim Txt As TextDocument = CurrentVBInstance.ActiveDocument.Object()
Txt.CreateEditPoint().Insert("Option Explicit On" & Chr(13))
```

With this final change, you now have a project that is ready to build. The add-in should now compile correctly and be ready to plug in to the Visual Studio .NET IDE. To do so, you will have to register it to make the functionality available. You can register your add-in with the following steps:

1. Build the dynamic link library (DLL) file for the add-in.
2. Open a command prompt.
3. At the command prompt, enter **regasm *name*.dll** where *name* is the name of the add-in.
4. Edit the registry. You can use the **regedit** command to start the registry editor.
5. Find the key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1\AddIns**. If the key **AddIns** is not already created, you must create it yourself.
6. Add a key with the same name of the **ProgId** class attribute.
7. In the new key, add the following string value: **FriendlyName** and **Description**.
8. Set the display name of the add-in in the **FiendlyName** option.
9. Set a descriptive comment about the add-in in the **Description** option.
10. Load Visual Studio .NET.
11. On the **Tools** menu, click **Add-in Manager**.
12. Select your add-in in the list to have it added to the Visual Studio .NET IDE.

By applying these steps, you can customize the Visual Studio .NET IDE just as you can the Visual Basic 6.0 IDE.

Summary

Several Visual Basic 6.0 language features have been changed or removed in Visual Basic .NET. This does not mean that you do not have options for upgrading code that contains these features; however, it does mean you will likely have to make manual adjustments to your code.

An important step in understanding how to deal with changed features is to learn which features were changed. Just as important is to understand what the change was and how it impacts upgrades.

Each feature identified in this chapter will require manual correction. This chapter has given some ideas about how to address these issues. Remember, there may be more than one way to solve problems that arise from these issues, so if a particular suggestion does not suit you, a Web search may help you find a more appropriate solution.

More Information

For more information about parameterized default properties in Visual Basic .NET, see "Default Property Changes in Visual Basic" in *Visual Basic Language Concepts* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vaconDefaultProperties.asp.

For more information about the new collections available in Visual Basic .NET, see "System.Collections Namespace" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp.

For more information about creating and registering Visual Basic 6.0 add-ins, see "Creating a Basic Add-In" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon98/html/vbconcreatingaddin.asp.

9

Upgrading Visual Basic 6.0 Forms Features

Many Microsoft Windows-based applications are based on forms. One of the key advantages of Microsoft Visual Basic is the ease with which forms can be built, making it a common language for rapid application development.

There have been big changes in the basic forms architecture since Visual Basic 6.0. Forms applications in Visual Basic .NET are created with Windows Forms architecture. Windows Forms is a richer, more extensible architecture than that available in Visual Basic 6.0, but the differences require making adjustments when upgrading applications. This chapter will introduce you to some of the changes in forms, and what you need to do to upgrade your forms applications.

Handling Changes to Graphics Operations

Visual Basic 6.0 provides a number of graphics methods for drawing lines and shapes on a form. It also provides the **PictureBox** and **Image** controls, which allow you to apply drawing methods on other controls. There are several changes to how drawing operations are performed in Visual Basic .NET. The next sections describe these changes.

Removal of the Line Control in Visual Basic .NET

The **Line** control in Visual Basic 6.0 allows you to draw a line segment on a form. It has no equivalent control in Visual Basic .NET. When the upgrade wizard is applied, vertical and horizontal **Line** controls are replaced with the Windows Forms **Label** control, with the **Text** property set to an empty string, the **BorderStyle** property set to **None**, and the **BackColor**, **Width**, and **Height** properties set to match the original control. Line controls that are not vertical or horizontal are not upgraded; the upgrade wizard replaces such lines with a **Label** control, with the **BackColor** property set to **Red** to highlight the issue.

Line controls can be replaced with lines drawn in the **Paint** event using graphics functions available in Visual Basic .NET. The following example shows how a diagonal line is painted in Visual Basic .NET.

```
Private Sub Form1_Paint(ByVal sender As Object, _
        ByVal e As System.Windows.Forms.PaintEventArgs) _
        Handles MyBase.Paint
    ' Create a Pen object.
    Dim pen As New Drawing.Pen(System.Drawing.Color.Brown, 1)
    e.Graphics.DrawLine(pen, 0, 0, 100, 100)
    pen.Dispose()
End Sub
```

Using the **DrawLine** method may be more work than using a **Line** control, but it is necessary for lines that are not vertical or horizontal. It will also be necessary for lines that have non-default values for attributes such as **BorderWidth** or **BorderStyle**. Furthermore, a **Label** control is significantly different from a graphic object such as a **Line**. Although it may initially appear the same in your upgraded project, the behavior will not be quite the same. Keep in mind that upgrading the **Line** control to a **Label** control is a temporary solution that should be reimplemented using the **DrawLine** method.

Removal of the Shape Control in Visual Basic .NET

The **Shape** control in Visual Basic 6.0 has no equivalent in Visual Basic .NET. Rectangular or square **Shape** controls in code are replaced with Windows Forms **Label** controls where the **BorderStyle** is set to **FixedSingle**, and the **BackColor**, **Width**, and **Height** properties are set to match the original control. Oval and circle **Shape** controls cannot be upgraded; they are replaced with a placeholder **Label** control with its **BackColor** property set to **Red**.

You can replace **Shape** controls with drawing methods in the **Paint** event. This strategy can be used for any **Shape** control, but it is mandatory for oval and circle **Shape** controls. The following code example demonstrates how to draw a circle in Visual Basic .NET.

```
Private Sub Form1_Paint(ByVal sender As Object, _
                        ByVal e As System.Windows.Forms.PaintEventArgs) _
                        Handles MyBase.Paint
   ' Create a Pen object.
   Dim pen As New Drawing.Pen(System.Drawing.Color.Brown, 1)
   e.Graphics.DrawEllipse(pen, 0, 0, 100, 100)
   pen.Dispose()
End Sub
```

Note: It may be necessary to use the **ZOrder** method to achieve the same visual layout as the original application.

Table 9.1 lists Visual Basic 6.0 graphics properties and methods and their Visual Basic .NET equivalents.

Table 9.1: Graphics Properties and Methods

Visual Basic 6.0	Visual Basic .NET equivalent
AutoRedraw property	No equivalent. To create persistant graphics, put graphics methods in the **Paint** event handler.
Circle method	**Graphics.DrawEllipse** method
ClipControls property	No equivalent. To repaint only part of an object, create a clip in the **Paint** event handler using the **Graphics.SetClip** method.
Cls method	**Graphics.Clear** method
CurrentX property	The **x** parameter of various graphics methods. For example, **DrawRectangle**(pen, x, y, width, height).
CurrentY property	The **y** parameter of various graphics methods. For example, **DrawRectangle**(pen, x, y, width, height).
DrawMode property	**Pen.Color** property
DrawStyle property	**Pen.PenType** property
DrawWidth property	**Pen.Width** property
FillColor property	**SolidBrush.Color** property
FillStyle property	**Pen.Brush** property
HasDC property	No equivalent. Device contexts are no longer necessary with GDI+.
HDC property	No equivalent. Device contexts are no longer necessary with GDI+.
Image property	**Image** property. In Visual Basic 6.0, the **Image** property returned a handle; in Visual Basic .NET, it returns a **System.Drawing.Image** object.
Line method	**Graphics.DrawLine** method
PaintPicture method	**Graphics.DrawImage** method
Point method	No equivalent for forms or controls. For bitmaps, use **Bitmap.GetPixel** method.
Print method	**Graphics.DrawString** method
Pset method	No equivalent for forms or controls. For bitmaps, use **Bitmap.SetPixel** method.
TextHeight property	**Graphics.MeasureString** method. This method returns a **System.Drawing.SizeF** structure that represents the size, in pixels, of the specified string as drawn with a given font. The **SizeF.Height** property can be used to get the text height.
TextWidth property	**Graphics.MeasureString** method. This method returns a **System.Drawing.SizeF** structure that represents the size, in pixels, of the specified string as drawn with a given font. The **SizeF.Width** property can be used to get the text width.

Handling Changes to the PopupMenu Method

One of the most frequently used features of Visual Basic 6.0 **Menu** controls is the **PopupMenu** method, which displays a pop-up menu on an **MDIForm** or **Form** object. The pop-up menu can be displayed at the current mouse location or specified coordinates. Usually, this method is used in combination with mouse events.

The following code example displays a pop-up menu at the cursor location when the user right-clicks a form. This example assumes the existence of a form that includes a **Menu** control named **mnuFile** with at least one menu item. The form's code contains the following **MouseDown** event handler.

```
…
Private Sub Form_MouseDown (Button As Integer, Shift As Integer, _
                            X As Single, Y As Single)
   If Button = 2 Then
      PopupMenu mnuFile
   End If
End Sub
…
```

When the code is upgraded to Visual Basic .NET, the **PopupMenu** instruction does not change, but it is marked with upgrade warnings as shown here.

```
' UPGRADE_ISSUE: Form method Form1.PopupMenu was not upgraded.
```

The upgraded code causes compile time errors in Visual Basic .NET. To get the same functionality, you have to replace the upgraded code with instructions that use the **ContextMenu** object. The following code sample demonstrates this strategy.

```
Private Sub Form1_MouseDown(ByVal eventSender As System.Object, _
        ByVal eventArgs As System.Windows.Forms.MouseEventArgs) _
        Handles MyBase.MouseDown
    Dim Button As Short = eventArgs.Button \ &H100000
    Dim Shift As Short = _
        System.Windows.Forms.Control.ModifierKeys \ &H10000
    Dim X As Single = VB6.PixelsToTwipsX(eventArgs.X)
    Dim Y As Single = VB6.PixelsToTwipsY(eventArgs.Y)
    If Button = 2 Then
        Dim popupmenu As ContextMenu
        popupmenu = New ContextMenu
        popupmenu.MergeMenu(mnuFile)
        popupmenu.Show(Me, New Point(eventArgs.X, eventArgs.Y))
    End If
End Sub
```

The preceding code example uses a **ContextMenu** object to allow a pop-up menu to be displayed. The **MergeMenu** instruction adds the items that are in the **mnuFile Menu** control.

If your original code displayed the pop-up menu at specified coordinates, you can specify the desired coordinates in the **Show** method. For example, the original pop-up menu code may look like this.

```
PopupMenu mnuFile, , 10, 10
```

If it does, you can specify the display coordinates in Visual Basic .NET, as shown here.

```
popupmenu.Show(Me, New Point(10, 10))
```

Not all features of Visual Basic 6.0 pop-up menus are available in Visual Basic .NET. ***Flags*** and ***boldcommand*** arguments have no equivalent. Menu items in Visual Basic .NET can not be set to bold with version 1.1 of the .NET Framework unless you use owner-drawn menus. However, this ability has been added to version 2.0. The pop-up menu position depends entirely on the position specified in the parameters of the **Show** method, and the pop-up menu reacts to a mouse click only when you use the left mouse button. Additional visual effects, such as bold items, can be achieved in Visual Basic .NET by using the **MenuItem.OwnerDraw** property to specify that a menu item will be drawn by the owner. This technique requires a user-defined event handler for the **MenuItem.DrawItem** event and the usage of the **Graphics** object. For more information, see "MenuItem.DrawItem Event" in the *.NET Framework Class Library* on MSDN.

Handling Changes to the ClipControls Property

Clipping is the process of determining which parts of a form or container, such as a **Frame** or **PictureBox** control, are painted when the form is displayed. An outline of the form and controls is created in memory. The Windows operating system uses this outline to paint some parts, such as the background, without affecting other parts, such as the contents of a **TextBox** control. Because the clipping region is created in memory, setting this property to **False** can reduce the time to paint or repaint a form.

The **ClipControls** property can be set at design time. At run time, this property returns a value that determines whether graphics methods in **Paint** events repaint the entire object or only newly exposed areas. It also determines whether the Windows operating system creates a clipping region that excludes nongraphical controls contained by the object.

Visual Basic .NET does not support the **ClipControls** property. When your application is upgraded and this property is referenced in the code, it is left as-is and is marked with upgrade warnings, as shown here.

```
' UPGRADE_ISSUE: Form property Form1.ClipControls was not upgraded.
```

To correct this issue, you must replace references to the property with a call to the **Graphics.SetClip** method. By providing an event handler for the **Paint** event, you can access the **Graphics** object associated with the control that is going to be drawn. Using this object, the **SetClip** method can be called to specify a clipping region.

This is a somewhat advanced technique that requires additional work; before writing custom drawing routines you should test the drawing performance of your upgraded application. The **ClipControls** property is used to improve performance in hardware that has limited capabilities. Modern high-performance computers and video cards are often sufficient to render displays without the need for clipping. It is possible that your upgraded application has enough performance on the current target systems without low-level handling of clipping regions. If this is the case, you can remove the code related to the **ClipControls** property instead of upgrading it.

Drag-and-Drop Functionality

Drag-and-drop capability is an important feature of most modern form-based applications. There are a number of important differences in drag-and-drop functionality between Visual Basic 6.0 and Visual Basic .NET. This section provides an overview of the differences so that you can better understand how they will affect your application.

Drag-and-Drop Functionality in Visual Basic 6.0

Visual Basic 6.0 supports two kinds of drag-and-drop operations: standard drag-and-drop functionality, which supports moving items between controls within a single form, and OLE drag-and-drop functionality, which supports moving items between applications. This section focuses on OLE drag-and-drop functionality.

The standard Visual Basic 6.0 controls have varying degrees of support for OLE drag-and-drop functionality.

Table 9.2 lists the standard Visual Basic 6.0 controls and their support for manual and automatic drag-and-drop operations.

Table 9.2: Visual Basic 6.0 Controls: Support for OLE Drag-and-Drop Functionality

Controls	OLEDragMode	OLEDropMode
TextBox, **PictureBox**, **Image**, **RichTextBox**, **MaskedBox**	**VbManual, vbAutomatic**	**vbNone, vbManual, vbAutomatic**
ComboBox, **ListBox**, **DirListBox**, **FileListBox**, **DBCombo**, **DBList**, **TreeView**, **ListView**, **ImageCombo**, **DataList**, **DataCombo**	**VbManual, vbAutomatic**	**vbNone, vbManual**
Form, **Label**, **Frame**, **CommandButton**, **DriveListBox**, **Data**, **MSFlexGrid**, **SSTab**, **TabStrip**, **Toolbar**, **StatusBar**, **ProgressBar**, **Slider**, **Animation**, **UpDown**, **MonthView**, **DateTimePicker**, **CoolBar**	Not supported	**vbNone**, **vbManual**

Figure 9.1 on the next page outlines the life cycle of an OLE drag-and-drop operation in Visual Basic 6.0.

Although drag-and-drop functionality is supported in Visual Basic .NET, there are differences. The next section details drag-and-drop functionality in Visual Basic .NET.

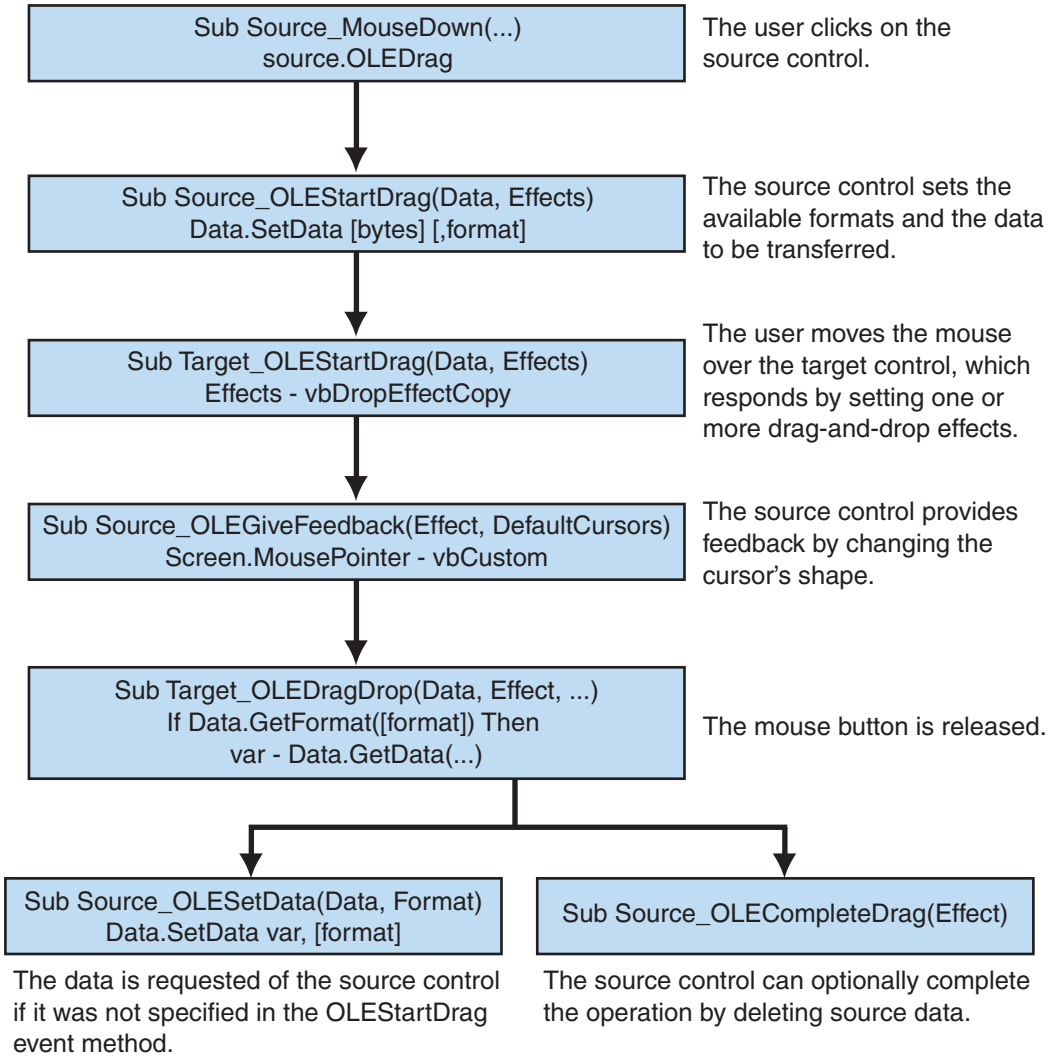

Figure 9.1
The life cycle of a Visual Basic 6.0 drag-and-drop operation

Drag-and-Drop Functionality in Visual Basic .NET

In Visual Basic .NET, the drag-and-drop operations are consolidated into a single framework. Drag-and-drop operations between controls is handled the same as drag-and-drop operations between applications. This change simplifies the programming burden for new development, but it requires rewriting drag-and-drop code for existing applications.

Figure 9.2 outlines the life cycle of a drag-and-drop procedure in Visual Basic .NET. Note that this cycle applies to both control-to-control and application-to-application drag-and-drop operations.

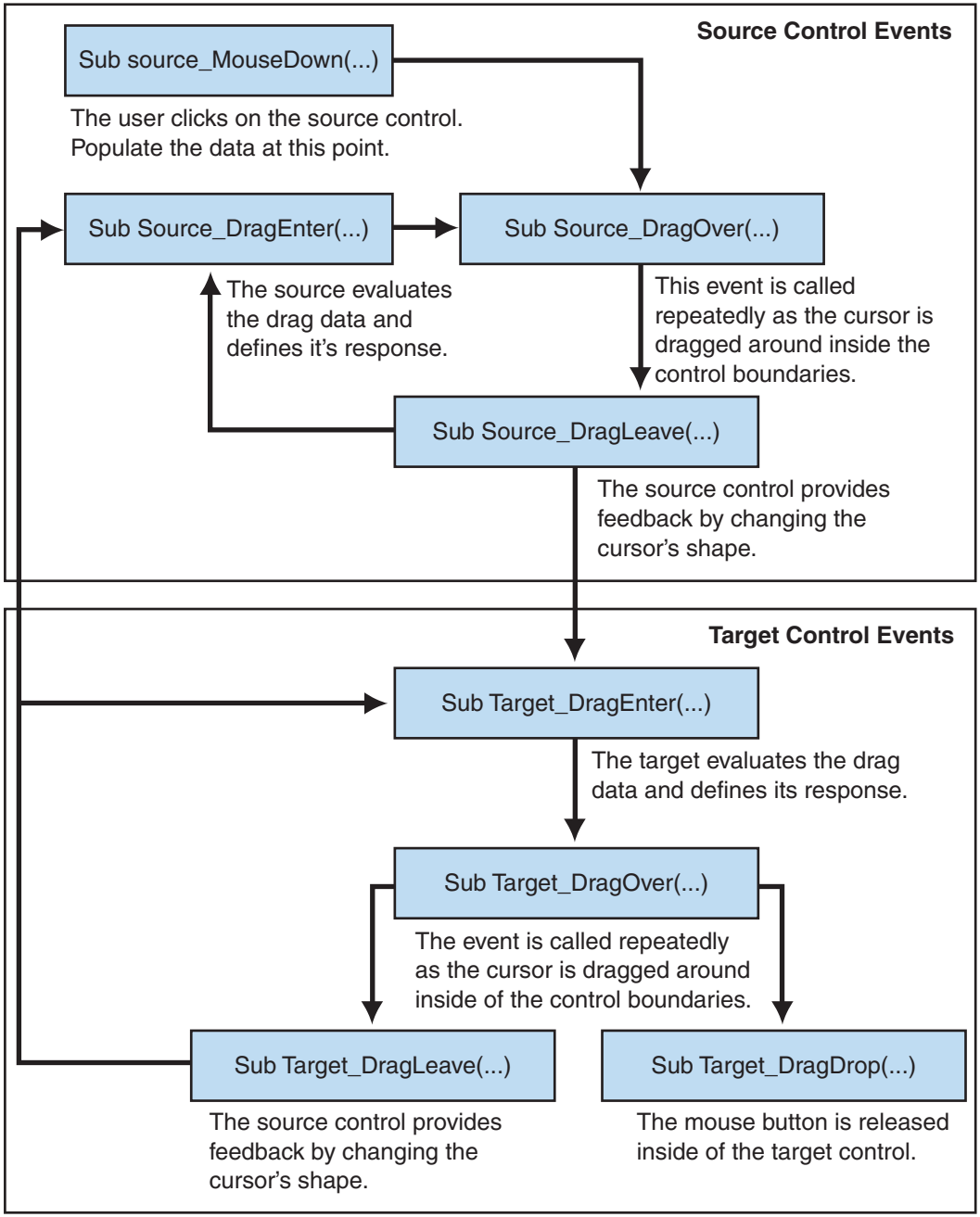

Figure 9.2
Life cycle of a Visual Basic .NET drag-and-drop operation

The changes necessary to make the drag-and-drop code work in Visual Basic .NET are fairly significant. Because of changes in the drag-and-drop programming model, the upgrade wizard cannot automatically make the modifications. Instead, the

methods are unchanged, and you have to implement the logic in the Visual Basic .NET drag-and-drop event model. The following code example shows how drag-and-drop operations are handled in Visual Basic 6.0.

```
Private Sub MyPictureBox_MouseDown(Button As Integer, Shift As Integer, _
            X As Single, Y As Single)
    MyPictureBox.OLEDrag
End Sub

Private Sub MyPictureBox_OLEDragDrop(Data As DataObject, Effect As Long, _
            Button As Integer, Shift As Integer, X As Single, Y As Single)

    If Data.GetFormat(vbCFFiles) Then
        Dim i As Integer
        For i = 1 To Data.Files.Count
            Dim ImagePath As String
            Dim Index As Integer

            ImagePath = Data.Files(i)
            Index = ImageIndexOf(ImagePath)
            If Index = -1 Then
                Select Case UCase(Right(ImagePath, 4))
                    Case ".BMP", ".JPG", ".GIF"
                        ImageList.AddItem ImagePath
                        ImageList.ListIndex = ImageList.ListCount - 1

                        Set MyPictureBox.Picture = LoadPicture(ImagePath)
                End Select
            Else
                ImageList.ListIndex = Index
            End If
        Next
    End If

End Sub

Private Sub MyPictureBox_OLEStartDrag(Data As DataObject, AllowedEffects As Long)
    Data.Files.Clear
    Data.Files.Add ImageList
    Data.SetData , vbCFFiles
    AllowedEffects = vbDropEffectCopy
End Sub

Private Function ImageIndexOf(ImagePath As String) As Integer
    Dim i As Integer

    For i = 0 To ImageList.ListCount
        If ImageList.List(i) = ImagePath Then
            ImageIndexOf = i
            Exit Function
        End If
    Next
    ImageIndexOf = -1
End Function
```

Instead of upgrading the Visual Basic 6.0 code, it is necessary to replace it with the Visual Basic .NET drag-and-drop event model. In the Visual Basic .NET event model, information is passed back and forth in a different way. The following code example shows how to implement the equivalent drag-and-drop operation demonstrated in the preceding Visual Basic .NET.

```
Private Sub MyForm_Load(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    MyPictureBox.AllowDrop = True
    Application.DoEvents()
End Sub

Private Sub MyPictureBox_DragDrop(ByVal sender As Object, _
            ByVal e As System.Windows.Forms.DragEventArgs) _
            Handles MyPictureBox.DragDrop
    ' Clear out the existing image and ensure that any resources
    ' are released.
    If Not MyPictureBox.Image Is Nothing Then
        MyPictureBox.Image.Dispose()
        MyPictureBox.Image = Nothing
    End If

    Dim Images() As String = e.Data.GetData(DataFormats.FileDrop, True)

    Dim i As Integer
    Dim Index As Integer

    For i = 0 To Images.Length - 1
        Index = ImageIndexOf(Images(i))
        If Index = -1 Then
            ImageList.SelectedIndex = ImageList.Items.Add(Images(i))
        Else
            ImageList.SelectedIndex = Index
        End If
    Next

    ' Set the picture to the last image that was added.
    MyPictureBox.Image = Image.FromFile(ImageList.SelectedItem)
End Sub

Private Sub PictureDisplay_DragEnter(ByVal sender As Object, _
            ByVal e As System.Windows.Forms.DragEventArgs) _
            Handles MyPictureBox.DragEnter
    If e.Data.GetDataPresent(DataFormats.FileDrop) Then
        Dim FileType As String
        Dim file() As String = e.Data.GetData(DataFormats.FileDrop, True)

        FileType = UCase(Microsoft.VisualBasic.Right(file(0), 4))
        Select Case FileType
            Case ".BMP", ".JPG", ".GIF"
                e.Effect = DragDropEffects.Copy
            Case Else
                e.Effect = DragDropEffects.None
```

```
            End Select
        Else
            e.Effect = DragDropEffects.None
        End If
    End Sub

    Private Sub PictureDisplay_MouseDown(ByVal sender As Object, _
                ByVal e As System.Windows.Forms.MouseEventArgs) _
                Handles MyPictureBox.MouseDown
        Dim file(0) As String
        file(0) = ImageList.SelectedItem

        Dim d As New DataObject(DataFormats.FileDrop, file)
        ImageList.DoDragDrop(d, DragDropEffects.Copy)
    End Sub

    Private Function ImageIndexOf(ByRef ImagePath As String) As Short
        Dim i As Short

        For i = 0 To ImageList.Items.Count
            If VB6.GetItemString(ImageList, i) = ImagePath Then
                ImageIndexOf = i
                Exit Function
            End If
        Next
        ImageIndexOf = -1
    End Function
```

This example should give you an idea of the kind of work you have to do to replace drag-and-drop functionality in Visual Basic .NET.

Handling Changes to the MousePointer and MouseIcon Properties

In Visual Basic 6.0, you can use the **MousePointer** and **MouseIcon** properties to display a custom icon, cursor, or any one of a variety of predefined mouse pointers. Changing the mouse pointer gives you a way to inform the user of a variety of activities or possible activities. For example, it can inform the user that long background tasks are processing, that a control or window can be resized, or that a particular control does not support drag-and-drop operations. Using custom icons or mouse pointers, you can express an endless range of visual information about the state and functionality of your application. Each pointer option is represented by an integer value setting.

At run time, you can set the value of the mouse pointer either by using the integer values or the Visual Basic mouse pointer constants. For example, the following line of code sets the mouse pointer to an hourglass display.

```
Form1.MousePointer = vbHourglass
```

To use a custom icon or cursor, set the **MousePointer** property to **vbCustom** and load an icon file or cursor file into the **MouseIcon** property. The following code example demonstrates setting the mouse pointer to a custom image.

```
Form1.MousePointer = vbCustom
Form1.MouseIcon = LoadPicture("c:\Icons\EW_06.CUR")
```

The upgrade of predefined mouse pointers of Visual Basic 6.0 is automatic. However, additional work is required for custom mouse pointers. When such code is upgraded with the upgrade wizard, the code is unchanged but marked with upgrade warnings as shown here.

```
' UPGRADE_ISSUE: Form property Form1.MousePointer does not support
' custom mousepointers.

' UPGRADE_ISSUE: Form property Form1.MouseIcon was not upgraded.
```

The upgraded code causes compile errors in Visual Basic .NET. To get the same functionality, you must replace it with code that uses the **System.Windows.Forms.Cursor** object, as shown here.

```
Me.Cursor = New Cursor("c:\Icons\EW_06.CUR")
```

Note that custom mouse pointers are not supported at design time in Visual Basic .NET; the **MousePointer** property is replaced by the **Cursor** property, and the **MouseIcon** property no longer exists. However, you can load your custom mouse pointer in the **Load** event.

Table 9.3 lists the Visual Basic 6.0 mouse pointer constants and the Visual Basic .NET equivalents.

Table 9.3: Mouse Pointer Constants

Visual Basic 6.0	Value	Visual Basic .NET Equivalent
vbDefault	0	**Windows.Forms.Cursors.Default**
vbArrow	1	**Windows.Forms.Cursors.Arrow**
vbCrosshair	2	**Windows.Forms.Cursors.Cross**
vbIbeam	3	**Windows.Forms.Cursors.IBeam**
vbIconPointer	4	**Windows.Forms.Cursors.Default**
vbSizePointer	5	**Windows.Forms.Cursors.SizeAll**
vbSizeNESW	6	**Windows.Forms.Cursors.SizeNESW**
vbSizeNS	7	**Windows.Forms.Cursors.SizeNS**

continued

Visual Basic 6.0	Value	Visual Basic .NET Equivalent
vbSizeNWSE	8	**Windows.Forms.Cursors.SizeNWSE**
vbSizeWE	9	**Windows.Forms.Cursors.SizeWE**
vbUpArrow	10	**Windows.Forms.Cursors.UpArrow**
vbHourglass	11	**Windows.Forms.Cursors.WaitCursor**
vbNoDrop	12	**Windows.Forms.Cursors.No**
vbArrowHourglass	13	**Windows.Forms.Cursors.AppStarting**
vbArrowQuestion	14	**Windows.Forms.Cursors.Help**
vbSizeAll	15	**Windows.Forms.Cursors.SizeAll**
vbCustom	99	No direct equivalent. Use the example shown in this section to replace code.

Handling Changes to Property Pages

Visual Basic 6.0 property pages allow you to work around the limitations of the Visual Basic **Property Browser**. For example, you can use property pages to give users a way to add a collection of colors to a color list user control. In the property page, you would write code that manages the collection, something beyond the capabilities of the Visual Basic 6.0 **Property Browser**. In contrast, the Visual Basic .NET **Property Browser** can be used to edit any .NET variable type or class. Property pages are no longer needed.

Property pages in user controls are not automatically upgraded by the upgrade wizard. To enable your control to edit the values that used to be exposed in the control's property pages, you need to add properties to your user control. The good news is that the Visual Basic .NET **Property Browser** automatically recognizes the types of your control's properties and allows users to edit them in the **Property Browser**. In the case of a color collection, you simply add the property to your user control, and the Visual Basic .NET **Property Browser** provides a color collection editor.

The following code example demonstrates this mechanism. Add the following code to a Windows Forms user control.

```
Dim ColorsCollection() As Color
Property Colors() As Color()
    Get
        Return ColorsCollection
    End Get
    Set(ByVal Value As Color())
        ColorsCollection = Value.Clone
    End Set
End Property
```

If you add the user control to a form, you will notice that the control has a new property, named **Colors**. When you access this property, the Collection editor is displayed, with all colors that are contained in the collection. If you try to change the color of a collection's item, Visual Studio .NET automatically shows the Colors editor, as shown in Figure 9.3.

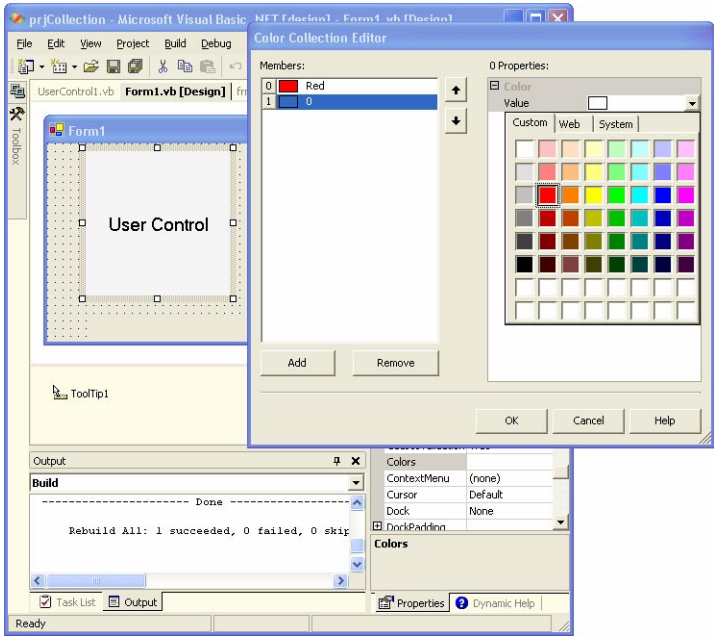

Figure 9.3
Adding a color collection to your user control

The advantage of showing properties in the **Property Browser** instead of in property pages is that all of the properties for your control are visible and easily discoverable. Compare this with Visual Basic 6.0 property pages, which hide functionality from clients who use your controls.

In Visual Basic 6.0, the data included in property pages is usually persisted using a **PropertyBag** object. The **ReadProperties** and **WriteProperties** events are used to retrieve or save a **UserControl**'s values to a **PropertyBag** object. In Visual Basic .NET, the **PropertyBag** object is no longer supported, and the **ReadProperties** and **WriteProperties** events no longer exist.

However, you can still persist an object's data using the **System.Runtime.Serialization** and **System.IO** namespaces. Serialization in .NET requires using the **Serializable** class attribute. A **Stream** object and a **BinaryFormatter** or **SoapFormatter** object can be used to retrieve the object's data from a file when instantiating the corresponding object. **Stream** and

BinaryFormatter objects can be used to write the object's values to the file before destroying the object. For more information about making an object's data persist, see "Property Bag Changes in Visual Basic .NET" on MSDN.

To change and manage properties in Visual Basic .NET, use **Property Editors** and **Designers**. When editing properties, a visual designer should create a new instance of the specified editor through a dialog box or drop-down list. The base class for editors is **System.Drawing.Design.UITypeEditor**, which is used in conjunction with **System.ComponentModel.EditorAttribute**. For example, this Visual Basic .NET code creates the **MyImage** class, which is marked with an **EditorAttribute** that specifies the **ImageEditor** as its editor.

```
<Editor("System.Windows.Forms.ImageEditorIndex, System.Design", _
    GetType(UITypeEditor))> _
Public Class MyImage
    …
End Class 'MyImage
```

The **Designer** attribute (**System.ComponentModel.DesignerAttribute**) specifies the class used to implement design-time services for a component. The following Visual Basic .NET code creates a class named **MyForm**. **MyForm** that has two attributes: a **DesignerAttribute** that specifies this class uses the **DocumentDesigner**, and a **DesignerCategoryAttribute** that specifies the **Form** category.

```
<Designer("System.Windows.Forms.Design.DocumentDesigner,
System.Windows.Forms.Design.DLL", _
    GetType(IRootDesigner)), DesignerCategory("Form")> _
Public Class MyForm
    Inherits ContainerControl
    …
End Class 'MyForm
```

To implement a functionality equivalent to Visual Basic 6.0 property pages in Visual Basic .NET, it is necessary to use some .NET Framework classes and member attributes. The first class you should consider is **System.Drawing.Design.UITypeEditor**. This class provides basic functionality that you can use or derive from to implement a custom editor for the design-time environment. Custom editors can be applied to the data types specified by the developer and are useful in situations where simple text value editors are insufficient. The following code example presents a class derived from **UITypeEditor** that provides a color chooser for design-time property value edition. For a complete example of these techniques, see ".NET Samples – Windows Forms: Control Authoring" in *.NET Framework QuickStarts* on MSDN.

```
…
Public Class FlashTrackBarValueEditor
    Inherits UITypeEditor
```

```
    Private edSvc As IWindowsFormsEditorService

    ' This method is used to handle the user interface, user
    ' input processing, and value assignment.
    Public Overloads Overrides Function EditValue(ByVal context As _
        ITypeDescriptorContext, ByVal provider As IServiceProvider, _
        ByVal value As Object) As Object
        If (Not (context Is Nothing) And Not (context.Instance Is Nothing) And _
            Not (provider Is Nothing)) Then
            edSvc = CType(provider.GetService( _
                GetType(IWindowsFormsEditorService)), IWindowsFormsEditorService)
            If Not (edSvc Is Nothing) Then
                ' This is where the corresponding control can be visually updated.
                Dim trackBar As FlashTrackBar = New FlashTrackBar
                SetEditorProps(CType(context.Instance, FlashTrackBar), TrackBar)
                edSvc.DropDownControl(trackBar)
                ' Usually, performing this assignment on the affected property
                ' requires additional validations.
                value = trackBar.Value
            End If
        End If
        Return value
    End Function

    ' This method is used to inform the Properties window of the type
    ' of editor style that the editor will use.
    Public Overloads Overrides Function GetEditStyle( _
        ByVal context As ITypeDescriptorContext) As UITypeEditorEditStyle
        If (Not (context Is Nothing) And Not (context.Instance Is Nothing)) Then
            Return UITypeEditorEditStyle.DropDown
        End If
        Return MyBase.GetEditStyle(context)
    End Function
End Class
…
```

The previous class can be used to specify a custom editor for one of the properties of a **UserControl**. In this case, a **UserControl** named **FlashTrackBar** is defined to provide functionality similar to a progress bar with a gradient color. This **UserControl** has two properties that indicate the start and end color whose value will be modified at design time with the custom editor that was previously defined. To specify the custom editor for a property, the **EditorAttribute** attribute class must be used, as shown here.

```
…
<Category("Flash"), _
Editor(GetType(FlashTrackBarDarkenByEditor), GetType(UITypeEditor)), _
DefaultValue(200)> _
Public Property DarkenBy() As Byte
    Get
        Return myDarkenBy
    End Get
```

```
    Set(ByVal Value As Byte)
        …
    End Set
End Property
…
```

Design-time services for a component can be specified with the **DesignerAttribute** and **DesignerCategoryAttribute** class attributes. For example, the following code example specifies that an instance of the **DocumentDesigner** class will provide design time behavior and view of the **MyDialog** control. The **DocumentDesigner** class is a base designer class used for extending the design mode behavior of a control that supports nested controls and can handle scroll messages. This class also provides design mode view for the affected control.

```
<Designer("System.Windows.Forms.Design.DocumentDesigner, _
System.Windows.Forms.Design.DLL", GetType(IRootDesigner)), _
DesignerCategory("Form")> _
Public Class MyDialog
    Inherits ContainerControl
    …
End Class
```

Handling Changes to the OLE Container Control

The OLE Container control is used to dynamically add objects to a form at run time. For example, you can use an OLE Container to create a control that contains an embedded Microsoft Word document, create a control that contains a linked Word document, or bind to an OLE object in a database.

Visual Basic .NET does not support the OLE Container control. As a result, the upgrade wizard is unable to upgrade an OLE Container control. Instead, it replaces the control with a red label to highlight the need to rewrite the code.

Figure 9.4 illustrates a sample Visual Basic 6.0 application with an embedded OLE Container control holding an embedded Microsoft Word document.

Figure 9.5 illustrates the Visual Basic .NET version after applying the upgrade wizard.

The code produced by the upgrade wizard is marked with UPGRADE_ISSUE comments wherever OLE Container controls appear. The following is an example of such a comment.

```
' UPGRADE_ISSUE: Ole method OLE1.CreateLink was not upgraded.
```

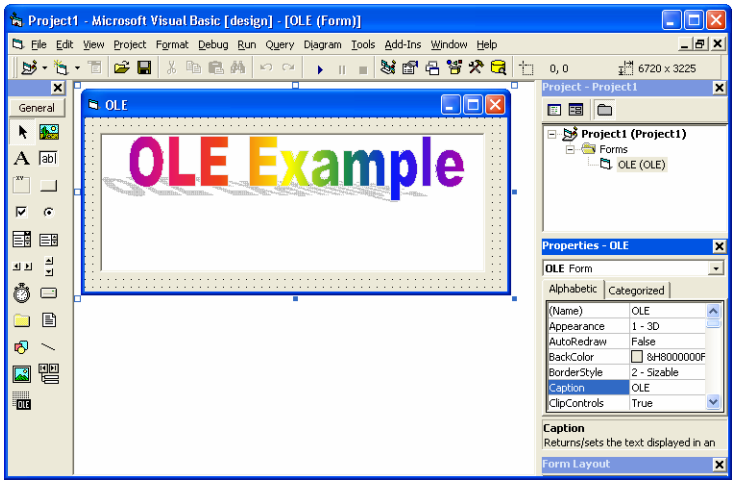

Figure 9.4
OLE Container control with linked object in Visual Basic 6.0

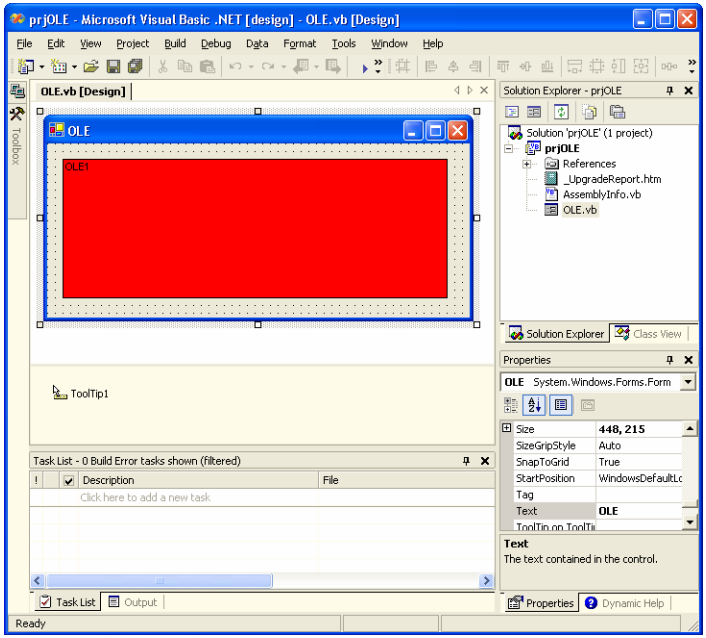

Figure 9.5
OLE Container control after applying the upgrade wizard

A strategy for replacing the OLE Container is to use a **WebBrowser** ActiveX control in place of the OLE Container to display the Word document in the form. Through this control, you can display, edit, and save documents displayed in the control.

Figure 9.6 illustrates an updated version of the Visual Basic .NET code that has replaced the OLE Container with a **WebBrowser** ActiveX control to display the Word document.

Figure 9.6
A Microsoft Word document displayed using the WebBrowser control

► To replace the OLE Container with a WebBrowser ActiveX control

1. Remove the red label inserted by the upgrade wizard as a placeholder for the OLE Container.
2. Add a **WebBrowser** control to the form. You can do this by right-clicking in the **ToolBox**, clicking **Add/Remove Items**, clicking the **COM Components** tab, and then clicking **Microsoft WebBrowser Control** in the **Customize Toolbox** component picker.
3. After you add the control, add the following code to the **Form_Load** subroutine (or wherever you want to bind to the document):

```
Me.AxWebBrowser1.Navigate("C:\Temp\LinkedDoc.doc")
```

   Be sure to substitute the file name in this example for the location of the object to which you are linking. After completing these steps, you should be able to rebuild the solution.

Note that this method works for most objects that can be displayed in Microsoft Internet Explorer, such as HTML files, Word documents, spreadsheets, GIFs, and JPEG files.

Handling Changes to Control Arrays

Visual Basic 6.0 provides control arrays for managing the controls on your forms. Control arrays allow sharing event procedures for a set of controls that share the same name and type. Also, they provide a mechanism for iterating through a set of controls and for adding controls at run time.

There were limits to control arrays. You can only put controls of the same type in a control array. So, for example, if one of your input boxes is a masked edit control, it cannot be in the same control array as your text boxes.

Control arrays are not consistent in Visual Basic 6.0. They are not quite collections and not quite arrays. This is why they are not provided in Visual Basic .NET. Instead, Visual Basic .NET has a rich set of features that provide all of the benefits of control arrays without the limitations.

In Visual Basic 6.0 and earlier versions, there were three main reasons that developers used control arrays:

1. To allow one event routine to be hooked to several controls
2. To access controls as a collection in a For Each loop
3. To dynamically add and remove controls in forms

Each of these tasks can be achieved in Visual Basic .NET using the available mechanisms for event handling, collections, and dynamic control creation. These techniques are explained in the rest of this section. For more information, see "Getting Back Your Visual Basic 6.0 Goodies" on MSDN.

Event Handling

Control arrays in Visual Basic 6.0 allow you to define one set of event procedures for all controls in the control array. In Visual Basic .NET, this functionality can be achieved for controls defined at design time and created at run time. The event handling mechanism allows controls to share event handler procedures without the need for control arrays. The following code demonstrates an event handler in Visual Basic .NET.

```
Private Sub ProcessGotFocus(ByVal sender As Object, ByVal e As System.EventArgs)
_
                           Handles Text1.GotFocus
   'Event handling code goes here
End Sub
```

Visual Basic .NET event handlers use the **Handles** keyword to define which events the handler will manage. In the preceding example, the **GotFocus** event for **Text1** is handled by the **ProcessGotFocus** subroutine. If you need share an event handler

with other controls, you should add the other events to the **Handles** clause. The following sample code demonstrates this.

```
Private Sub ProcessGotFocus(ByVal sender As Object, ByVal e As System.EventArgs)
_
                            Handles Text1.GotFocus, Text2.GotFocus,
MaskBox1.GotFocus
    ' Event handling code goes here.
End Sub
```

Now, the **ProcessGotFocus** subroutine will handle the **GotFocus** event for the controls **Text1**, **Text2**, and **MaskBox1**. With the handles clause, you can share event handlers for controls of different types and even for different events as long as the handler procedure parameters correspond to the parameters of the event declaration.

Accessing Control Arrays as a Collection

Visual Basic .NET provides extensive support for collections and other types of structures that can be used to hold groups of controls. It is convenient to loop through the controls in a set like that to update common properties or modify the behavior of the controls.

Because there are no control arrays in Visual Basic .NET, the user must explicitly build a collection of objects and add all the required objects. In this way, the user can create different groupings of objects according to the current needs.

The following code example shows a method that groups three **Label** controls in an **ArrayList** class that can be used as a collection in a **For Each** loop. The example assumes that the code is part of a form that has three **Label** controls. The **Demo_Load** procedure sets the **UseMnemonic** property to **True** for all labels.

```
Dim labelCollection As ArrayList
Private Sub GroupLabels()
   labelCollection = New ArrayList
   labelCollection.Add(Label1)
   labelCollection.Add(Label2)
   labelCollection.Add(Label3)
End Sub

Private Sub Demo_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   Dim obj As Control
   For Each obj In labelCollection
      Dim curLabel As Label
      curLabel = CType(obj, Label)
      curLabel.UseMnemonic = True
   Next
End Sub
```

Visual Studio .NET provides a collection that holds all the controls that have been included in a particular form; the management of this collection is done in an automatic way. It is exposed as the **Controls** property of the **Form** class, and its type is **ControlsCollection**. For more information about this collection, see the "Upgrading the Controls Collection" section in Chapter 7, "Upgrading Commonly-Used Visual Basic 6.0 Objects."

Adding Controls Dynamically

Another important goal of control arrays in Visual Basic 6.0 is that they provide an easy way to add controls at run time. You can create the control array at design time and then add controls of the same type to the array at run time. In Visual Basic .NET, you do not need a control array to add controls at run time. You can use the **Controls** collection to add controls of any type at run time; this will be shown in the next code example.

In the case where controls are dynamically added to a form according to the original Visual Basic 6.0 application logic, it will be necessary to dynamically connect the new controls with the preexisting event handlers. This can be done with the **AddHandler** statement, which ties events raised by controls with specific procedures that can handle the event. An example of this technique is shown in the following Visual Basic .NET code example.

```
Sub Addbutton()
    Dim newButton As New System.Windows.Forms.Button
    AddHandler newButton.Click, AddressOf General_Click_Handler
    Me.Controls.Add(newButton)
End Sub
Private Sub General_Click_Handler(ByVal sender As System.Object, _
ByVal e As System.EventArgs)
    System.Windows.Forms.MessageBox.Show("A control was clicked")
End Sub
```

The **AddButton** procedure creates a new **Button** control and connects its **Click** event with the **General_Click_Handler** event handler; it then adds the new control to the corresponding form's **Controls** collection, which will display the button on the form.

Visual Basic .NET supports control arrays through the Visual Basic .NET compatibility library. With it, you can simplify the upgrade process while retaining some of the Visual Basic 6.0 features. However, as mentioned in the "Upgrading the Controls Collection" section of Chapter 7, "Upgrading Commonly-Used Visual Basic 6.0 Objects," use of the compatibility library instead of the .NET Framework should be analyzed according to the available resources and the advantages for the future advancement of the application.

There are third-party components for Visual Studio .NET that provide functionality similar to Visual Basic 6.0 control arrays, including design-time support. One of these components is the **ControlArray** control that is available on the .NET Framework Windows Forms Web site. This component allows the user to create logical groupings of controls on a form that will share the **ControlArray** event handlers. It also allows the user to set properties or call methods for all the contained controls at the same time through the **ControlArray** container, using the **System.Windows.Forms.Control** interface.

Handling Changes to DDE Functionality

At one time, Dynamic Data Exchange (DDE) was one of the few ways you could pass data between applications without using a shared file. Support for DDE in Visual Basic has been present since version 1. However, DDE has since been replaced by COM.

With COM, you can share data and invoke methods between applications in a more efficient and scalable manner. Because most Windows-based applications that expose public data and methods do so by way of COM, DDE is not the primary way that you retrieve public data and invoke public methods in a Windows-based application. As a consequence, COM can be used as a means of communicating with most Windows-based applications.

After upgrading your application to Visual Basic .NET, it is highly recommended that you replace all DDE communication with COM. For example, a Visual Basic 6.0 application can call **LinkExecute** to send a command in a DDE conversation with a server application. In this case, it is advisable to look at the documentation or type library for the server application and see whether the functionality provided by the DDE command can be replicated with one or more of the COM methods. If such functionality is available, a COM reference can be added to the Visual Basic .NET project and the corresponding methods can be called.

Neither Visual Basic .NET nor the .NET Framework provides support for DDE. If you must communicate with another application using DDE, you will have to implement DDE in your Visual Basic .NET application in one of the following two ways:

- Interoperate with a user-defined ActiveX EXE server built in Visual Basic 6.0 that manages the DDE conversation.
- Declare and implement the DDE-related Windows API functions.

If you want to reuse your Visual Basic 6.0 DDE code, you can create a Visual Basic 6.0 ActiveX EXE project and add a public class to exchange the DDE-related information you need with your Visual Basic .NET application. For example, if you want

to perform a DDE **LinkExecute** operation in your Visual Basic .NET code, you create a DDE helper class written in Visual Basic 6.0 as follows:

1. Create a Visual Basic 6.0 ActiveX EXE project.
2. Add a public method to the class (default name **Class1**) named **LinkExecute** that takes two parameters: **ServerTopic** and **ExecCommand**.
3. Add a form, for example named **Form1**, to the ActiveX EXE project.
4. Add a **TextBox**, for example named **Text1**, to the form.
5. Add the following code to the **LinkExecute** method in **Class1**.

```
Dim f As New Form1
f.Text1.LinkMode = 0                'None
f.Text1.LinkTopic = ServerTopic
f.Text1.LinkMode = 2                'Manual
f.Text1.LinkExecute ExecCommand
```

6. Build the ActiveX EXE server, by default named **Project1.dll**.

Note that you can, and should, name the classes, forms, and controls more appropriately for your project. However, the name for the method **LinkExecute** should not change to keep it consistent with existing DDE code.

To perform the **LinkExecute** operation in Visual Basic .NET, add a COM reference to **Project1.dll** to your Visual Basic .NET project **References** list, and then enter the following code.

```
Dim DDEClientHelper As New Project1.Class1Class()
DDEClientHelper.LinkExecute("MyDDEServer•MyTopic", "MyCommand")
```

Implementing DDE in a Visual Basic .NET application requires calling the Windows DDE-related API functions; at minimum, it is necessary to call the methods **DdeInitialize** and **DdeUninitialize**. To call **DdeInitialize**, you have to implement **DdeCallbackProc**. Additionally, it will be necessary to call a number of other DDE-related API functions to establish a connection and send data. To implement functionality to perform **LinkExecute**, for example, requires calling at least eight other DDE-related API functions. This task also requires a deep understanding of Windows messaging architecture and memory management. This understanding is in addition to knowledge of how to represent Windows types — such as handles and structures — in Visual Basic code. This undertaking is not for the faint of heart. To learn more about the DDE-related Windows API, search for "Dynamic Data Exchange Management Functions" on MSDN. For more information about issues related to using **Declare** statements for Windows API function calls in your code, see the section "Type Changes" in Chapter 13, "Working with the Windows API."

Summary

Most Visual Basic 6.0 applications use forms. The forms architecture has been completely redone in Visual Basic .NET as the Windows Forms package to conform to the .NET Framework. The result is a flexible architecture for building robust forms-based applications, but the trade-off is that some Visual Basic 6.0 forms features must be manually upgraded.

The techniques provided in this chapter will help you achieve functional equivalence for those forms features that are no longer supported in Visual Basic .NET. However, you may find that this is an ideal time to consider redesigning the forms that drive your applications. If you are considering redesigning your forms, you can find a wealth of information about the Windows Forms package and tutorials to get you started on MSDN. A good starting point is "Building Windows Forms Applications" in the Microsoft .NET Framework Developer Center on MSDN.

More Information

For more information about the **MenuItem.DrawItem** event, see "MenuItem.DrawItem Event" in the *.NET Framework Class Library* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemwindowsformsmenuitemclassdrawitemtopic.asp.*

For more information about making an object's data persist, see "Property Bag Changes in Visual Basic .NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbtchpropertybagchangesinvisualbasicnet.asp.*

For more information about techniques for control authoring, see ".NET Samples – Windows Forms: Control Authoring" in *.NET Framework QuickStarts* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpqstart/html/cpsmpNETSamples-WindowsFormsControlAuthoring.asp.*

For more information about control arrays, see "Getting Back Your Visual Basic 6.0 Goodies" on MSDN: *http://msdn.microsoft.com/vbasic/using/columns/adventures/default.aspx?pull=/library/en-us/dnadvnet/html/vbnet05132003.asp.*

To download the **ControlArray** control, go to the .NET Framework Windows Forms Web site: *http://www.windowsforms.net/default.aspx?tabindex=6&tabid=47&ItemID=16&mid=142.*

For more information about the Windows Forms package, see "Building Windows Forms Applications" in the Microsoft .NET Framework Developer Center on MSDN: *http://msdn.microsoft.com/netframework/programming/winforms/.*

10

Upgrading Web Applications

Visual Basic 6.0 includes several features that support programming for the Web. These include Microsoft Internet Information Services (IIS) applications (Web classes), DHTML applications, ActiveX documents, and ActiveX controls that can be downloaded to Web pages.

Visual Basic .NET was built to support Web programming with features such as ASP.NET Web applications, XML Web services, and much more. Because Visual Basic .NET is built on a new architecture, Visual Basic 6.0 Web features either are no longer supported or have changed substantially; however, your knowledge of Web programming should help you to quickly transition to the new Web technologies.

In Visual Basic 6.0, IIS applications used the Active Server Pages (ASP) model to create applications that ran on IIS. In Visual Basic .NET, ASP.NET technology allows you to create an application using Web Forms pages and to create components using XML Web services. These technologies make programming for the Web very similar to programming for Windows in Visual Basic 6.0.

Visual Basic 6.0 DHTML applications used the Dynamic HTML object model and Visual Basic code to create applications that could respond to actions performed in a Web browser by a user. Visual Basic .NET Web Forms expand on the DHTML model, providing richer dynamic user-interface capabilities as well as client-side validation.

Visual Basic 6.0 ActiveX documents are not supported in Visual Basic .NET. You can still interoperate with ActiveX documents from your Visual Basic .NET Web applications, but development should be maintained in Visual Basic 6.0.

Like Visual Basic 6.0, Visual Basic .NET allows you to create ActiveX controls that can be downloaded to Web pages and to use existing ActiveX controls in your applications.

This chapter will explain the changes you will have to deal with when upgrading Web applications.

Upgrading ActiveX Documents

A feature available in Visual Basic 6.0 is the ability to create ActiveX documents; these are forms that can appear within Web browsers. ActiveX documents offer features such as built-in viewport scrolling, hyperlinks, and menu negotiation.

ActiveX documents are not supported in Visual Basic .NET. Instead, Web user controls can be used to achieve the same effect as ActiveX documents. Therefore, when dealing with applications that use ActiveX documents, you have three upgrade options:

- You can use ActiveX documents from your Visual Basic .NET Web applications through interoperability, but development must be maintained in Visual Basic 6.0. You can navigate from a Visual Basic .NET Web Form to a Visual Basic 6.0 ActiveX document, and vice-versa.
- You can rewrite your ActiveX documents as Web user controls. You will have to include the new controls in a Web Form to simulate the behavior of your ActiveX documents.
- You can create a Windows Form control that corresponds to the original ActiveX document and then host it in Microsoft Internet Explorer. This approach takes advantage of the similar features in both the source and target components, but nonetheless is limited with respect to deployment and compatibility.

To illustrate the upgrade of an ActiveX document to a Web user control, assume that you have an ActiveX document project that contains a user document named docCalculate, with three **TextBox** controls, a **Line** control, and **Button** control, as shown in Figure 10.1.

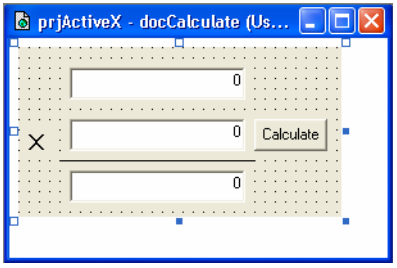

Figure 10.1
A sample Visual Basic 6.0 ActiveX document application

In the **Button**'s code, you the multiply the values of the first two **TextBox** controls, and display the result in the third **TextBox**. The event code looks like following.

```
Option Explicit
Private Sub btnCalculate_Click()
    On Error GoTo Exception
    If (txtFirstNumber.Text <> "") Then
```

```
        If (txtSecondNumber.Text <> "") Then
            Dim res As Single
            res = CSng(txtFirstNumber.Text) * CSng(txtSecondNumber.Text)
            txtResult.Text = CStr(res)
        End If
    End If
    Exit Sub
Exception:
    txtResult.Text = "####ERROR####"
End Sub
```

If you try to convert this project with the upgrade wizard, your user document will not be upgraded; however, the wizard will copy it unchanged to the upgraded project. This code will generate a compilation error stating that **Sub Main** was not found in **prjActiveX**. Also, the resulting project will be a Windows Forms project that does not exhibit the same behavior as an ActiveX document.

To upgrade this code, the first step is to create a user control for each user document that exists in the ActiveX document. A recommended naming strategy is to set the name for each control to the original name of the user document prefixed with "UC." You can then copy the controls and the code for each ActiveX document to its respective user control.

The next step is to convert your ActiveX document project using the upgrade wizard. This will only upgrade the user controls, so you will need to perform additional steps to create a document project. You can add an ASP.NET Web application to the solution to achieve this affect. A recommended naming strategy is to use the name of your original ActiveX document prefixed with "Web." In this example, the recommended name would be WebprjActiveX. Note that adding the ASP.NET Web application to your solution will result in Visual Studio .NET creating a virtual directory in your IIS with the same name. Additionally, you will need to add a Web Form to the project for each user document in the original Visual Basic 6.0 project. A suggested naming strategy for each Web Form is to use the original name of the user document. For this example, only one Web Form would be added, with its name set as docCalculate.aspx.

You will need to add a Web user control to the project for each user document in the original Visual Basic 6.0 project. For each such control, set the name as the original name. In this example, only one Web user control will be added, and its name will be **docCalculate**. When all the Web user controls have been added, you can copy the design of your user documents in its respective Web user control. In this example, you can add a grid layout panel to the control, three text box controls, a button control, and a label control. The respective names for each control will be the same as in the original Visual Basic 6.0 project. Figure 10.2 illustrates an example of the final resulting Visual Basic .NET form.

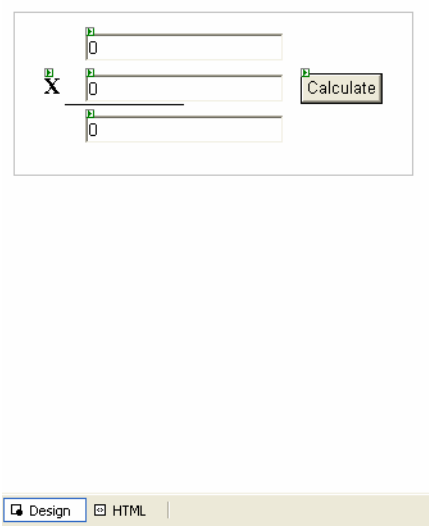

Figure 10.2
A sample Visual Basic .NET upgrade of a Visual Basic 6.0 ActiveX document application

As a final step, you will have to copy the code, upgraded in the user controls, to the respective Web user control. After the converted code is copied, you can add each Web user control to the appropriate Web Form. The project should rebuild, and any issues that had been generated should be fixed, and the document should now be viewable in Internet Explorer.

Upgrading Web Classes

In Visual Basic 6.0, WebClass projects (also known as IIS application projects) are used to create Web applications based on ASP technology. In Visual Basic .NET, ASP.NET Web application projects are used to create Web applications based on the newer ASP.NET technology. When a Visual Basic 6.0 WebClass project is upgraded to Visual Basic .NET, it is converted to an ASP.NET Web application project.

Visual Basic 6.0 WebClass projects have a **StateManagement** property that can be used to keep an instance of the WebClass alive between requests. This is done by setting the **StateManagement** property to **wcRetainInstance** at design time. The **ReleaseInstance** method is used to terminate an instance.

In Visual Basic .NET, ASP.NET Web applications do not have a **StateManagement** property. The model for managing application state differs considerably. Any code related to state management will need to be replaced; there are many options for this. Some options involve keeping information on the client (for example, directly in the page or in a cookie), and others involve storing information on the server between round trips.

If you decide store your instance on the client, you can use some of the following techniques:

- **View state**. The **control.ViewState** property provides a dictionary for retaining values between multiple requests for the same page. This information is automatically stored. When the page is processed, the current state of the page and controls is hashed into a string and saved in the page as a hidden field. When the page is posted back to the server, the page parses the view state string at page initialization and restores property information on the page.
- **Hidden form fields**. You can create hidden fields on a form that are not visibly rendered by the browser. With this technique, you can set properties just as you can with a standard control. When the page is submitted to the server, the content of the hidden field is sent in the HTTP **Form** collection along with the values of the other controls, which allows you to store information directly in the page by way of the hidden fields.
- **Cookies**. You can use cookies to store information about a particular client, session, or application. The cookies are saved on the client device, and when the browser requests a page, it sends the information in the cookie along with the requested information. The server can read the cookie and extract the necessary value.
- **Query strings**. You use query strings to maintain some state information, but they are limited to the capacity of the browser and client devices. This imposes a 255 character limit on the length of the URL. For query string values to be available during page processing, you must submit the page using an HTTP GET method. The consequence of this is that you cannot take advantage of this option if a page is processed in response to an HTTP POST method.

Alternatively, if you decide that the best option is to store the information on the server, you can use any of the following methods:

- **Application state**. With ASP.NET, you can save values using application state (an instance of the **HttpApplicationState** class for each active Web application). Application state is a global storage mechanism accessible from all pages in the Web application and is useful for storing information that needs to be maintained between server round trips and between pages. Application state is a key-value dictionary structure created during each request to a specific URL. You can add your information to this structure to store it between page requests.
- **Session state**. With ASP.NET, you can save values using session state (an instance of the **HttpSessionState** class for each active Web application session).
- **Database support**. You can maintain the state of the page using database technology when you are storing a large amount of information. Database storage is particularly useful for maintaining long-term state or state that must be preserved even if the server must be restarted. The database approach is often used in conjunction with cookies.

When upgrading WebClass projects, **Function** and **Sub** procedures in your Visual Basic 6.0 code (for example, **ProcessTags** or **Respond**) will have their scope changed from **Private** to **Public** to allow the WebClass Compatibility runtime to execute them.

Certain Visual Basic 6.0 WebClass events are not supported in ASP.NET. These include **Initialize**, **BeginRequest**, **EndRequest**, and **Terminate**. These event procedures will be upgraded by the upgrade wizard, but they will not be called at run time. After upgrading, you will need to move any code in these events to equivalent ASP.NET events, such as **Init** or **Unload**.

The upgrade wizard will also add several declarations to your project: one for the WebClass and one for each of the WebItems and templates in the original project. A **Page_Load** event procedure will be added to the project, creating first a **WebClass** object and then **WebItem** objects for each of the WebItems and templates associated with original project. Finally, in the **Page_Load** event procedure, you will see a call to the WebClass Compatibility runtime, **WebClass.ProcessEvents**. This allows the runtime to render the WebItem specified in the request URL. This code is the only new code added to your upgraded project and only serves to emulate the underlying behavior of the Visual Basic 6.0 WebClass runtime.

Summary

Upgrading your Web-based applications will require manual effort. ActiveX documents, which are not supported in Visual Basic .NET, must be rewritten to use Web user controls to recreate similar functionality. However, Web classes can be at least partially upgraded automatically with the help of the Visual Basic Upgrade Wizard and the WebClass compatibility runtime. The techniques presented in this chapter should help you address issues with upgrading these types of Web-based applications to Visual Basic .NET.

11

Upgrading String and File Operations

A typical Microsoft Visual Basic 6.0 application will have to perform some types of string operations. It is also not uncommon for applications to use or process files while performing their tasks.

When upgrading to Visual Basic .NET, there are two approaches to upgrading code that performs string or file operations. The first option is to allow the upgrade wizard to automatically upgrade the code. The resulting code will have the same behavior as your original code with the help of the Visual Basic Compatibility library (which is discussed in Chapter 8, "Upgrading Commonly-Used Visual Basic 6.0 Language Features"). The second option is to replace these operations with new Visual Basic .NET features. This option requires more work, but it has no dependence on the compatibility library.

This chapter describes how to upgrade string and file operations to Visual Basic .NET. Both the compatibility library approach and the replacing with new functionality approach will be demonstrated, and pointers to sources of further information will be provided.

Operations Handled by the Upgrade Wizard

The simplest approach is to allow the upgrade wizard to do the work for you. This section discusses those features that are automatically handled by the upgrade wizard.

Auto-Upgraded String Operations

The most basic string operations involve concatenating strings and obtaining substrings. These are the simplest types of string manipulations, and they are also the most commonly used. The following code example demonstrates how to obtain left, middle, and right substrings, and how to concatenate strings.

```
Private Sub Command1_Click()
    Dim length As Integer
' Divide the string into equal size parts
    Dim chunk As Integer

    length = Len(Label1.Caption)
    chunk = length / 3
    lblLeft.Caption = Left(Label1.Caption, chunk)
    lblMid.Caption = Mid(Label1.Caption, chunk + 1, chunk)
    lblRight.Caption = Right(Label1.Caption, chunk)
    labelRandom.Caption = lblMid.Caption & lblRight.Caption & lblLeft.Caption
End Sub
```

The upgrade wizard can be applied to the preceding code, and the result can be built and executed with no issues. The result produced by the upgrade wizard is shown here.

```
Option Strict Off
Option Explicit On
Imports VB = Microsoft.VisualBasic
…
    Private Sub Command1_Click(ByVal eventSender As System.Object, _
        ByVal eventArgs As System.EventArgs) Handles Command1.Click

        Dim length As Short
        Dim chunk As Short

        length = Len(Label1.Text)
        chunk = length / 3
        lblLeft.Text = VB.Left(Label1.Text, chunk)
        lblMid.Text = Mid(Label1.Text, chunk + 1, chunk)
        lblRight.Text = VB.Right(Label1.Text, chunk)
        labelRandom.Caption = lblMid.Text & lblRight.Text & lblLeft.Text
    End Sub
…
```

Note the inclusion of the **Microsoft.VisualBasic** namespace with the **Import** statement. This namespace contains the core Visual Basic .NET functionality, including the string functions used in this example. The Visual Basic 6.0 string functions are upgraded by the upgrade wizard to the equivalent functions in this namespace. When executed, the preceding code has the exact same behavior as the original code.

A complete listing of the string operations that are included in the **Microsoft.VisualBasic** namespace can be found by viewing Help for this namespace.

Auto-Upgraded File Operations

File input and output is supported through the **Microsoft.VisualBasic** library. This section focuses on how the upgrade wizard handles file input and output in your Visual Basic 6.0 applications.

Accessing Text Files

Visual Basic 6.0 provides several I/O commands to manage the access to files and directories. These commands include: **Open**, **Close**, **Reset**, **Get**, **Put**, **Print**, **Write**, **Input**, **LineInput**, **Lock**, **Unlock**, and **Width**.

The following Visual Basic 6.0 code example demonstrates the use of some of these file operations to open a text file, read its contents one character at a time, and send the characters read to the **Immediate** window.

```
Dim FileName as String
Dim MyChar as String

FileName = "C:\Temp\TextBox.txt"
' Open file.
Open FileName For Input As #1
' Loop until end of file.
Do While Not EOF(1)
' Get one character.
   MyChar = Input(1, #1)
' Print to the Immediate window.
   Debug.Print MyChar
Loop
Close #1
```

Please note that the **FileName** variable should be set to a valid name before attempting to execute this example.

This code can be upgraded automatically by the upgrade wizard. The result produced is shown in the following code example.

```
Dim FileName As String
Dim MyChar As String

FileName = "C:\Temp\TextBox.txt"
' Open file.
FileOpen(1, filename, OpenMode.Input)
' Loop until end of file.
Do While Not EOF(1)
' Get one character.
```

```
    MyChar = InputString(1, 1)
' Print to the Immediate window.
    System.Diagnostics.Debug.WriteLine(MyChar)
Loop
FileClose(1)
```

This code compiles and executes with the same behavior as the original code, thanks to the support provided by the compatibility library. As previously stated, whenever the resources allow it, it is possible to rewrite the output using only core Microsoft .NET Framework functionality. The preferred method for working with files in Visual Basic .NET is to use streams. A stream is a more generic view of a sequence of bytes that allows reading, writing, and seeking. Streams can be associated with a file, a network connection, and memory, to name just a few examples. Visual Basic .NET even provides for reading and writing cryptographic streams.

The Visual Basic .NET code generated by the upgrade wizard can be rewritten to use streams instead of text file functions. For more information, see the "Improving File I/O with Streams" section later in this chapter.

Accessing Fixed-Length Records Using User-Defined Types

It is not unusual to find Visual Basic 6.0 applications that communicate with legacy systems through flat files. Visual Basic 6.0 provided useful tools for that type of interaction, including random, binary, and sequential file access mode. This section presents the steps necessary to upgrade random file access code to Visual Basic .NET.

The following Visual Basic 6.0 sample code includes one method for storing a record and another method to read it from a flat file.

```
Type employee
    name As String * 15
    last_name As String * 20
    department As String * 6
    phone_ext(3) As Long
    salary As Integer
End Type

Private Sub Form_Load()
    WriteRecord 3
    ReadRecord 3
End Sub
Sub ReadRecord(recordnum As Integer)
    Dim fileh As Integer
    Dim emp As employee
    fileh = FreeFile
    Open "C:\recordtest.dat" For Random As fileh Len = 73
    ' The record will be read from the position indicated by recordnum:
    Get fileh, recordnum, emp
```

```
    MsgBox ("Employee: " & emp.name & " " & emp.last_name & _
    ", Dept: " & emp.department)
    Close fileh
End Sub
Sub WriteRecord(recordnum As Integer)
    Dim fileh As Integer
    Dim emp As employee
    fileh = FreeFile
    Open "C:\recordtest.dat" For Random As fileh Len = 73
    emp.name = "John"
    emp.last_name = "Doe"
    emp.department = "First aids"
    emp.phone_ext(0) = 123
    emp.salary = 1000

    ' The record will be saved in the position indicated by recordnum:
    Put fileh, recordnum, emp
    Close fileh
End Sub
```

The employee user-defined type (UDT) contains only fixed length fields that allow the UDT to be accessed in a random way when it is stored inside a file. The following code sample presents the output that was obtained with the upgrade wizard and then was adjusted by the user according to the upgrade warnings.

```
Structure employee
    <VBFixedString(15),System.Runtime.InteropServices.MarshalAs( _
    System.Runtime.InteropServices.UnmanagedType.ByValTStr, _
    SizeConst:=15)> Public name As String

    <VBFixedString(20),System.Runtime.InteropServices.MarshalAs( _
    System.Runtime.InteropServices.UnmanagedType.ByValTStr, _
    SizeConst:=20)> Public last_name As String

    <VBFixedString(6),System.Runtime.InteropServices.MarshalAs( _
    System.Runtime.InteropServices.UnmanagedType.ByValTStr, _
    SizeConst:=6)> Public department As String

    <VBFixedArray(3)> Dim phone_ext() As Integer
    Dim salary As Short

    ' UPGRADE_TODO: "Initialize" must be called to initialize instances of
    ' this structure. Click for more:
    ' 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="vbup1026"'
    Public Sub Initialize()
        ReDim phone_ext(3)
    End Sub
End Structure

Private Sub Form1_Load(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    WriteRecord(3)
```

```
    ReadRecord(3)
End Sub
Sub ReadRecord(ByRef recordnum As Short)
    Dim fileh As Short
    Dim emp As employee
    emp.Initialize()
    fileh = FreeFile
    FileOpen(fileh, "C:\recordtest.dat", OpenMode.Random, , , 73)
    ' The record will be read from the position indicated by recordnum:
    ' UPGRADE_WARNING: Get was upgraded to FileGet and has a new behavior. Click
for
    ' more: 'ms-help://MS.VSCC.2003/commoner/redir/
redirect.htm?keyword="vbup1041"'
    FileGet(fileh, emp, recordnum)
    MsgBox("Employee: " & emp.name & " " & emp.last_name & ", Dept: " & _
    emp.department)
    FileClose(fileh)
End Sub
Sub WriteRecord(ByRef recordnum As Short)
    Dim fileh As Short
    Dim emp As employee
    emp.Initialize()
    fileh = FreeFile
    FileOpen(fileh, "C:\recordtest.dat", OpenMode.Random, , , 73)
    emp.name = "John"
    emp.last_name = "Doe"
    emp.department = "First aids"
    emp.phone_ext(0) = 123
    emp.salary = 1000
    ' The record will be saved in the position indicated by recordnum:
    ' UPGRADE_WARNING: Put was upgraded to FilePut and has a new behavior. Click
for
    ' more: 'ms-help://MS.VSCC.2003/commoner/redir/
redirect.htm?keyword="vbup1041"'
    FilePut(fileh, emp, recordnum)
    FileClose(fileh)
End Sub
```

To initialize the employee structure and set the size of the **phone_ext** field, the **Initialize** method must be called as shown with the statements in bold. Notice that an upgrade warning about behavior differences was included before the **FilePut** and **FileGet** method calls. The warning refers to the situation when these functions receive dynamic arrays or strings as arguments; in this case, a two-byte length descriptor is added and the obtained file will have a different size. However, because the example is using fixed-length fields, this difference is not applicable, so the warning can be removed.

Manual String and File Operation Changes

As with all programming, there are multiple ways to accomplish various string and file manipulation tasks. The previous section demonstrated the types of changes the upgrade wizard applies to automatically upgrade string and file code from Visual Basic 6.0 to Visual Basic .NET. However, the new version of the language includes many other powerful string and file methods and classes to improve the performance, maintainability, and simplicity of file operations. This section will introduce just a few of the possibilities.

Replacing Strings with StringBuilders

In Visual Basic 6.0 and Microsoft Visual Studio .NET, strings are immutable. This means that whenever a string is declared and a value is assigned to it, the value that the string holds in memory never changes. The following code example demonstrates the impact of this characteristic.

```
Dim sMyString As String
sMyString = "Hello"
sMyString = sMyString + " World"
```

In the preceding code, although it seems as if a single string is being modified, in fact three strings are actually created. The first string creation occurs in the first line where the string is declared. The second string is created when the value "Hello" is assigned to the **sMyString** variable, and a third when the string is concatenated with the " World" value.

This characteristic of strings has some drawbacks. For starters, you are using more memory than what you actually require to perform a simple task. There are two remaining objects that require cleanup by the garbage collector. There is also a performance overhead because of the extra CPU cycles that are needed to create the object, create the copies of the object, and clean the old strings from memory.

String variables can be used when the stored values will not be changed frequently during the program execution. In cases like the previous example, where a fixed quantity of strings is concatenated, the usage of a **String** data type is appropriate.

When the quantity of operations that a string will participate in is not known beforehand, and a high quantity of modifications is expected, the **StringBuilder** class is the best choice because no new objects are created when the string value is altered. In operations such as appending text to a string, the **StringBuilder** will most likely outperform **String** concatenation because there is less overhead in terms of creating new objects in memory for each operation.

In Visual Studio .NET, the **StringBuilder** class can be found in the **System.Text** namespace. The following code example demonstrates how to modify the previous code to use the **StringBuilder** class.

```
Dim sMyString As New StringBuilder("Hello ")
sMyString.Append(" World")
```

In contrast to the original code example, this code does not create a new object every time the **StringBuilder** object is modified. The same data in memory is modified for any content-changing operation performed on the **StringBuilder**. If you ever needed to fetch the **String** contents from the **StringBuilder**, you would only need to use the **toString()** method to retrieve this information.

Significant performance improvements will be obtained when **StringBuilder** is used for series of string operations that modify one string variable, as in the following string replacement function.

```
Public Shared Function ReplaceSymbols(str As String) As String
    Dim tmpStrB As New StringBuilder(str)
    tmpStrB.Replace("#"c, "!"c, 15, 29)
    tmpStrB.Replace("!"c, "o"c)
    tmpStrB.Replace("cat", "dog")
    tmpStrB.Replace("dog", "fox", 15, 20)
    Return tmpStrB.ToString()
End Function
```

Does this mean that you should change every instance of **String** in your old Visual Basic 6.0 code to use **StringBuilders** in Visual Basic .NET? No; if you are not modifying your strings or if you are concatenating small numbers, you might leave your strings as they are. If, on the other hand, you are performing many manipulations (concatenating, changing case, replacing/inserting characters, and so on), you might be better off changing your **String** instances to **StringBuilders**. By following this approach, you would be assured that your application will not suffer performance degradation due to **String** manipulation.

Replacing Complex String Manipulation with Regular Expressions

Visual Studio .NET programming languages, including Visual Basic .NET, are able to fully exploit the power of regular expressions. To take advantage of the classes that provide access to the .NET Framework regular expression engine, you need to import the **System.Text.RegularExpressions** namespace into your code.

Regular expressions allow a programmer to define patterns to be found in a general manner, which is great for several reasons. For starters, the amount of coding required to perform a specific task is greatly reduced because regular expressions allow a programmer to write in a few lines of code in what usually took many lines before. For example, the following code example shows a function written in Visual

Basic .NET that finds the number of occurrences of a character pattern within a string.

```
Function findOccurrences(ByVal sHaystack As String, ByVal sNeedle As String) _
         As Integer
    Dim iPosition As Integer
    Dim iCount As Integer
    For i As Integer = 1 To sHaystack.Length
        iPosition = InStr(i, sHaystack, sNeedle)
        If (iPosition > 0) Then
            iCount = iCount + 1
            i = iPosition
        End If
    Next i
    Return iCount
End Function
```

By utilizing regular expressions, the function can be rewritten more succinctly in the following manner.

```
Function findOccurrences2(ByVal sHaystack As String, ByVal sNeedle As String) _
         As Integer
    Return Regex.Matches(sHaystack, sNeedle).Count
End Function
```

By comparing the two functions, you can clearly see that the amount of code required for the second function is minimal compared to the original function. There is no need to have temporary variables to store the current result and there is no need to code the looping section of the procedure; thus, the code has not only been reduced, but it is also more efficient code and easier to understand. The preceding code example is a very simple one; a more complex problem could easily convert hundreds of lines of code into a simplified expression.

If you wanted to modify the function to carry out the same search but with different options, you can easily do so by changing the regular expression options. For instance, if you wanted to have the function perform the search without being case sensitive, you would only need to pass **RegexOptions.IgnoreCase** as an option parameter to the **Matches** function — a procedure that would otherwise require many changes to the code found in the original **findOccurences** method. There are many options in the **RegularExpressions** namespace that can aid you in carrying out a complex procedure with strings. For information about the different enumerated values available in this particular namespace, see "System.Text.RegularExpressions Namespace" in the *.NET Framework Class Library* on MSDN.

You should seriously consider using regular expressions if your upgraded code handles any type of complex input validation or complex string modification. For

example, visualize a scenario in which you are upgrading a Visual Basic 6.0 password verification routine that has intricate requirements: the password must be at least eight characters long, must contain at least one digit, and should contain at least one special character. The non-regular expression code for that function may span many lines of code to parse the password chosen by a user. The available Visual Basic 6.0 procedures used to code this type of algorithm might leave no other choice but to write very hard to read code that might be unnecessarily complicated. By using the classes under the **RegularExpression** namespace, you could carry out the verification with one line of code if your regular expression has been well planned.

The **Regex.Replace** function can also help you perform complex string manipulations at a fraction of the code that would otherwise take. The following code example shows the order of the last name and first name passed to a function is reversed.

```
' Assumes input is in the (last_name,first_name) format
Function switchPlaces(ByVal sInput As String) As String
   return Regex.Replace(sInput, "(?<last>.+\D),(?<first>.+\D)", _
        "${first} ${last}")
End Function
```

The **Replace** function receives one input string and a regular expression that specifies the pattern to be detected in the input string. The last argument indicates how the elements of the instance that was found must be replaced. Character escapes and substitutions are the only special constructs recognized in a replacement pattern. For example, the replacement pattern a*${test}b inserts the string "a*" followed by the substring matched by the "test" capturing group, if any, followed by the string "b". Other examples are: $123 substitutes the last substring matched by group number 123 (decimal), and ${name) substitutes the last substring matched by a (?<name>) group. For more information about regular expressions, see "Regular Expression Language Elements" in the *.NET Framework General Reference* on MSDN.

This code is another example of how easy string manipulation can be if regular expressions are used. The parameter is parsed using a regular expression and backreferences are used to store the substrings retrieved. Matching strings are replaced based on the format specified in the method call.

The .NET implementation of regular expressions allows regular expressions to be written in a more efficient and maintainable manner than using string functions to perform intricate string manipulation. Becoming comfortable with regular expressions may take some time, but the ultimate reward will come as an increased ability to carry out complex string manipulations proficiently and effortlessly.

Improving File I/O with Streams

The upgrade wizard is able to upgrade most of the file access methods using the Visual Basic 6.0 Compatibility library available in .NET. However, the result produced by the upgrade wizard uses methods in the Visual Basic Compatibility library. An alternative to upgrading file access is to use new Visual Basic .NET methods for file access. The preferred method in .NET is to use streams.

For example, consider the following code.

```
Dim FileName As String
Dim MyChar As String

FileName = "C:\Temp\TextBox.txt"
FileOpen(1, filename, OpenMode.Input)          ' Open file.
Do While Not EOF(1)                            ' Loop until end of file.
    MyChar = InputString(1, 1)                 ' Get one character.
    System.Diagnostics.Debug.WriteLine(MyChar) ' Print to the Immediate window.
Loop
FileClose(1)
```

As shown earlier in this chapter, this code can be automatically upgraded by the upgrade wizard. However, to achieve the same effect using streams, you will have to manually rewrite the operations.

Streams are the recommended choice for reading and writing data from files in .NET for most new development projects. A stream is a generic view of a sequence of bytes that allows reading, writing, and seeking. Streams can be associated with a file or other sources (such as network connections) and have highly optimized methods that make them faster than the compatibility library methods for file access.

The original Visual Basic 6.0 file code sample can be rewritten to use streams instead of text file functions in Visual Basic .NET. A revised version of this code is shown here.

```
Dim fs As FileStream
Dim MyChar(1) As Char
Dim reader As System.IO.StreamReader

fs = New FileStream("C:\Temp\TextBox.txt", FileMode.Create)
fs.Seek(0, SeekOrigin.Begin)
reader = StreamReader(fs, System.Text.Encoding.ASCII)
Do While reader.Peek <> - 1
   reader.Read(MyChar, 0, 1)
   System.Diagnostics.Debug.WriteLine(MyChar(0))
Loop
reader.Close()
```

As demonstrated in this code sample, reading from a stream is fairly straightforward. Instead of opening a file associated with a file handle, you create a **StreamReader** object to open the file. After the file is opened, the **StreamReader** method **Read** can be used to obtain the next character. The **Peek** property allows you to look ahead one character to determine whether the end of the file has been reached (indicated if **Peek** returns -1). After the file contents are read, the **Close** method closes the stream.

For Visual Basic 2005:

The **System.IO.File.ReadAll** method provides functionality to open a file, read the contents into a string variable, and close the file using a single method.

```
Dim path As String
Dim fileContents As String
fileContents = File.ReadAll(path)
```

The analogous method for writing files is **System.IO.File.WriteAll.**

The **StreamReader** class's **Read**, **ReadBlock**, **ReadLine**, and **ReadToEnd** methods are used to read the contents of a file. The **StreamWriter** class provides methods for writing to streams. The process is similar to reading. First, create a **StreamWriter** object, use the appropriate methods for writing to the stream, and close the stream when finished. You can use the **StreamWriter** class's **Write** and **WriteLine** methods to write to the stream.

Traditional Visual Basic 6.0 sequential I/O has some characteristics in common with streaming. Streaming also has similarities to binary I/O. Sequential access with streaming functionality can be achieved using the **FileStream** and **StreamWriter** classes, as in the previous example. Notice how the file access is done with ASCII encoding; this is necessary because .NET strings use Unicode encoding and to access a file that was written with Visual Basic 6.0 in mind, ASCII encoding must be used.

Streams maintain a current position that is a pointer to the location where the next operation will occur. Both **StreamWriter** and **StreamReader** objects update the current position after every operation that is executed.

Streams have an important limitation with respect to Visual Basic 6.0 sequential access. There is no built-in capability to read delimited data. When processing delimited data with streams, it is the programmer's responsibility to parse the data.

Sequential writing of a file can be achieved with the **StreamWriter.WriteLine** method, which can write a string followed by a line terminator. Binary writing of a file, which allows any amount of data to be placed in the file during a writing operation, can be achieved with **StreamWriter.Write**; this method can write different data types at the current position.

Stream classes are available in the **System.IO** namespace. Additionally, this namespace includes classes for other file operations, such as directory handling. The following code example illustrates how you can use the **GetDirectories** method of the **Directory** class to get a listing of subdirectories that are contained in the C:\Temp directory.

```
Dim directories() As String
Dim i As Integer
directories = System.IO.Directory.GetDirectories("C:\temp")
For i = 0 To directories.Length - 1
    System.Diagnostics.Debug.WriteLine(directories(i))
Next
```

For more information about the **System.IO** namespace, see the Visual Basic .NET documentation.

File Access Through the File System Object Model

Another option available in Visual Basic .NET for accessing and manipulating files is to use the File System Object (FSO) model. This model provides objects and methods for working with files and folders (directories) in an object-oriented way. This model is provided through the Visual Basic Scripting type library (Scrrun.dll).

Using the FSO model, you can create or delete files and folders, obtain information about files and folders (such as path information), and perform other manipulations such as copying and moving files and folders. The model also provides a class, **TextStream**, which allows you to create objects for reading and writing text files. Note that binary file reading and writing is not supported by the FSO model.

The following code example demonstrates a very simple use of the FSO model of file and directory access. It creates a **FileSystemObject** object to facilitate the use of the FSO model. It also creates a **TextStream** object to open a text file whose contents are displayed to the console.

```
Private Sub Command1_Click(ByVal eventSender As System.Object, _
    ByVal eventArgs As System.EventArgs) Handles Command1.Click

    Dim fs As Scripting.FileSystemObject
    Dim inFile As Scripting.TextStream
    Dim filename As String

    ' Obtain the file name.
    …
    ' Create a FileSystemObject for file manipulation.
    fs = CreateObject("Scripting.FileSystemObject")
    ' Open a text file for reading.
    inFile = fs.OpenTextFile(filename, Scripting.IOMode.ForReading)
    ' Display the contents of the text file to the console.
    While Not inFile.AtEndOfStream
```

```
            System.Console.WriteLine(inFile.ReadLine)
        End While
        inFile.Close()

        ' Copy the file to a temp directory.
        fs.CopyFile(filename, "C:\Temp\", True)
    End Sub
```

Using methods and properties of the **FileSystemObject**, you can also perform activities such as creating temporary file and folder names, check for the existence of a particular file or folder, delete a file or folder, and more. The FSO model of file access available through the scripting library is too extensive to cover here. For more information about the **FileSystemObject** model of file access, or to get tips about how to choose between the different file access models available in Visual Basic .NET, see "Accessing Files with FileSystemObject" and "Choosing Among File I/O Options in Visual Basic .NET" in Visual Basic .NET Help. These documents are also available on MSDN.

Summary

Changes in Visual Basic .NET have made several features of earlier versions of the language obsolete. The compatibility library is provided to help minimize the effort in upgrading some of these features. For those features not available in the compatibility library, Visual Basic .NET almost always offers the same functionality through new objects and functions. This chapter has described the most common features of Visual Basic 6.0 that are now obsolete, and how to replace these features in Visual Basic .NET.

More Information

For more information about the different enumerated values available in the **RegularExpressions** namespace, see "System.Text.RegularExpressions Namespace" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemtextregularexpressions.asp.

For more information about regular expressions, see "Regular Expression Language Elements" in the *.NET Framework General Reference* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconregularexpressionslanguageelements.asp.

For more information about the **FileSystemObject** model of file access, or to get tips about how to choose between the different file access models available in Visual Basic .NET, see "Accessing Files with FileSystemObject" and "Choosing Among File I/O Options in Visual Basic .NET" in Visual Basic .NET Help or on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vbconintroductiontofilesystemobjectmodel.asp
and:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vatskchoosingamongfileiooptionsinvisualbasicnet.asp.

12

Upgrading Data Access

Almost every business application has some sort of data access. This is not surprising when you consider that conceptually most businesses are based around data. Customers and orders, inventory and purchase information; all of this business information needs to be stored in a way that it can easily be retrieved and updated. Countless Visual Basic applications have been developed to provide a friendly face for information stored in databases.

In Visual Basic, three data access interfaces are available to you: ActiveX Data Objects (ADO), Remote Data Objects (RDO), and Data Access Objects (DAO). A data access interface is an object model that represents various facets of accessing data. Using Visual Basic, you can programmatically control the connection, statement builders, and returned data for use in any application.

Each new version of Visual Basic has delivered enhancements to data access; Visual Basic .NET is no exception. This newest version continues support for ActiveX Data Objects (ADO) and data binding. It also supports DAO and RDO, although it does not support data binding for these technologies. In terms of enhancement, Visual Basic .NET introduces a new type of data model, ADO.NET.

This chapter looks at upgrading the three major data access technologies (ADO, DAO, and RDO) and how you can convert your Data Reports in Visual Basic 6.0 to Crystal Reports in Visual Basic .NET. Because this part of the guide assumes you are achieving functional equivalence for your application, upgrading to ADO.NET is not covered in this chapter. However, for more information about ADO.NET see section "ADO to ADO.NET" in Chapter 20, "Common Technology Scenario Advancements."

General Considerations

There are many Visual Basic applications that store their data in a database system such as SQL Server or Microsoft Access. Each one uses different approaches to access the data, such as ADO with data binding or RDO without data binding. Before upgrading your data access code, you should analyze it to identify what type of data access it is using. The process to upgrade data access code depends on the technology used and whether data binding is used. This chapter examines the methods for upgrading data access.

ActiveX Data Objects (ADO)

Microsoft ActiveX Data Objects (ADO) was first introduced as the data access interface in Microsoft Internet Information Server (IIS). ADO is a simple, easy-to-use interface to access data. As the need for an interface that spans many tools and languages grows, ADO continues to be enhanced to combine the best features of, and eventually replace the most widely used data access interfaces today; RDO and DAO. ADO is similar to RDO and DAO in many ways. For example, it uses similar language conventions. ADO provides simpler semantics that make it easy to learn for today's developers.

ADO is designed to be the application-level interface to OLE DB, Microsoft's newest and most powerful data access paradigm. OLE DB provides high-performance access to any data source. Together ADO and OLE DB form the foundation of the Universal Data Access strategy. OLE DB enables universal access to any data. ADO makes it easy for developers to program data access. Because ADO is built on top of OLE DB, it benefits from the rich universal data access infrastructure that OLE DB provides.

The ADO object model defines a collection of programmable objects that you can use in any platform that supports both COM and Automation, like Visual Basic. The ADO object model is designed to expose the most commonly used features of OLE DB.

With ADO you can manipulate data objects, data binding, and design-time tools such as the ADO data environment. The users of your applications may regard the run-time behavior as another component of data access, but in this instance you can consider the run-time behavior as an element of each of the three components. The following sections examine each of these three components and demonstrate where they differ between Visual Basic 6.0 and Visual Basic .NET.

Upgrading ADO Data Binding

Data binding is one of the most powerful uses of data access in Visual Basic 6.0. Essentially, using data binding allows a property of a control to bind to a specific field in a database. When the bound control property (such as the text box's **Text** property) is modified by the user of the control, the control notifies the database that the value has changed. It then requests that the relevant field of the current record be updated. This process takes place, and in return the database notifies the control of the success or failure of the update. The main properties for data binding in a control are: **DataChanged**, **DataField**, **DataFormat**, **DataMember**, and **DataSource**.

This upgrade scenario includes any application in which a reference to Microsoft ActiveX Data Objects has been added to the project, and at least one form contains an **ADO Data Control,** and one or more controls use data binding, using some of the following properties: **DataSource**, **DataMember**, and **DataField**.

Normally, most ADO code is upgraded automatically by the upgrade wizard. With minimal manual changes, your application and Data Access technology will work the same way that it works in Visual Basic 6.0. For example, it may be necessary to add a reference to the **ADODB.NET** assembly (Adodb.dll) instead of the **Interop.ADODB.dll** that is automatically referenced by the upgrade wizard.

Visual Basic .NET only supports ADO data binding. DAO or RDO data binding are not supported. ADO data binding is supported because it is built into the COM library that manages ADO data binding in Visual Basic 6.0. Its equivalent is available in the compatibility library in Visual Basic .NET, **Microsoft.VisualBasic.Compatibility.Data**. This library manages data binding in Visual Basic .NET. As a result, the properties, methods, and events are compatible between the two versions of the language, which allows ADO Control and Data Environment to be automatically upgraded.

The upgrade wizard adds some support functions to maintain the same functionality of Visual Basic 6.0. This is because of the need in Visual Basic .NET for all objects referenced in the code to be instantiated before they are used. The code added by the upgrade wizard ensures this happens. Also, when your code has data binding created at design time, the upgrade wizard adds the binding variables and procedures that are used to do the data binding with ADO.

Consider a form example that contains a **TextBox** with an ADO Data Control set as the **DataSource** property of the **TextBox** control, and where the **DataField** property of the **TextControl** is set to **Title**. When you use the upgrade wizard to upgrade such a form, variables and procedures are added to the upgraded code to enable the data

binding to continue to work in Visual Basic .NET, as shown in the following code example.

```
Private ADOBind_Adodc1 As VB6.MBindingCollection

Public Sub VB6_AddADODataBinding()
    ADOBind_Adodc1 = New VB6.MBindingCollection()
    ADOBind_Adodc1.DataSource = CType(Adodc1, msdatasrc.DataSource)
    ADOBind_Adodc1.Add(Text1, "Text", "Title", Nothing, "Text1")
    ADOBind_Adodc1.UpdateMode = VB6.UpdateMode.vbUpdateWhenPropertyChanges
    ADOBind_Adodc1.UpdateControls()
End Sub
Public Sub VB6_RemoveADODataBinding()
    ADOBind_Adodc1.Clear()
    ADOBind_Adodc1.Dispose()
    ADOBind_Adodc1 = Nothing
End Sub
```

However, there are problems with the upgrade of data binding in intrinsic controls set at run time. These kinds of controls are upgraded to native Visual Basic .NET, and the **DataSource**, **DataMember**, and **DataField** are not upgraded by the upgrade wizard. Data binding at design time is also not supported, and the upgrade wizard does not add any binding variable or procedures, requiring you to make changes to the upgraded code.

To illustrate this situation, assume that you have a project with a form that contains a Label control named **labAuthor** and an ADO Data Control named **Adodc1**. **Adodc1** is connected to the Microsoft Access database **Northwind**. At run time, you set the ADO Data Control to the **DataSource** property and set **EmployeeID** as the **DataField**, as shown in the following code example.

```
Private Sub Form_Load()
    Set Me.labAuthor.DataSource = Adodc1
    Me.labAuthor.DataField = "EmployeeID"
End Sub
```

When this code is upgraded, the resultant code has issues that you must fix. The upgraded code is shown here.

```
Private Sub frmADO_Load(ByVal eventSender As System.Object, _
            ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    ' UPGRADE_ISSUE: Label property labAuthor.DataSource was not upgraded.
    Me.labAuthor.DataSource = Adodc1
    ' UPGRADE_ISSUE: Label property labAuthor.DataField was not upgraded.
    Me.labAuthor.DataField = "EmployeeID"
End Sub
```

The upgrade wizard does not automatically create the variables and procedures needed for the data binding do function. Instead, you must add this code manually.

Later, you will have to add the **Text** property of the label control to binding, using the ADO Data Control, and you will have to eliminate the instructions that have issues. The modified code for the current example is shown here.

```
Private ADOBind_Adodc1 As VB6.MBindingCollection

Public Sub VB6_AddADODataBinding()
    ADOBind_Adodc1 = New VB6.MBindingCollection
    ADOBind_Adodc1.DataSource = CType(Adodc1, msdatasrc.DataSource)
    ADOBind_Adodc1.Add(labAuthor, "Text", "EmployeeID", Nothing, "labAuthor")
    ADOBind_Adodc1.UpdateMode = VB6.UpdateMode.vbUpdateWhenPropertyChanges
    ADOBind_Adodc1.UpdateControls()
End Sub

Public Sub VB6_RemoveADODataBinding()
    ADOBind_Adodc1.Clear()
    ADOBind_Adodc1.Dispose()
    ADOBind_Adodc1 = Nothing
End Sub
```

After these subroutines and variables are added, you have to invoke the appropriate subroutines in the same place where the **DataSource** and **DataField** properties were assigned in the original code. In the current example, these were assigned in the **Load Event** code. The code following example shows the appropriate adjustments.

```
Private Sub frmADO_Load(ByVal eventSender As System.Object, _
                        ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    VB6_AddADODataBinding()
End Sub
```

After these changes are made, the upgraded code has the same behavior in Visual Basic .NET as it did in Visual Basic 6.0.

Projects without ADO Data Binding

The previous section explains how to upgrade code with data binding and the changes that are necessary to achieve the same behavior in Visual Basic .NET. However, not all database access requires data binding.

An application falls into this upgrade scenario when its project references Microsoft ActiveX Data Objects but does not contain an ADO Data Control. In this case, all data access is managed directly at run time.

The automatic upgrade of a project that accesses a database without the use of data binding does not require changes in the upgraded code. The upgrade wizard automatically adds the reference to **Microsoft.VisualBasic.Compatibility** library, which contains ADO. All ADO code works without any modifications and has the same behavior as in the original Visual Basic 6.0 project.

However, there is a difference between the upgraded code and the original source code with respect to how a field in a recordset is accessed. In Visual Basic 6.0, it is common to access fields using the shorthand coding convention *RecordsetName*!*FieldName*. In Visual Basic .NET the code needs to be expanded to resolve the default properties. The resulting code looks a little different after it is upgraded. This is shown in the following code example.

```
Dim cn As New Connection
Dim rs As Recordset
cn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Temp\Northwind.mdb"
Set rs = cn.Execute("Select * From Products where ProductID = 1")
txtProduct.Text = rs!ProductName
txtCompany.Text = rs!SupplierID
rs.Close
cn.Close
```

This Visual Basic 6.0 code retrieves the product name and supplier ID for a specified product. This information is assigned to two **TextBox** controls. When it is upgraded, the new code looks like the following example.

```
Dim cn As New ADODB.Connection
Dim rs As ADODB.Recordset
cn.Open("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Temp\Northwind.mdb")
rs = cn.Execute("Select * From Products where ProductID = 1")
txtProduct.Text = rs.Fields("ProductName").Value
txtCompany.Text = rs.Fields("SupplierID").Value
rs.Close()
cn.Close()
```

Notice that the fields of the recordset are expanded when you use the upgrade wizard.

Upgrading Data Environment

Data Environment is a Visual Basic 6.0 feature that provides an interactive, design time environment for creating programmatic, run-time data access. Visual Basic .NET provides an even more powerful environment for building database applications, but because the underlying data access library is conceptually different (Visual Basic .NET uses ADO.NET while Visual Basic 6.0 uses ADO), there is no direct mapping between Data Environment in Visual Basic 6.0 and data access features in Visual Basic .NET.

With Data Environment, Visual Basic 6.0 developers can accomplish the following tasks:

- Create ADO Connection objects.
- Create ADO Command objects that are based on stored procedures, tables, views, synonyms and SQL statements.

- Create hierarchies of commands that are based on a grouping of Command objects or by relating one or more Command objects together.
- Write and run code that reads and sets properties of objects hosted in Data Environment.
- Execute commands included in Data Environment as programmatic, run-time methods.
- Bind Form controls to commands hosted in Data Environment.
- Create aggregates that automatically calculate values within any command hierarchy.

When you use the upgrade wizard to upgrade an application that uses Data Environment, the following changes are made:

- For each Data Environment in your Visual Basic 6.0 project, a class with the name **DataEnvironment** prefixed to the Data Environment's name is created and a variable named with Data Environment's name is instanced in a module.
- For each connection and recordset hosted by a Data Environment object, a **Public WithEvents** member variable with the same name is created.
- For each command, a public method with the same name is created. It will be available in the same way that its respective command is available in Visual Basic 6.0.

For example, consider an application that has a Data Environment named DE with one connection named **Access** and a **Command** named **Orders**. When it is upgraded using the upgrade wizard, you obtain a class that looks like the following code example.

```
Module DataEnvironment_test_Module
    Friend DE As DataEnvironment_DE = New DataEnvironment_DE()
End Module

Friend Class DataEnvironment_DE
    Inherits VB6.BaseDataEnvironment
    Public WithEvents Access As ADODB.Connection
    Public WithEvents rsOrders As ADODB.Recordset
    Private m_Orders As ADODB.Command

    Public Sub New()
        MyBase.New()
        Dim par As ADODB.Parameter
        Dim connectStr as String

        Access = New ADODB.Connection()
        connectString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
            "Data Source=C:\Temp\Northwind.mdb;Persist Security Info=False;"
        Access.ConnectionString = connectString
        m_Connections.Add(Access, " Access")
        m_Orders = New ADODB.Command()
```

```
        rsOrders = New ADODB.Recordset()
        m_Orders.Name = "Orders"
        m_Orders.CommandText = "SELECT OrderID, OrderDate FROM Orders"
        m_Orders.CommandType = ADODB.CommandTypeEnum.adCmdText
        rsOrders.CursorLocation = ADODB.CursorLocationEnum.adUseClient
        rsOrders.CursorType = ADODB.CursorTypeEnum.adOpenStatic
        rsOrders.LockType = ADODB.LockTypeEnum.adLockReadOnly
        rsOrders.Source = m_Orders
        m_Commands.Add(m_Orders, "Orders")
        m_Recordsets.Add(rsOrders, "Orders")
    End Sub

    Public Sub Orders()
        If Access.State = ADODB.ObjectStateEnum.adStateClosed Then
            Access.Open()
        End If
        If rsOrders.State = ADODB.ObjectStateEnum.adStateOpen Then
            rsOrders.Close()
        End If
        m_Orders.ActiveConnection = Access
        rsOrders.Open()
    End Sub
End Class
```

All uses of Data Environment in the code are maintained with the same format and the same behavior; however, there is a big difference between a Data Environment object and this new class: you can not visually manipulate the new class as you can in Visual Basic 6.0. So, what happens if your code uses Data Environment objects with data binding? The data binding works fine, but you cannot add new commands visually. The upgrade process is discussed in the following section.

Upgrading Data Environment with Data Binding

Visual Basic 6.0 Data Environment exposes a data source interface to enable the binding of data-aware controls directly to a Data Environment object. This can be accomplished by setting the **DataSource** property of the control to the Data Environment object name, the **DataMember** property to the name of the **Command** object (hosted by a Data Environment object) that acts as a source of the recordset that contains the desired data, and the **DataField** property to the name of the column that the data should be fetched from. The Data Environment runtime detects that it is used as a data source and initiates the following actions prior to displaying the Visual Basic 6.0 form with bound controls:

- All database connections that are used by the commands that the user interface controls are bound to are opened.
- All necessary commands are executed to build the recordsets.
- All user interface controls are bound to the recordsets that are obtained in the previous task, according to the **DataMember** and **DataField** properties.

Because the WinForms library that is used by Visual Basic .NET is optimized for the ADO.NET data access library, WinForms controls cannot be directly bound to ADO Recordsets. To enable binding between WinForms and ADO, the upgrade wizard uses the **MBindingCollection** and **MBinding** objects of the **Microsoft.VisualBasic.Compatibility.Data** library. It adds two new procedures that are used to add and remove the data binding of your controls. However, the visual management of the commands of Data Environment is lost. If you want to manage your data and data binding visually, you have to move your ADO code to ADO.NET.

Data Access Objects and Remote Data Objects

This section focuses on upgrading projects with DAO or RDO technology to Visual Basic .NET. However, before the upgrade process is defined, it is important to explain why these technologies are used in Visual Basic 6.0.

DAO was the first object-oriented interface to expose the Microsoft Jet database engine used by Microsoft Access. It allows Visual Basic developers to directly connect to Access tables — as well as other databases — through ODBC. DAO is suited best for either single-system applications or for small, local deployments.

RDO is an object-oriented data access interface to ODBC. It offers the easy-to-use style of DAO to provide an interface that exposes virtually all of ODBCs low-level power and flexibility. The limits of RDO are that it does not access Jet or ISAM databases very well, and it can access relational databases only through existing ODBC drivers. However, RDO has proven to be the favorite interface choice for a large number of well-known relational database developers, such as SQL Server and Oracle. RDO provides the objects, properties, and methods that are needed to access the more complex aspects of stored procedures and complex result sets.

In most of the cases, the best strategy for converting DAO and RDO data access technologies to .NET Framework is to first replace them with ADO and to then upgrade them to ADO.NET. This way, the long term goal of upgrading DAO and RDO technologies is broken into intermediate goals that provide various levels of functionality along the way.

Data Binding Upgrade Considerations

DAO and RDO have both been around for some time. DAO data binding was introduced in Visual Basic 3.0, and RDO data binding debuted in Visual Basic 4.0. When the Visual Basic development team first implemented these forms of data binding, they built them into the forms package. This implementation allowed for a seamless integration, but it also tied the data binding technology to Visual Basic Forms.

In Visual Basic .NET, Visual Basic Forms has been replaced with Windows Forms, a redesigned forms package. The designers of Windows Forms decided not to build DAO and RDO data binding into Windows Forms; therefore, DAO and RDO data binding are not supported.

Because of this, you might wonder how you can upgrade an application that uses these technologies, if you can upgrade your application without making changes, or how to change your DAO-based or RDO-based code with ADO. The following sections address these points.

DAO/RDO in Visual Basic .NET

It is possible to keep using DAO and RDO in Visual Basic .NET, as long as you do not use data binding with these technologies. But given the effort that is required to keep these technologies when you upgrade your application to Visual Basic .NET, it is probably easier to replace them with ADO in Visual Basic 6.0 before you start the upgrade. To do this, you must re-implement the data binding in your code. However, because these technologies are so similar to each other in Visual Basic 6.0, it requires that you make only minimal changes to your original code base. After you replace DAO and RDO with ADO in your original code, almost all of the remaining upgrade work is done for you automatically by the upgrade wizard.

The upgrade process includes replacing the Data Controls with ADO Data Controls and replacing the DAO/RDO technologies with ADO technology. These replacements are detailed in the following sections.

Replacing the Data Control with ADO Data Control in Visual Basic 6.0

The intrinsic Data Control implements data access by using the Microsoft Jet Database engine — the same database engine that powers Microsoft Access. This technology gives you seamless access to many standard database formats and allows you to create data-aware applications without having to write any code.

This control can be used in combination with DAO technologies to bind data to controls in a form. However, the Data Control is not supported in Visual Basic .NET, so this control is not upgraded automatically by the upgrade wizard. Instead, the upgrade wizard replaces the Data Control with a label on the form. The label's **BackColor** property is set to **red** to indicate that an upgrade issue is present.

The best way to resolve this issue is to handle it before you use the upgrade wizard. To do this, you replace the Data Control with an ADO Data Control by first adding the ADO Data Control to your Visual Basic 6.0 project. After the ADO Data Control is available to your Visual Basic 6.0 project, you need to replace each Data Control in your code and form design with an ADO Data Control.

In the **ConnectionString** property of the ADO Data Control, set the database or data provider as it is specified in the **Connect** and **DatabaseName** properties of the Data

Control. For example, if your Data Control has **Access** specified as its **Connect** property and **C:\Temp\Northwind.mdb** specified as its **DatabaseName,** you construct the connection string for your new ADO Data Control as shown in the following procedure.

► Constructing the connection string for an ADO Control

1. Open the property page for the **ConnectionString** property, and then click **Build**.
2. In the **Data Link Properties** of the **Provider** tab, select a Microsoft Jet provider (3.51 or 4.0). The version you select will depend on the version of Microsoft Office that you have installed on your computer. For example, if Microsoft Office 97 is installed, you must select version 3.51, and for all other cases, you must select version 4.0. After you select the correct provider, click the **Next** button.
3. On the **Connection** tab, navigate to your Microsoft Access database, or simply set the path to the correct database. After you specify the database to connect to, click the **OK** button in the **Data Link Properties,** and then click the **OK** button on the **Property Pages**.

After you set the **ConnectionString**, set the value and the type of the **RecordSource** that will be returned by the ADO Data Control. You can do this on the **Property Pages** of the **RecordSource** property where you can choose the two values simultaneously. Figure 12.1 shows an example of the **Property Pages**.

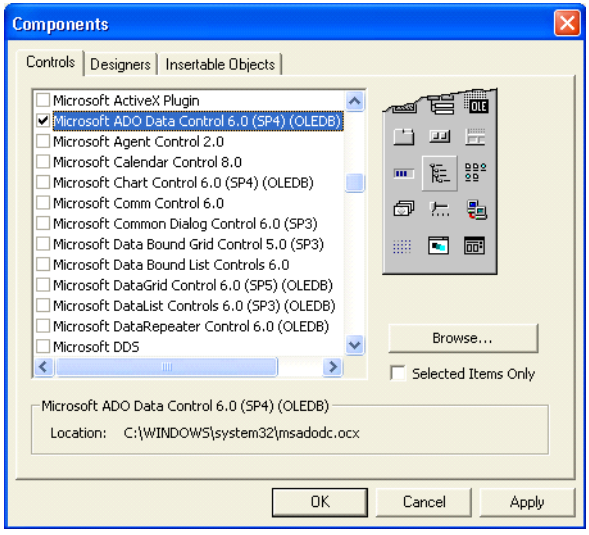

Figure 12.1
PropertyPages *for the RecordSource property of an ADO Data Control*

After you complete this procedure, you can remove the original Data Control from your form. At this point, you should rebuild and retest the application to ensure everything functions correctly before you begin the upgrade process.

Replacing DAO/RDO with ADO in Visual Basic 6.0

ADO is the successor to DAO/RDO. Functionally, ADO 2.0 is most similar to RDO, and there is generally a direct mapping between the two models. ADO "flattens" the object model used by DAO and RDO; this means that it contains fewer objects and more properties, methods (and arguments), and events. For example, ADO has no equivalent functionality to the **rdoEngine** and **rdoEnvironment** objects that exposed the ODBC driver manager and the **hEnv** interfaces. It is also not possible to create ODBC data sources from ADO, despite the fact that your interface might be through the ODBC OLE DB service provider.

Much of the functionality contained in the DAO and RDO models is consolidated into single objects, resulting in a much simpler object model. However, because of this, you might initially find it difficult to find the appropriate ADO object, collection, property, method, or event to map to DAO/RDO. Also, unlike DAO and RDO, it is possible to create ADO objects outside the scope of the ADO object hierarchy.

However, note that ADO does not currently support all of the functionality available in DAO. Instead, ADO more closely supports RDO-style functionality to interact with OLE DB data sources, remoting, and DHTML technology.

Because ADO does not currently support the data definition language (DDL), users, or groups of DAO, you should not replace DAO with ADO. Two exceptions to this rule are if your application uses ODBCDirect or it is only for client-server applications that do not rely on the Jet database engine or use DDL.

Upgrading DAO/RDO without Data Binding

Handling DAO and RDO code that does not use data binding is relatively straightforward in upgrade projects. DAO and RDO are implemented as COM libraries, so most of the code will work exactly the same in Visual Basic .NET as it did in Visual Basic 6.0. For example, the following DAO code opens the **Biblio** database and then executes a **Select** statement that returns the list of **Authors**. The name of the **Author** is then shown in a message box.

```
Dim dbsExample As Database
Dim rstExample As Recordset
Dim fldExample As Field

Set dbsExample = OpenDatabase("c:\temp\Biblio.mdb")
Set rstExample = dbsExample.OpenRecordset("Authors", dbOpenDynaset)
Set fldExample = rstExample.Fields("Author")
MsgBox CStr(fldExample.Value)
dbsExample.Close
```

This code is upgraded automatically by the upgrade wizard as shown in the following example.

```
Dim dbsExample As DAO.Database
Dim rstExample As DAO.Recordset
Dim fldExample As DAO.Field

dbsExample = DAODBEngine_definst.OpenDatabase("c:\temp\Biblio.mdb")
rstExample = dbsExample.OpenRecordset("Authors",
DAO.RecordsetTypeEnum.dbOpenDynaset)
fldExample = rstExample.Fields("Author")
MsgBox(CStr(fldExample.Value))
dbsExample.Close()
```

After the code is upgraded with the upgrade wizard, it looks essentially the same as it did in Visual Basic 6.0, but it is more explicit. The upgrade wizard also automatically adds a reference to the **Microsoft.VisualBasic.Compatibility** library, so the resulting code works perfectly in Visual Basic .NET without any manual modifications.

One major difference between Visual Basic .NET and all of the other data access technologies is how you access fields of a recordset (or result set for RDO). In Visual Basic 6.0, it is common to access fields using the shorthand coding convention *RecordsetName*!*FieldName*. In Visual Basic .NET, this code needs to be expanded to resolve the default properties. The upgrade wizard does this for you, but in this case the code looks a little different after upgrading. Consider the following Visual Basic 6.0 code example.

```
Dim rstExample As Recordset
Dim strExample As String
strExample = rstExample!Author
```

The code assigns the author's name of the recordset **rstExample** to the field **fldExample**. When the upgrade wizard is applied, the result is similar to the following code example.

```
Dim rstExample As DAO.Recordset
Dim strExample As String
strExample = rstExample.Fields("Author").Value
```

Notice that **rstExample**!**Author** was expanded to **rstExample**.**Fields**("Author").**Value**. The upgrade wizard will handle these types of expansions for you.

However, keep in mind that data binding for RDO and DAO is not supported in Visual Basic .NET. When your code uses this technology, you must research the options that allow your code to run after you upgrade the application.

Upgrading Data Access Objects (DAO)

This section covers issues that are specific to upgrading DAO technology to ADO.

DAO with Data Binding

In the following scenario, an application's project contains a reference to the Microsoft DAO library and at least one form that contains a data control. Furthermore, at least one control on the form is bound to the data control through the **DataSource** and **DataField** properties of the control.

When you upgrade these applications, you must make some manual changes in the upgraded code because DAO data binding is not supported in Visual Basic .NET. Code that uses DAO at run time is upgraded correctly by the upgrade wizard; it uses a wrapper to the DAO library, but you must manually recode all of the data control code if you want to maintain the data binding in your project.

The best strategy is to replace your data control with an ADO Data Control before you upgrade your project to Visual Basic .NET. The ADO Data Control is similar to a data control and is supported in Visual Basic .NET. Replacing this control minimizes the changes that you need to make in your code. For more information about upgrading a data control, see the "Replacing the RDO Remote Data Control with ADO Data Control in Visual Basic 6.0" section later in this chapter.

DAO without Data Binding

The following scenario applies to your application if the project contains a reference to the DAO library, but none of its forms contain a data control, and all of the data access is managed directly at run time.

When an application like this is upgraded to Visual Basic .NET, the upgrade wizard adds DAO library a wrapper to the new project. As a result, code that uses DAO data objects does not require manual changes, and the behavior is the same as it is in Visual Basic 6.0.

Upgrading Remote Data Objects (RDO)

This section covers issues that are specific for the upgrade of RDO technology to ADO.

RDO with Data Binding

Applications that fall into this scenario have projects that reference the Microsoft Remote Data Objects. In them, at least one form contains the Remote Data Control for data binding and at least one control in the form uses data binding with Remote Data Control that uses the properties, **DataSource** and **DataField** .

After you upgrade an application like this, the Remote Data Control appears to be upgraded. However, the new control is read-only at run time, and you will not be able to navigate between the results. You can correct this by changing the Remote Data Control of the upgraded project to an ADODC control in the **Microsoft.VisualBasic.Compatibility.Data** library. However, this approach might

require a significant amount of work. It is more efficient to replace the RDO technology with ADO technology in the Visual Basic 6.0 application before you upgrade the application. For more information on this approach see "Replacing the RDO Remote Data Control with ADO Data Control in Visual Basic 6.0" later in this chapter.

RDO without Data Binding

In this scenario, your application must contain in its project a reference to Microsoft Remote Data Objects. Additionally, it may not contain any RDO Data Controls; all of the data access is managed at run time.

With an application like this the upgrade of the RDO code is automatic, because the upgrade wizard creates a reference to the RDO library. Your application maintains the same RDO instructions in its code, which results in the same Visual Basic 6.0 functionality.

Replacing RDO with ADO in Visual Basic 6.0

Because RDO maps closely to ADO, it is generally a good idea to replace RDO with ADO before you upgrade your application. The following scenarios demonstrate how RDO is used to resolve data access issues and how you can achieve the same behavior with ADO. The examples assume that the application project has a reference to Microsoft ADO 2.0.

Establishing a Connection to a Database

To open a connection in RDO, you must supply a connection string with parameters. A connection is not required by RDO to create an **rdoQuery** object, but it is required to initially create an **rdoResultset** object. The following code example demonstrates opening a connection using RDO.

```
Dim WithEvents cn As rdoConnection
Dim cnB As New rdoConnection
Const ConnectString = "uid=myname;pwd=mypw;driver={SQL Server};" & _
    "server=myserver;database=pubs;dsn=''"
```

This connection string accesses a specific SQL Server and permits ODBC to open a connection without a Data Source Name (DSN). This example represents a typical ODBC connection string with all of the standard arguments.

To continue the example, consider the following form Load event code. The code establishes the type of cursor driver and the login timeout. By default, RDO uses the **rdUseIfNeeded** cursor type, which invokes server-side cursors on SQL Server. However, this default is overridden in the following example by specifying **rdUseNone**. The **rdDriverNoPrompt** flag means that the application generates an error if the user ID and password do not match.

```
Private Sub Form_Load()
    Set cn = New rdoConnection
    With cn
        .Connect = ConnectString
        .LoginTimeout = 10
        .CursorDriver = rdUseNone
        .EstablishConnection rdDriverNoPrompt
    End With
```

Within the Load event, a second connection performs client-batch updates as shown in this example.

```
    With cnB
        .Connect = ConnectString
        .CursorDriver = rdUseClientBatch
        .EstablishConnection
    End With
End Sub
```

The last event occurs when the connection operation completes. It handles any errors that occur when the connection is opened. With this event, you can test to see if the connection was established before attempting to perform any actions. For example, you can enable any buttons that rely on an open connection. This is demonstrated in the following code example.

```
Private Sub cn_Connect(ByVal ErrorOccurred As Boolean)
    If ErrorOccurred Then
        MsgBox "Could not open connection", vbCritical
    Else
        RunOKFrame.Enabled = True
    End If
End Sub
```

However, to establish a database connection in ADO, you must first create a set of ADO objects that are referenced from the ADODB object. These are used later to set specific properties that open connections and generate result sets. This is shown in the following code.

```
Dim cn As New ADODB.Connection
Dim rs As New ADODB.Recordset
Dim cnB As New ADODB.Connection
Dim Qy As New ADODB.Command
```

The next line creates a connection string, just like the one that was created in the previous RDO example. In both cases, the examples are using ODBCs "non-DSN" connection strategy to save time and to increase performance.

```
Const ConnectString= "uid=myname;pwd=mypw;driver={SQL Server};" & _
    "server=myserver;database=pubs;dsn=''"
```

Now that the variables and connection string have been created, you can open an ADO connection to a database in the form Load event. This is demonstrated here.

```
Private Sub Form_Load()
    With cn
        ' Establish DSN-less connection
        .ConnectionString = ConnectString
        .ConnectionTimeout = 10
        .Properties("Prompt") = adPromptNever
        ' This is the default prompting mode in ADO.
        .Open
    End With
    With cnB
        .ConnectionString = ConnectString
        .CursorLocation = adUseClient
        .Open
    End With
End Sub
```

Notice how this converted code is very similar to the RDO code except that the constants are prefaced with **ad** instead of **rd**. For example, the **rdDriverNoPrompt** was changed to **adPromptNever**. There is no need to specify the prompting behavior because ADO defaults to no prompt. If you do elect to change this, use the ADO Properties collection to establish the desired prompt behavior. In RDO, you can set the behavior using the OpenConnection argument. In ADO, you must set the Properties ("Prompt") property. Also, there is no need to specify a cursor driver if you do not want to use one (such as the RDO CursorDriver = rdUseNone) because ADO defaults to no cursor driver by default.

Running a Basic Query

In RDO, you can run queries using the method **OpenResultset**, which returns a **Resultset** that allows you to access all results. The following example shows how to return a **Resultset** based on a SQL statement. Notice that building a **Resultset** requires an open connection.

```
…
Private Sub RunButton_Click()
    Dim rs As rdoResultset
    Set rs = cn.OpenResultset("select * from titles where title like '%h'")

    ' Perform operations on the Resultset obtained from the query here.

    rs.Close
End Sub
…
```

To execute a query with ADO, you must use the **Open** method of a recordset, using the database connection. The following event procedure is very similar to the previous RDO code example. However, in this case, you use the new ADO **Open** method that takes the SQL query and the ADO Connection object as arguments, instead of using the **rdoConnection** object's **OpenResultset** method. You can also opt to use the ADO Connection object's **Execute** method, just as you could in RDO, as long as it does not return a rowset.

```
Private Sub RunButton_Click()
   Dim rs As New ADODB.Recordset
   rs.Open "select * from titles where title like '%h'", cn
   Set MyMSHFlexGrid.Recordset = rs
   rs.Close
End Sub
```

You can run this query and process its recordset asynchronously in ADO. When you specify the **adFetchAsynch** option on **rs.Open**, ADO causes the cursor provider to automatically populate the recordset in the background.

Displaying a Result Set in a MSHFlexGrid Control

The following code example uses the **ShowData** method of a custom ActiveX control to display data from a result set in an **MSHFlexGrid** control using RDO technology. The code sets up the grid based on the names in the **rdoColumns** property and initializes the grid, preparing it for the data. Notice the use of the **OrdinalPosition** property to index the resultset's **rdoColumns** property. The code uses the **GetClipString** method of the **rdoResultset** object to add the rows to the **MSHFlexGrid** control.

```
…
Public Function ShowData(Resultset As rdoResultset) As Variant
    Dim cl As rdoColumn
    Static GridSetup As Boolean
    Dim MaxL As Integer
    Dim rsl As rdoResultset
    Dim Rows As Variant
    On Error GoTo ShowDataEH
    Set rsl = Resultset
    If GridSetup = False Then
        FGrid1.Rows = 51
        FGrid1.Cols = rsl.rdoColumns.Count
        FGrid1.Row = 0
        For Each cl In rsl.rdoColumns
            FGrid1.Col = cl.OrdinalPosition - 1
            FGrid1 = cl.Name
            If rsl.rdoColumns(cl.OrdinalPosition - 1).ChunkRequired Then
                MaxL = 1
            Else
                MaxL = rsl.rdoColumns(cl.OrdinalPosition - 1).Size + 4
            End If
```

```
            If MaxL > 20 Then MaxL = 20
            FGrid1.ColWidth(FGrid1.Col) = TextWidth(String(MaxL, "n"))
        Next cl
        GridSetup = True
    End If
    FGrid1.Rows = 1     'Clear Grid of data (except titles)
    FGrid1.Rows = 51
    FGrid1.Row = 1
    FGrid1.Col = 0
    FGrid1.RowSel = FGrid1.Rows - 1
    FGrid1.ColSel = FGrid1.Cols - 1
    FGrid1.Clip = rsl.GetClipString(50, , , "-")

ExitShowData:
    FGrid1.RowSel = 1
    FGrid1.ColSel = 0
    Exit Function

ShowDataEH:
    Select Case Err
        Case 40022:
            FGrid1.Clear
            Resume ExitShowData
        Case 13
            FGrid1.Text = "< >"
            Resume Next
        Case Else
            MsgBox "Could not display data: " & Err & vbCrLf & Error$
            Resume ' ExitShowData
    End Select
End Function
…
```

By way of comparison, the following code example implements the **ShowData** method of a custom ActiveX control that is adapted from an RDO control, but it uses ADO. Note that the RDO **GetClipString** method is superseded in ADO by the **GetString** method. Because you then have to parse the resulting **Variant** array, the routine is noticeably slower.

Notice how you can no longer use the **OrdinalPosition** as an index on the **Fields** collection to pull out the column titles as you could in RDO. To resolve this issue, you can substitute a new integer counter to address the column being manipulated. Use the **DefinedSize** and **ActualSize** properties to find the TEXT and IMAGE data type fields that do not fit in a column. These new properties make it easier to determine the details of specific field values.

The following example adds code to handle BLOB types if they are encountered while working through the data columns.

```
Public Function ShowData(Resultset As Recordset) As Variant
    Dim cl As Field
```

```
    Static GridSetup As Boolean
    Dim MaxL As Integer
    Dim Op As Integer
    Dim rsl As Recordset
    Dim rows As Variant
    On Error GoTo ShowDataEH
    Set rsl = Resultset

    If GridSetup = False Then
        FGrid1.rows = 51
        FGrid1.Cols = rsl.Fields.Count
        FGrid1.Row = 0
        Op = 0
        For Each cl In rsl.Fields
            FGrid1.Col = Op
            FGrid1 = cl.Name
            If rsl.Fields(Op).DefinedSize > 255 Then
                MaxL = 1
            Else
                MaxL = rsl.Fields(Op).ActualSize + 4
            End If
            If MaxL > 20 Then MaxL = 20
             FGrid1.ColWidth(FGrid1.Col) = TextWidth(String(MaxL, "n"))
            Op = Op + 1
        Next cl
        GridSetup = True
    End If
    FGrid1.rows = 1
    FGrid1.rows = 51
    FGrid1.Row = 1
    FGrid1.Col = 0
    FGrid1.RowSel = FGrid1.rows - 1
    FGrid1.ColSel = FGrid1.Cols - 1
    With FGrid1
        ' You can also use the ADO2 GetString method here in lieu of the
        ' following.
        FGrid1.Clip = rsl.GetString(adClipString, 50, , , "-")
    End With

ExitShowData:
    FGrid1.RowSel = 1
    FGrid1.ColSel = 0
    Exit Function

ShowDataEH:
    Select Case Err
        Case 3021:
            FGrid1.Clear
            Resume ExitShowData
        Case 13, Is < 0
            rows(j, i) = "< >"
            Resume 'Next
        Case Else
            MsgBox "Could not display data: " & Err & vbCrLf & Error$
```

```
            Resume ' ExitShowData
    End Select
End Function
```

For more information about converting RDO 2.0 to ADO 2.0, see "Visual Basic Concepts: Converting from RDO 2.0 to ADO 2.0" on MSDN.

Replacing the RDO Remote Data Control with ADO Data Control in Visual Basic 6.0

If you upgrade an application that contains Remote Data Control technology, it appears the controls are upgraded by the upgrade wizard; however, the new controls are read-only at run time. In this case, you will not be able to navigate between results. A better option is to replace the RDO technology with ADO technology in the Visual Basic 6.0 application before you upgrade it.

The following example illustrates the process that you can use to upgrade a Remote Data Control. The example contains a Remote Data Control named **RDC1**, and a TextBox control named **OrderId**.

In this example, **RDC1** receives all of the orders that are stored in the Orders table, which is part of the Northwind Microsoft Access database that uses a DSN named **Access**. The control's properties are set as listed in Table 12.1.

Table 12.1: Remote Data Control Properties for the Upgrade Example

RemoteData Control properties	Value
Connect	DSN=Access
CursorDriver	0-rdUseIfNeeded
DataSourceName	MS Access Database
Password	<empty>
SQL	Select * from Orders
UserName	Admin

In the **TextBox** control, the **DataSource** property is set to **RDC1** and the **DataField** property is set to **OrderID**.

To use an ADO data control, perform the following procedure.

► Using an ADO data control

1. Add the ADO reference to the project, as explained in the previous section, "Replacing the Data Control with ADO Data Control in Visual Basic 6.0."
2. Add an ADO data control to the form, and set its name with a representative name. In the current example, it is **ADORDC1**.

3. In the **ConnectionString** property of the ADO data control, set the Access DSN that is used in the **RemoteData** control. Figure 12.2 shows how to do this through the **Property Page**s.

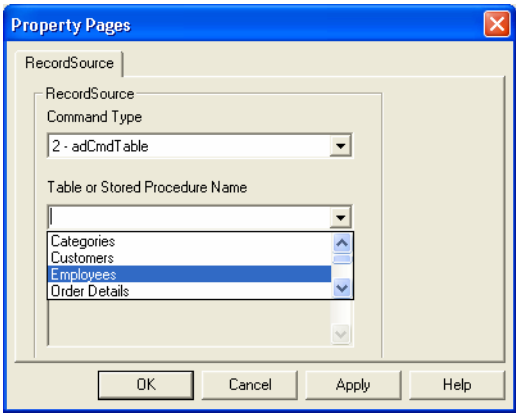

Figure 12.2
*Setting the **ConnectionString** of an ADO data control through the property page.*

4. Set the **UserName** property of the ADO data control to **Admin**.
5. In the **RecordSource** property of the ADO data control, set the query "**select * from Orders**".
6. Delete the **RemoteData** control.
7. Change the value of the **DataSource** property of the **TextBox** control to **ADORDC1**.

After you have completed this procedure, rebuild and retest the application to ensure everything functions correctly before you begin the upgrade process.

The next section discusses upgrading all DAO and RDO technology to ADO.

Custom Data Access Components

Custom data access components are those components that encapsulate data access technologies such as ADO, DAO, or RDO inside a DLL or an OCX that your application uses. These components may have been developed internally in a specific language, or they may have been purchased from another company with or without the source code. There are various alternatives for upgrading applications that use custom data access components. The approach that you take will depend on whether you have access to the source code of the component and the language that was used to develop it.

One of these three scenarios might apply to your application:

- You do not have the source code for the custom component.
- You have the source code for the custom component and it was developed in Visual Basic 6.0.
- You have the source code for the component, but it was not developed in Visual Basic 6.0.

For all of these cases, you can choose from two upgrade options:

- Upgrade your application and use a wrapper for the component.
- Upgrade the application and the component separately, and replace the old component with a new Visual Basic .NET based component.

The following section discusses each option.

Upgrading to a .NET Version of the Component

When you have the source code for the custom data access components that are in your application and they are written in Visual Basic 6.0, you can often upgrade them to Visual Basic .NET. Typically, you can do this apart from upgrading the rest of the application. This is called a staged upgrade and is discussed in Chapter 2, "Practices for Successful Upgrades." However, the determining factor in how you choose to upgrade these components depends heavily on the underlying data access technology that was used to build them.

As discussed in previous sections, code that uses DAO/RDO is most easily upgraded if you first replace the DAO/RDO with ADO in the Visual Basic 6.0 code. On the other hand, if the component directly accesses custom database files, it might make the most sense to replace the data access with a connection to a database server, such as Microsoft SQL Server. In each of these scenarios, the disadvantage is that you must spend the time and effort to prepare the code before you perform the upgrade. There is also the possibility that your component's behavior will change as a result of the preparation. However, this could turn out to be an advantage because as you resolve behavior issues you have the opportunity to add new data access functionality. Another advantage to upgrading a data access component to Visual Basic .NET is that you have the opportunity to use the newest data access technology, ADO.NET. ADO.NET is entirely integrated with Visual Basic .NET and provides easy access to databases.

If the custom data access component you need to upgrade was purchased from a third party, it is possible that a new version of the component has been released. In this case, it is also possible that the new version was developed in Visual Basic .NET. If that is the case, the amount of work you have to do to upgrade your application may be considerably reduced.

Using COM Interop with Custom Data Access Components

When you do not have the source code for the custom data access components in your application, or the code is developed in a language that is not Visual Basic 6.0, your best option is to upgrade your application with the upgrade wizard. The upgrade wizard creates a wrapper to the data access component, which allows you to use the existing statements and retain the same Visual Basic 6.0 functionality in Visual Basic .NET through COM interop.

The advantage of this approach is that the changes that you need to make in the upgraded code with respect to the data access component are minimal. At the same time, your application retains the same functionality that it had in Visual Basic 6.0. The disadvantages are that the component continues to use old technologies and that the overhead of using COM interop will decrease the performance of the application. For more information about COM interop, see Chapter 14, "Interop Between Visual Basic 6.0 and Visual Basic .NET."

Of course, you can use this strategy even if the custom component source code is available and developed in Visual Basic 6.0, but the best method for this scenario is to upgrade the component to Visual Basic .NET as discussed in the previous section, "Upgrading to a .NET Version of the Component."

Upgrading Mixed Data Access Technologies

When your application uses different technologies such as ADO, RDO, DAO and custom data access, you should consolidate all of the data access into a single technology before beginning the upgrade. Your best option in this case is to use ADO because it can be upgraded automatically by the upgrade wizard. This in turn, will reduce the number of manual changes you will need to make in the upgraded code.

As an alternative, you can choose to upgrade each technology separately. This requires that you create a new Visual Basic 6.0 project for each technology that your application uses. Each project must be upgraded separately. After all of the projects have been upgraded, you will have to integrate them into a single application in Visual Basic .NET.

Converting Data Reports to Crystal Reports

Visual Basic 6.0 provides the Microsoft Data Report Designer, a versatile data report generator that features the ability to created banded hierarchical reports. Used in conjunction with a data source such as the Data Environment Designer, you can create reports from several different relational tables. In addition to creating printable reports, you can also export the report to HTML or text files.

In Visual Basic .NET, the standard reporting tool is Crystal Reports. This tool allows the creation of robust reports that adapt to an application's needs. There are many advantages to using Crystal Reports.

One of the advantages of Crystal Reports is the ability to convert a Microsoft Data Report (.dsr file) to a Crystal Report (.rpt file) through a relatively easy automated process. When a Data Report is converted, an equivalent Crystal Report is generated. The resulting report has the same design and data access as the original report. Database connections and the general structure of the report are kept under a similar model.

However, the Crystal Reports tool cannot convert Data Reports that contain a query that uses parameters to retrieve data. For example, a Data Report with the following query cannot be converted automatically:

```
SELECT Products.* FROM Products WHERE (ProductID = ?)
```

In these cases, your best option is to manually re-implement the reports in Crystal Reports. For more information about how to create Crystal Reports, see:

- "Reports in Windows Applications" in *Crystal Reports for Visual Studio .NET* on MSDN at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/crystlmn/html/crconreportsinwindowsapplications.asp.*
- "Add Professional Quality Reports to Your Application with Visual Studio .NET" in *MSDN Magazine* at *http://msdn.microsoft.com/msdnmag/issues/02/05/Crystal/.*

In cases where Crystal Reports can be used for automatic conversion, the process consists of several phases:

1. You will first need to upgrade all of the associated Visual Basic 6.0 projects, forms, and data access components to Visual Basic .NET.
2. Next, you will need to upgrade the Data Reports to Crystal Reports.
3. Finally, you will have to make some changes in the project to allow the upgraded code to work with the converted Crystal Reports.

The following example shows you how to upgrade Data Reports. In the example, a Visual Basic 6.0 project contains a report named ProductReport that is used to show the information about products in the Northwind database, a sample database included with Microsoft Office. A DataEnvironment object named **DataBase** connects to the database, and a form is used to display the products. The form includes two buttons that allow a user to either display (**btnShow**) or export (**btnFile**) the report.

The **Show Report** button displays the report on the screen; the **Export Report** button exports the report to a text file. The code for each of these events follows.

```
Private Sub btnFile_Click()
```

```
    ProductReport.ExportReport rptKeyText, "c:\temp\Reports\Products.doc"
End Sub

Private Sub btnShow_Click()
    ProductReport.Show
End Sub
```

When you use the upgrade wizard to upgrade the project, the DataReport is not upgraded. Instead, it is copied into the new project unchanged. The code in the form and the DataEnvironment are upgraded and now require the same amount of work as a typical upgrade project.

After you upgrade the code base, the next phase is to convert the DataReport to a Crystal Report. Use the following procedure to convert the DataReport.

► Converting a DataReport to a Crystal Report

1. In the **Solution Explorer** pane of the Visual Studio IDE, click the **Add New Item** option to add the Crystal Report to your upgraded project. When you add the new report, the **Crystal Report Gallery** dialog box displays, as shown in Figure 12.3.

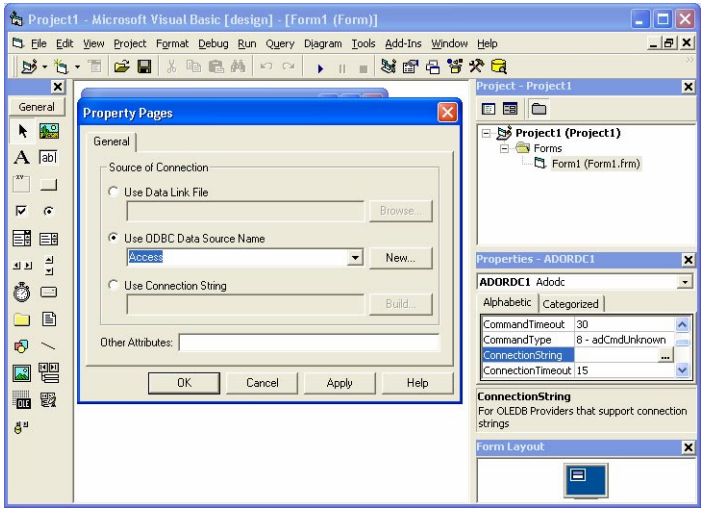

Figure 12.3
The Crystal Report gallery

2. In the **Crystal Report Gallery** dialog box, click the **From an Existing Report** option.
3. In the **Search** dialog box, navigate to the DataReport file and select it. The DataReport file is part of your Visual Basic 6.0 application's source code. After selecting the appropriate file, click **OK**.

After you complete these steps, the Crystal Report Engine automatically upgrades your DataReport object. When the upgrade process is done, you can remove the DataReport that was copied to your project by the upgrade wizard.

The final phase of converting a DataReport to a Crystal Report is to make changes in the upgraded project that allow it to work with the newly created Crystal Report. To do so, add the following references to your project:

- CrystalDecisions.Shared
- CrystalDecisions.ReportSource
- CrystalDecisions.CrystalReports.Engine
- CrystalDecisions.Windows.Forms

To replicate the behavior of the **Show Report** button, you have to add a new form to the project. In this example, you can add a form named **DataReport**. You will need to add Crystal Report Viewer to this form, and set its **Dock** property to **Fill** and its **Modifiers** property to **Public**. Optionally, you can also set its name to **Viewer**. After the form and the viewer are added, you will need to adjust the button event handler code to display the report in the new form. After you apply the upgrade wizard to the original code, the **Show Report** button code gives the following warning.

```
Dim ProductReport As Object
' UPGRADE_WARNING: Couldn't resolve default property of object ProductReport.Show.
ProductReport.Show()
```

To resolve this upgrade issue, you have to declare variables for the new report and form. You then have to assign the report to **ReportSource** property of the Crystal Report Viewer. The final modified code is shown here.

```
Dim ProductReport As New ProductReport
Dim report As New DataReport
report.Viewer.ReportSource = ProductReport
report.Show()
```

With these changes, the Crystal Report exhibits the same behavior as the original Visual Basic 6.0 report when the **Show Report** button is clicked.

To replicate the behavior of the **Export File** button, you also have to manually adjust the upgraded code. In this case, it is necessary to use the **Export** function of the Crystal Report. The following code example shows the result of applying the upgrade wizard to the original Visual Basic 6.0 event handler for the **Export File** button.

```
Dim ProductReport As Object
'UPGRADE_WARNING: Couldn't resolve default property of object
ProductReport.ExportReport.
ProductReport.ExportReport(MSDataReportLib.ExportKeyConstants.rptKeyText,
"c:\temp\Reports\Products.doc")
```

To resolve this upgrade issue, you need to declare a variable for the Crystal Report. To export the report to a text file, invoke the **Export** method of the Crystal Report in place of the **ExportReport** function in the event handler, as shown in the following code.

```
Dim ProductReport As New ProductReport
ProductReport.ExportToDisk(ExportFormatType.RichText, _
    "c:\temp\Reports\Products.doc")
```

Summary

More than likely, you will need to upgrade Visual Basic 6.0 applications that use some form of data access. This chapter provides the strategies that you can use to upgrade your data access code.

More Information

For more information about how to create Crystal Reports, see:

- "Reports in Windows Applications" in *Crystal Reports for Visual Studio .NET* on MSDN:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/crystlmn/html/crconreportsinwindowsapplications.asp.
- "Add Professional Quality Reports to Your Application with Visual Studio .NET" in *MSDN Magazine*:
  http://msdn.microsoft.com/msdnmag/issues/02/05/Crystal/.

For information about the ArtinSoft Visual Basic .NET Ready program, see the ArtinSoft Web site:
http://www.artinsoft.com/iproducts/vb6todotnet/laboratories.asp.

For more information about converting RDO 2.0 to ADO 2.0, see "Converting from RDO 2.0 to ADO 2.0" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon98/html/vbconconvertingfromrdotoado.asp.

13

Working with the Windows API

The Windows API is the foundation that every Windows application is built on. Visual Basic applications are no exception. Visual Basic, through the language and forms package, abstracts the Windows API to a set of easy-to-use statements, components, and controls. In many cases, you can build a Visual Basic application without having to directly call a Windows API function. However, because the Visual Basic language and forms packages do not represent every Windows API function available, Visual Basic supports directly calling Windows API functions; it has done this since version 1.0. This enables you to add capabilities to your application that cannot be implemented by the Visual Basic runtime.

Visual Basic .NET moves forward with this capability by enabling you to declare and call Windows API functions in the same way as before. Furthermore, access to many of these APIs has been exposed through the more comprehensive .NET Framework class library. You now have two options for upgrading these API function calls.

The first option is to continue to use the Windows API. This option requires that you use interoperability techniques to access the API. This option requires the least amount of effort to upgrade API function calls, but results in a dependence on unmanaged code. (For more information about managed code, see the "Increased Productivity" section of Chapter 1, "Introduction.")

The second option is to replace Windows API calls with Visual Basic .NET alternatives. This option requires more effort to upgrade API function calls, but the result is fully managed code that is not dependent on an unmanaged library.

This chapter demonstrates both approaches for working with the Windows API. It focuses on the language changes that affect the way you create **Declare** statements so that you can continue to use the Windows API functions. It also shows you how to use classes and methods that are contained in the .NET Framework to complement or replace the Windows API calls you make in your application.

Type Changes

Visual Basic .NET contains changes in the data types that affect several Windows API functions. Some of these changes are related to the **Integer** and **Long** data types, and others are related to fixed-length strings. You must use caution when upgrading to Visual Basic .NET to ensure proper invocation of Windows API functions.

Changes to Integer and Long Data Types

The storage sizes for the **Integer** and **Long** data types have changed between Visual Basic 6.0 and Visual Basic .NET to maintain consistency with the .NET Framework. These changes affect almost every Windows API function declaration that involves the numeric types **Integer** and **Long**. In Visual Basic 6.0, an **Integer** is 16 bits and a **Long** is 32 bits. In Visual Basic .NET, an **Integer** is 32 bits and a **Long** is 64 bits. Visual Basic .NET adds a new type named **Short**, which, in terms of size, is the replacement for the Visual Basic 6.0 **Integer** type. In Visual Basic .NET, when you create a new **Declare** statement for a Windows API function, you need to be mindful of this difference. Any parameter type or user-defined type member that was formerly a **Long** needs to be declared as **Integer**; any member formerly declared as **Integer** needs to be declared as **Short**.

Take, for example, the following Visual Basic 6.0 **Declare** statement, which contains a mix of **Integer** and **Long** parameters.

```
Declare Function GetKeyState Lib "user32" (ByVal nVirtKey As Long) _
    As Integer
```

The equivalent Visual Basic .NET declaration is shown here.

```
Declare Function GetKeyState Lib "user32" (ByVal nVirtKey As Integer) _
    As Short
```

The Visual Basic Upgrade Wizard will automatically change all the variable declarations in your code to use the correct size (changes in the code are marked in **bold**). You need to consider the size of a type only when you are creating new **Declare** statements or you are modifying existing statements.

Changes to Fixed-Length Strings

Visual Basic 6.0 has a fixed-length string data type that is not supported in Visual Basic .NET. Any fixed-length string declarations in your code must be upgraded to a fixed-length string wrapper class that is provided in Visual Basic .NET. Consider the following Visual Basic 6.0 function declaration.

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Long) As Long
Function GetUser()
    Dim Ret As Long
    Dim UserName As String
    Dim Buffer As String * 25
    Ret = GetUserName(Buffer, 25)
    UserName = Left$(Buffer, InStr(Buffer, Chr(0)) - 1)
    MsgBox (UserName)
End Function
```

The upgrade wizard upgrades the code to the following.

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA"(ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Function GetUser() As Object
    Dim Ret As Integer
    Dim UserName As String
    Dim Buffer As New VB6.FixedLengthString(25)
    Ret = GetUserName(Buffer.Value, 25)
    UserName = Left(Buffer.Value, InStr(Buffer.Value, Chr(0)) - 1)
    MsgBox(UserName)
End Function
```

The preceding code has the same behavior as the original Visual Basic 6.0 code, but results in a dependency on the Visual Basic 6.0 compatibility runtime. Another approach is to remove this dependency on the Visual Basic 6.0 compatibility runtime. You can do this either by preparing your code prior to the upgrade or by changing the Visual Basic .NET code that is produced by the upgrade wizard.

To prepare your Visual Basic 6.0 code prior to the upgrade, you can rewrite this code by using a normal string explicitly set to length 25 instead of a fixed-length string, as shown here.

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Long) As Long
Function GetUser()
    Dim Ret As Long
    Dim UserName As String
    Dim Buffer As String
    Buffer = String$(25, " ")
    Ret = GetUserName(Buffer, 25)
    UserName = Left$(Buffer, InStr(Buffer, Chr(0)) - 1)
    MsgBox (UserName)
End Function
```

Alternatively, you can remove the Visual Basic 6.0 compatibility runtime dependency by changing the upgraded Visual Basic .NET code to the following.

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA"(ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Function GetUser() As Object
   Dim Ret As Integer
   Dim UserName As String
   Dim Buffer As New String(" ", 25)
   Ret = GetUserName(Buffer, 25)
   UserName = Left(Buffer, InStr(Buffer, Chr(0)) - 1)
   MsgBox(UserName)
End Function
```

Either option reduces your code's dependence on the Visual Basic 6.0 compatibility runtime.

Variable Type "As Any" Is No Longer Supported

Visual Basic 6.0 allows you to declare parameter types using the **As Any** variable type. This declaration allows you to pass an argument of any type; Visual Basic 6.0 passes the correct information when the call is made. This gives you greater flexibility, but no type checking is performed on the argument when you call the API function. Thus, if you pass an incompatible argument type, the application may generate a run-time exception.

Visual Basic .NET does not support declaring Windows API parameters with the **As Any** variable type. However, a similar effect can be achieved by declaring the same API function multiple times, using different types for the same parameter in each declaration.

The Windows API function **SendMessage** is an example of an instance where you would use the **As Any** type in the API function declaration in Visual Basic 6.0 code. Depending on the Windows message you are sending, the parameter types required for the message will vary. For example, the **WM_SETTEXT** message requires you to pass a string as the last parameter. By contrast, the **WM_GETTEXTLENGTH** message requires you to pass 0, a numeric value, for the last parameter. To handle both of these messages, you create the Visual Basic 6.0 **Declare** statement for **SendMessage** as shown here.

```
Private Declare Function SendMessage Lib "user32" _
Alias "SendMessageA" _
   (ByVal hwnd As Long, _
    ByVal wMsg As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Any) As Long
```

Notice how the last parameter has been declared with the **As Any** type. The following code demonstrates two different invocations of this function, passing a different type for the last parameter in each invocation. This example uses **SendMessage** to set the caption of the current form and then retrieve the length of the caption set.

```
Dim TextLen As Long
Const WM_SETTEXT = &HC
Const WM_GETTEXTLENGTH = &HE

' Invoke SendMessage using a String value for the last parameter.
SendMessage Me.hwnd, WM_SETTEXT, 0, "My text"
' Invoke SendMessage using a numeric value for the last parameter.
TextLen = SendMessage(Me.hwnd, WM_GETTEXTLENGTH, 0, 0&)
```

To implement the equivalent functionality in Visual Basic .NET, you create multiple **Declare** statements for the **SendMessage** function. In the case of the **SendMessage** API, you create two **Declare** statements, using two different types for the last parameter, **String** and **Integer**, as shown here.

```
Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
   (ByVal hwnd As Integer, _
    ByVal wMsg As Integer, _
    ByVal wParam As Integer, _
    ByVal lParam As String) As Integer

Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
   (ByVal hwnd As Integer, _
    ByVal wMsg As Integer, _
    ByVal wParam As Integer, _
    ByVal lParam As Integer) As Integer
```

Note: You also need to change the other parameter types from **Long** to **Integer**, as discussed in the "Changes to Fixed-Length Strings" section earlier in this chapter.

For this example, the equivalent Visual Basic .NET code uses **SendMessage** to set the caption of the current form and then retrieve the length of the caption set.

```
Dim TextLen As Integer
Const WM_SETTEXT = &HC
Const WM_GETTEXTLENGTH = &HE

' Invoke SendMessage using a String value for the last parameter
SendMessage(Me.Handle.ToInt32, WM_SETTEXT, 0, "My text")
' Invoke SendMessage using a numeric value for the last parameter
TextLen = SendMessage(Me.Handle.ToInt32, WM_GETTEXTLENGTH, 0, 0)
```

Creating multiple **Declare** statements for the same function is more work than using the **As Any** variable type, but you benefit by getting both the flexibility of using the same function name and type checking when the call is made.

Passing User-Defined Types to API Functions

When you pass a Visual Basic 6.0 user-defined type to an API function, Visual Basic passes a pointer to the memory that contains the user-defined type. The API function sees the members of the user-defined type in the same order that they were declared in Visual Basic. However, this is not the case for Visual Basic .NET. If you declare a user-defined type, the order of the members is not guaranteed to stay the same in the code. The common language runtime (CLR) may reorganize the members of a user-defined type in a way that is most efficient for the user-defined type to be passed to a function. To guarantee that the members are passed exactly as declared in the code, you need to use marshaling attributes.

For example, if your code is calling an API function named **MyFunction** (written in C or C++), that takes a 2-byte **Boolean** type (**VARIANT_BOOL**) parameter, you can use the **MarshalAs** attribute to specify the parameter type that the API function expects. Usually, a **Boolean** parameter is passed using 4 bytes, but if you include **UnmanagedType.VariantBool** as a parameter to the **MarshalAs** attribute, the parameter is marshaled — or, passed — as a 2-byte **VARIANT_BOOL** argument. The following code demonstrates how to apply the **MarshalAs** attribute to your API function declarations.

```
Declare Sub MyFunction Lib "MyLibrary.dll" _
   (<MarshalAs(UnmanagedType.VariantBool)> ByVal MyBool As Boolean)
```

The **MarshalAs** attribute is contained in the **System.Run-time.InteropServices** namespace. To call **MarshalAs** without qualification, you need to add an **Imports** statement to the top of the module for **System.Runtime.InteropServices**, as shown here.

```
Imports System.Runtime.InteropServices
```

For any structure passed to API functions, the structure should be declared using the **StructLayout** attribute to ensure compatibility with the Windows API. The **StructLayout** attribute takes a number of parameters, but the two most significant attributes for ensuring compatibility are **LayoutKind** and **CharSet**. For example, to specify that the structure members should be passed in the same order that they are declared, set the **LayoutKind** attribute to **LayoutKind.Sequential**. To ensure that string parameters are marshaled as ANSI strings, set the **CharSet** attribute to **CharSet.Ansi**.

Note: It is important to know that the .NET Framework uses Unicode strings when it communicates with external APIs. It may be necessary to use different string marshaling options, such as ANSI encoding, or an encoding that is determined by the underlying operating system.

The following code snippet shows a sample external function declaration that requires the ANSI **CharSet** attribute because the function expects an ANSI string.

```
Declare Function CharUpperA Lib "user32" (<MarshalAs(UnmanagedType.LPStr)> _
ByVal s As String) As String
```

This function receives one ANSI string and returns a corresponding string with all of the characters converted to uppercase. However, if a Unicode version of the same function is provided in the API, it is a better choice when it is accessed from the .NET Framework. It will have better performance, because there are no conversions between encodings, and the declaration is simplified because no marshaling is required, as shown in the following example.

```
Declare Unicode Function CharUpperW Lib "user32" (ByVal s As String) As String
```

In general, whenever possible, use Unicode versions of the API functions and use **<MarshalAs(UnmanagedType.LPWStr)>** instead of **<MarshalAs(UnmanagedType.LPStr)>** for string arguments. Apply the same reasoning to structure fields that are necessary for API interaction. Whenever possible, use Unicode versions of structures and use **<StructLayout(LayoutKind.Sequential, CharSet:= CharSet.Unicode)>** instead of **<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)>** as the marshaling attributes for structures and string fields.

Suppose you are calling an API function named **EnumFontFamilies,** in a DLL named **GDI32**, that takes as a parameter a structure named **LOGFONT**. The **LOGFONT Structure** contains several fields of integer and byte types and one fixed-length array of byte type, declared in Visual Basic 6.0, as shown here.

```
Public Const LF_FACESIZE = 32
Type LOGFONT
        lfHeight As Long
        lfWidth As Long
        lfEscapement As Long
        lfOrientation As Long
        lfWeight As Long
        lfItalic As Byte
        lfUnderline As Byte
        lfStrikeOut As Byte
        lfCharSet As Byte
        lfOutPrecision As Byte
        lfClipPrecision As Byte
        lfQuality As Byte
        lfPitchAndFamily As Byte
        lfFaceName(LF_FACESIZE) As Byte
End Type
```

The upgrade wizard would upgrade the declaration to this approximate equivalent.

```
Structure LOGFONT
   Dim lfHeight As Integer
   Dim lfWidth As Integer
   Dim lfEscapement As Integer
   Dim lfOrientation As Integer
   Dim lfWeight As Integer
   Dim lfItalic As Byte
   Dim lfUnderline As Byte
   Dim lfStrikeOut As Byte
   Dim lfCharSet As Byte
   Dim lfOutPrecision As Byte
   Dim lfClipPrecision As Byte
   Dim lfQuality As Byte
   Dim lfPitchAndFamily As Byte
   <VBFixedArray(LF_FACESIZE)> Dim lfFaceName() As Byte
   Public Sub Initialize()
       ReDim lfFaceName(LF_FACESIZE)
   End Sub
End Structure
```

The preceding declaration is approximate because it is missing attributes to make it fully equivalent with the Visual Basic 6.0 declaration. Specifically, it is missing attributes in the structure to specify layout. It is also missing an attribute on **lfFaceName** member to mark it as a fixed-length array.

There is no guarantee that the CLR will pass the members of the structure in the same order that they were declared. For example, the CLR may reorganize the structure in memory so that the **lfFaceName** member comes first. To ensure that the order of the members is preserved, you need to add marshaling attributes to force the CLR to use the declaration order when passing structures to functions. The following code demonstrates how to accomplish this.

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure LOGFONT
   Dim lfHeight As Integer
   Dim lfWidth As Integer
   Dim lfEscapement As Integer
   Dim lfOrientation As Integer
   Dim lfWeight As Integer
   Dim lfItalic As Byte
   Dim lfUnderline As Byte
   Dim lfStrikeOut As Byte
   Dim lfCharSet As Byte
   Dim lfOutPrecision As Byte
   Dim lfClipPrecision As Byte
   Dim lfQuality As Byte
   Dim lfPitchAndFamily As Byte
   <VBFixedArray(LF_FACESIZE)> Dim lfFaceName() As Byte
   Public Sub Initialize()
```

```
        ReDim lfFaceName(LF_FACESIZE)
    End Sub
End Structure
```

Another issue arises with respect to the **lfFaceName** member. As mentioned earlier in this chapter, Visual Basic .NET does not natively support fixed-length arrays. However, when you pass a structure containing an array to an API function that requires fixed-length strings, you can change the byte member to a character array and include a **MarshalAs** attribute to tell the CLR to pass a byte array of a fixed size. To do this, declare the structure member (in this case, **lfFaceName**) as a **Char** array instead of a **Byte** array. It is necessary to include the **UnmanagedType.ByValArray** and **SizeConst** arguments to the **MarshalAs** attribute. The modified structure declaration is as follows.

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure LOGFONT
    Dim lfHeight As Integer
    Dim lfWidth As Integer
    Dim lfEscapement As Integer
    Dim lfOrientation As Integer
    Dim lfWeight As Integer
    Dim lfItalic As Byte
    Dim lfUnderline As Byte
    Dim lfStrikeOut As Byte
    Dim lfCharSet As Byte
    Dim lfOutPrecision As Byte
    Dim lfClipPrecision As Byte
    Dim lfQuality As Byte
    Dim lfPitchAndFamily As Byte
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=32)> _
    Dim lfFaceName() As Char
    Public Sub Initialize()
        ReDim lfFaceName(LF_FACESIZE)
    End Sub
End Structure
```

With both the **StructLayout** and **MarshalAs** attributes in place, the structure will be passed in memory to an API function in the same way the equivalent Visual Basic 6.0 **Type…End Type** structure is passed.

Note: The upgrade wizard incorrectly upgrades fixed-length arrays contained in structures. Instead of declaring the arrays as a **Char** array, the wizard declares the upgraded fixed-length arrays as a **Byte**. The wizard applies the **VBFixedArray** attribute instead of the **ByValArray** attribute. You will have to manually correct any occurrences of this in your code.

Changes to "AddressOf" Functionality

Certain Windows API functions, such as **EnumFontsFamilies**, require a pointer to a callback function. When you call the API function, Windows invokes the callback function that you provided. In the case of **EnumFontsFamilies**, Windows will call the function for each available font.

Visual Basic 6.0 allows you to declare Windows API functions that take callback function pointers by declaring the function pointer parameter as **Long**, to represent a 32-bit pointer. Your code calls the API function and by qualifying the subroutine name with the **AddressOf** keyword, it passes the subroutine to serve as the callback function.

Visual Basic .NET still supports the **AddressOf** keyword, but instead of returning a 32-bit integer, it returns a *delegate*. A delegate is a new type in Visual Basic .NET that allows you to declare pointers to functions or to class members. This means that you can still create **Declare** statements for Windows API functions that take a callback function pointer as a parameter. The difference is that instead of declaring the parameter type as a 32-bit integer, you need to declare the function parameter as a delegate type.

The following Visual Basic 6.0 code demonstrates how to use **AddressOf** to pass a pointer to a callback function.

```
Public Const LF_FACESIZE = 32
Type LOGFONT
    lfHeight As Long
    lfWidth As Long
    lfEscapement As Long
    lfOrientation As Long
    lfWeight As Long
    lfItalic As Byte
    lfUnderline As Byte
    lfStrikeOut As Byte
    lfCharSet As Byte
    lfOutPrecision As Byte
    lfClipPrecision As Byte
    lfQuality As Byte
    lfPitchAndFamily As Byte
    lfFaceName(LF_FACESIZE) As Byte
End Type

Declare Function EnumFontFamilies Lib "gdi32" Alias "EnumFontFamiliesA" _
    (ByVal hDC As Long, ByVal lpszFamily As String, _
    ByVal lpEnumFontFamProc As Long, LParam As Integer) As Long
Declare Function GetDC Lib "user32" (ByVal hWnd As Long) As Long
Declare Function ReleaseDC Lib "user32" (ByVal hWnd As Long, _
    ByVal hDC As Long) As Long

Dim FontList As New Collection
```

```
Function EnumFontFamProc(lpNLF As LOGFONT, lpNTM As Long, _
         ByVal FontType As Long, LParam As Integer) As Long
    Dim FaceName As String
    Dim FullName As String
    FaceName = StrConv(lpNLF.lfFaceName, &H40)
    FontList.Add Left$(FaceName, InStr(FaceName, vbNullChar) - 1)
    EnumFontFamProc = 1
End Function

Sub FillListWithFonts(LB As ListBox)
    Dim font As Variant
    Dim hDC As Long
    LB.Clear
    hDC = GetDC(LB.hWnd)
    EnumFontFamilies hDC, vbNullString, AddressOf EnumFontFamProc, 0
    For Each font In FontList
      LB.AddItem font
    Next
    ReleaseDC LB.hWnd, hDC
End Sub
```

The upgrade wizard upgrades the preceding code to the following code.

```
Public Const LF_FACESIZE As Short = 32
Structure LOGFONT
    Dim lfHeight As Integer
    Dim lfWidth As Integer
    Dim lfEscapement As Integer
    Dim lfOrientation As Integer
    Dim lfWeight As Integer
    Dim lfItalic As Byte
    Dim lfUnderline As Byte
    Dim lfStrikeOut As Byte
    Dim lfCharSet As Byte
    Dim lfOutPrecision As Byte
    Dim lfClipPrecision As Byte
    Dim lfQuality As Byte
    Dim lfPitchAndFamily As Byte
    <VBFixedArray(LF_FACESIZE)> Dim lfFaceName() As Byte

    Public Sub Initialize()
        ReDim lfFaceName(LF_FACESIZE)
    End Sub
End Structure

Declare Function EnumFontFamilies Lib "gdi32" Alias _
    "EnumFontFamiliesA"(ByVal hDC As Integer, ByVal lpszFamily As String, _
    ByVal lpEnumFontFamProc As Integer, ByRef LParam As Short) As Integer
Declare Function GetDC Lib "user32" (ByVal hWnd As Integer) As Integer
Declare Function ReleaseDC Lib "user32" (ByVal hWnd As Integer, _
    ByVal hDC As Integer) As Integer

Dim FontList As New Collection
```

```
Function EnumFontFamProc(ByRef lpNLF As LOGFONT, ByRef lpNTM As Integer, _
         ByVal FontType As Integer, ByRef LParam As Short) As Integer
    Dim FaceName As String
    Dim FullName As String
    FaceName = _
        StrConv(System.Text.UnicodeEncoding.Unicode.GetString(lpNLF.lfFaceName), _
        &H40s)
    FontList.Add(Left(FaceName, InStr(FaceName, vbNullChar) - 1))
    EnumFontFamProc = 1
End Function

Sub FillListWithFonts(ByRef LB As System.Windows.Forms.ListBox)
    Dim font As Object
    Dim hDC As Integer
    LB.Items.Clear()
    hDC = GetDC(LB.Handle.ToInt32)
    'UPGRADE_WARNING: Add a delegate for AddressOf EnumFontFamProc
    EnumFontFamilies(hDC, vbNullString, AddressOf EnumFontFamProc, 0)
    For Each font In FontList
        LB.Items.Add(font)
    Next font
    ReleaseDC(LB.Handle.ToInt32, hDC)
End Sub
```

The upgraded code has several issues that must be fixed before it will compile and run without error. The first issue relates to the structure LOGFONT. This structure requires that you attach marshaling attributes to the structure to specify how it is stored in memory and passed to the API function. For information about attaching marshaling attributes, see "Passing User-Defined Types to API Functions" earlier in this chapter.

The second issue relates to the conversion of the fixed-byte array to a string. The error occurs in the following line.

```
FaceName = _
    StrConv(System.Text.UnicodeEncoding.Unicode.GetString(lpNLF.lfFaceName), _
    &H40S)
```

Change the preceding line to the following line.

```
FaceName = New String(lpNLF.lfFaceName)
```

The final issue is associated with the **AddressOf** expression. This compile error occurs because the **AddressOf** expression cannot be converted to **Integer** because **Integer** is not a delegate type. The error occurs in the following statement.

```
EnumFontFamilies(hDC, vbNullString, AddressOf EnumFontFamProc, 0)
```

To fix the error, you have to change the **Declare** declaration for **EnumFontFamilies** to accept a delegate type for the **EnumFontFamProc** parameter instead of an **Integer** type. The easiest way to create a delegate declaration for the **EnumFontFamProc** function is to perform the following steps:

1. Copy and paste the subroutine declaration in your code that will serve as the callback function.
2. Insert the **Delegate** keyword at the beginning of the declaration.
3. Change the function name by appending the word **Delegate** to it. The delegate name must be unique.

Using the preceding example, we can create the delegate declaration by copying and pasting the following function signature to the section of code containing the **Declare** statements.

```
Function EnumFontFamProc(ByRef lpNLF As LOGFONT, ByRef lpNTM As Integer, _
    ByVal FontType As Integer, ByRef LParam As Short) As Integer
```

Insert the keyword **Delegate** at the beginning of the function declaration and change the name by appending **Delegate** to the original function name; **EnumFontFamProc** becomes **EnumFontFamProcDelegate**. You end up with a delegate declaration for a function with the structure, name, and parameters of **EnumFontFamProc**, as shown here.

```
Delegate Function EnumFontFamProcDelegate(ByRef lpNLF As LOGFONT, _
         ByRef lpNTM As Integer, ByVal FontType As Integer, _
         ByRef LParam As Short) As Integer
```

Now that the delegate declaration for the callback function is in place, you have to change the parameter type for the **EnumFontFamilies Declare** statement from **Integer** to the delegate type **EnumFontFamProcDelegate**, as shown here.

```
Declare Function EnumFontFamilies Lib "gdi32" Alias "EnumFontFamiliesA" _
        (ByVal hDC As Integer, ByVal lpszFamily As String, _
         ByVal lpEnumFontFamProc As EnumFontFamProcDelegate, _
         ByRef LParam As Short) As Integer
```

With these changes, the code will compile and run without error, and it will produce the same results as the original Visual Basic 6.0 code.

Functions ObjPtr, StrPtr, and VarPtr Are No Longer Supported

In Visual Basic 6.0, it is possible to obtain the 32-bit address of an object, a string, or a variable or user-defined value through the undocumented helper functions **ObjPtr**, **StrPtr**, and **VarPtr**, respectively. Visual Basic .NET does not support these functions,

nor does it allow you to obtain the memory address for any type. At first, this may seem overly restrictive. However, the benefits of this restriction are significant. Leaving memory management to the CLR frees you from having to worry about allocating and freeing memory resources; this lets you concentrate on application and business logic. Also, this restriction increases application reliability and security. Managed application memory spaces are isolated from one another. No longer can an application accidentally (or maliciously) use pointers to access another application's memory space.

In cases where you must have control over underlying memory, the .NET Framework provides a pointer type named **System.IntPtr**. **System.IntPtr** is a platform-specific type that is used to represent a pointer or handle. This type can be used in conjunction with the **System.Runtime.InteropServices.Marshal** class, which contains methods for unmanaged memory operations, to obtain a pointer for a given type. For example, the functions **AllocHGlobal**, **GetComInterfaceForObject**, and **OffSetOf** all return pointers to memory. You can also use the **GCHandle** structure to obtain a handle to an element that is contained in garbage collector – managed memory. In addition, you can obtain a pointer to the memory by having the garbage collector pin the object to a single memory location to prevent it from being moved.

Consider the following Visual Basic 6.0 example module named "Module1." It calls **CopyMemory** to copy a source user structure to a destination structure by using **VarPtr** to obtain the memory addresses of the source and destination variables.

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
   (ByVal lpDest As Long, ByVal lpSource As Long, ByVal cbCopy As Long)

Type MyType
    valid As Boolean
    x As Long
    y As Long
End Type

Sub Test()
   Dim source As MyType
   Dim dest As MyType

   source.valid = True
   source.x = 10
   source.y = 10

   CopyMemory ByVal VarPtr(dest), _
              ByVal VarPtr(source), LenB(source)

   MsgBox "Valid = " & dest.valid & ": X = " & dest.x & " - Y = " & dest.y
End Sub
```

The following code is the output of the Visual Basic Upgrade Wizard.

```
Option Strict Off
Option Explicit On
Module Module1
    Declare Sub CopyMemory Lib "kernel32"  Alias "RtlMoveMemory"(ByVal lpDest As
Integer, ByVal lpSource As Integer, ByVal cbCopy As Integer)


    Structure MyType
        Dim valid As Boolean
        Dim x As Integer
        Dim y As Integer
    End Structure


    Sub Test()
        Dim source As MyType
        Dim dest As MyType

        source.valid = True
        source.x = 10
        source.y = 10

        ' UPGRADE_ISSUE: LenB function is not supported.
        ' UPGRADE_ISSUE: VarPtr function is not supported.
        CopyMemory(VarPtr(dest), VarPtr(source), LenB(source))

        MsgBox("Valid = " & dest.valid & ": X = " & dest.x & " - Y = " & _
                    dest.y)
    End Sub
End Module
```

► To perform a complete upgrade

1. Import the **InteropServices** namespace so that you can use the Marshal class to work with unmanaged memory.

```
Imports System.Runtime.InteropServices
```

2. Create and initialize the following variables.

```
Dim addressOfSource As IntPtr
Dim addressOfDest As IntPtr
Dim MyStrucSize As Integer

MyStrucSize = Marshal.SizeOf(GetType(MyType))
addressOfSource = IntPtr.Zero
addressOfDest = IntPtr.Zero

addressOfDest = Marshal.AllocHGlobal(MyStrucSize)
addressOfSource = Marshal.AllocHGlobal(MyStrucSize)
```

The **addressOfSource** and **addressOfDest** variables are used to store the addresses of two **MyType** instances. **MyStrucSize** variable is used to store the size of the **MyType** structure. **IntPtr.Zero** is a read-only field that represents a pointer or handle that has been initialized to zero. The **Marshal.AllocHGlobal** method allocates a block of memory from the unmanaged memory of the process by using **GlobalAlloc**.

3. Marshal the data from the source **MyType** instance to the unmanaged block of memory that is represented by the address variable.

```
Marshal.StructureToPtr(source, addressOfSource, True)
```

4. Copy the data from **source** to **dest** using unmanaged memory.

```
CopyMemory(addressOfDest.ToInt32, addressOfSource.ToInt32, MyStrucSize)
```

5. Marshal the data from the unmanaged block of memory **addressOfDest** to the managed object **dest**.

```
dest = Marshal.PtrToStructure(addressOfDest, GetType(MyType))
```

6. Free the allocated memory as shown in this code example.

```
Marshal.FreeHGlobal(addressOfSource)
Marshal.FreeHGlobal(addressOfDest)
```

Here is the equivalent Visual Basic .NET code for the Visual Basic 6.0 code.

```
Option Strict Off
Option Explicit On

Imports System.Runtime.InteropServices

Module Module1
    Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" (ByVal lpDest As
Integer, ByVal lpSource As Integer, ByVal cbCopy As Integer)

    Structure MyType
        Dim valid As Boolean
        Dim x As Integer
        Dim y As Integer
    End Structure

    Sub Test()
        Dim source As MyType
        Dim dest As MyType

        Dim addressOfSource As IntPtr
        Dim addressOfDest As IntPtr
        Dim MyStrucSize As Integer
```

```
        MyStrucSize = Marshal.SizeOf(GetType(MyType))
        addressOfSource = IntPtr.Zero
        addressOfDest = IntPtr.Zero

        source.valid = True
        source.x = 10
        source.y = 10

        addressOfDest = Marshal.AllocHGlobal(MyStrucSize)
        addressOfSource = Marshal.AllocHGlobal(MyStrucSize)

        Marshal.StructureToPtr(source, addressOfSource, True)

        CopyMemory(addressOfDest.ToInt32, addressOfSource.ToInt32, MyStrucSize)

        dest = Marshal.PtrToStructure(addressOfDest, GetType(MyType))

        MsgBox("Valid = " & dest.valid & ": X = " & dest.x & " - Y = " & dest.y)

        Marshal.FreeHGlobal(addressOfSource)
        Marshal.FreeHGlobal(addressOfDest)
    End Sub
End Module
```

You can use the same approach with other similar Visual Basic 6.0 functions, such as **ObjPtr** and **VarPtrArray**.

Note: If you choose to use unmanaged memory operations, be aware that you are responsible for manually allocating and freeing memory resources. For information about using managed memory operations, see the next section, "Moving API Calls to Visual Basic .NET."

Moving API Calls to Visual Basic .NET

Another option when upgrading API calls from Visual Basic 6.0 to Visual Basic .NET is to use Visual Basic .NET alternatives instead of the **Declare** statement. An advantage of this approach is that all the Visual Basic .NET alternatives result in fully managed code even when the APIs are unmanaged. Many of the APIs have equivalents in Visual Basic .NET or the .NET Framework.

The replacement of API functions by Visual Basic .NET alternatives should be done on a case-by-case basis. The following example illustrates how some API calls can be upgraded to equivalent Visual Basic .NET operations. This example uses the **FindFirstFile** and **FindNextFile** API functions to list the files of a specific directory.

The Visual Basic 6.0 **Declare** statements of these API functions are shown here.

```
Private Const MAX_PATH = 260
Private Const ERROR_NO_MORE_FILES = 18&
```

```
Private Type FILETIME
    dwLowDateTime As Long
    dwHighDateTime As Long
End Type

Private Type WIN32_FIND_DATA
    dwFileAttributes As Long
    ftCreationTime As FILETIME
    ftLastAccessTime As FILETIME
    ftLastWriteTime As FILETIME
    nFileSizeHigh As Long
    nFileSizeLow As Long
    dwReserved0 As Long
    dwReserved1 As Long
    cFileName As String * MAX_PATH
    cAlternate As String * 14
End Type

Private Declare Function FindFirstFile Lib "kernel32" _
    Alias "FindFirstFileA" (ByVal lpFileName As String, _
    lpFindFileData As WIN32_FIND_DATA) As Long
Private Declare Function FindNextFile Lib "kernel32" Alias _
    "FindNextFileA" (ByVal hFindFile As Long, _
    lpFindFileData As WIN32_FIND_DATA) As Long
```

The following code demonstrates how these functions might be used in a Visual Basic 6.0 application. The **ListFilesAndDirs** procedure receives a directory path as a parameter and lists all the directories and files contained in the path. Here is the Visual Basic 6.0 code for the procedure **ListFilesAndDirs**.

```
Public Sub ListFilesAndDirs(ByVal DirPath As String)
    Dim FirstFile As Long
    Dim NextFile As Long
    Dim FileData As WIN32_FIND_DATA
    Dim pos As Integer

    DirPath = DirPath & "*.*"
    FirstFile = FindFirstFile(DirPath, FileData)
    If FirstFile <> -1 Then
        NextFile = FindNextFile(FirstFile, FileData)
        While (NextFile <> ERROR_NO_MORE_FILES) And (NextFile <> 0)
            pos = InStr(FileData.cFileName, vbNullChar)
            If pos = 0 Then
                Debug.Print FileData.cFileName
            Else
                If pos > 1 Then
                    Debug.Print Left$(FileData.cFileName, pos - 1)
                End If
            End If
            NextFile = FindNextFile(FirstFile, FileData)
            Wend
    End If
End Sub
```

After the Visual Basic 6.0 code is converted by the upgrade wizard, you should perform several steps to eliminate the dependencies on the API calls and to replace them with Visual Basic .NET equivalents. The first step removes all the **Declare** statements and structures because you are not using API calls.

The second step is to substitute the call to the **FindFirstFile** and **FindNextFile** API functions with the .NET Framework functions **System.IO.Directory.GetFiles** and **System.IO.Directory.GetDirectories**. Because the Visual Basic .NET functions return a string array, the new Visual Basic .NET function is the following.

```
Dim List As String() = System.IO.Directory.GetFiles(DirPath, "*.*")
Dim ListofDirs As String() = _
    System.IO.Directory.GetDirectories(DirPath, "*.*")
pos = List.Length
ReDim Preserve List(List.Length + ListofDirs.Length - 2)
ListofDirs.CopyTo(List, pos - 1)
```

Notice that the Visual Basic .NET functions receive the **Path** and the **Search** pattern parameters. As a result, the following Visual Basic 6.0 line should be removed from the code.

```
DirPath = DirPath & "*.*"
```

And because the Visual Basic .NET functions return an array of strings, you can replace the first **If** and **While** statements with a **For** statement, as shown here.

```
For i = 0 To list.Length - 1
```

As a final step, you should change the code that retrieves and prints the file name, because you are not using structures. Here is the Visual Basic 6.0 code.

```
FileName = list(i)
pos = FileName.LastIndexOf("\\")
If pos > 1 Then
    FileName = FileName.Substring(pos + 2, FileName.Length-pos-2)
End If
System.Diagnostics.Debug.WriteLine(FileName)
```

The equivalent Visual Basic .NET code is shown here.

```
Public Function ListFiles(ByVal DirPath As String) As String()
    Dim pos As Short
    Dim i As Integer
    Dim FileName As String

    Dim List As String() = System.IO.Directory.GetFiles(DirPath, "*.*")
    Dim ListofDirs As String() = _
        System.IO.Directory.GetDirectories(DirPath, "*.*")
    pos = List.Length
    ReDim Preserve List(List.Length + ListofDirs.Length - 2)
```

```
    ListofDirs.CopyTo(List, pos - 1)

    For i = 0 To List.Length - 1
        FileName = List(i)
        pos = FileName.LastIndexOf("\\")
        If pos > 1 Then
            FileName = FileName.Substring(pos + 2, FileName.Length-pos-2)
        End If
        System.Diagnostics.Debug.WriteLine(FileName)
    Next
End Function
```

The preceding example shows only a fraction of the possible API replacements that you can use to remove dependence on the Windows API. For a list of the Microsoft .NET Framework version 1.0 or 1.1 APIs that provide similar functionality to Win32 functions, see "Microsoft Win32 to Microsoft .NET Framework API Map" on MSDN.

Summary

This chapter demonstrates how you can upgrade Visual Basic 6.0 applications that are built on the Windows API to Visual Basic .NET, using two different methods.

The first method involves the continued use of the Windows API from within Visual Basic .NET code. When you apply this approach, you must take care to appropriately update data types and apply marshaling attributes to ensure the proper operation of API calls.

The second method is to replace Windows API calls with equivalent Visual Basic .NET function calls. This approach requires that you apply more effort during the upgrade process than with the first method, but it reduces an application's dependence on an older, unmanaged API.

Using either method, you will be able to achieve the functional equivalence in Visual Basic .NET for Visual Basic 6.0 applications built on the Windows API.

More Information

For more information about preparing your application before you begin the upgrade process, see "Preparing Your Visual Basic 6.0 Applications for the Upgrade to Visual Basic .NET" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvb600/html/vb6tovbdotnet.asp.

For a list of the Microsoft .NET Framework version 1.0 or 1.1 APIs that provide functionality that is similar to Win32 functions, see "Microsoft Win32 to Microsoft .NET Framework API Map" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/win32map.asp.

14

Interop Between Visual Basic 6.0 and Visual Basic .NET

In Chapter 1, "Introduction," you were introduced to the options available when you decide to upgrade an application to Microsoft Visual Basic .NET. The challenges and risks of each approach were outlined, and guidelines were provided to help you choose the upgrade strategy that would be best for your particular application. As discussed in that chapter, you can mitigate risk and reduce the magnitude of upgrade challenges by approaching the upgrade process incrementally, or by performing a partial upgrade. With these types of upgrades, Visual Basic 6.0 modules must interoperate with Microsoft .NET Framework assemblies. It may also be necessary for Visual Basic .NET modules to interoperate with Visual Basic 6.0 modules. There are many differences between the traditional Visual Basic 6.0 execution environment and the managed execution environment used by Visual Basic .NET. These differences make it impossible to achieve direct invocation of .NET assemblies from Visual Basic 6.0 code modules. Therefore, a mechanism must be used to allow .NET assemblies to continue to execute within the common language runtime (CLR)-managed context, yet still be accessible to Visual Basic 6.0 code modules.

Be sure to always perform an adequate functional analysis of your application prior to an upgrade. This will enable you to identify the functional components of the application, and to build a more effective plan for the overall upgrade process. This also helps you improve your application by identifying redundant or unused code modules that should be eliminated to reduce the amount of work that will be required for the upgrade. After you identify the components that must be upgraded to Visual Basic .NET and those that can remain in Visual Basic 6.0 (either temporarily or permanently), you can begin to create an effective interoperability between components.

This chapter provides information about how to achieve interoperability between Visual Basic 6.0 code modules and .NET assemblies. Interoperability can be achieved

for assemblies written in any .NET language, but this chapter focuses only on Visual Basic .NET.

Note: The Visual Basic 6.0 code samples in this chapter require that the Microsoft Service Pack 6 for Visual Basic 6.0 is installed and functioning properly. You can download this service pack from MSDN.

Calling .NET Assemblies from Visual Basic 6.0 Clients

The .NET Framework was designed from the ground up to interoperate with COM. COM components can call .NET assemblies, and vice versa. Furthermore, .NET assemblies can be built to be referenced just like COM components are referenced. To do this, you build your assembly so that it is COM-callable, and your code will be able to reference the assembly's functionality and interfaces.

COM-callable assemblies can be referenced from Visual Basic 6.0 applications as if they were ordinary COM components. In this case, interoperability may be as simple as referencing the correct assembly's type library (.tlb file) and accessing the desired functionality. However, there is a drawback to using this approach; it will be covered later in this chapter.

Calling Visual Basic 6.0 Libraries from Visual Basic .NET Clients

As already mentioned, Visual Basic .NET applications are able to communicate with COM components. This ability includes COM components created with Visual Basic 6.0.

To access Visual Basic 6.0 code from Visual Basic .NET clients, you must expose the Visual Basic 6.0 code as a COM component, and then reference it as such from the Visual Basic .NET client.

How to Achieve Interoperability

Achieving interoperability between Visual Basic 6.0 and Visual Basic .NET is not a difficult process. There are two approaches you can take.

The first approach is to use direct access through COM. This approach requires that the .NET assembly to be referenced is created as a COM-callable assembly.

The second approach is an extension of the first. Instead of creating a COM-callable assembly, you create a COM-callable wrapper using a .NET language for the functionality you want to access. With this approach, there is no special build requirement for the assemblies you want to access, but the wrapper itself must be built as COM-callable.

There are some requirements that must be fulfilled to achieve interoperability. These will be presented next, followed by a discussion of the different approaches to achieving interop mentioned in the preceding paragraphs.

Access Requirements

Every .NET component that is going to be accessed from Visual Basic 6.0 must be registered for interoperability in the system. Registration takes place at a global level and will make the component available to all Visual Basic 6.0 applications. An assembly is the primary building block of a .NET application and it can contain different classes that are available to COM clients. Registering an assembly will make the classes in a registered assembly available to all Visual Basic 6.0 applications on the system. For more information on native dynamic link libraries and assemblies see Chapter 4, "Common Application Types."

The process to register a .NET assembly and make it COM-callable involves creating a type library (.tlb) and including the corresponding entries in the system registry. Fortunately, this is easily accomplished from within the Visual Studio .NET IDE by setting the appropriate build configuration option.

► **To register a component for COM interop**

1. Open the Visual Basic .NET project that contains the classes you want to make COM-callable.
2. From the **Project** menu, choose **Properties**.
3. Under the Configuration Properties folder, click **Build**.
4. Select the **Register for COM Interop** check box, and then click **OK**.
5. On the **File** menu, click **Save** to save your project with the configuration changes.
6. On the **Build** menu, choose **Build Solution**.

Following the steps of this procedure will create the Visual Basic .NET DLL and register it for COM interoperability. The assembly's **AssemblyName** property value will be the name with which it is registered under COM. This property can be modified by performing the following procedure.

► **To change they AssemblyName property of an assembly**

1. Right-click the project name in Solution Explorer, and then click **Properties**.
2. In the Common Properties folder, click **General**.
3. In the **Assembly name** text box, specify the name for the assembly.
4. Click **OK** to save your changes.

An assembly can also be registered at a Windows command line by using the Assembly Registration tool (Regasm.exe) distributed with the .NET Framework. This would typically only be necessary during deployment, because the steps outlined

above can be used to register an assembly during the build. The command for registering an assembly with the Regasm.exe utility is as follows:

regasm *Assembly.dll* **/tlb:** *Assembly.tlb*

This command will register an assembly and generate a type library so that it appears as a regular COM object. Items in italics should be replaced with the appropriate names for your project. The **/tlb** option is used to specify that a type library file should be generated.

Note: Be careful of naming conflicts between assemblies. Be sure to unregister any older versions of the same assembly before registering the new one.

Should it be necessary to unregister the assembly, this can be achieved by using the following **regasm** command:

regasm /u *Assembly.dll*

For the Visual Basic 6.0 application to find the .NET assembly at run time, the assembly must be in either the global assembly cache, or in the application's executable folder. If you wish to install the assembly to another location, and still have it globally accessible to Visual Basic 6.0 applications, you can use the **/codebase** option with the Regasm.exe utility. This option allows the user to specify a file path that will be used by the common language runtime to locate the corresponding assembly. However, the details of the **/codebase** option are beyond the scope of this guide, and it is recommended that COM-callable assemblies be placed in the global assembly cache when applications are deployed.

Requirements for Interoperability with COM

For a .NET managed type (such as a class, interface, structure, or enumeration) to be exposed to COM clients, it must meet the following requirements:

- **Managed types must be public**. Only public types in an assembly are registered and exported to the type library. As a result, only public types are visible to COM.

  Managed types expose features to other managed code that might not be exposed to COM. For instance, parameterized constructors, static methods, and constant fields are not exposed to COM clients. Also, be aware that interop marshaling can introduce behavior differences.

- **Methods, properties, fields, and events must be public**. Members of public types must also be public if they are to be visible to COM. You can restrict the visibility of an assembly, a public type, or public members of a public type by applying the **ComVisibleAttribute**. By default, all public types and members are visible.

- **Types must have a public default constructor to be activated from COM**. Managed public types are visible to COM. However, without a public default

constructor (a constructor with no arguments), COM clients cannot create the type. COM clients can still use the type if it is activated by some other means, for example if it is indirectly created by other class methods.

- **Types cannot be abstract**. Neither COM clients nor .NET clients can create abstract types.
- **Certain Visual Basic .NET features will prevent a method from being COM callable**. Shared (static) Visual Basic .NET methods cannot be called from COM objects. Also, methods should take and return simple data types such as **Integer** or **String**. If a method takes or returns a type that was not designed for COM interop, the method will not be callable from Visual Basic 6.0. Visual Basic .NET methods should also not be overloaded. COM does not support method overloading, and as a result, each overload will appear in Visual Basic 6.0 with an automatically-generated name different from the original name. It is better to give the methods unique names when building the .NET wrapper class.

Also, note that when exported to COM, the inheritance hierarchy of a managed type is flattened; as a result, derived classes will expose their base class members as if they were declared in the derived class.

Accessing .NET Assemblies Directly from Visual Basic 6.0

The process for calling .NET assemblies from Visual Basic 6.0 is fairly straightforward. The general process is as follows:

1. Determine the assembly you want to access.
2. Locate the assembly's type library.

> **Note:** The assembly must be registered using the Assembly Registration utility (regasm.exe) to be accessed from COM clients. For more information about registering assemblies, see the "Access Requirements" section earlier in this chapter.

3. In your Visual Basic 6.0 application, add the reference to the assembly's desired type library.
4. Invoke the required functionality from the assembly.

A drawback to this approach is that you might have access to only a limited percentage of the Visual Basic .NET and .NET Framework functionality. The reasons for this are as follows:

- Not all .NET assemblies are COM-callable. The **ComVisible** attribute controls the accessibility of the members of an assembly to COM clients. If this attribute is set to **False** for an assembly, all public types within the assembly will be hidden, making them inaccessible to COM clients. The **ComVisible** attribute can be set for assemblies, interfaces, classes, and contained members in an individual way.

- There are Visual Basic 6.0 limitations to the object-oriented programming paradigm used by Visual Basic .NET. For example, some .NET classes may be abstract and require subclassing to use, and this is not possible in Visual Basic 6.0.
- There is limited Visual Basic 6.0 support of .NET exceptions. The .NET assemblies can be built to use exceptions as the mechanism to signal error conditions, but Visual Basic 6.0 does not support them as structured object-oriented exceptions. This means that to detect error conditions after a call to the .NET component, it is necessary to specifically check for the occurrence of an exception immediately after the call or to use **On Error Goto** statements, just as with any Visual Basic 6.0 component. This diminishes the advantages obtained from .NET error management mechanisms.

Despite this drawback, this approach might still be the best alternative to a complete upgrade that would be overly complex and costly.

The process for accessing .NET assemblies from Visual Basic 6.0 is detailed in the following procedure. In this example, the project will reference the .NET System type library that contains the fundamental and base classes used by .NET code. This example assumes that Visual Basic 6.0 and Visual Studio .NET (or a version of the .NET Framework) are installed on the same system.

► **To access a .NET assembly from a Visual Basic 6.0 application**

1. Register the assembly to be accessed. This allows Visual Basic 6.0 to instantiate and access the .NET assembly. Use the following steps to register an assembly:
   a. In Windows, open a command prompt. By default, the shortcut for the command prompt is in the **Accessories** category of the **Programs** or **All Programs** (depending on your version of Windows) submenu of the **Start** menu.
   b. Change the current directory to the .NET Framework directory where the System.dll file is located. To do this, enter the following at the command prompt: **cd *DriveLetter*:\Windows\Microsoft.NET\Framework\v1.1.4322**.

      Note: *DriveLetter* represents the drive that the .NET Framework is installed on, and the name v1.1.4322 corresponds to the .NET Framework version that is installed. The actual name may be different on your system. If you are not sure where your installation resides, search the file system for the Microsoft.NET\Framework folder.

   c. Register the System.dll file by entering the following command at the command prompt: **regasm System.dll**.
2. Start Visual Basic 6.0.
3. In the **New Project** dialog box, click **Standard.exe** to create the new project.
4. On the **Project** menu, click **References**. This displays the **References** dialog box where you can select the COM components and the registered COM-callable .NET assemblies that your application will reference.

5. In the **References** dialog box, click **Browse**.
6. In the **Add Reference** dialog box, navigate to the directory where the System.tlb file is located. This should be the same directory that you identified in step 1.b.
7. Click the .tlb file for the assembly you want to reference, and then click **Open**. In this example, you would click **System.tlb**.
8. In the **Available References** list, select the check box for the Visual Basic .NET library you want to reference, and then click **OK**. For this example, select the **System.dll** check box.

After completing these steps, you can reference any interface and public class contained within the assembly from your Visual Basic 6.0 project. For example, by referencing the System type library, you can create an instance of the **System.WebClient** class. You can then call any **System.WebClient** methods through that object, such as the **downloadFile** method, as shown in the following sample program.

```
Dim downloader As System.WebClient
Dim targetFile As String
Dim localFile As String

targetFile = _
    "http://www.msdn.com/library/toolbar/3.0/images/banners/" & _
    "msdn_masthead_ltr.gif"
Set downloader = New System.WebClient
downloader.downloadFile targetFile, App.Path & "\msdnLogo.gif"
```

Note: To run this sample program inside the Visual Basic 6.0 development environment, the Visual Studio 6.0 Service Pack 6 must be installed on the system. If the service pack is not installed and the sample program is executed within the Visual Basic 6.0 IDE, an automation runtime error will be generated. In this case, the sample program can only successfully be run outside the development environment by running the application's .exe file. You can download this service pack from MSDN.

Invoking .NET functionality from your Visual Basic 6.0 applications is fairly straightforward. If the assemblies you want to reference are COM-callable, accessing their functionality from Visual Basic 6.0 is greatly simplified.

Despite the ease of referencing .NET assemblies in Visual Basic 6.0 projects, there is a price to be paid in terms of application performance. The COM interoperability that takes place between Visual Basic 6.0 and Visual Basic .NET requires additional steps that are not executed in components that do not interoperate. These steps include data marshaling, calling convention adjustment, register protection, thread handling, and exception-handling frame management. Because of this, the overhead of a Visual Basic 6.0 to Visual Basic .NET call is higher than the overhead for a direct Visual Basic 6.0 to Visual Basic 6.0 call. You may want to minimize the number and

frequency of calls that your application makes; this reduces the impact of this additional overhead. For more information about performance improvement, see Chapter 7, "Improving Interop Performance," of *Improving .NET Application Performance and Scalability* on MSDN.

Creating Interoperability Wrappers in .NET

Directly accessing the .NET Framework Class Library (FCL) may limit what you can do in terms of functionality. Recall that Visual Basic 6.0 is not object-oriented: therefore you might not be able to customize the behavior of the classes that you import through this method. This is especially true if the class you want to utilize is an abstract class, and you are required to subclass it to be able to use it. In addition, not all .NET assemblies are built to be COM-callable. This will limit the classes you can access directly from your Visual Basic 6.0 code without additional steps. In such cases, there is an alternate approach that you can use: create a COM-callable wrapper object in Visual Basic .NET, and reference the wrapper object from Visual Basic 6.0. Creating a wrapper involves several steps, including creating a type library (.tlb) file for the wrapper class and registering the class for COM interoperability.

You can easily create a Visual Basic .NET wrapper for the **System.Text.StringBuilder** class to perform the necessary string handling operations for you. A wrapper will also minimize the parameters that you must enter from Visual Basic 6.0 to successfully use the .NET functionality. The following code is an example of a Visual Basic .NET wrapper class. (The name of the file where this class is defined should be named *wrappers* to be consistent with the Visual Basic 6.0 code that will be presented after this code example.)

```
Imports System.text

<ComClass(StringBuilderWrapper.ClassId, _
    StringBuilderWrapper.InterfaceId, StringBuilderWrapper.EventsId)> _
Public Class StringBuilderWrapper

#Region "COM GUIDs"
    Public Const ClassId As String = "963BBA03-8FA9-45F3-94FC-2E68D4940515"
    Public Const InterfaceId As String = "E7B75972-6A52-4E7D-96E1-9E7DBBBACBC7"
    Public Const EventsId As String = "48760051-DD7F-4EA1-B612-9CD7F2F1BAD3"
#End Region
    Private stringb As StringBuilder
    Public Sub New()
        stringb = New StringBuilder
    End Sub
    Public Sub SetValue(ByVal s As String)
        stringb = New StringBuilder(s)
    End Sub
    Public Sub Append(ByVal s As String)
        stringb.Append(s)
    End Sub
    Public Overrides Function ToString() As String
```

```
        Return stringb.ToString()
    End Function
End Class
```

After the class is compiled and registered for COM interop, you can import the corresponding .tlb file into your Visual Basic 6.0 project references. The following Visual Basic 6.0 code example shows how the wrapper class functionality can be accessed. Notice that only a very limited functionality of the **StringBuilder** class is supported by the wrapper, it offers just what is needed by the Visual Basic 6.0 client.

```
Private Sub TestStringOperations()
    Dim s As New wrappers.StringBuilderWrapper
    s.SetValue "This is "
    s.Append "a test"
    MsgBox s.ToString
End Sub
```

To make a Visual Basic .NET assembly COM-callable, it must be registered for COM interop. This process involves creating a .tlb file and including the corresponding entries in the system registry. Fortunately, this is easily accomplished from within the Visual Studio .NET IDE by setting the appropriate build configuration option. The following procedure details the steps for registering a component for COM interop.

► **To register a Visual Basic .NET component for COM interop**

1. Open the project for your Visual Basic .NET wrapper class.
2. On the **Project** menu, click **Properties**.
3. In the **Configuration Properties** folder, click **Build**.
4. Select the **Register for COM Interop** check box, and then click **OK**.
5. On the **File** menu, click **Save** to save your project with the configuration changes.
6. On the **Build** menu, choose **Build Solution**.

Using this procedure will create the Visual Basic .NET DLL and register it for COM interoperability. The assembly's **AssemblyName** property value will be the name that it is registered with under COM. This property can be modified by accessing the corresponding project properties (by right-clicking the project name in Solution Explorer) and then clicking **Properties**. In the **Common Properties** folder, click **General**.

Note: You must use care to avoid creating name conflicts when registering wrapper classes. Be sure that each assembly's name is unique, and unregister any earlier versions of the same assembly before registering the new one.

Creating Visual Basic .NET wrappers for use by Visual Basic 6.0 is probably your best option if you want to access a Visual Basic .NET assembly from Visual Basic 6.0 without limiting yourself to those that are built as COM-callable. Using wrappers gives you access to the full range of available .NET assemblies through Visual Basic .NET. Using wrappers also allows you to better modularize your code by avoiding granular invocations of the .NET assemblies where they're not necessary. You can also mask FCL functionality by using your own classes, and even replace the implementation of the wrappers while keeping the interfaces the same. This allows you to evolve your wrapper code without worrying about compatibility with the client code.

As with any interoperability procedure, there are caveats to be aware of when developing .NET wrapper classes. Aside from the drawback listed in the previous section (you may have limited access to the functionality in the assembly), there are a few additional issues to be mindful of:

- Additional effort may be required to implement wrappers if your code is not well modularized.
- Refactoring code can be time-consuming, though potentially well worth the investment.
- Wrapper design and planning becomes critical. You do not want to wrap around too much or too little functionality. Both cases will increase your risk and the amount of work required to create a functional wrapper.

Command Line Registration

As previously stated, you must register your wrapper component within the system after you create it so that it is globally accessible to all of your Visual Basic 6.0 applications. The process for registering a component from within the Visual Studio .NET IDE was shown in the previous section. However, an assembly can also be registered at a Windows command line by using the Assembly Registration tool (regasm.exe) that is distributed with the .NET Framework. The procedure for this approach is outlined here.

In this example, artificial module names are used for illustrative purposes only. Items in italics should be replaced with the appropriate names for your project.

► **To register a .NET assembly using command-line tools**

1. Compile the wrapper class using the Visual Basic .NET compile command vbc.exe.

```
vbc assemblyinfo.vb sourcecode.vb /r:System.dll
   /target:library /out:wrappers.dll
```

The **/r** option specifies that the compilation should reference the System.dll. The **/target** option specifies that a library (.dll) should be generated. The **/out** option specifies the name of the output file.

2. Register the assembly and generate a type library so that it appears as a regular COM object.

```
regasm wrappers.dll /tlb:wrappers.tlb
```

The **/tlb** option is used to specify that a type library file should be generated.

Note: You must use care to avoid creating name conflicts when registering assemblies. Be sure that each assembly's name is unique, and unregister any earlier versions of the same assembly before registering the new one.

If it is necessary to remove the .tlb file information that corresponds to an assembly from the system registry, you can do so by using the following **regasm** command.

```
regasm /u wrappers.dll
```

Data Type Marshaling

Marshaling is the process of packing and unpacking parameters and return values in a way that enables the execution of a cross platform invocation. As mentioned earlier, COM is the key to achieving interoperability between Visual Basic 6.0 and Visual Basic .NET. It is because of the presence of COM that data type marshaling becomes a non-issue in cases where components in different thread apartments interact. The .NET interoperability marshaling enables the interaction between different data types in managed and unmanaged memory. Interop marshaling is performed at run time by the CLR's marshaling service when functions are invoked within the same COM apartment. When the interaction takes place between managed code and unmanaged code in a different COM apartment or a different process, both the interop marshaler and the COM marshaler are executed.

Most Visual Basic data types have similar representations in both managed and unmanaged memory. The interop marshaler automatically handles these types. Other types can have significant differences when upgraded to Visual Basic .NET. An ambiguous data type can have different unmanaged representations that map to a single managed type, or unavailable type information, such as the size of a string or an array.

Custom Marshaling

Assumptions or dependences about the internal characteristics of a data type can lead to the need for custom marshaling. Characteristics that are commonly taken for granted are internal representation, element size, and element position. When upgrading a part of a Visual Basic 6.0 application that depends on a piece of code that has special assumptions about a data type, it may be necessary to make the two pieces of code interact through .NET interoperability. In this type of scenario, the part of the application that was upgraded to Visual Basic .NET needs to stay compatible with the component that is not upgraded. This will require the use of user-defined marshaling for some of the data types involved in the interaction between both components.

The recommended approach to do the necessary data conversions in this kind of situation is to write a wrapper class that includes all the code to perform these conversions. This class, called a marshaling wrapper class, will provide a bridge between an old and a new interface. It allows clients that expect a particular interface to work with components that implement a different interface.

In most cases, a marshaling wrapper class will be enough to handle all interaction and compatibility issues. In the cases where a declarative, low level technique is necessary to handle marshaling you can use custom marshalers.

The **MarshalAsAttribute** attribute can be used to apply a custom marshaler to a parameter or function return value. The custom marshaler is identified by this attribute and implements the **ICustomMarshaler** interface to provide the appropriate wrappers to the .NET CLR. It will call the **MarshalNativeToManaged** and **MarshalManagedToNative** methods on the custom marshaler to activate the correct wrapper for handling calls between interoperating components. For more information, see "Custom Marshaling" in the *.NET Framework Developer's Guide* on MSDN.

The following example demonstrates when the custom interop marshaling must be used to stay compatible with a component that is not upgraded to Visual Basic 6.0. The original application has a COM class named **Converter** that defines the function **ConvertCurrency** that receives and returns values with the **Currency** Visual Basic 6.0 data type. This function is used by the rest of the application to perform financial calculations. The original declaration of **ConvertCurrency** is shown here.

```
Public Function ConvertCurrency(amount As Currency, curFrom As String, _
                curTo As String) As Currency
    ConvertCurrency = amount * GetConversionFactor(curFrom, curTo)
End Function
```

The **Converter** class is used in the client code as shown here.

```
Private Sub Form_Load()
    Dim converter As New Utils.Converter
```

```
    Dim res As Currency
    Dim amount As Currency
    amount = GetInitialAmount()
    res = converter.ConvertCurrency(amount, "DOL", "EUR")
End Sub
```

The following class is based on the result obtained from an automatic upgrade of the **Converter** class to Visual Basic .NET.

```
<System.Runtime.InteropServices.ProgId("Converter_NET.Converter"), _
ClassInterface(ClassInterfaceType.AutoDual)> Public Class Converter
    Public Function ConvertCurrency(ByRef amount As Decimal, _
                    ByRef curFrom As String, ByRef curTo As String) As Decimal
        ConvertCurrency = amount * _
                          GetConversionFactor(curFrom, curTo)
    End Function
End Class
```

Notice that the class attribute **ClassInterface(ClassInterfaceType.AutoDual)** has been manually included to make the class member declarations available to Visual Basic 6.0. The resulting project is also automatically registered for COM interop. Note that in order to expose this class to COM clients, the **ComClassAttribute** attribute may also have been used; the preceding example uses the **ClassInterfaceType.AutoDual** attribute only to simplify the example.

Assume that the application's upgrade plan determined that the new **Converter** class will be accessed by unmanaged COM clients, as shown in the previous **Form_Load** example. According to this plan, the references to the original **Converter** class must be replaced by the new managed one. However, when the resulting application is compiled the following error message is shown for the invocation of **ConvertCurrency**: "Compile error: Function or interface marked as restricted, or the function uses an Automation type not supported in Visual Basic."

The problem is that the COM client identifies the **Decimal** data type as an unsupported **Variant** type. The solution to this problem requires the use of custom marshaling in the definition of the Visual Basic .NET component, as shown here.

```
<System.Runtime.InteropServices.ProgId("Converter_NET.Converter"), _
ClassInterface(ClassInterfaceType.AutoDual)> Public Class Converter
    Public Function ConvertCurrency( _
                    <MarshalAs(UnmanagedType.Currency)> ByRef amount As Decimal _
                    ByRef curFrom As String, ByRef curTo As String) _
                    As <MarshalAs(UnmanagedType.Currency)> Decimal
        ConvertCurrency = amount * _
                          GetConversionFactor(curFrom, curTo)
    End Function
End Class
```

The **MarshalAsAttribute** attribute indicates that one of the function parameters and the function return value must be marshaled as a COM currency data type instead of **Decimal**. With this adjustment, the component can still be used from Visual Basic 6.0 clients.

Error Management

As mentioned previously, .NET uses a structured exception handling approach for indicating, and reacting to, both anticipated and unanticipated error conditions. This is not the case in Visual Basic 6.0. (Although the **OnError** handler provides a process that can be considered similar to exception handling, it does not behave the same way.)

Because interoperability between Visual Basic .NET and Visual Basic 6.0 is achieved using COM-based mechanisms, there is an inherent value, named **HRESULT**, which is passed back and forth between the Visual Basic 6.0 code and the .NET assembly. Even though this happens behind the scenes, it is still there. Examining the value of **HRESULT** for the existence of an error condition is done by the CLR itself — thus, the value is never directly exposed to the client code in either side. In Visual Basic 6.0, this value can be accessed through the **Err.Number** property. In Visual Basic .NET, this value is treated differently depending on whether or not the original exception is a standard exception. When it is a standard exception, Visual Basic .NET clients will receive a standard .NET exception. If it is a custom exception, the Visual Basic .NET client will receive a generic exception with the **ErrorCode** property set in accordance with the **HRESULT** value.

Catching .NET Exceptions in Visual Basic 6.0

When **HRESULT** signals that an exception has been raised by the .NET assembly, the **On Error** clause is invoked in Visual Basic 6.0 if it has been declared and defined. The actual error condition can be examined by using the Visual Basic 6.0 intrinsic **Err** object and by inspecting its **Description** and **Number** properties.

Note: There is a well-defined mapping of common .NET exceptions to Visual Basic 6.0 error codes that can be used to determine the type of system error encountered. For more information about the particulars of the mapping mechanism, see "HRESULTs and Exceptions" on MSDN.

For example, assume you want to use the following assembly that performs a mathematical division between its two arguments.

Visual Basic .NET component

```
Imports System.IO
Imports System.Runtime.InteropServices

<ComClass(DivisorTool.ClassId, DivisorTool.InterfaceId, DivisorTool.EventsId)> _
```

```
Public Class DivisorTool

#Region "COM GUIDs"
    Public Const ClassId As String = "D45B5767-62C9-4871-90DA-EFB56E2F8860"
    Public Const InterfaceId As String = "AA43EC7A-54B0-4E82-BC54-00087E27645C"
    Public Const EventsId As String = "9C4D24A8-F325-409F-8A00-62DE8B938F1D"
#End Region

    Public Sub New()
        MyBase.New()
    End Sub
    Public Function Divide(ByVal term As Double, _
                    ByVal divisor As Double) As Double
        Dim res As Decimal
        res = (term / divisor)
        Return res
    End Function
End Class
```

After the assembly is built, registered, and referenced within your Visual Basic 6.0 project, you can access the exception information using Visual Basic 6.0 code similar to the example shown here.

Visual Basic 6.0 client

```
Private Function Perform_Division() As Double
    On Error GoTo Error_Handler
    Dim div As New DivisorTool
    Perform_Division = div.Divide(10, 0)
    Exit Function

Error_Handler:
    MsgBox "Error : " & Err.Description, _
           vbOKOnly, "(code = " & CStr(Err.Number) & ")"
End Function
```

If the **Divisor.Divide()** function was called with a divisor value of 0, a **System.OverflowException** would be raised from the .NET component, and propagated to the Visual Basic 6.0 client code. You could then check for it as you would normally check for an overflow error condition within Visual Basic 6.0. The internal exception-to-error-code mapping mechanism ensures that the error will remain consistent from one code tier to the other, because this is a well-known system error.

Application-defined exceptions are slightly different. To handle application errors that are raised in the form of .NET exceptions in a granular fashion, you *must* inspect the **Number** property of the Visual Basic 6.0 **Err** object, and to select an execution path based on its value. This technique can also be applied to standard exceptions like overflow or divide-by-zero exceptions and will be explained in the forthcoming examples.

The previous example is repeated here to illustrate how different error conditions can be detected using simple code modifications. Imagine that for specific business logic reasons, all divisors are acceptable except the value 3, and all terms are acceptable, except the value 3. Thus, the routine is coded to raise different exceptions if the divisor is 3 or the term is 3. The following code example demonstrates this.

Visual Basic .NET component

```
Imports System.IO
Imports System.Runtime.InteropServices

<ComClass(DivisorTool.ClassId, DivisorTool.InterfaceId, DivisorTool.EventsId)> _
Public Class DivisorTool

#Region "COM GUIDs"
    Public Const ClassId As String = "D45B5767-62C9-4871-90DA-EFB56E2F8860"
    Public Const InterfaceId As String = "AA43EC7A-54B0-4E82-BC54-00087E27645C"
    Public Const EventsId As String = "9C4D24A8-F325-409F-8A00-62DE8B938F1D"
#End Region

    Public Sub New()
        MyBase.New()
    End Sub
    Public Function Divide(ByVal term As Double, _
                    ByVal divisor As Double) As Double
        If divisor = 3 Then
            Dim e As System.Runtime.InteropServices.COMException = _
            New System.Runtime.InteropServices.COMException( _
                "Bad Divisor", &H80000001)
            Throw e
        ElseIf term = 3 Then
            Dim e As System.Runtime.InteropServices.COMException = _
            New System.Runtime.InteropServices.COMException( _
                "Bad Term", &H80000002)
            Throw e
        Else
            Return (term / divisor)
        End If
    End Function
End Class
```

The hex values &H80000001 and &H80000002 correspond to the **HRESULT** error codes that will be received in Visual Basic 6.0 as exceptions. The lower 16 bits of these values correspond to the specific error code that was detected; in this case, the error codes are 1 and 2.

Note: For a description of the fields contained in a **HRESULT** value, see "HRESULT" on MSDN.

If a Visual Basic 6.0 client application referenced this class and invoked the **Divide** function, the error handler would have to determine the correct error that was raised to decide how to handle it. This is demonstrated in the following code example.

Visual Basic 6.0 client

```
Private Function Perform_Division() As Double
    On Error GoTo Error_Handler
    Dim div As New DivisorTool
    Perform_Division = div.Divide(10, 0)
Exit Function

Error_Handler:
    If (Err.Number And &H7FFF) = 1 Then
        ' Handle the "bad divisor" error
    ElseIf (Err.Number And &H7FFF) = 2 Then
        ' Handle the "bad term" error
    Else
        ' There is another, unknown error…
        MsgBox "Error : " & Err.Description, _
            vbOKOnly, "(code = " & CStr(Err.Number) & ")"
    End If
End Function
```

It is necessary to extract the original error code by doing a bitwise **And** operation between **Err.Number** and the hex value &H7FFF. The .NET interoperability turns on the first bit of the error code that is going to be sent to the COM client, and the **And** operation turns off this bit to obtain the original error code.

As demonstrated in the previous code example, this approach can increase the amount of code needed for interoperability, and the complexity of the overall code. Error handlers will likely grow in complexity, and it may become difficult to debug exactly which error conditions are producing which unexpected results.

There is additional risk if the documentation for the .NET assembly does not indicate which error codes (such as **Err.Number** values) correspond to which error conditions. This can complicate effective handling of some exceptions raised from the .NET component in your Visual Basic 6.0 client code.

Catching OnError Conditions Raised from Visual Basic 6.0 in Visual Basic .NET

The converse case must also be analyzed: how would a .NET assembly detect and react to an error condition raised from a Visual Basic 6.0 code module? COM is also involved, so there are some advantages in terms of handling common system errors.

For common system errors, the marshaling layer automatically translates these error conditions to appropriate .NET exceptions. This means that you can call a Visual Basic 6.0 subroutine from .NET, and use **try/catch/finally** blocks to detect normal system errors raised by the code module.

For example, consider a Visual Basic 6.0 version of the **DivisorTool** as shown in the previous section. If such a component was used in a .NET assembly and the **Divide** function invoked with a divisor value of 0, a **divide-by-zero** error would be raised by the Visual Basic 6.0 component. This error would be automatically translated to a **System.DivideByZeroException** on the .NET side, so you could easily check for that system exception and handle the error gracefully in your client code.

The following code example illustrates this approach.

Visual Basic 6.0 component

```
Public Function Divide(ByVal term As Double, ByVal divisor As Double) As Double
    Divide = term / divisor
End Function
```

Visual Basic .NET client

```
Private Function Perform_Division( _
    ByVal term As Double, _
    ByVal divisor As Double) As Double

    Dim x As Double
    Dim div As VB6Module.DivisorTool
    div = New VB6Module.DivisorTool
    Try
        Return div.Divide(term, divisor)
    Catch e As System.DivideByZeroException
        System.Windows.Forms.MessageBox.Show("Division by 0")
        Return 0
    End Try
End Function
```

As shown in the Visual Basic .NET code example, the invocation of the Visual Basic 6.0 function **Divide** is enclosed in a **try/catch** block, with a **Catch** clause for the **System.DivideByZeroException**.

As mentioned before, application-specific errors are a little more complicated than just dealing with general errors. Visual Basic 6.0 application errors are raised as an **InteropServices.COMException** in Visual Basic .NET code. This code example expands the previous example, adding an application exception when the specified divisor, or the specified term is 3.

Visual Basic 6.0 component

```
Public Function Divide(ByVal term As Double, ByVal divisor As Double) As Double
    If divisor = 3 Then
        Err.Raise 1, "Project1", "Bad divisor", "", ""
    ElseIf term = 3 Then
        Err.Raise 2, "Project1", "Bad term", "", ""
    Else
```

```
        Divide = term / divisor
    End If
End Function
```

Visual Basic .NET client

```
Private Function Perform_Division( _
    ByVal term As Double, _
    ByVal divisor As Double) As Double

    Dim x As Double
    Dim div As VB6Module.DivisorTool
    div = New VB6Module.DivisorTool
    Try
        Return div.Divide(term, divisor)
    Catch e As System.DivideByZeroException
        System.Windows.Forms.MessageBox.Show("Division by 0")
        Return 0
    Catch e As System.Runtime.InteropServices.COMException
        System.Windows.Forms.MessageBox.Show("App Error")
    End Try
End Function
```

In the previous code example, the Visual Basic .NET client code does not distinguish between the two Visual Basic 6.0 application errors; therefore, it cannot gracefully handle the error condition. In the previous example, it would be impossible to tell whether it was a "bad divisor error" (Error #1) or a "bad term error" (Error #2).

The differentiation between error conditions can be achieved by using the **COMException.ErrorCode** value. Thus, a mechanism is provided that will allow Visual Basic .NET client code to differentiate error conditions by their error code values, as specified in the **Err.Raise** statements in the Visual Basic 6.0 component code.

The following code example shows an enhanced version of the Visual Basic .NET client code. It inspects the error code value to appropriately react to application-specific error conditions.

Visual Basic .NET client (with specific error handling)

```
Private Function Perform_Division( _
    ByVal term As Double, _
    ByVal divisor As Double) As Double

    Dim x As Double
    Dim div As VB6Module.DivisorTool
    div = New VB6Module.DivisorTool
    Try
        Return div.Divide(term, divisor)
    Catch e As System.DivideByZeroException
        System.Windows.Forms.MessageBox.Show("Division by 0")
```

```
        Return 0
    Catch e As System.Runtime.InteropServices.COMException
        If (e.ErrorCode And &HFFFF) = 1 Then
            'Handle the "Bad Divisor" error
        ElseIf (e.ErrorCode And &HFFFF) = 2 Then
            'Handle the "Bad Term" error
        Else
            System.Windows.Forms.MessageBox.Show("App Error")
        End If
    End Try
End Function
```

As shown in the previous code example, a bit operation must be performed to extract the actual error code from the exception object. A bitwise **And** operation between the **ErrorCode** and the hexadecimal value FFFF (&HFFFF) checks the lower 16 bits of the **ErrorCode** value. These bits contain the actual error code raised from Visual Basic 6.0's **Err.Raise** call. By examining the lower 16 bits, you can determine the exact error that was raised in the Visual Basic 6.0 component and react appropriately to application-specific error conditions. This way, you can keep your business logic consistent across the .NET boundary, even in regard to error handling.

Sinking COM Events

Sinking events for COM client consumption from Visual Basic 6.0 is done through the **RaiseEvent** statement. These events can also be caught and handled by .NET code as if they were originating from a COM component.

The following Visual Basic 6.0 code example illustrates how to raise an event and the subsequent Visual Basic .NET code example shows how that event would be caught and processed. The Visual Basic 6.0 event producer code raises a **Divide** event whenever the **DoDivide** function is invoked and raises a **Multiply** event when the **DoMultiply** function is invoked. The consumer code, written in Visual Basic .NET, provides the event handlers **OnDivide** and **OnMultiply** to catch and handle each event whenever they occur.

Visual Basic 6.0 event producer

```
Option Explicit

Public Event Divide(ByVal x As Double, ByVal y As Double)
Public Event Multiply(ByVal x As Double, ByVal y As Double)

Public Function DoDivide(x As Double, y As Double) As Double
    RaiseEvent Divide(x, y)
    DoDivide = x / y
End Function

Public Function DoMultiply(x As Double, y As Double) As Double
    RaiseEvent Multiply(x, y)
```

```
    DoMultiply = x * y
End Function
```

Visual Basic .NET event consumer

```
Option Explicit
Option Strict

Imports System
Imports System.Runtime.InteropServices

Namespace EventConsumer

    Public Class Consumer

        Dim WithEvents myProducer As New VB6Module.Producer

        Private Sub ConsumeTest()
            myProducer.DoDivide(1,3)
            myProducer.DoMultiply(5,7)
        End Sub

        Private Sub OnDivide(x As Double, y As Double) _
            Handles myProducer.Divide
            Console.WriteLine("Division: {0}", x / y)
        End Sub

        Private Sub OnMultiply(x As Double, y As Double) _
            Handles myProducer.Multiply
            Console.WriteLine("Multiplication: {0}", x * y)
        End Sub

    End Class
End Namespace
```

Similarly, it is possible to raise events in Visual Basic .NET components using the **Delegate** directive. These events can be caught in Visual Basic 6.0 client code using the **WithEvents** directive. The following code example demonstrates this. In this code example, the producer code is the Visual Basic .NET version of the event producer in the previous example, and the consumer code is the Visual Basic 6.0 version of the consumer in the previous example.

Visual Basic .NET event producer

```
Option Explicit On
Option Strict On

Namespace EventProducer
    <ComClass(Producer.ClassId, Producer.InterfaceId, Producer.EventsId)> _
    Public Class Producer
```

```
#Region "COM GUIDs"
        Public Const ClassId As String = "591AC4AC-949A-45E6-AE5F-FFB5B7635E9B"
        Public Const InterfaceId As String = _
        "D3DC65BE-F49E-4356-8308-80B9B6198139"
        Public Const EventsId As String = "58D8EC7F-821A-453A-BFC0-6AF8A8D43D1A"
#End Region

        Public Event Divide(ByVal x As Double, ByVal y As Double)
        Public Event Multiply(ByVal x As Double, ByVal y As Double)

        Public Sub New()
            MyBase.New()
        End Sub

        Public Sub SendDivide(ByVal x As Double, ByVal y As Double)
            RaiseEvent Divide(x, y)
        End Sub

        Public Sub SendMultiply(ByVal x As Double, ByVal y As Double)
            RaiseEvent Multiply(x, y)
        End Sub

        ' Add more custom functions to this class, which then
        ' send the event notifications by calling SendXXXX.
    End Class
End Namespace
```

Visual Basic 6.0 event consumer

```
Public WithEvents myProd As Producer

Private Sub Class_Initialize()
    Set myProd = New Producer
End Sub

' Events and methods are matched by event name and signature.
Private Sub myProd_Divide(ByVal x As Double, ByVal y As Double)
    MsgBox "Division: " & (x / y)
End Sub

Private Sub myProd_Multiply(ByVal x As Double, ByVal y As Double)
    MsgBox "Multiplication: " & (x * y)
End Sub

Public Sub ProducerTest()
    myProd.SendDivide 1, 2
    myProd.SendMultiply 3, 5
End Sub
```

As shown in the previous code example, after the **SendMultiply** and **SendDivide** functions are defined, you can raise events, when necessary, to any COM client that

is listening. This also applies to Visual Basic 6.0 clients because they would connect to the Visual Basic .NET event producer through COM interop.

Because the CLR is present between the components written in Visual Basic 6.0 and Visual Basic .NET, event handling between each run time environment is transparent to the other. It does not matter to the component catching the event if it was raised from a managed .NET component or an unmanaged Visual Basic 6.0 component. The CLR's COM interop layer handles the details, and this simplifies the upgrade process.

OLE Automation Call Synchronization

The Visual Basic 6.0 **App** object provides a group of properties that can be used to specify parameters for the interaction and synchronization with an OLE Automation server during the invocation of one of its methods.

The first set of properties is related to the time that the application will retry a failed automation call request. After the specified time elapses, a customizable dialog box automatically displays in the application. This dialog box informs the user about the busy state of the OLE server. This behavior is controlled with the properties: **OleServerBusyMsgText**, **OleServerBusyMsgTitle**, **OleServerBusyRaiseError**, and **OleServerBusyTimeout**.

The second set of properties is related to the time an application will wait for an OLE Automation request to be completed. This group of properties works in a similar way to the previous group and includes: **OleRequestPendingMsgText**, **OleRequestPendingMsgTitle**, and **OleRequestPendingTimeout**.

The functionality provided by these properties does not have an equivalent in Visual Basic .NET and requires reimplementation using the currently available mechanisms.

The objective of this functionality is to produce a responsive application that allows the user to be aware of the state of the application. One way to provide similar functionality with Visual Basic .NET is to use the .NET multithreading capabilities.

The following code example shows how to use Visual Basic .NET to make an asynchronous call to an OLE Automation server; this allows the application to wait and then provide useful information to the user about the state of the application and pending requests.

Visual Basic 6.0 component

```
Private Sub Form_Load()
    App.OleRequestPendingMsgText = "Pending"
    App.OleRequestPendingTimeout = 500
    App.OleServerBusyMsgText = "Busy"
    App.OleServerBusyRaiseError = False
End Sub
```

```
Private Sub Command1_Click()
    Dim c As New ComponentExe.ClassComExe
' This method takes a lot of time.
    c.myMethod
    MsgBox "myMethod ComExe finished"
End Sub
```

The previous **Form_Load** method sets custom parameters that affect the dialog box that is displayed when the application is waiting for the *ClassComExe.myMethod* method to execute. In Visual Basic .NET, this functionality needs to be reimplemented in a different way to let the user know whether the application is still running.

The following code example invokes the *ClassComExe.myMethod* method using an intermediary class that is executed in a different thread and allows the main application to remain responsive to the user. Notice how the auxiliary class for asynchronous calls notifies the application about the process completion; this raises an event that is handled by the main application.

Visual Basic .NET component

```
Dim WithEvents AsynchCallerObj As AsynchCaller

Class AsynchCaller
    Public Event ThreadDone()
    Sub DoCall()
        Dim c As New ComponentExe.ClassComExe
        c.foo()
        RaiseEvent ThreadDone()
    End Sub
End Class

Private Sub Form1_Load(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) Handles MyBase.Load
    ' Code removed by the user because it is not supported.
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
    AsynchCallerObj = New AsynchCaller
    Dim Thread As New System.Threading.Thread(AddressOf AsynchCallerObj.DoCall)
    Thread.Start()
    ' Here a progress monitor could be started to let the user know about the
    ' process state.
End Sub

Sub AsynchCallerDone() Handles AsynchCallerObj.ThreadDone
    MsgBox("foo ComExe finished")
End Sub
```

Resource Handling

The Common Language Runtime (CLR) provides a resource handling mechanism that frees programmers from the burden of manual memory management. This mechanism is known as garbage collection and is an essential part of .NET.

Visual Basic .NET objects and Visual Basic 6.0 objects are created with the **New** keyword. Usually, before an object can be used it requires an initialization stage. This initialization can include tasks such as opening files, connecting to a database, or acquiring other system resources like memory for internal structures. Visual Basic .NET controls the initialization of new objects using procedures called constructors.

Objects end life after they leave scope and are released by the CLR. Visual Basic .NET controls the release of system resources using procedures called destructors. Together, constructors and destructors support the creation of robust and predictable classes that use the system resources in an efficient manner.

Constructors and Destructors in Visual Basic .NET

The **Sub New** and **Sub Finalize** procedures in Visual Basic .NET are the constructor and destructor methods respectively; they replace the **Class_Initialize** and **Class_Terminate** methods used in Visual Basic 6.0. As opposed to **Class_Initialize**, the **Sub New** constructor can run only once when a class is created, and it cannot be called explicitly anywhere other than in the first line of code in another constructor from either the same class or from a derived class.

Before releasing objects, the CLR automatically calls the **Finalize** method for objects that define a **Sub Finalize** procedure. The **Finalize** method can contain code that needs to execute just before an object is destroyed, such as closing files and saving state information or releasing unmanaged components (for example, COM components created with Visual Basic 6.0). The execution of the **Sub Finalize** will consume time, so you should define a **Sub Finalize** method only when you need to release objects explicitly, as in the case of contained COM components. Unlike the Visual Basic 6.0 **Class_Terminate**, which is executed as soon as an object reference is set to **Nothing**, there is usually a delay between when an object loses scope and when Visual Basic .NET invokes the **Finalize** destructor.

Visual Basic .NET allows for a second kind of destructor, named **Dispose**, which must be explicitly called whenever allocated resources need to be released immediately. Any .NET classes can implement the **IDisposable** interface in order to release allocated unmanaged resources. This interface defines the **Dispose** method that users of the component should call when a component's instance is no longer needed. The implementation of **Dispose** can contain code to immediately release limited system resources like allocated memory, file handles, and database connections.

Garbage Collection

In .NET, the garbage collection mechanism is responsible for the invocation of component's destructors when the component cannot be accessed by any executing code. This condition is reached when all references to the component have been released or belong to objects that are isolated from all running code.

The .NET Framework uses a technique called reference-tracing garbage collection to periodically release resources that will no longer be used. (For information about garbage collection, see "Object Lifetime: How Objects Are Created and Destroyed" on MSDN.) Visual Basic 6.0 used a different system called reference counting with the same purpose. These systems have the following main differences that must be considered in order to achieve successful interoperation:

- **Non-deterministic object lifetime**. The CLR destroys objects more quickly when system resources decrease to a certain level; on the other hand, objects are destroyed slower when system resources abound. The result of this scheme is that it is impossible to determine in advance when an object will actually be destroyed and its resources freed under Visual Basic .NET. In this case, .NET objects are said to have a non-deterministic lifetime. This behavior does not affect the development process, as long as it is understood that the **Finalize** destructor may not execute immediately when an object goes out of scope under Visual Basic .NET.
- **Assignment of the Nothing value**. In Visual Basic 6.0, programmers can affect the reference count for an object by assigning **Nothing** to one of the corresponding object variables. When the reference count reaches zero, the object resources are immediately released. In Visual Basic .NET, an assignment with a **Nothing** value will never produce an immediate release; if this is necessary, the **Dispose** method must be implemented and explicitly invoked at the desired time.

Note: The **System.Runtime.InteropServices.Marshal** class provides methods to manipulate the reference count associated with COM objects and manually determine an object's lifetime. These methods include: **AddRef**, **Release** and **ReleaseComObject**. The **ReleaseComObject** method is often used to release unmanaged resources that are allocated in a .NET component. This will force the release of the COM object without waiting for the garbage collector to clean resources.

Summary

Partial and staged upgrade strategies can help minimize the risks of upgrading an application while still allowing it to benefit from new language features and technologies. A key necessity in these strategies is the ability to have Visual Basic 6.0 components interact with new Visual Basic .NET components.

With COM interop in Visual Basic .NET, partial upgrades can be painlessly achieved. Visual Basic 6.0 components that are complex or otherwise costly to upgrade can remain in Visual Basic 6.0, while more straightforward components can be upgraded to Visual Basic .NET. Through COM interop, the older and newer components can seamlessly interact with each other, leveraging your Visual Basic 6.0 components with new Visual Basic .NET features. The techniques presented in this chapter can help you achieve this interoperability.

More Information

To download the Microsoft Service Pack 6 for Visual Basic 6.0 from MSDN: *http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp6/default.aspx*.

For more information about performance improvement, see Chapter 7, "Improving Interop Performance," of *Improving .NET Application Performance and Scalability* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt07.asp*.

For more information on custom marshals, see "Custom Marshaling" in the *.NET Framework Developer's Guide* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconCustomMarshaling.asp*.

For more information about mapping common .NET exceptions to Visual Basic 6.0 error codes, see "HRESULTs and Exceptions" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhresultsexceptions.asp*.

For a description of the fields contained in a **HRESULT** value, see "HRESULT" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/APISP/html/sp__mapi1book_c_type_hresult.asp*.

For information about garbage collection, see "Object Lifetime: How Objects Are Created and Destroyed" in *Visual Basic Language Concepts* on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vaconFinalizeDestructors.asp*.

15

Upgrading MTS and COM+ Applications

Microsoft Transaction Server (MTS) is a technology primarily intended for multi-user server applications. In addition to supporting distributed transactions, MTS extends COM's security model and provides scalability services, connection management, thread pooling, and an administration structure. This environment provides an easy way to build and deploy scalable server applications, making available a suitable distributed-enabled atmosphere for COM objects running in the middle tier. When Microsoft Transaction Server is installed, middle-tier components and objects should be run inside the MTS environment whether or not they are involved in transactions.

COM+ is an extension to the Component Object Model (COM) and the MTS infrastructure. The consolidation of these two programming models makes it easier to develop distributed applications by unifying the development, deployment, debugging, and maintenance of an application that would formerly have relied on COM for certain services and MTS for others.

This chapter examines how to upgrade Visual Basic 6.0 COM+ and MTS applications to Visual Basic .NET.

Using MTS/COM+ in Visual Basic 6.0

Visual Basic 6.0 can be used with MTS to build n-tier applications. Using Visual Basic 6.0, you develop COM DLL components that run on middle-tiered servers under the control of MTS. When clients call these COM DLLs, the Windows operating system routes the requests to MTS automatically. Other services provided by MTS that are relevant to upgrade efforts include:

- Component transactions.
- Object brokering.

- Resource pooling.
- Just-in-time activation.
- Administration.

A possible upgrade strategy is to use proxy COM classes that are accessed through interoperability. While this strategy is adequate in some scenarios, using proxy classes is not always possible or appropriate. In some cases you need to re-implement the MTS and COM+ services using the .NET Framework instead.

The System.EnterpriseServices namespace in the .NET Framework contains classes that offer similar functionality and provide an interface to mechanisms used by COM+. This makes it possible to build enterprise applications using .NET objects while retaining access to COM+ services.

The following is a code example of a COM+ component written using Visual Basic 6.0 that has the MTSTransactionMode property set to RequiresTransaction:

```
Public Sub transferfunds(acc1 As String, acc2 As String, _
    amount As Long)

    On Error GoTo ErrHandler

    withdrawFunds acc1, amount
    depositFunds acc2, amount
    GetObjectContext.SetComplete

    Exit Sub
ErrHandler:
    GetObjectContext.SetAbort
    Err.Raise Err.Number, Err.Source, Err.Description, _
        Err.HelpFile, Err.HelpContext
End Sub

Private Sub withdrawFunds(acc As String, amount As Long)
    ' Perform withdrawal...
End Sub

Private Sub depositFunds(acc As String, amount As Long)
    ' Perform deposit...
End Sub
```

Using COM+ in Visual Basic .NET

The .NET Framework relies on COM+ to provide it with distributed transactions, component instance management, role-based security, queued components and events. The System.EnterpriseServices namespace makes COM+ part of the .NET Framework, in essence, by supplying all the types, classes, and objects necessary to use COM+ services.

A .NET component that uses COM+ services is referred to as a serviced component. The following code example shows a serviced component as a class in Visual Basic .NET that transfers funds between two bank accounts. Please note that although it appears similar to the preceding example, it is not yet functionally equivalent because the transaction services are not being used. These will be added a little later.

```
Imports System.EnterpriseServices

Public Class MyComponent
    Inherits ServicedComponent

    Public Sub transferfunds(ByRef acc1 As String, _
        ByRef acc2 As String, ByRef amount As Integer)

        On Error GoTo ErrHandler

        withdrawFunds(acc1, amount)
        depositFunds(acc2, amount)

        Exit Sub
    ErrHandler:
        Err.Raise(Err.Number, Err.Source, Err.Description, _
           Err.HelpFile, Err.HelpContext)
    End Sub

    Private Sub withdrawFunds(ByRef acc As String, _
        ByRef amount As Integer)
        ' Perform withdrawal...
    End Sub

    Private Sub depositFunds(ByRef acc As String, _
         ByRef amount As Integer)
        ' Perform deposit...
    End Sub

End Class
```

Notice that building this class requires that you add a reference to the System.EnterpriseServices namespace.

Consider the follow information when you create serviced components:

- When an instance of a serviced component is no longer needed, the client should call the Dispose() method to free the resources used by the component.
- The serviced component must contain a default constructor (a constructor with no arguments).
- Static methods are not remotable and are not associated with a particular instance of a class; therefore, they cannot take advantage of any of the COM+ services such as transactions and object pooling.

- Those services that flow between computers, such as transactions and Windows authentication, only do so when DCOM is used.
- At run time, the user executing the COM+ application must have permission to run unmanaged code.

The following code example is a slightly more complex version of the Visual Basic 6.0 code example from the previous section. In this example, the code was upgraded to a .NET serviced component.

```
Option Strict Off
Option Explicit On
Imports System.EnterpriseServices

<Transaction(TransactionOption.Required), EventTrackingEnabled()> _
Public Class Account
    Inherits ServicedComponent

    Public Sub transferfunds(ByRef acc1 As String, _
        ByRef acc2 As String, ByRef amount As Integer)
        On Error GoTo ErrHandler

        withdrawFunds(acc1, amount)
        depositFunds(acc2, amount)
        ContextUtil.SetComplete()

        Exit Sub
    ErrHandler:
        ContextUtil.SetAbort()
        Err.Raise(Err.Number, Err.Source, Err.Description, _
           Err.HelpFile, Err.HelpContext)
    End Sub

    Private Sub withdrawFunds(ByRef acc As String, _
        ByRef amount As Integer)
        ' Perform withdrawal...
    End Sub

    Private Sub depositFunds(ByRef acc As String, _
         ByRef amount As Integer)
        ' Perform deposit...
    End Sub
End Class
```

The first important instruction here is the reference to the System.EnterpriseServices namespace. It supplies the infrastructure for enterprise applications and provides .NET objects with access to COM+ services.

Next, the class extends the ServiceComponent class, which enables it to be used as a COM+ application. The ServiceComponent class, in the System.EnterpriseServices namespace, is the base object of all classes using COM+ services.

Finally, the Transaction and the EventTrackingEnabled attributes denote the type of transaction that is available to the Account class (in this example the transaction is Required) and enable event tracking for the Account class, respectively.

To deploy and use a ServicedComponent object it is necessary to sign the corresponding component with a strong name, consisting of the assembly's identity (its simple text name and version number, which are required, and the culture information if it is provided) plus a public key and a digital signature. This task can be accomplished by using the Strong Name tool (Sn.exe), a command-line tool distributed with the .NET Framework to create a key pair file with the file name extension .snk. The generated file must be copied to the component's source code folder. This important step is included in the procedures within this chapter that require it but for more information about the Strong Name tool, see "Strong Name Tool (Sn.exe)" in .NET Framework Tools on MSDN.

When using ServicedComponent classes it is necessary to specify additional configuration options by using assembly attributes like the following code example:

```
<Assembly: AssemblyKeyFile("MyAssembly.snk")>
```

The AssemblyKeyFileAttribute class is used to specify the file that is generated with the Strong Name Tool (Sn.exe) and contains a key pair that is used to generate a strong name. These attributes can be added to a class file, but the recommended place to include these assembly attributes is the AssemblyInfo.vb file that is included in all Visual Basic .NET projects.

General Considerations

This section explores some of the common issues concerning COM+, such as services, applications, and deployment and how they are affected by the upgrade to Visual Basic .NET.

For Visual Basic 2005:

The Visual Basic Upgrade Wizard included in Visual Studio 2005 upgrades Visual Basic 6.0 Microsoft Transaction Server and COM+ Services projects into EnterpriseServices projects in Visual Basic 2005. Most of the tasks described in this chapter are performed automatically by the upgrade wizard, with the exception of the tasks described in the "COM+ Events" and "Message Queuing and Queued Components" sections of this chapter. These tasks still require a manual upgrade.

In addition, the manual tasks for deploying and configuring an application are still required after you use the Visual Basic 2005 Upgrade Wizard.

COM+ Application Types

The following list describes the four basic types of COM+ applications:

- Library applications. A library application runs in the process of the client that creates it. Library applications can use role-based security but do not support remote access or queued components.
- Server applications. A server application runs in a dedicated process that COM+ creates. Cross-process communication is the major factor that affects performance differences and limitations between library and server applications. Server applications can support all COM+ services.
- Application proxies. An application proxy is a set of files that contains registration information that allows a client to remotely access a server application.
- COM+ preinstalled applications. COM+ includes a set of preinstalled applications that handle internal functions. The preinstalled applications are listed in the COM+ Applications folder in the Component Services administrative tool, but they cannot be modified or deleted. The following applications are included:
  - .NET Utilities
  - Analyzer Control Publisher Application
  - COM+ Explorer
  - COM+ QC Dead Letter Queue Listener
  - COM+ Utilities
  - IIS In-Process Applications
  - IIS Out-Of-Process Pooled Applications
  - System Application

These applications belong to the COM+ infrastructure and are not bound to any specific languages, so there are no special factors to consider when you upgrade Visual Basic 6.0 code.

Using SOAP Services

It is often useful for a client application to be able to call a method that is implemented on a remote server. Sometimes the method uses volatile information that is stored on the remote server (for example, a method that returns the current currency exchange that corresponds to a given currency). At other times, the developer wants to be able to upgrade the methods implementation without having to redeploy all the applications that use it.

XML Web services are published on Web servers, such as IIS, and accessed using HTTP. These HTTP packets contain the input and output parameters of calls to a method that is implemented on the server and are encoded in SOAP. To use an XML Web service, you need to know the URL where the service is exposed and the name of the method you want to call, and you must provide the input parameters to the method.

Note: The SOAP services functionality is supported in the following operating systems: Microsoft Windows 2003, Microsoft Windows 2000 and Microsoft Windows XP. SOAP services are not supported in Microsoft Windows NT.

SOAP

While it is helpful to have some understanding of the infrastructure that underlies XML Web services, COM+ makes it easy to create and use them.

Any COM+ application can be exposed as an XML Web service. Clients can then make remote calls to the methods in the default interfaces of the application's configured components. You can use the Component Services administrative tool to create an IIS virtual root directory that calls the component methods by using SOAP. You do not need to do any special programming when writing the components, except that the methods that you want to expose must be in the default interface and the component must be configured in the server's COM+ catalog. You do not need to write code to communicate through a network interface or to parse SOAP.

When you expose a COM+ application as an XML Web service, detailed information about the syntax of all the methods that are available from an XML Web service is published automatically, using the Web Services Description Language (WSDL). Clients use this information to communicate with the XML Web service.

COM+ provides the following two ways to access and use a remote XML Web service:

- The well-known object (WKO) mode can be used to access any XML Web service that publishes its syntax using WSDL, even if that XML Web service was not created using COM+ or even using Microsoft Windows.
- The client-activated object (CAO) mode can be used only to access XML Web services created by exposing COM+ applications. CAO mode increases performance by using persistent connections, which is a feature that is not supported by the current SOAP standard.

Both methods allow client applications to call the methods of XML Web services without having to write code to communicate through a network interface or to parse SOAP.

The following code is a COM+ and SOAP example. It shows how to use the ApplicationActivation attribute with the SoapVRoot element to automatically publish the components as a Web service.

To use COM+ in Visual Basic .NET, it is necessary to sign the component with a strong name. To generate the keys required for the strong name you must use the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter).

```
sn -k SOAPServices.snk
```

The following code is a COM+/SOAP example.

```
Imports System
Imports System.Reflection
Imports System.Runtime.InteropServices
Imports System.EnterpriseServices

Namespace SOAPServices

    Public Interface ICalc
        Function Add(ByVal Value1 As Integer, ByVal Value2 As Integer)
            As Integer
    End Interface

    Public Class Calc
        Inherits ServicedComponent
        Implements ICalc

        Public Function Add(ByVal Value1 As Integer, _
            ByVal Value2 As Integer) As Integer Implements ICalc.Add

            Return (Value1 + Value2)
        End Function
    End Class
End Namespace
```

For this example, you must include the following lines in the AssemblyInfo.vb file.

```
Imports System.EnterpriseServices
<Assembly: ApplicationName("SOAP Services")>
<Assembly: ApplicationActivation(ActivationOption.Server, SoapVRoot := _
    "SOAPServices")>
<Assembly: AssemblyKeyFile("SOAPServices.snk")>
```

COM+ Application Proxies in .NET

To access a COM+ server application remotely from another computer, the client computer must have a subset of the attributes of the server application installed, including proxy/stub DLLs and type libraries for DCOM interface remoting. This subset is called an application proxy.

You do not have to write special code to create an application proxy when you create a serviced component and register it on the server; the ServicedComponent class contains MarshalByRefObject in its inheritance tree and can therefore be accessed from remote clients. However, to prevent the data included in the COM+ application proxy from being included in the actual server code, you should make it a practice to separate the interface definitions from the class implementations.

Upgrading MTS/COM+ Services

Using the upgrade wizard and manual modifications to upgrade a COM+ component is mostly a straightforward process, but there are a few issues to bear in mind.

The code samples in this chapter explore these issues in detail. Each scenario lists a Visual Basic 6.0 component, the output of the upgrade wizard after it is applied to the component, and the manual changes that you must make to complete the upgrade to Visual Basic .NET.

Note: To try the .NET component code samples that are used in this chapter, you need to first register them for COM interop. For information about how to register .NET components, see the "Creating Interoperability Wrappers in .NET" section in Chapter 14, "Interop between Visual Basic 6.0 and Visual Basic .NET."

COM+ Example Scenario

In this scenario, a Visual Basic 6.0 server component exposes a function. This component is added to the Component Services as a COM+ application, which is used by a client application. First, the upgrade wizard is used to upgrade the server component, and then the necessary manual changes to create an equivalent component in Visual Basic .NET are made. The scenario is completed by upgrading the client component.

The following highly simplified code sample contains a server component in Visual Basic 6.0 named COMTest. The component in this example does not require or use any transaction.

```
Private OpCounter As Integer

Public Sub Init()
    OpCounter = 0
End Sub

Public Function Add(ByVal value1 As Long, ByVal value2 As Long) _
    As Long

    Dim SumResult As Long
    SumResult = value1 + value2
    OpCounter = OpCounter + 1
    add = SumResult
End Function
```

The core of the component is an addition function that receives two parameters of type Long; the component also exposes an initialization method (Init).

The following code sample lists the Visual Basic 6.0 application that instantiates the server component that resides within a COM+ application.

```
Public Sub Execute()
    Dim MyCOM As Object
    Dim result As Long

    Set MyCOM = CreateObject("Test.COMTest")
    MyCOM.Init
    result = MyCOM.Add(10, 5)

End Sub
```

In this scenario, the name of the server component module is COMTest, and the name of the dynamic link library is Test. To instantiate the object in a COM+ application, CreateObject is called with Test.COMTest as the parameter

Upgrading the Server Component

When you upgrade the server component with the upgrade wizard, the process results in the following output.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")>
Public Class COMTest
    Private OpCounter As Short

    Public Sub Init()
        OpCounter = 0
    End Sub

    Public Function Add(ByVal value1 As Integer, ByVal value2 As Integer) _
        As Integer

        Dim SumResult As Integer
        SumResult = value1 + value2
        OpCounter = OpCounter + 1
        add = SumResult
    End Function
End Class
```

The upgrade wizard upgrades most of the module's functionality correctly, but there are several minor changes that you must make before you have a fully functional and accurate server component.

► **To make a functionally equivalent component**

1. Add the System.EnterpriseServices reference to the project and specify the Imports statement.

```
Imports System.EnterpriseServices
```

2. Change the class so that it inherits from the ServicedComponent class.

```
Public Class COMTest
    Inherits ServicedComponent
```

3. Create a strong name for the COMTest component: (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k COMKey.snk
```

4. Link the server component with the key pair file that you just created. To do this, add the AssemblyKeyFile attribute to the AssemblyInfo.vb file.

```
<Assembly: AssemblyKeyFile("COMKey.snk")>
```

These changes result in the following listing.

```
Option Strict Off
Option Explicit On
Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")>
Public Class COMTest
    Inherits ServicedComponent

    Private OpCounter As Short
    Public Sub Init()
        OpCounter = 0
    End Sub

    Public Function Add(ByVal value1 As Integer, ByVal value2 As Integer) _
        As Integer

        Dim SumResult As Integer
        SumResult = value1 + value2
        OpCounter = OpCounter + 1
        add = SumResult
    End Function
End Class
```

After completing all of the manual changes, you can build the component, create a new COM+ application, and add the DLL to the application.

Keep in mind this very important detail: In the Visual Basic 6.0 component, the Instancing attribute is set to Multiuse, which allows other applications to create objects from the class. One instance of your component can provide any number of objects that are created in this fashion. An out-of-process component can supply

multiple objects to multiple clients; an in-process component can supply multiple objects to the client and to any other components in its process. This behavior is code-independent, which means that you can switch the COM+ application activation mechanism without modifying the component or the client source code.

In contrast, Visual Basic .NET components use the library activation type by default. This means that if you specify that the COM+ application should be activated as a server, the .NET client application will crash when you try to create the component. To specify how the .NET component will be activated you need to modify the component's source code by using the ApplicationActivation attribute with the AssemblyInfo.vb file:

- For a server component, use it as shown here.

```
<Assembly: System.EnterpriseServices.ApplicationActivation _
    (System.EnterpriseServices.ActivationOption.Server)>
```

- For a library component, use it as shown here.

```
<Assembly: System.EnterpriseServices.ApplicationActivation _
    (System.EnterpriseServices.ActivationOption.Library)>
```

Upgrading the Client Application

Upgrading the client application is more straightforward than upgrading the server component. The following code example shows the output that is produced by the upgrade wizard.

Note: In the following code example and throughout this chapter, some of the longer code lines and comments that are automatically produced by the upgrade wizard are reformatted to make them easier to read and understand. Your output may differ slightly from the code shown in this chapter.

```
Public Sub Execute()
    Dim COM As Object
    Dim result As Integer

    COM = CreateObject("Test.COMTest")
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object COM.init
    COM.Init()
    ' UPGRADE_WARNING: Could not resolve default property of
    ' object COM.Add
    result = COM.Add(10, 5)
End Sub
```

The following statement is the only one that requires modification.

```
COM = CreateObject("Test.COMTest")
```

The parameter used with the CreateObject method must match the name of the .NET component inside the COM+ application. Refer to the ProgID attribute in the Server Component for the correct name to use.

```
COM = CreateObject("COMTest_NET.COMTest")
```

The final client application is as follows.

```
Public Sub Execute()
    Dim COM As Object
    Dim result As Integer

    COM = CreateObject("COMTest_NET.COMTest")
    COM.Init()
    result = COM.Add(10, 5)
End Sub
```

For Visual Basic 2005:

The version of the Visual Basic .NET Upgrade Wizard that ships with Visual Studio 2005 will automatically perform most of the manual changes described in this chapter.

COM+ Compensating Resource Manager

The COM+ Compensating Resource Manager (CRM) Services can be used to integrate application resources with Microsoft Distributed Transaction Coordinator (DTC) transactions.

There are some classes and interfaces provided by the COM+ Services Library that allow developers to take advantage of CRM Services, such as CRMClerk, CRMRecoveryClerk, ICrmFormatLogRecords, ICrmMonitorLogRecords and tagCrmLogRecordRead. These services are also provided in the .NET Framework in the System.EnterpriseServices.CompensatingResourceManager namespace.

Developers can use the CRMClerk class to handle log-related operations. The equivalent of this class in the .NET Framework is System.EnterpriseServices.CompensatingResourceManager.Clerk.

Visual Basic 6.0 developers can implement ICrmCompensatorVariants interface to handle log-related issues. To get similar functionality in Visual Basic .NET, the class supporting ICrmCompensatorVariants must inherit from System.EnterpriseServices.CompensatingResourceManager.Compensator.

A CRM consists of two separate COM co-classes, Worker and Compensator, which must be implemented. The Worker class initiates changes, and the Compensator class either commits the changes or rolls them back.

Upgrading a Worker Component

The following code example shows a CRM Worker component that is defined inside a Visual Basic 6.0 class named COMTest.

```
Public Clerk As CRMClerk

Public Sub Register()
    On Error GoTo ErrorHandler
    Set Clerk = New CRMClerk

    Clerk.RegisterCompensator "MyCompensator.Comp", _
        "This is my Compensator", CRMREGFLAG_ALLPHASES
Exit Sub

ErrorHandler:
    ' Error handling code goes here.
End Sub
```

The following example shows the code after it is upgraded.

```
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest
    Public Clerk As COMSVCSLib.CRMClerk

    Public Sub Register()
        On Error GoTo ErrorHandler
        Clerk = New COMSVCSLib.CRMClerk

        Clerk.RegisterCompensator("MyCompensator.Comp", _
            "This is my Compensator", _
            COMSVCSLib.tagCRMREGFLAGS.CRMREGFLAG_ALLPHASES)
    Exit Sub

    ErrorHandler:
        ' Error handling code goes here.
    End Sub
End Class
```

► **To create a functionally equivalent component**

1. Add the EnterprisesServices reference to the project and a corresponding Imports statement to the class.

```
Imports System.EnterpriseServices
```

2. Change the class so that it inherits from the ServicedComponent class.

```
Inherits ServicedComponent
```

3. Delete all the COMSVCSLib.CRMClerk references because .NET Framework equivalent classes now replace those references.
4. Replace the Visual Basic 6.0 Clerk object with the .NET CompensatingResourceManager.Clerk object.
5. Remove the call to the COMSVCSLib.CRMClerk constructor and the call to the RegisterCompensator method and replace them with just a single call to the Clerk class constructor. The Clerk constructor has the following signature.

```
Public Sub New( _
    ByVal compensator As String, _
    ByVal description As String, _
    ByVal flags As CompensatorOptions _
    )
```

6. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

These changes result in the following, cleanly upgraded component.

```
Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits ServicedComponent

    Public Clerk As CompensatingResourceManager.Clerk

    Public Sub Register()
        On Error GoTo ErrorHandler
        Clerk = New CompensatingResourceManager.Clerk("", "", _
            CompensatingResourceManager.CompensatorOptions.AllPhases)
        Exit Sub

    ErrorHandler:
        ' Error handling code goes here.
    End Sub
End Class
```

Upgrading a Compensator Component

The following code example shows a CRM Compensator component in Visual Basic 6.0.

```
Option Explicit

Implements ICrmCompensatorVariants

Dim CrmLogControl As ICrmLogControl

Private Function ICrmCompensatorVariants_AbortRecordVariants( _
    pLogRecord As Variant) As Boolean

End Function

Private Sub ICrmCompensatorVariants_BeginAbortVariants( _
    ByVal bRecovery As Boolean)

End Sub

Private Sub ICrmCompensatorVariants_BeginCommitVariants( _
    ByVal bRecovery As Boolean)

End Sub

Private Sub ICrmCompensatorVariants_BeginPrepareVariants()
End Sub

Private Function ICrmCompensatorVariants_CommitRecordVariants( _
    pLogRecord As Variant) As Boolean

    ICrmCompensatorVariants_CommitRecordVariants = False
End Function

Private Sub ICrmCompensatorVariants_EndAbortVariants()
End Sub

Private Sub ICrmCompensatorVariants_EndCommitVariants()
End Sub

Private Function ICrmCompensatorVariants_EndPrepareVariants() _
    As Boolean

    ICrmCompensatorVariants_EndPrepareVariants = True
End Function

Private Function ICrmCompensatorVariants_PrepareRecordVariants( _
    pLogRecord As Variant) As Boolean

End Function

Private Sub ICrmCompensatorVariants_SetLogControlVariants( _
```

```
    ByVal pLogControl As COMSVCSLib.ICrmLogControl)
    Set CrmLogControl = pLogControl
End Sub
```

Upgrading this component with the upgrade wizard results in the following Visual Basic .NET code.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest Implements COMSVCSLib.ICrmCompensatorVariants

    Dim CrmLogControl As COMSVCSLib.ICrmLogControl

    Private Function ICrmCompensatorVariants_AbortRecordVariants( _
        ByRef pLogRecord As Object) As Boolean Implements _
        COMSVCSLib.ICrmCompensatorVariants.AbortRecordVariants
    End Function

    Private Sub ICrmCompensatorVariants_BeginAbortVariants(_
        ByVal bRecovery As Boolean) Implements _
        COMSVCSLib.ICrmCompensatorVariants.BeginAbortVariants

    End Sub

    Private Sub ICrmCompensatorVariants_BeginCommitVariants(_
        ByVal bRecovery As Boolean) Implements _
        COMSVCSLib.ICrmCompensatorVariants.BeginCommitVariants

    End Sub

    Private Sub ICrmCompensatorVariants_BeginPrepareVariants() _
        Implements _
        COMSVCSLib.ICrmCompensatorVariants.BeginPrepareVariants

    End Sub

    Private Function ICrmCompensatorVariants_CommitRecordVariants( _
        ByRef pLogRecord As Object) As Boolean Implements _
        COMSVCSLib.ICrmCompensatorVariants.CommitRecordVariants

        ICrmCompensatorVariants_CommitRecordVariants = False
    End Function

    Private Sub ICrmCompensatorVariants_EndAbortVariants() Implements _
        COMSVCSLib.ICrmCompensatorVariants.EndAbortVariants
    End Sub

    Private Sub ICrmCompensatorVariants_EndCommitVariants() _
        Implements COMSVCSLib.ICrmCompensatorVariants.EndCommitVariants
    End Sub

    Private Function ICrmCompensatorVariants_EndPrepareVariants() _
```

```
        As Boolean Implements _
        COMSVCSLib.ICrmCompensatorVariants.EndPrepareVariants

        ICrmCompensatorVariants_EndPrepareVariants = True
    End Function

    Private Function ICrmCompensatorVariants_PrepareRecordVariants( _
        ByRef pLogRecord As Object) As Boolean Implements _
        COMSVCSLib.ICrmCompensatorVariants.PrepareRecordVariants
    End Function

    Private Sub ICrmCompensatorVariants_SetLogControlVariants( _
        ByVal pLogControl As COMSVCSLib.ICrmLogControl) Implements _
        COMSVCSLib.ICrmCompensatorVariants.SetLogControlVariants

        CrmLogControl = pLogControl
    End Sub
End Class
```

► **To complete the upgrade**

1. Add the EnterpriseServices reference to the project and the following Imports statement to the class.

```
Imports System.EnterpriseServices.CompensatingResourceManager
```

2. Next, change the class to inherit from the Compensator class.

```
Inherits Compensator
```

   The Compensator class is the base class for all Compensating Resource Manager (CRM) Compensators.

3. Delete all of the COMSVCSLib references from the project.
4. Replace methods according to Table 15.1.

Table 15.1: Visual Basic 6.0 Method Equivalents in the System.EnterpriseServices.CompensatingResourceManger Namespace

Visual Basic 6.0 Methods	Equivalent .NET System.EnterpriseServices. CompensatingResourceManager Methods
ICrmCompensatorVariants_ AbortRecordVariants	Compensator.AbortRecord
ICrmCompensatorVariants_ BeginAbortVariants	Compensator.BeginAbort
ICrmCompensatorVariants_ BeginCommitVariants	Compensator.BeginCommit

Visual Basic 6.0 Methods	Equivalent .NET System.EnterpriseServices.CompensatingResourceManager Methods
ICrmCompensatorVariants_BeginPrepareVariants	Compensator.BeginPrepare
ICrmCompensatorVariants_CommitRecordVariants	Compensator.CommitRecord
ICrmCompensatorVariants_EndAbortVariants	Compensator.EndAbort
ICrmCompensatorVariants_EndCommitVariants	Compensator.EndCommit
ICrmCompensatorVariants_EndPrepareVariants	Compensator.EndPrepare
ICrmCompensatorVariants_PrepareRecordVariants	Compensator.PrepareRecord
ICrmCompensatorVariants_SetLogControlVariants	Not supported; use the Compensator read-only property Clerk.
COMSVCSLib.ICrmLogControl	No need to upgrade; use the Compensator read-only property Clerk

5. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

The following code is the final Visual Basic .NET Compensator.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices.CompensatingResourceManager

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits Compensator

    Public Overrides Function AbortRecord(ByVal rec As LogRecord) _
        As Boolean

    End Function

    Public Overrides Sub BeginAbort(ByVal fRecovery As Boolean)
    End Sub
```

```
    Public Overrides Sub BeginCommit(ByVal fRecovery As Boolean)
    End Sub

    Public Overrides Sub BeginPrepare()
    End Sub

    Public Overrides Function CommitRecord(ByVal rec As LogRecord) _
        As Boolean

        CommitRecord = False
    End Function

    Public Overrides Sub EndAbort()
    End Sub

    Public Overrides Sub EndCommit()
    End Sub

    Public Overrides Function EndPrepare() As Boolean
        EndPrepare = True
    End Function

    Public Overrides Function PrepareRecord(ByVal rec As LogRecord) _
        As Boolean

    End Function
End Class
```

COM+ Object Pooling

Components that are developed using Visual Basic 6.0 cannot be pooled because COM+ object pooling requires multi-threaded apartment (MTA) components, and Visual Basic 6.0 components use the single-threaded apartment model. However, a Visual Basic 6.0 application can detect whether an object can be pooled or not, using the IObjectControl interface.

In Visual Basic 6.0, the ObjectControl class provides the same functionality as IObjectControl. In Visual Basic .NET, the ServicedComponent class in the System.EnterpriseServices namespace provides similar functionality. Therefore, a Visual Basic .NET class that inherits from ServicesComponent is capable of pooling.

To make an object poolable, decorate the class with the ObjectPooling attribute and add a reference to the System.EnterpriseServices namespace. Also, pooled objects may only run in server applications — they may not be configured to run as a library. In the following Visual Basic .NET class, the object pooling minimal size is 2 and the maximum size is 4.

```
Imports System.EnterpriseServices
Imports System.Windows.Forms
```

```
<ObjectPooling(2, 4), _
    System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits ServicedComponent
    Public counter As Integer = 0

    <AutoComplete()> Public Sub IncrementCounter() As Integer
        counter = counter + 1
        IncrementCounter = counter
    End Sub

    Protected Overrides Function CanBePooled() As Boolean
        Return True
    End Function
End Class
```

Under normal circumstances, an object that was not pooled in Visual Basic 6.0 does not need to be pooled in Visual Basic .NET. Doing this should be considered a new feature over and above that required for functional equivalence and should be tested thoroughly.

COM+ Application Security

COM+ applications can provide controlled access to resources. Using roles, developers can administratively construct an authorization policy for an application, choosing which users can access which resources, even down to the method level, if necessary. Additionally, roles provide a framework for enforcing security-checking within code if an application requires finer-grained access control.

Role-based security is built on a general mechanism that enables developers to retrieve security information that is related to all upstream callers in the chain of calls to one particular component.

Visual Basic 6.0 developers can handle user-role information using the SecurityCallContext class. In Visual Basic 6.0, developers can access the members of SecurityCallContext without declaring a specific instance of this type, by using the GetSecurityCallContext global function provided by the IGetSecurityCallContext interface and accessing the object returned.

Visual Basic .NET developers can get similar functionality from GetSecurityCallContext through the CurrentCall property of a static instance of System.EnterpriseServices.SecurityCallContext.

Visual Basic 6.0 developers can use SecurityCallers and SecurityIdentity to manipulate information about a caller in a secured application. In Visual Basic .NET, the SecurityCallers and SecurityIdentity classes are included in the System.EnterpriseServices namespace and provide similar functionality.

Upgrading a Visual Basic 6.0 component that uses security objects is a fairly straightforward procedure. The following code example lists a Visual Basic 6.0 component that checks whether the security feature is enabled or disabled.

```
Public Function SecurityEnabled(ByVal caller As String) As Boolean
    Dim SecContext As SecurityCallContext

    On Error GoTo ErrorHandler

    Set SecContext = GetSecurityCallContext
    SecurityEnabled = SecContext.IsSecurityEnabled
Exit Function

ErrorHandler:
    ' Error handling code goes here.
End Function
```

Upgrading this component with the upgrade wizard results in the following Visual Basic .NET code.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest
    Public Function SecurityEnabled(ByVal caller As String) _
        As Boolean

        Dim SecContext As COMSVCSLib.SecurityCallContext

        On Error GoTo ErrorHandler

        SecContext = _
            COMSVCSLibGetSecurityCallContextAppObject_definst. _
            GetSecurityCallContext
        SecurityEnabled = SecContext.IsSecurityEnabled
    Exit Function

    ErrorHandler:
        ' Error handling code goes here.
    End Function
End Class
```

► To complete the upgrade

1. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

   ```
   Imports System.EnterpriseServices
   ```

2. Change the class to inherit from the ServicedComponent class.

   ```
   Inherits ServicedComponent
   ```

3. Delete all COMSVCSLib references from the project.
4. Replace the Visual Basic 6.0 SecurityCallContext object with the .NET System.EnterpriseServices.SecurityCallContext.
5. Replace the Visual Basic 6.0 GetSecurityCallContext method with the .NET static method SecurityCallContext.CurrentCall.

```
System.EnterpriseServices.SecurityCallContext.CurrentCall()
```

6. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

The following example shows the final code.

```
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest
  Inherits ServicedComponent
  Public Function SecurityEnabled(ByVal caller As String) As Boolean

      Dim SecContext As System.EnterpriseServices.SecurityCallContext

      On Error GoTo ErrorHandler

      SecContext =
          System.EnterpriseServices.SecurityCallContext.CurrentCall()
      SecurityEnabled = SecContext.IsSecurityEnabled
    Exit Function

    ErrorHandler:
        ' Error handling code goes here.
    End Function
End Class
```

COM+ Shared Property Manager

The COM+ Shared Property Manager (SPM) can be used to manage shared transient state for objects. Global variables cannot be used in a distributed environment because of concurrency and name collision issues. The SPM eliminates name collisions by providing shared property groups, which establish unique namespaces for the shared properties they contain. It also implements locks and semaphores as synchronization mechanisms.

On single processor servers with small numbers of clients, the Shared Property Manager works quite well as a mechanism for storing state. However, it should be

noted that it does not scale well and applications that expect frequent concurrent users should instead use a database to store state information.

In Visual Basic 6.0, the classes for implementing SPM-related functionality are the SharedPropertyManager, SharedPropertyGroup, and SharedProperty classes from the COM+ Services Type Library. Visual Basic .NET counterparts for these classes can be clearly identified with the same names, and are included in the System.EnterpriseServices namespace.

The following Visual Basic 6.0 component uses SPM objects.

```
Public Function SPMTest(ByVal strGrpName As String, _
    ByVal strPrpName As String, ByVal vntPrpValue As String, _
    ByRef blnExists As Boolean)

    Dim spmMgr As COMSVCSLib.SharedPropertyGroupManager
    Dim spmGrp As COMSVCSLib.SharedPropertyGroup
    Dim spmPrp As COMSVCSLib.SharedProperty

    Set spmMgr = New COMSVCSLib.SharedPropertyGroupManager
    Set spmGrp = spmMgr.CreatePropertyGroup(strGrpName, LockSetGet, _
        Process, blnExists)
    Set spmMgr = Nothing
    Set spmPrp = spmGrp.CreateProperty(strPrpName, blnExists)

    spmPrp.Value = vntPrpValue
    SPMTest = spmPrp.Value

    Set spmPrp = Nothing
    Set spmGrp = Nothing
End Function
```

After you upgrade the component, the class contains several statements that need to be resolved using the Security classes provided by the .NET Framework inside the System.EnterpriseServices namespace.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest
    Public Function SPMTest(ByVal strGrpName As String, _
        ByVal strPrpName As String, ByVal vntPrpValue As String, _
        ByRef blnExists As Boolean) As Object

    Dim spmMgr As COMSVCSLib.SharedPropertyGroupManager
    Dim spmGrp As COMSVCSLib.SharedPropertyGroup
    Dim spmPrp As COMSVCSLib.SharedProperty

    spmMgr = New COMSVCSLib.SharedPropertyGroupManager

    spmGrp = spmMgr.CreatePropertyGroup(strGrpName, _
        COMSVCSLib.__MIDL___MIDL_itf_autosvcs_0408_0002.LockSetGet, _
        COMSVCSLib.__MIDL___MIDL_itf_autosvcs_0408_0003.Process, _
```

```
        blnExists)

    ' UPGRADE_NOTE: Object spmMgr may not be destroyed until it is
    ' garbage collected.
    spmMgr = Nothing

    spmPrp = spmGrp.CreateProperty(strPrpName, blnExists)

    spmPrp.Value = vntPrpValue

    ' UPGRADE_WARNING: Could not resolve default property of
    ' object SPMTest
    SPMTest = spmPrp.Value

    ' UPGRADE_NOTE: Object spmPrp may not be destroyed until it is
    ' garbage collected.
    spmPrp = Nothing

    ' UPGRADE_NOTE: Object spmGrp may not be destroyed until it is
    ' garbage collected.
    spmGrp = Nothing
    End Function
End Class
```

As mentioned earlier, the name of the classes used in the SPM infrastructure are the same in .NET.

► **To complete the upgrade**

1. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

   ```
   Imports System.EnterpriseServices
   ```

2. Change the class to inherit from the ServicedComponent class.

   ```
   Inherits ServicedComponent
   ```

3. Delete all of the COMSVCSLib references from the project.
4. Replace the Visual Basic 6.0 LockSetGet constant with its .NET equivalent, PropertyLockMode.SetGet.

   ```
   System.EnterpriseServices.PropertyLockMode.SetGet
   ```

5. Replace the Visual Basic 6 Process constant with its .NET equivalent, PropertyReleaseMode.Process.

   ```
   System.EnterpriseServices.PropertyReleaseMode.Process
   ```

6. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

The following example shows the final code.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits ServicedComponent

    Public Function SPMTest(ByVal strGrpName As String, _
        ByVal strPrpName As String, ByVal vntPrpValue As String, _
        ByRef blnExists As Boolean) As Object

        Dim spmMgr As SharedPropertyGroupManager
        Dim spmGrp As SharedPropertyGroup
        Dim spmPrp As SharedProperty

        spmMgr = New SharedPropertyGroupManager
        spmGrp = spmMgr.CreatePropertyGroup(strGrpName, _
            PropertyLockMode.SetGet, PropertyReleaseMode.Process, _
            blnExists)
        spmMgr = Nothing
        spmPrp = spmGrp.CreateProperty(strPrpName, blnExists)
        spmPrp.Value = vntPrpValue

        SPMTest = spmPrp.Value

        spmPrp = Nothing
        spmGrp = Nothing
    End Function
End Class
```

COM+ Object Constructor Strings

COM+ object constructor strings are initialization strings that are administratively specified for a component. These object constructor strings can be used to write a single component with a degree of generality that allows it to be later customized for a particular task; in other words, it allows you to create parameterized object constructors.

Visual Basic 6.0 developers can use this COM+ feature by implementing the IObjectConstruct and IObjectConstructString interfaces. In Visual Basic .NET, the ServicedComponent class in the System.EnterpriseServices namespace provides the Construct method, which provides a similar functionality. A Visual Basic .NET class that supports construction strings must inherit from System.EnterpriseServices.ServicedComponent. You can emulate the Construct method from IObjectConstruct by overriding the Construct method from the ServicedComponent class.

The following Visual Basic 6.0 component implements the IObjectConstruct interface.

```
Implements COMSVCSLib.IObjectConstruct

Private MyConnString As String

Private Sub IObjectConstruct_Construct(ByVal objConstructor As Object)

    Dim cs As COMSVCSLib.IObjectConstructString
    Set cs = objConstructor
    MyConnString = cs.ConstructString

End Sub
```

Upgrading this component with the upgrade wizard results in the following Visual Basic .NET code.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest_
    Implements COMSVCSLib.IObjectConstruct

    Private MyConnString As String

    Private Sub IObjectConstruct_Construct(ByVal objConstructor _
        As Object) Implements COMSVCSLib.IObjectConstruct.Construct

        Dim cs As COMSVCSLib.IObjectConstructString
        cs = objConstructor
        MyConnString = cs.ConstructString

    End Sub
End Class
```

► To complete the upgrade

1. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

   ```
   Imports System.EnterpriseServices
   ```

2. Change the class to inherit from the ServicedComponent class.

```
Inherits ServicedComponent
```

3. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

4. Follow either the construct approach or the attribute approach, or both, as explained in the next section.

Construct Approach

The ServicedComponent class implements the IObjectConstruct interface as a virtual method. Your derived serviced component can override the Construct method, as shown in this code example.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits ServicedComponent

    Private MyConnString As String

    Protected Overrides Sub Construct(ByVal objConstructor As String)
        MyConnString = objConstructor
    End Sub

End Class
```

If the Enable object construction check box on the component Activation tab is selected, the Construct method is called after the component's constructor is called. This provides the component with the configured construction string.

Attribute Approach

You can also enable construction string support and provide a default construction string using the ConstructionEnabled attribute.

```
Option Strict Off
Option Explicit On
```

```
Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest"), _
    ConstructionEnabled(True, [Default]:="My string")> _
    Public Class COMTest
    Inherits ServicedComponent

End Class
```

The ConstructionEnabled attribute has two public properties: Enabled, which enables construction string support for your serviced component in the Component Services Explorer after the component is registered, and Default, which provides an initial string value.

COM+ Transactions

In Visual Basic 6.0, you can use the MTSTransactionMode property to set the transactional behavior of a user class. This property is only used by components that are running in the Microsoft Transaction Server, and has no effect if the component is run outside of the MTS.

In Visual Basic .NET, developers can use System.EnterpriseServices namespace to get transactional functionality for their .NET classes. To be treated as a transactional class, a Visual Basic .NET class should inherit from ServicedComponent and include the proper transaction attribute. Transactional Visual Basic 6.0 classes should be upgraded to serviced components.

The MTSTransactionMode values correspond to the System.EnterpriseServices.TransactionOption .NET Framework enumeration that contains the automatic transaction type requested by a COM+ component. The MTSTransactionMode property can be set to any of the constant values specified in Table 15.2, which presents the equivalences between both groups of values.

Table 15.2: MTSTransactionMode Value Equivalents in the System.EnterpriseServices.TransactionOption Namespace

MTSTransactionMode value	System.EnterpriseServices.TransactionOption equivalent
NotAnMTSObject (0)	Disabled
NoTransactions (1)	NotSupported
RequiresTransaction (2)	Required
UsesTransaction (3)	Supported
RequiresNewTransaction (4)	RequiresNew

Use the following information to create a Visual Basic .NET project that references transactional classes:

- Have transactional classes inherit from ServicedComponent.
- Include transactional attributes in transactional classes.
- Eliminate calls to GetObjectContext; instead use the ContextUtil class.

The Visual Basic 6.0 MTSTransactionMode attribute controls whether transactions are required for a class. This attribute should be mapped to the corresponding value from System.EnterpriseServices.TransactionOption.

All COM+ transaction related classes are supported by Visual Basic 6.0, and almost every class has an equivalent class in the .NET Framework. The only exception is the ITransaction interface, which does not have a semantic equivalent in Visual Basic .NET.

Adding transaction support to the simple COM+ server component as described in the "Using MTS/COM+ in Visual Basic 6.0" section earlier in this chapter results in the following Visual Basic 6.0 code:

Notice that the MTSTransactionMode property value is set to RequiresTransaction and that the name of the Server is COMTest.

```
Private Sub transferfunds(acc1 As String, acc2 As String, _
    amount As Long)
    On Error GoTo ErrHandler

    withdrawFunds acc1, amount
    depositFunds acc2, amount
    GetObjectContext.SetComplete

    Exit Sub
ErrHandler:
    GetObjectContext.SetAbort
    Err.Raise Err.Number, Err.Source, Err.Description, _
        Err.HelpFile, Err.HelpContext
End Sub

Private Sub withdrawFunds(acc As String, amount As Long)
   ' Perform withdrawal...
End Sub

Private Sub depositFunds(acc As String, amount As Long)
   ' Perform deposit...
End Sub
```

Upgrading this server component with the upgrade wizard results in the following Visual Basic .NET code.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
Public Class COMTest
    Public Sub transferfunds(ByRef acc1 As String, _
        ByRef acc2 As String, ByRef amount As Integer)
        On Error GoTo ErrHandler

        withdrawFunds(acc1, amount)
        depositFunds(acc2, amount)
        ' UPGRADE_WARNING: Could not resolve default property of
        ' object GetObjectContext.SetComplete.
        GetObjectContext.SetComplete()

        Exit Sub
    ErrHandler:
        ' UPGRADE_WARNING: Could not resolve default property of
        ' object GetObjectContext.SetAbort.
        GetObjectContext.SetAbort()
        Err.Raise(Err.Number, Err.Source, Err.Description, _
           Err.HelpFile, Err.HelpContext)
    End Sub

    Private Sub withdrawFunds(ByRef acc As String, _
        ByRef amount As Integer)
        ' Perform withdrawal...
    End Sub

    Private Sub depositFunds(ByRef acc As String, _
         ByRef amount As Integer)
        ' Perform deposit...
    End Sub
End Class
```

► To complete the upgrade

1. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

```
Imports System.EnterpriseServices
```

2. Change the class to inherit from the ServicedComponent class.

```
Inherits ServicedComponent
```

3. Use one of the following two approaches to resolve the GetObjectContext warnings:
   a. The first approach is to use the System.EnterpriseServices.ContextUtil class, which is the .NET Framework equivalent of GetObjectContext. The ContextUtil class obtains information about the COM+ object context.

      ```
      ContextUtil.SetComplete()
      ContextUtil.SetAbort()
      ```

   b. The second approach is to use System.EnterpriseServices.AutoCompleteAttribute. This attribute marks the attributed method as an AutoComplete object. The transaction automatically calls SetComplete if the method call returns normally. If the method call throws an exception, the transaction is aborted.
4. Add the TransactionOption.Supported attribute to the .NET class. You do this because the Visual Basic 6.0 server component has the MTSTransactionMode property set to RequiresTransaction.

   ```
   System.EnterpriseServices.Transaction(TransactionOption.Supported)
   ```

5. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

   ```
   sn -k COMKey.snk
   ```

The final Visual Basic .NET code, shown here, lists a solution that uses the ContextUtil class.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest"), _
    System.EnterpriseServices.Transaction(TransactionOption.Supported)> _
    Public Class COMTest
    Inherits ServicedComponent

    Public Sub transferfunds(ByRef acc1 As String, _
        ByRef acc2 As String, ByRef amount As Integer)
        On Error GoTo ErrHandler

        withdrawFunds(acc1, amount)
        depositFunds(acc2, amount)
        ContextUtil.SetComplete()
```

```
        Exit Sub
    ErrHandler:
        ContextUtil.SetAbort()
        Err.Raise(Err.Number, Err.Source, Err.Description, _
           Err.HelpFile, Err.HelpContext)
    End Sub

    Private Sub withdrawFunds(ByRef acc As String, _
        ByRef amount As Integer)
        ' Perform withdrawal...
    End Sub

    Private Sub depositFunds(ByRef acc As String, _
        ByRef amount As Integer)
        ' Perform deposit...
    End Sub
End Class
```

The following code example lists a solution that uses the AutoComplete attribute.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest"), _
    System.EnterpriseServices.Transaction(TransactionOption.Supported)> _
    Public Class COMTest
    Inherits ServicedComponent

    <AutoComplete()> Public Sub transferfunds(ByRef acc1 As String, _
        ByRef acc2 As String, ByRef amount As Integer)
        On Error GoTo ErrHandler

        withdrawFunds(acc1, amount)
        depositFunds(acc1, amount)

        Exit Sub
    ErrHandler:
        Err.Raise(Err.Number, Err.Source, Err.Description, _
           Err.HelpFile, Err.HelpContext)
    End Sub

    Private Sub withdrawFunds(ByRef acc As String, _
        ByRef amount As Integer)
        ' Perform withdrawal...
    End Sub

    Private Sub depositFunds(ByRef acc As String, ByRef amount As Integer)
        ' Perform deposit...
    End Sub
End Class
```

Additional COM+ Functionality

Some features of COM+ in Visual Basic 6.0 are not supported by the upgrade wizard. When these features are encountered by the upgrade wizard, they are not automatically upgraded and instead require manual effort to upgrade.

Table 15.3 lists Visual Basic 6.0 COM+ objects that have equivalent objects in Visual Basic .NET that you can use to upgrade them. Other objects require more extensive modification than an object replacement.

Table 15.3: Visual Basic .NET Equivalents for Visual Basic 6.0 COM+ Objects

Visual Basic 6.0 object	Visual Basic .NET equivalent
Service Configuration: CserviceConfig	System.EnterpriseServices.ServiceConfig
Context: IMTxAS	System.EnterpriseServices.ContextUtil
Exceptions and Errors: Error_Constants	Use these classes from the System.EnterpriseServices namespace: RegistrationErrorInfo RegistrationException ServicedComponentException
COM+ Application Types: tagCOMPLUS_APPTYPE	System.Runtime.InteropServices.IMoniker

Table 15.4 provides pointers on where to find information about how to upgrade these objects.

Table 15.4: Pointers to Information for Upgrading Visual Basic 6.0 COM+ Objects

To upgrade these Visual Basic 6.0 objects:	See the following sections in this chapter:
COM+ Events: COMEvents, ComServiceEvents, COMSVCSEVENTINFO, ComSystemAppEventData, IEventServerTrace, IMtsEventInfo, CoMTSLocator, MtsGrp	"COM+ Events" later in this chapter.
Compensating Resource Manager (CRM): ICrmFormatLogRecords, ICrmMonitorLogRecords, CRMRecoveryC!lerk	"COM+ Compensating Resource Manager" earlier in this chapter
Security: IsecurityProperty, SecurityCertificate, SecurityIdentity, SecurityProperty	"COM+ Application Security" earlier in this chapter and "COM+ Security" following this section.
SOAP: SoapMoniker	"Using SOAP Services" earlier in this chapter.

To upgrade these Visual Basic 6.0 objects:	See the following sections in this chapter:
Transactions: TransactionContext, TransactionContextEx	"Context Components" later in this chapter
MSMQ: MessageMover	"Message Queuing and Queued Components" later in this chapter
Pooling: PoolMgr	"COM+ Object Pooling" earlier in this chapter

COM+ Security

COM+ provides several security features that you can use to help protect COM+ applications. The following Visual Basic 6.0 component uses the SecurityCallContext and SecurityIdentity objects to obtain information about the direct component caller. The method GetSecurityCallContext returns the current security context that is used to obtain the direct caller.

```
Public Function DirectCallerInfo() As String
    Dim ctx As SecurityCallContext
    Dim idx As SecurityIdentity
    Dim AccName, AuServ, ImpL, AuLev As String

    Set ctx = GetSecurityCallContext
    Set idx = ctx.Item("DirectCaller")

    AccName = "Account Name: " & idx("AccountName") & " – "
    AuServ = "Authentication Service: " & CLng(idx("AuthenticationService")) & _
        " –– "
    ImpL = "Impersonation Level: " & CLng(idx("ImpersonationLevel")) & " – "
    AuLev = "Authentication Level: " & CLng(idx("AuthenticationLevel"))

    DirectCallerInfo = AccName & AuServ & ImpL & AuLev
End Function
```

The upgraded Visual Basic .NET code has several warnings and references to the COMSVCSLib library. You must resolve these issues to obtain a functional .NET assembly that can be exported as a COM+ application.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> Public Class
COMTest
    Public Function DirectCallerInfo() As String
        Dim ctx As COMSVCSLib.SecurityCallContext
        Dim idx As COMSVCSLib.SecurityIdentity
        Dim AuServ, AccName, ImpL As Object
        Dim AuLev As String
```

```
        ctx = _
           COMSVCSLibGetSecurityCallContextAppObject_definst.GetSecurityCallContext
        idx = ctx.Item("DirectCaller")

        ' UPGRADE_WARNING: Could not resolve default property of object idx().
        ' UPGRADE_WARNING: Could not resolve default property of object AccName.
        AccName = "Account Name: " & idx("AccountName") & " - "
        ' UPGRADE_WARNING: Could not resolve default property of object idx().
        ' UPGRADE_WARNING: Could not resolve default property of object AuServ.
        AuServ = "Authentication Service: " & CInt(idx("AuthenticationService")) &
_
           " - "
        ' UPGRADE_WARNING: Could not resolve default property of object idx().
        ' UPGRADE_WARNING: Could not resolve default property of object ImpL.
        ImpL = "Impersonation Level: " & CInt(idx("ImpersonationLevel")) & " - "
        ' UPGRADE_WARNING: Could not resolve default property of object idx().
        AuLev = "Authentication Level: " & CInt(idx("AuthenticationLevel"))

        ' UPGRADE_WARNING: Could not resolve default property of object ImpL.
        ' UPGRADE_WARNING: Could not resolve default property of object AuServ.
        ' UPGRADE_WARNING: Could not resolve default property of object AccName.
        DirectCallerInfo = AccName & AuServ & ImpL & AuLev
    End Function
End Class
```

► To create a functionally equivalent component

1. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

```
Imports System.EnterpriseServices
```

2. Change the class to inherit from the ServicedComponent class.

```
Inherits ServicedComponent
```

3. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

5. Remove all references to COMSVCSLib. (Because the names of the COMSVCSLib security classes are the same inside the EnterpriseServices namespace, you do not need to modify them.)

6. Use the SecurityCallContext.CurrentCall static method to obtain the current security context.

```
System.EnterpriseServices.SecurityCallContext.CurrentCall
```

7. Use the DirectCaller to return an instance of that object.

```
SecurityCallContext.DirectCaller
```

8. Obtain the account name, authentication service, impersonation level, and authentication level values from their respective properties inside the SecurityIdentity class.

```
SecurityIdentity.AccountName
SecurityIdentity.AuthenticationService
SecurityIdentity.ImpersonationLevel
SecurityIdentity.AuthenticationLevel
```

After applying these steps to the example, the code should appear similar to the following.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices

<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits ServicedComponent
    Public Function DirectCallerInfo() As String
        Dim ctx As SecurityCallContext
        Dim idx As SecurityIdentity
        Dim AuServ, AccName, ImpL, AuLev As String

        ctx = SecurityCallContext.CurrentCall
        idx = ctx.DirectCaller

        AccName = "Account Name: " & idx.AccountName & " - "
        AuServ = "Authentication Service: " & idx.AuthenticationService.ToString()
_
            & " - "
        ImpL = "Impersonation Level: " & idx.ImpersonationLevel.ToString & " - "
        AuLev = "Authentication Level: " & idx.AuthenticationLevel.ToString

        DirectCallerInfo = AccName & AuServ & ImpL & AuLev
    End Function
End Class
```

Context Components

This section explains the transaction context object, and how to upgrade the Commit and Abort methods to use in Visual Basic .NET.

This Visual Basic 6.0 component uses the TransactionContext object to create an instance of the .NET security component whose upgrade was discussed in the previous section, "COM+ Security." If no errors occur during the transaction, the changes will be committed. Otherwise, the transaction will be aborted.

```
Public Function TestTC() As String
    Dim ctx As TransactionContext
    Dim Com As Object
    Dim str As String

    On Error GoTo ErrorHandler

    Set ctx = New TransactionContext
    Set Com = ctx.CreateInstance("COMTest_NET.COMTest")
    str = Com.DirectCallerInfo
    ctx.Commit
    TestTC = str
Exit Function

ErrorHandler:
    ctx.Abort
End Function
```

Using the upgrade wizard to upgrade this server component results in the following Visual Basic .NET code.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> Public Class
COMTest
    Public Function TestTC() As String
        Dim ctx As COMSVCSLib.TransactionContext
        Dim Com As Object
        ' UPGRADE_NOTE: str was upgraded to str_Renamed.
        Dim str_Renamed As String

        On Error GoTo ErrorHandler

        ctx = New COMSVCSLib.TransactionContext
        Com = ctx.CreateInstance("COMTest_NET.COMTest")
        ' UPGRADE_WARNING: Could not resolve default property of object
        ' Com.DirectCallerInfo.
        str_Renamed = Com.DirectCallerInfo
        ctx.Commit()
        TestTC = str_Renamed
    Exit Function
```

```
    ErrorHandler:
        ctx.Abort()
    End Function
End Class
```

► To complete the upgrade

1. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

```
Imports System.EnterpriseServices
```

2. Change the class to inherit from the ServicedComponent class.

```
Inherits ServicedComponent
```

3. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

5. Remove all references to COMSVCSLib.
6. Remove the following statements.

```
Dim ctx As COMSVCSLib.TransactionContext
ctx = New COMSVCSLib.TransactionContext
```

7. Use the CreateObject method to instantiate the Visual Basic 6.0 Security component that was created in the previous section, "COM+ Security."

```
CreateObject("Test.COMTest")
```

8. Replace the TransactionContext Commit and Abort methods with the System.EnterpriseServices.ContextUtil SetComplete and SetAbort methods respectively.

```
ContextUtil.SetComplete()
ContextUtil.SetAbort()
```

The final Visual Basic .NET code is shown here.

```
Option Strict Off
Option Explicit On

Imports System.EnterpriseServices
```

```
<System.Runtime.InteropServices.ProgId("COMTest_NET.COMTest")> _
    Public Class COMTest
    Inherits ServicedComponent

    Public Function TestTC() As String
        Dim Com As Object
        Dim str_Renamed As String

        On Error GoTo ErrorHandler

        Com = CreateObject("Test.COMTest")
        str_Renamed = Com.DirectCallerInfo
        ContextUtil.SetComplete()
        TestTC = str_Renamed
        Exit Function

ErrorHandler:
        ContextUtil.SetAbort()
    End Function
End Class
```

COM+ Events

The COM+ event system introduces Visual Basic programmers to the concept of a loosely coupled event (LCE) system in which event consumers (called subscribers) do not have to declare a variable against the event provider (called a publisher). Instead, the subscriber and publisher both rely on an interface definition in the form of a COM+ event class.

Figure 15.1 illustrates a COM+ event system concept.

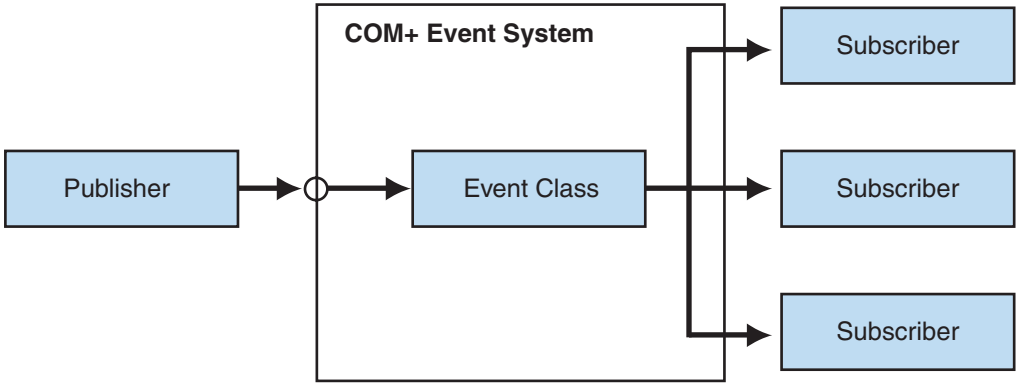

Figure 15.1
COM+ event system conceptual diagram

As event classes define the interface that the publisher calls and the subscriber implements they form the primary mechanism for decoupling the event. You must install the event component inside the Component Services dialog box by selecting the Install new event class(es) option.

After you install the event class, you must create a subscriber to receive the new event and implement the interface defined by the event class.

The publisher (the object that performs the event publication) can be a standard module, dynamic link library, or a regular COM+ application, which is a COM+ application that you create by selecting the Install new class(es) option inside the Component Service dialog box. The user application in the client tier is the subscriber that will capture the event later.

The following scenario illustrates the upgrade process of a Visual Basic 6.0 COM+ event.

The Event Component

You define an event class in Visual Basic as you would any other interface: simply define the function signature for the events you want to raise. In fact, event classes are nothing more than interfaces built into an abstract class. The module must contain a reference to COM+ Services Type Library, as shown in this code example.

```
Public Sub ContactInfoUpdated(name As String)
End Sub
```

The Event Publisher

Creating a publisher is also a simple process. To raise an event, the publisher creates an instance of the event class and calls the exposed method. You must add to the project references to EventComponent and the COM+ Services Type Library. You must also be sure the publisher is hosted inside the COM+ catalog as a COM+ application.

```
Public Sub UpdateContact(name As String)
    ' Update ContactInfo
    ' ...

    Dim e As EventComponent.EventClass
    Set e = CreateObject("EventComponent.EventClass")
    e.ContactInfoUpdated (name)
End Sub
```

The Event Subscriber and Test

After the event class is installed, you must create a subscriber to receive the new event and implement the interface defined by the event class. In this implementation, the subscriber defines the actions for the event.

In addition to the subscriber, the following code contains statements to test the event. The project must be an EXE application with two buttons.

► **To make the project and EXE application with two buttons**

1. Use the following code to subscribe to the event.

```
Private Sub Command2_Click()
    Subscribe
End Sub
```

2. Use the following code to test the event.

```
Private Sub Command1_Click()
    Update "joe"
End Sub
```

3. Use the following code to add to the project references to EventComponent, PublisherComponent, COM+ Services Type Library, and COM+ Admin Type Library.

```
Implements EventComponent.EventClass

Private Sub Command1_Click()
    Update "joe"
End Sub

Private Sub Command2_Click()
    Subscribe
End Sub

Public Sub Update(name As String)
    Dim COM As Object
    Set COM = CreateObject("PublisherComponent.Publisher")
    COM.UpdateContact (name)
End Sub

Public Sub Subscribe()
    Dim subscriptions As ICatalogCollection
    Dim s As ICatalogObject
    Dim comAdm As COMAdmin.COMAdminCatalog

    Set comAdm = New COMAdmin.COMAdminCatalog
    Set subscriptions = comAdm.GetCollection("TransientSubscriptions")
    Set s = subscriptions.Add
    ' For illustrative purposes the CLSID value is hard coded in this example.
```

```
        ' This CLSID value is generated when the library that contains the Event
        ' class is registered. After the value is generated it can be read
        ' from the registry.
        s.Value("EventCLSID") = "{2D080D19-C950-4B0A-9009-67C30CF5DCEA}"
        s.Value("Name") = "Form subscription"
        s.Value("SubscriberInterface") = Me
        subscriptions.SaveChanges
    End Sub

    Private Sub EventClass_ContactInfoUpdated(name As String)
        MsgBox name
    End Sub
```

Upgrading each module with the upgrade wizard results in the following Visual Basic .NET code.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("EventClass_NET.EventClass")> Public Class
EventClass
    Public Sub ContactInfoUpdated(ByRef name As String)
    End Sub
End Class
```

The code for the event publisher is shown here.

```
Option Strict Off
Option Explicit On
<System.Runtime.InteropServices.ProgId("Business_NET.Business")> Public Class _
    Business
    Public Sub UpdateContact(ByRef name As String)
        Dim EventComponent As Object
        ' Update ContactInfo
        ' ...

        Dim e As EventComponent.EventClass
        e = CreateObject("EventComponent.EventClass")
        ' UPGRADE_WARNING: Could not resolve default property of object
        ' e.ContactInfoUpdated. Click for more: 'ms-
        ' help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword="vbup1037"'
        e.ContactInfoUpdated(name)
    End Sub
End Class
```

Finally, the code for the event subscriber and test is shown here.

```
Option Strict Off
Option Explicit On

Friend Class Form1
    Inherits System.Windows.Forms.Form
```

```
    Implements EventComponent.EventClass
#Region "Windows Form Designer generated code "
    ...
#End Region
#Region "Upgrade Support "
    ...
#End Region

    Private Sub Command1_Click(ByVal eventSender As System.Object, ByVal eventArgs
_
        As System.EventArgs) Handles Command1.Click
        Update_Renamed("joe")
    End Sub

    ' UPGRADE_NOTE: Update was upgraded to Update_Renamed.
    ' UPGRADE_NOTE: name was upgraded to name_Renamed.
    Public Sub Update_Renamed(ByRef name_Renamed As String)
        Dim COM As Object

        COM = CreateObject("BusinessComponent.Business")
        ' UPGRADE_WARNING: Could not resolve default property of object
        ' COM.UpdateContact.
        COM.UpdateContact(name_Renamed)

    End Sub

    Public Sub Subscribe()
        Dim subscriptions As COMAdmin.ICatalogCollection
        Dim s As COMAdmin.ICatalogObject
        Dim comAdm As COMAdmin.COMAdminCatalog

        comAdm = New COMAdmin.COMAdminCatalog
        subscriptions = comAdm.GetCollection("TransientSubscriptions")
        s = subscriptions.Add
        ' UPGRADE_WARNING: Could not resolve default property of object s.Value().
        ' For illustrative purposes the CLSID value is hardcoded in
        ' this example.
        ' This CLSID value is generated when the library that contains the Event
        ' class is registered. After the value is generated it can be read
        ' from the registry.
        s.Value("EventCLSID") = "{2D080D19-C950-4B0A-9009-67C30CF5DCEA}"
        ' UPGRADE_WARNING: Could not resolve default property of object s.Value().
        s.Value("Name") = "Form subscription"
        ' UPGRADE_WARNING: Could not resolve default property of object s.Value().
        s.Value("SubscriberInterface") = Me
        subscriptions.SaveChanges()

    End Sub

    Private Sub Command2_Click(ByVal eventSender As System.Object, ByVal eventArgs
_
        As System.EventArgs) Handles Command2.Click
        Subscribe()
    End Sub
```

```
    ' UPGRADE_NOTE: name was upgraded to name_Renamed.
    Private Sub EventClass_ContactInfoUpdated(ByRef name_Renamed As String) _
        Implements EventComponent.EventClass.ContactInfoUpdated
        MsgBox(name_Renamed)
    End Sub
End Class
```

You must modify every upgraded project to achieve the functionality of the Visual Basic 6.0 COM+ event. For the following components, this includes changing some names to ensure a better explanation of building COM+ events in .NET.

Event Interface

The Event Component, which is a Class Library, defines the event interface infrastructure. For this particular component you do not need to extend the ServicedComponent class, but it is necessary to add a strong name to the assembly.

► **To add a strong name to the assembly**

1. Use the Strong Name tool (Sn.exe) to create a key pair file for the component. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

   ```
   sn -k MyKey.snk
   ```

2. Add the AssemblyKeyFile attribute to the AssemblyInfo.vb file to link the server component with the key pair file.

   ```
   <Assembly: AssemblyKeyFile("MyKey.snk")>
   ```

The following code example shows an event component.

```
Public Interface IEvent
    Sub ContactInfoUpdated(ByVal name As String)
End Interface
```

Event Class

The next step is to create an event class that implements the previous interface as detailed in the following procedure.

► **To create an event class**

1. Add a reference to the event interface.
2. Implement the interface.
3. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

   ```
   Imports System.EnterpriseServices
   ```

4. Change the class to inherit from the ServicedComponent class.

```
Inherits ServicedComponent
```

5. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

7. Add the EventClass attribute to the class. This attribute marks the attributed class as an event class. Notice that method calls on an event class are never delivered to the implementation; instead, they are delivered to event subscribers.
8. Add the EventTrackingEnabled attribute to the class. This attribute enables event tracking for a component.

```
Imports System.EnterpriseServices

<EventClass(), EventTrackingEnabled()> _
Public Class COMEventClass
    Inherits ServicedComponent
    Implements EventInterface.IEvent

Public Sub ContactInfoUpdated(ByVal name As String) Implements
EventInterface.IEvent.ContactInfoUpdated
    End Sub
End Class
```

The Publisher

Next, create a publisher component to be hosted by a COM+ application as detailed in the following procedure.

► **To create the publisher component that is to be hosted by a COM+ application**

1. Add a reference to the event interface.
2. Add the EnterpriseServices reference to the project and a corresponding Imports statement to the class.

```
Imports System.EnterpriseServices
```

3. Change the class to inherit from the ServicedComponent class.

```
Inherits ServicedComponent
```

4. Sign the component with a strong name by adding the corresponding AssemblyKeyFile attribute to the AssemblyInfo.vb file. Generate the key file with the following command. (For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.)

```
sn -k MyKey.snk
```

The following code is an example of a Publisher class.

```
Imports System.EnterpriseServices

Public Class Publisher
    Inherits ServicedComponent

    Public Sub Update(ByVal name As String)
        Dim evt As New EventClass.COMEventClass
        evt.ContactInfoUpdated(name)
    End Sub
End Class
```

After the event and the publisher are ready, you must upgrade the subscriber/test application. It will be a Windows Forms application with the same two buttons as the Visual Basic 6.0 application.

► **To create the application**

1. Add a reference to the event interface.
2. Add a reference to the COM+ Admin Type Library.
3. Use the Type.GetTypeFromProgID method to obtain the type of the event class, and then use the GUID property to obtain the COM.

```
Type.GetTypeFromProgID().GUID
```

   The statements inside the Subscribe method use COM+ Admin Type Library objects, so you do not have to make additional changes.

The following code example shows the application code.

```
Public Class Form1
    Inherits System.Windows.Forms.Form
    Implements EventInterface.IEvent

#Region " Windows Form Designer generated code "
    ...
#End Region

#Region "Upgrade Support "
    ...
#End Region
```

```
    Public Sub Update_Renamed(ByRef name_Renamed As String)
        Dim COM As Object
        COM = CreateObject("PublisherComponent_NET.Publisher")
        COM.UpdateContact(name_Renamed)
    End Sub

    Public Sub Subscribe()
        Dim subscriptions As COMAdmin.ICatalogCollection
        Dim s As COMAdmin.ICatalogObject
        Dim comAdm As COMAdmin.COMAdminCatalog

        comAdm = New COMAdmin.COMAdminCatalog
        subscriptions = comAdm.GetCollection("TransientSubscriptions")
        s = subscriptions.Add
        s.Value("EventCLSID") = "{" & _
            Type.GetTypeFromProgID("EventClass.COMEventClass").GUID.ToString() &
"}"
        s.Value("Name") = "Form subscription"
        s.Value("SubscriberInterface") = Me
        subscriptions.SaveChanges()
    End Sub

    Public Sub ContactInfoUpdated(ByVal name As String) Implements _
        EventInterface.IEvent.ContactInfoUpdated
        MsgBox(name)
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
        System.EventArgs) Handles Button1.Click
        Subscribe()
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As _
        System.EventArgs) Handles Button2.Click
        Update_Renamed("Joe")
    End Sub
End Class
```

Message Queuing and Queued Components

Message Queuing (also known as MSMQ) constitutes a platform that supports flexible and reliable communication between applications. Different applications can be integrated by implementing a communication environment where messages can be sent between applications in networks with varying degrees of reliability. Message Queuing enables applications that are running at different times to communicate across heterogeneous networks in a reliable manner, in spite of any unreliable components in the system.

A task is invoked asynchronously when it is called without waiting for it to terminate and produce a result. Using this type of execution allows applications to have better responsiveness and improved usage of the available resources. This mode of processing can be achieved using COM+ Queued Components or Message Queuing.

Using Message Queuing, an application can place data into a queue, which is a message store that allows applications to communicate indirectly. Queues will preserve messages until they are not needed anymore. While the data is in the queue it can be retrieved by the same application that generated it or by other applications. Message Queuing queues can receive messages that represent a task to be executed asynchronously by a server process that retrieved the message.

In an unreliable distributed computing environment, it is quite possible that not all of the servers involved in a transaction are available at a given time. For example, a customer purchase order may need to be temporarily saved in a message store while the necessary remote SQL server becomes available. Message Queuing was designed and developed with this kind of situation in mind.

Until recently, the primary ways to access the services provided by Message Queuing were through the MSMQ COM API and C API. The COM API is available to COM clients such as Visual Basic 6.0, but it can also be accessed in Visual Basic .NET using COM interoperability. However, with the introduction of the .NET Framework, the System.Messaging namespace provides a new and convenient way to access MSMQ in a managed environment. For more information about the System.Messaging namespace, see "System.Messaging Namespace" in the .NET Framework Class Library on MSDN.

Message Queuing supports different types of queues, including:

- Public queues. These queues can be accessed by everyone. Public queues are usually created and configured by the system administrator.
- Private queues. These queues can be accessed by applications that have data about the complete path of the queue to be accessed.
- Transactional queues. These queues can be used as the target of transactional messages. Sending messages as part of a transaction ensures that they are delivered in order, delivered only once and successfully retrieved from their destination queue. If one of these conditions is not met, then the transaction will be incomplete and the .NET Framework MessageQueueTransaction.Abort method will be called to rollback the entire transaction. Transactional queues can be used in conjunction with Microsoft Transaction Server (MTS) to make messages part of the MTS transaction. For more information about using transactional queues in .NET, see "MessageQueue.Transactional Property" in the .NET Framework Class Library on MSDN.

The System.Messaging namespace provides a group of classes that allow a developer to access most of the functionality offered by Message Queuing. Table 15.5 offers suggestions for upgrading the different classes available in the MSMQ COM API:

Table 15.5: How to Upgrade Message Queuing COM API Classes in Visual Basic .NET

MSMQ COM API Class	Suggested Upgrade
MSMQApplication Implements the Message Queuing application object. Provides global functionality.	No equivalent is available; it requires reimplementation.
MSMQCoordinatedTransactionDispenser Implements the DTC transaction dispenser. Supports creating new DTC transactions.	No equivalent is available; it requires reimplementation.
MSMQDestination Describes a Message Queuing destination.	The destination of a message is indicated by a System.Messaging.MessageQueue object itself, whose Send method no longer receives a destination object. The Send method is now part of the message queue as opposed to the MSMQ COM API which defined that method in the MSMQMessage class.
MSMQEvent Describes outgoing asynchronous events. Used for notification of asynchronous message arrival.	The System.Messaging.MessageQueue.Peek Completed event can be used to get notifications of asynchronous message arrivals.
MSMQManagementEncapsulates common administrative functionalities for outgoing and target queues. reimplementation.	No equivalent is available; it requires
MSMQMessageDescribes a message.	System.Messaging.Message
MSMQOutgoingQueueManagement Encapsulates administrative functionalities for outgoing queues.	No equivalent is available; it requires reimplementation.
MSMQQuery Provides Message Queuing lookup facilities that are used to locate public queues.	Query criteria is specified with System.Messaging.MessageQueueCriteria. The query is executed with the System.Messaging.MessageQueue. GetPublicQueues method.
MSMQQueue Describes an open queue that supports message retrieval.	It is a subset of System.Messaging.MessageQueue.
MSMQQueueInfo Describes a queue. Used to create, delete, and open queues.	It is a subset of System.Messaging. MessageQueue.

MSMQ COM API Class	Suggested Upgrade
MSMQQueueInfos Describes the collection of queues produced by MSMQQuery.LookupQueue.	System.Messaging.MessageQueueEnumerator
MSMQQueueManagement Encapsulates administrative functionalities for target queues.	No equivalent is available; it requires reimplementation.
MSMQTransaction Implements the Message Queuing transaction object.	System.Messaging.MessageQueueTransaction
MSMQTransactionDispenser Implements the Message Queuing transaction dispenser.	A new transaction is created with the constructor System.Messaging.MessageQueueTransaction.New. Transaction objects are then used as parameters in MessageQueue.Send and MessageQueue.Receive.

The following example shows the Visual Basic 6.0 source code for message sender and message receiver procedures that use a public Message Queuing queue as the communication medium.

```
Private Sub Send_Click()
    Dim dest As New MSMQDestination
    Dim msg As New MSMQMessage
    dest.ADsPath = _
         "LDAP://CN=MSMQTest,CN=msmq,CN=MyComputer,CN=Computers,DC=MyDomain,DC=com"
    dest.Open

    msg.Body = "This is a test"
    msg.Send dest
End Sub

Private Sub Receive_Click()
    Dim QueueQuery As New MSMQQuery
    Dim QueueCollection As New MSMQQueueInfos
    Dim QueueInfo As New MSMQQueueInfo
    Dim MsgQueue As New MSMQQueue
    Dim msg As New MSMQMessage

    Set QueueCollection = QueueQuery.LookupQueue(, , "MSMQTest")
    QueueCollection.Reset
    Set QueueInfo = QueueCollection.Next()
    Set MsgQueue = QueueInfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msg = MsgQueue.Receive()
    MsgBox "Received message: " & msg.Body
End Sub
```

Notice the different parts that are included in the path string (ADsPath) that is used in the Send_Click event handler. The arguments specified are the public queue name (MSMQTest), Message Queuing provider (msmq), the server name (MyComputer), an indicator that the path refers to a message queue (Computers), and the domain name (MyDomain). The above example uses a public queue that was created by a system administrator using the Computer Management configuration application. The steps for creating a queue are in the next section.

Note: You must have Message Queuing installed on your computer to do this procedure. For more information about installing Message Queuing, see "How to Install MSMQ 2.0 to Enable Queued Components" on MSDN.

► **To create a public queue**

1. On the Windows taskbar, click Start, and then click Control Panel.
2. In Control Panel, double-click Administrative Tools.
3. Double-click Computer Management.
4. Click the Services and Applications folder.
5. Double-click the Message Queuing folder.
6. Right-click the Public Queues folder, and then click New\Public Queue.
7. In the New Object window, type MSMQTest in the Queue name box, and then click OK.

You can manually upgrade this code sample to the following Visual Basic .NET code.

```
Private Sub Send_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
_
    Handles Send.Click

    Dim msgQueue As MessageQueue
    Dim msg As New Messaging.Message
    msgQueue = New MessageQueue(".\MSMQTest")
    msg.Formatter = New XmlMessageFormatter
    msg.Body = "This is a test"
    msgQueue.Send(msg)
End Sub

Private Sub Receive_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Receive.Click

    Dim msgQueue As MessageQueue
    Dim msg As Messaging.Message
    Dim types(0) As String
    msgQueue = New MessageQueue(".\MSMQTest")
    msg = msgQueue.Receive()
    types(0) = "System.String"
```

```
    msg.Formatter = New XmlMessageFormatter(types)
    MessageBox.Show("Received message: " & msg.Body)
End Sub
```

Notice how the message queue is identified using a different format, .\MSMQTest. This specifies a public message that is located on the current computer. The new format is MachineName\QueueName. For more information about the MessageQueue class, see "MessageQueue Class" in the .NET Framework Class Library on MSDN. Another important difference is the use of the Message.Formatter property that indicates the formatter used to serialize an object into — or de-serialize an object from — the message body. This formatter introduces differences in the message content with respect to the messages sent from Visual Basic 6.0. Messages sent from Visual Basic .NET can be received by a Visual Basic 6.0 application but additional work is necessary to read the contents of the message. The Visual Basic 6.0 application has to parse the internal XML format to obtain the information originally sent in the message. Additional work is also necessary when communicating in the inverse order. The recommended solution when it is necessary to communicate between applications written in Visual Basic 6.0 and Visual Basic .NET is to keep the entire MSMQ communication layer consistent. For example, all of the application's communication layer should be written in Visual Basic 6.0 or Visual Basic .NET. In such cases, it may be necessary to create new communication classes that will interact locally with components written in other languages.

The COM+ Queued Components service provides an efficient way to invoke and execute components asynchronously using Message Queuing. Setting up this kind of processing can be done relatively easily by deriving the asynchronous class from System.EnterpriseServices.ServicedComponent.

As shown in the following example, the MaxListenerThreads property indicates the maximum number of concurrent Queued Components listener threads. This example shows how to implement a QueuedComp class on the server to display a message asynchronously. It also shows a client method that invokes the DispMsg method on the remote Queued Component. The following code example shows how to do this.

First, the server code is shown here.

```
Imports System.Reflection
Imports System.EnterpriseServices
Imports System
Namespace QueuedCompDemo
    Public Interface IQueuedComp
        Sub DispMsg(msg As String)
    End Interface
    <InterfaceQueuing(Interface := "IQueuedComp")> _
    Public Class QueuedComp
    Inherits ServicedComponent Implements IQueuedComp
        Public Sub DispMsg(msg As String) implements _
```

```
        IQueuedComp.DispMsg
            MessageBox.Show(msg, "Processing message")
        End Sub
    End Class
End Namespace
```

Notice that for the server project, you must include the following lines in the AssemblyInfo.vb file.

```
<Assembly: ApplicationName("QueuedCompDemoSvr")>
<Assembly: ApplicationActivation(ActivationOption.Server)>
<Assembly: ApplicationQueuing(Enabled := True, _
            QueueListenerEnabled := True)>
<Assembly: AssemblyKeyFile("QueuedCompDemoSvr.snk")>
```

Next, the client code is shown.

```
Protected Sub Send_Click(sender As Object, e As System.EventArgs) _
Handles send.Click
    Dim iQc As IQueuedComp = Nothing
    Try        iQc = CType(Marshal.BindToMoniker( _
        "queue:/new: QueuedCompDemo.QueuedComp"), _
        IQComponent)
    Catch l as Exception
        Console.Writeline("Caught Exception: " & l.Message)
    End Try
    iQc.DispMsg("Message test")
    Marshal.ReleaseComObject(iQc)
End Sub
```

Notice that the server component must be signed with a strong name. For more information about strong names, see the "Using COM+ in Visual Basic .NET" section earlier in this chapter.

Summary

MTS and COM+ technologies allow you to build distributed applications to meet your business needs. If you built these applications with Visual Basic 6.0, you can upgrade them to Visual Basic .NET. To do this, you must consider several variables, such as COM+ application types, SOAP, COM+ deployment, but by understanding the upgrade strategies and issues involved, you ensure that the process of upgrading these important technologies goes smoothly. Applying the techniques presented in this chapter assists you in upgrading existing Visual Basic 6.0 MTS and COM+ applications to Visual Basic .NET with as little effort as possible.

More Information

For more information about the Strong Name tool, see ".NET Framework Tools Strong Name Tool (Sn.exe)" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfstrongnameutilitysnexe.asp.

For more information about the System.Messaging namespace, see ".NET Framework Class Library System.Messaging" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemmessaging.asp.

For more information about using transactional queues in .NET, see "MessageQueue.Transactional Property" in the .NET Framework Class Library on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemmessagingmessagequeueclasstransactionaltopic.asp.

For more information about installing Message Queuing, see "How to Install MSMQ 2.0 to Enable Queued Components" on MSDN
http://support.microsoft.com/default.aspx?scid=kb;en-us;256096.

For more information about the MessageQueue class, see "MessageQueue Class" in the .NET Framework Class Library on MSDN.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemmessagingmessagequeueclasstopic.asp.

16

Application Completion

Before the upgrade process is complete, you may need to revise some aspects of your application so that end users can use it more easily. These revisions include reorganizing assemblies to improve deployment, upgrading integrated Help, managing run-time dependencies, and dealing with deployment options. These revisions cannot be automatically handled; therefore, you must manually make these changes.

This chapter provides the information you need to complete these revisions.

Separating Assemblies

An assembly is the primary building block of a .NET Framework application. It is a component library that is built, versioned, and deployed as a single implementation unit. Every assembly contains a *manifest*, which describes that assembly.

A system architect must carefully plan the application deployment strategy so that it is both efficient and reliable. The architect must consider different aspects of the application and target platform, and choose the appropriate execution location for each assembly. The following sections provide an overview of the available options and their corresponding advantages and disadvantages. For more information about assemblies and the features they offer, see the "Native DLLs and Assemblies" section in Chapter 4, "Common Application Types."

No Assembly Separation

If an application or component is implemented as a single assembly but the components have low cohesion, the development, deployment and support aspects can be negatively affected. This can result in serious deployment problems if parts of the application are modified and new versions of those parts need to be installed. Visual Basic 6.0 supported this type of organization because it allowed the developer to define only a single user class per file.

Visual Basic .NET supports having more than one class per assembly, which allows developers to create groups of related classes with high cohesion. This allows complete groups of functionality to be managed together, which reduces the risk of breaking an application by having incorrect versions of components.

In contrast, when assemblies are used as containers for only one component or class, the application requires additional system resources. This is because more components results in more assemblies, and each assembly increases the processing workload for the .NET common language runtime.

Separation by Application Tier

Small and medium-size projects can have components organized into assemblies that correspond to each of the application tiers. Assembly separation by application tier is a natural consequence of the physical separation between the different application tiers. For information on application tiers see the "Desktop and Web Applications" section in Chapter 4, "Common Application Types."

The main reason to further divide a tier among multiple assemblies, which can correspond to one or more layers, is the size and complexity of the assembly. If an assembly is large and some of its functionality is rarely used, the assembly can be divided according to functional layers. You should consider locating the presentation tier, business logic, and data access layers in the same assembly if doing so will ease deployment and systems management concerns.

Separation by Functionality

When the complexity of an application increases, you should organize assemblies into functional groups. For example, the business logic and the data access functionality can be grouped into a single assembly that would be shared by multiple applications (this is simpler than sharing separate tier assemblies). A functional separation is useful from a design perspective because it allows developers to build new functionality based on blocks that operate at the same conceptual level as the new features.

Upgrading Integrated Help

If your Visual Basic 6.0 application includes support for a Help system — for example, if it displays Help documents when a user presses the F1 key —then the Visual Basic Upgrade Wizard cannot upgrade the system automatically. The upgrade wizard leaves the **HelpFile** property (used to specify the name and location of the Help document) and the **HelpContextID** properties (used to associate a specific topic in the Help file for each control on your forms) unchanged. In addition, the upgrade wizard generates a compilation error for each occurrence of these properties because of changes in the Help system in Visual Basic .NET. For example, in

Visual Basic .NET Help support is implemented on a per-form basis: it adds one or more **HelpProvider** components to a form. If you need to access a specific topic, you use the **HelpKeyword** and **HelpNavigator** properties that are provided for each form and control in design time or use the **SetHelpKeyword** and **SetShowHelp** methods in run time.

The following Visual Basic 6.0 code contains two **TextBox** controls. The **App** object sets the Help file and Help topics properties for an application are set at run time.

Note: If you plan to test this example, you must have a compiled HTML Help file (.chm) that contains sections with specific context IDs. In this code sample, the name of the file is MyHelp.chm. It contains two sections: the Help context ID for the first is 1000 and the context ID for the second is 1001. You should replace the path for the Help file and the Help context ID constants as appropriate.

```
' Actual context numbers from MyHelp.chm.
' Define Help context ID constants
Const HelpUserName = 1000
Const HelpPWD = 1001

Private Sub Form_Load()
    App.HelpFile = "C:\MyHelp.chm"

    Text1.HelpContextID = HelpUserName
    Text2.HelpContextID = HelpPWD
End Sub
```

If you use the upgrade wizard to upgrade this code, the following code and issues will result.

```
Option Strict Off
Option Explicit On
Friend Class Form1
    Inherits System.Windows.Forms.Form
    …
    Const HelpUserName As Short = 1000
    Const HelpPWD As Short = 1001

    Private Sub Form1_Load(ByVal eventSender As System.Object, _
                           ByVal eventArgs As System.EventArgs) Handles
MyBase.Load
        'UPGRADE_ISSUE: App property App.HelpFile was not upgraded.
        App.HelpFile = "C:\MyHelp.chm"

        'UPGRADE_ISSUE: TextBox property Text1.HelpContextID was not upgraded.
        Text1.HelpContextID = HelpUserName
        'UPGRADE_ISSUE: TextBox property Text2.HelpContextID was not upgraded.
        Text2.HelpContextID = HelpPWD
    End Sub
End Class
```

Note that the problematic statements are unchanged from the original code. You must address these compilation errors before you can build the project.

Integrating Help at Run Time

The following procedure allows you to correct the Help errors at run time, and include Help support in your Visual Basic .NET application.

► To integrate Help at run time

1. Add a **HelpProvider** component to the forms that display Help. This is demonstrated in Figure 16.1.

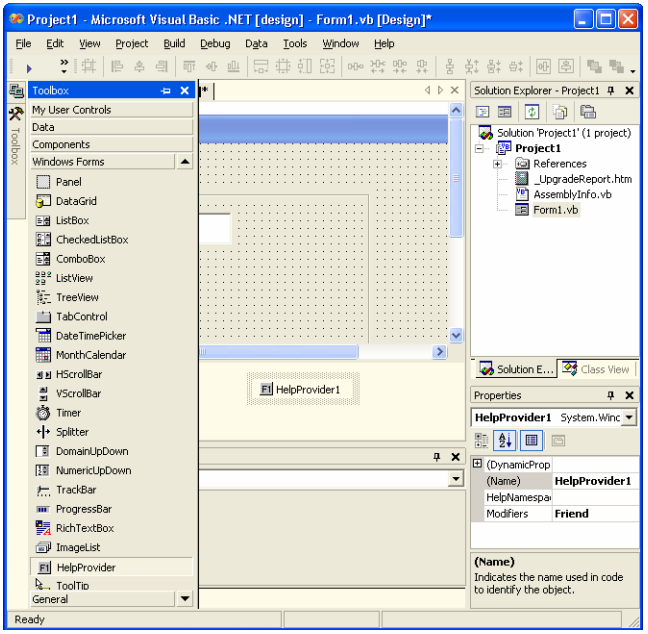

Figure 16.1
*Adding a **HelpProvider** component to a form*

2. Replace the **HelpFile** property of the **App** object with the **HelpNamespace** property of the **HelpProvider**. To do this, replace the following line.

```
App.HelpFile = "C:\MyHelp.chm"
```

Replace the preceding line with the following line.

```
Me.HelpProvider1.HelpNamespace = "C:\MyHelp.chm"
```

3. To upgrade the **HelpContextID** property, you must know how the Help file was made. In Visual Basic 6.0 you use an associated context number, but in Visual Basic .NET you must use the name of the document that has the required information. The example uses two constants to identify the specific topic that should display. Therefore, the constant **HelpUserName** shows the document **UserName.htm**, and the constant **HelpPWD** shows the **PWD.htm**.

   In the upgraded code, change the lines that refer to the constants to refer to the proper pages. First, change each constant's type from **Short** to **String**, and then replace the values with the appropriate document name. This is shown in the following code example.

```
Const HelpUserName As String = "UserName.htm"
Const HelpPWD As String = "PWD.htm"
```

4. Replace the **HelpContextID** property of each object with the **SetHelpKeyword**, **SetHelpNavigator** and **SetShowHelp** methods of the **HelpProvider** object you added in step 1. The following code example shows these changes.

```
Me.HelpProvider1.SetHelpKeyword(Me.Text1, HelpUserName)
Me.HelpProvider1.SetHelpNavigator(Me.Text1, HelpNavigator.Topic)
Me.HelpProvider1.SetShowHelp(Me.Text1, True)

Me.HelpProvider1.SetHelpKeyword(Me.Text2, HelpPWD)
Me.HelpProvider1.SetHelpNavigator(Me.Text2, HelpNavigator.Topic)
Me.HelpProvider1.SetShowHelp(Me.Text2, True)
```

After you complete these changes, the new version of your application should display Help just as the original version did.

Integrating Help at Design Time

There is another technique used to incorporate Help inside a Visual Basic 6.0 application. The previous example showed how to integrate Help at run time; however, you can include a Help file at design time by setting a property in the Visual Basic 6.0 **Project Properties** dialog box. This is shown in Figure 16.2 on the next page.

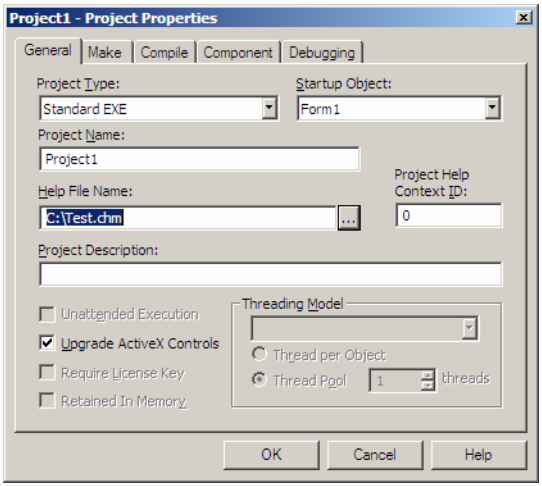

Figure 16.2
*Specifying a Visual Basic 6.0 project Help file in the **Project Properties** dialog box*

This requires that you modify the control's **HelpConmtextID** property inside the properties window, as shown in Figure 16.3.

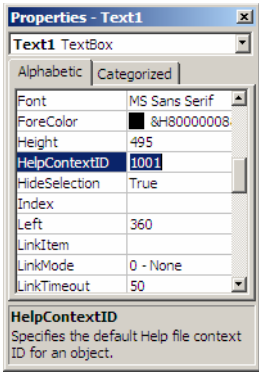

Figure 16.3
*Setting a **HelpContextID** for a Visual Basic 6.0 control*

This example is based on the previous example, which is a form with two **TextBoxes**. To make it run in Visual Basic 6.0, you must add the path of a valid Help file to the **Help File Name** property in the **Project Properties** dialog. Then, set the **HelpContextID** property of each **TextBox** to a valid entry inside the Help file. This sets the values correctly, just as the preceding example did, where the values were set at run time.

After you upgrade this example with the Visual Basic .NET Upgrade Wizard, the application will compile and execute. However, Help will not be available. To make

Help available in the upgraded application, you must follow a procedure similar to the one for integrating Help at run time.

► To integrate Help at design time

1. Add a **HelpProvider** component
2. Modify the **HelpProvider HelpNamespace** property by setting the path to the Help file, as shown in Figure 16.4.

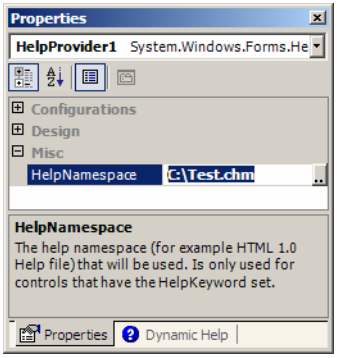

Figure 16.4
*Specifying the path of the **HelpProvider.HelpNamespace** property*

3. Specify the Topic ID as the **HelpKeyword** and set the **HelpNavigator** to the **Topic** value, as shown in Figure 16.5.

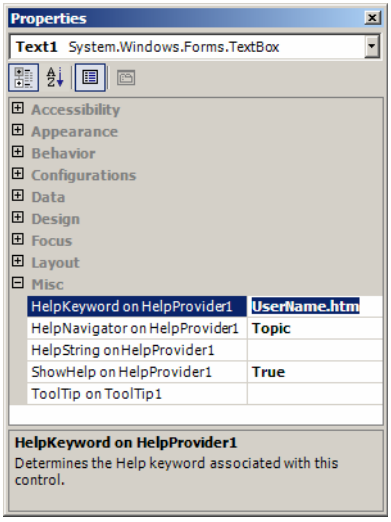

Figure 16.5
Setting the Help properties for a Visual Basic .NET control

The **HelpProvider** class provides Help for controls by extending their properties to the following:

- The **HelpKeyword** property provides the key information to retrieve the Help associated with this control from the Help file specified by **HelpNamespace**.
- The **HelpNavigator** property specifies the Help command to use when retrieving Help from the Help file for the specified control.
- The **HelpString** property specifies the string associated with the specified control.
- The **ShowHelp** property specifies whether Help is displayed for the specified control.

For more information, see "HelpProvider Class" in the *.NET Framework Class Library* on MSDN.

Upgrading WinHelp to HTML

Visual Basic .NET supports the HTML Help format only. Therefore, if your existing Help project is based on a Windows Help project (a .hpj file), you must change it before you upgrade the source code.

► To upgrade Windows Help to HTML format

1. Open the **HTML Help Workshop** tool, which is included as part of Visual Studio 6.0.
2. Create a new project.
3. Select **Convert WinHelp project**, and then click **Next**.
4. In the first field, type the name of the Windows Help file that you need to convert.
5. In the next text field, type the name of the folder that will contain the HTML files. Alternatively, you can click the **Browse** button to select a folder, and then type a name for your project.
6. Click the **Finish** button to begin the conversion. The HTML Help Workshop tool will convert your WinHelp project (.hpj) file to an HTML Help project (.hhp) file, the WinHelp topic (.rtf) files to HTML Help topic (.htm, .html) files, the WinHelp contents (.cnt) files to HTML Help contents (.hhc) files, and the WinHelp index to an HTML Help index (.hhk) file. In addition, WinHelp art files (.bmp or .wmf files) will be converted to HTML image files (.gif or .jpeg files) if your target browser program, such as Internet Explorer 3.0, requires this format, or to .png image files if your target browser program, such as Internet Explorer 4.0, supports them.
7. When the conversion process has finished, compile the project by using the **Compile** menu option. This will create the Compiled HTML Help (.chm) file.

Integrating Context-Sensitive Help

You can implement context-sensitive (pop-up) Help in your Visual Basic .NET application by using the **WhatsThisButton** and **WhatsThisHelp** properties of a form. The **What's This** button appears when you set these properties to **True** and you set the following properties to the specified values:

- Set the **ControlBox** property to **True**.
- Set the **BorderStyle** property to **Fixed Single** or **Sizable**.
- Set the **MinButton** and **MaxButton** to **False**, or set the **BorderStyle** property to **Fixed Dialog**.

The upgrade wizard does not upgrade the **WhatsThisButton** and **WhatsThisHelp** properties; however, the pop-up Help functionality is implemented in Visual Basic .NET by the **HelpButton** property of a Windows form. In Visual Basic .NET, the Help button appears only if the **HelpButton** property is set to **True** and both the **MaximizeBox** and **MinimizeBox** properties are set to **False**.

Run-time Dependencies

Visual Basic 6.0 applications must have a minimum set of files, referred to as *bootstrap* files, before they can be installed. In addition, Visual Basic 6.0 applications require application-specific files, such as an executable file (.exe), data files, ActiveX controls, and dynamic link library (.dll) files.

There are three main categories of required application files:

- **Run-time files**. These are files an application must have to work correctly after installation. All Visual Basic 6.0 applications require these files. The following are the run-time files for Visual Basic projects: Msvbvm60.dll, Stdole2.tlb, Oleaut32.dll, Olepro32.dll, Comcat.dll, Asycfilt.dll, and Ctl3d32.dll.
- **Setup files**. These files are required to set up a standard application on the end user's computer. These include the setup executables (Setup.exe and Setup1.exe), the setup file list (Setup.lst), and the uninstall program (St6unst.exe). For more information about setup, see the "Upgrading Application Setup" section later in this chapter.
- **Application-specific files**. These are files that are specific to your application that must be present for the application to run. These files include the application executable file, any data files, and any ActiveX controls used in the application. These also include other files that your project depends on, such as the library and other files used by the ActiveX controls that your application contains.

In Visual Basic .NET, applications also have dependencies. The run-time files are encapsulated in the .NET Framework runtime and Microsoft data access components (MDAC). You must make sure that the .NET Framework and MDAC are installed before your application can run on the target computer.

The setup files and application-specific files are also required in Visual Basic .NET; however, the setup file list has changed because the creation of deployment projects is different in Visual Basic .NET. The next section, "Upgrading Application Setup," explains these requirements.

Upgrading Application Setup

After you complete the automatic upgrade and manual changes to your projects, you should consider how you can distribute your new applications. Your old installers will no longer work and you cannot use the Package and Deployment Wizard to create an installer for your new application. However, Visual Basic .NET provides a new project type — called Setup and Deployment Projects — that allows you to create Windows installers or CAB files to distribute your applications.

The process of using the Setup and Deployment Projects option to create installers is easiest when you have used a Visual Basic 6.0 setup project to create installers without making any customizations to the resulting installer. For simple projects that require no customization in deployment, you need only indicate the executable files and the product name in the deployment project. However, if your original project's installer required customization, such as adding additional dialog boxes or changing standard dialog text, then you will need to perform similar customization when you use Setup and Deployment Projects.

The following subsections will show you how to use the Setup and Deployment Projects type in Visual Studio .NET to create an installer for your upgraded application, and how to customize the installer if you need to do so.

Creating a New Installer

To create an installer for your project, you need to perform several steps in the solution where the deployment project will reside. The first step is to add a new Setup and Deployment project to the application solution.

► **To add the setup and deployment project to your application**

1. Right-click the Solution node of the tree in the Solution Explorer, and then click **Add New Item** on the menu.
2. In the **Add New Project** dialog box, type the name of your setup project. You can also specify a location where the installer project will reside if you prefer to keep it separate from the rest of your application. Figure 16.6 shows a typical **Add New Project** dialog box.

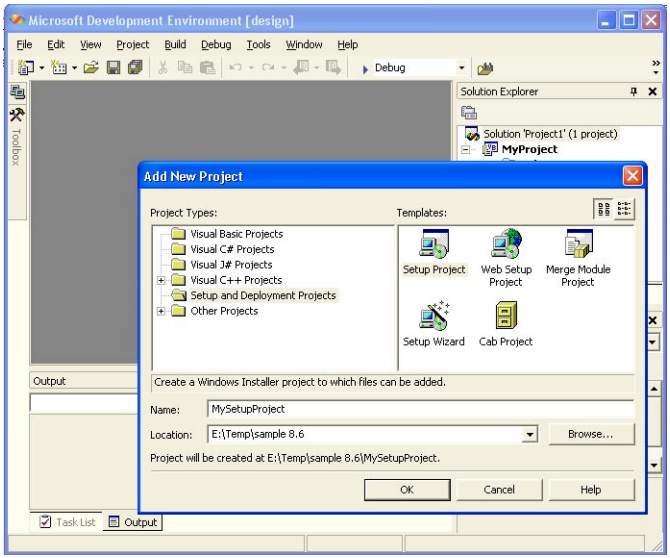

Figure 16.6
The Add New Project dialog box

3. Type the project information, and click the **OK** button. The project will be added to your solution.

After you add the setup project to your solution, you must associate the projects in your solution with the installer. This requires that you add your projects to the application folder as project output. Project output contains the files that your projects produce when the project is built, and typically consists of executable (.exe) or library (.dll) files.

► To add projects to the setup

1. Use Solution Explorer to go to your setup project.
2. On the **Project** menu, click **Add**, and then click **Project Output**. The **Add Project Output Group** dialog box appears, as shown in Figure 16.7 on the next page. The projects in your solution will appear in the drop-down list at the top of the dialog box, followed by a list of project output categories that you can add to the installer.

   For a typical installation, select the application project from the list, and select **Primary output** from the group list. You can also specify the configuration to use when preparing the installer, such as **Debug .NET** or **Release .NET**.

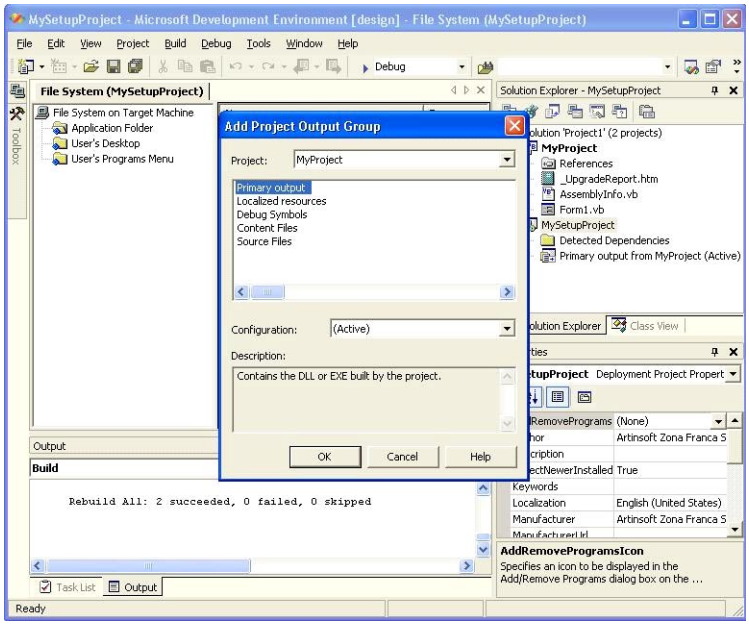

Figure 16.7
The Add Project Output Group dialog box

After you have added all the appropriate project output to your installer project, you are ready to build a typical application installer.

In the setup project's File System view, there are three folders: Application Folder, User's Desktop, and User's Programs Menu. The Application Folder specifies the path on the target computer where the executable files for your application will be installed. By default, the value for this path is Program Files*Manufacturer**ProductName* where *Manufacturer* is the company name you specified when you installed Visual Studio .NET and *ProductName* is the name that you used for the setup project.

To change these values, use the **DefaultLocation** property of the Application Folder or the **Manufacturer** and **ProductName** properties of your setup project. Be sure to use the same values that your old installer used in these properties to minimize the impact on your users.

The User's Desktop folder specifies any files or shortcuts you want placed on the user's desktop during installation. The User's Programs Menu folder specifies the items to be added to the user's Programs Menu when the application is installed.

You can now build your setup project, and all files required to distribute your application will be created. These files are: Setup.exe, Setup.ini, and *ProductName*.msi. Setup.exe is a wrapper for the *ProductName*.msi and the Windows Installer bootstrapping application, which installs the correct version of Windows

Installer if it is not already present on the target computer. All files generated by the setup project are required and must be distributed as a package.

Customizing Your Installer

In most cases, a standard installer with default options is not sufficient to meet the deployment requirements of an application. You may need to add dialog boxes, text, Help files, and more, to your setup program. In Visual Basic 6.0, this required you to either modify the setup script file or modify the Setup1.exe project template used by the wizard.

In Visual Basic .NET, you can use the project's properties and the editors provided in Visual Studio .NET to make these changes. The editors are the following:

- **File System Editor**. This allows you to specify installation directories and Start menu and desktop items.
- **Registry Editor**. This allows you to specify registry keys and values to be added to the target computer during installation.
- **File Types Editor**. This allows you to specify the file type associations for the application on the target computer, as well as the verbs used to identify possible actions for those file types.
- **User Interface Editor**. This allows you to add more dialog boxes to your installation process.
- **Custom Actions Editor**. This allows you to specify additional actions to be performed during installation.
- **Launch Conditions Editor**. This allows you to specify conditions necessary on the target computer before the installation can begin.

For more information about a particular editor, see "Editors Used in Deployment" on MSDN.

With the File System Editor, you can manage all reference to files that will reside on the target computer, set conditions for copying files, create shortcuts to the installed files, or add files to a specified directory. The paths you specify for these files are used on the target computer during installation.

The Registry Editor allows you to manipulate specified registry keys on the target computer during installation. You can also use the Registry Editor to customize your upgraded application's installation to match the previous version's behavior at installation. For example, if the previous installer adds the product version as a key in HKLM\Software*Manufacturer*\Version during the installation process, then by using the Registry Editor, you only need to expand the tree until you find the specified manufacturer. After you have located the manufacturer, you can add a new key with its name set to Version. You can add a new string value to this key in which you set the value for the product version. Figure 16.8 on the next page shows how to do this.

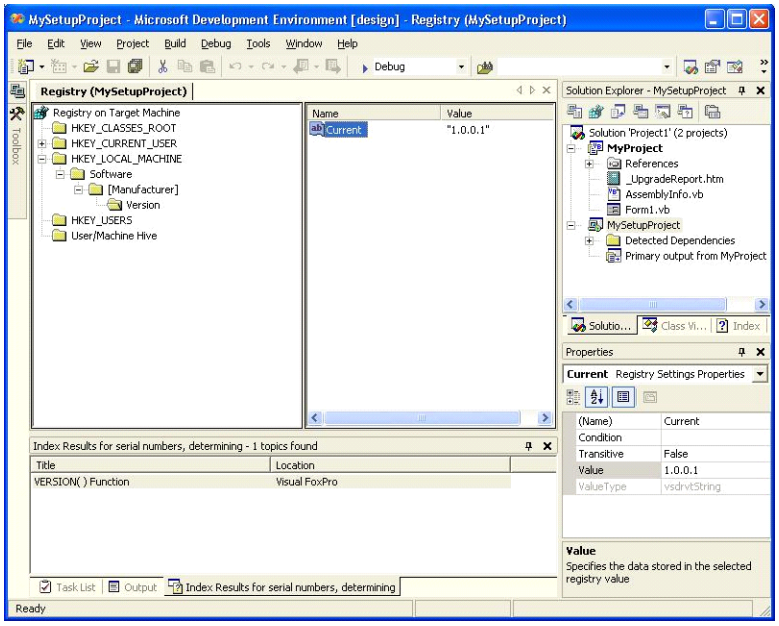

Figure 16.8

Using the Registry Editor to create a version number key in a deployment project

If you added dialogs to your original installer, you should use the User Interface Editor to add similar dialogs to your new installer. The User Interface Editor provides a number of predefined user interface dialog boxes that can be displayed during installation to present or gather additional information.

The available dialog boxes that you can add are the following:

- **Checkboxes**. These dialog boxes present up to four choices to the user and returns the values of those choices during installation. For more information, see "Checkboxes User Interface Dialog Box" on MSDN.
- **Confirm Installation**. This dialog box gives the user a chance to cancel installation or to go back to earlier dialog boxes and make changes before installation begins. For more information, see "Confirm Installation User Interface Dialog Box" on MSDN.
- **Customer Information**. This dialog box prompts the user for information such as name, company or organization, and product serial number. For more information, see "Customer Information User Interface Dialog Box" on MSDN.
- **Finished**. This dialog box notifies the user when the installation is complete. For more information, see "Finished User Interface Dialog Box" on MSDN.
- **Installation Address**. This dialog box allows the user to choose a Web location where application files will be installed. For more information, see "Installation Address User Interface Dialog Box" on MSDN.

- **Installation Folder**. This dialog box allows the user to choose a folder where application files will be installed. For more information, see "Installation Folder User Interface Dialog Box" on MSDN.
- **License Agreement**. This dialog box presents a license agreement for the user to read and acknowledge. For more information, see "License Agreement User Interface Dialog Box" on MSDN.
- **Progress**. This dialog box updates the user about the progress of the installation. For more information, see "Progress User Interface Dialog Box" on MSDN.
- **RadioButtons**. This dialog box presents up to four mutually exclusive choices to a user and returns the value of the selected choice during installation. For more information, see "RadioButtons User Interface Dialog Box" on MSDN.
- **Read Me**. This dialog box presents additional user information, such as errata and late-addition notes that a user should be aware of before running the application. For more information, see "Read Me User Interface Dialog Box" on MSDN.
- **Register User**. This dialog box allows the user to submit registration information by using an executable file that you supply. For more information, see "Register User User Interface Dialog Box" on MSDN.
- **Splash**. This dialog box presents an image to the user, typically to display a logo or branding information. For more information, see "Splash User Interface Dialog Box" on MSDN.
- **Textboxes**. This dialog box presents up to four text entry fields to a user and returns the contents of those fields during installation. For more information, see "Textboxes User Interface Dialog Box" on MSDN.
- **Welcome**. This dialog box presents introductory text and copyright information to the user. For more information, see "Welcome User Interface Dialog Box" on MSDN.

For more information about these dialog boxes, see "Deployment Dialog Boxes" on MSDN.

For example, if your previous installer included a readme dialog box during the installation process, and you want to include this behavior in Visual Basic .NET, do the following:

1. Save the readme content in a file.
2. Use the User Interface Editor to add a readme user interface dialog box to your project.
3. Set the path of your readme file to the **ReadmeFile** property of the dialog box.

You can access any of the editors by using the panel at the top of the Solution Explorer pane, as shown in Figure 16.9 on the next page.

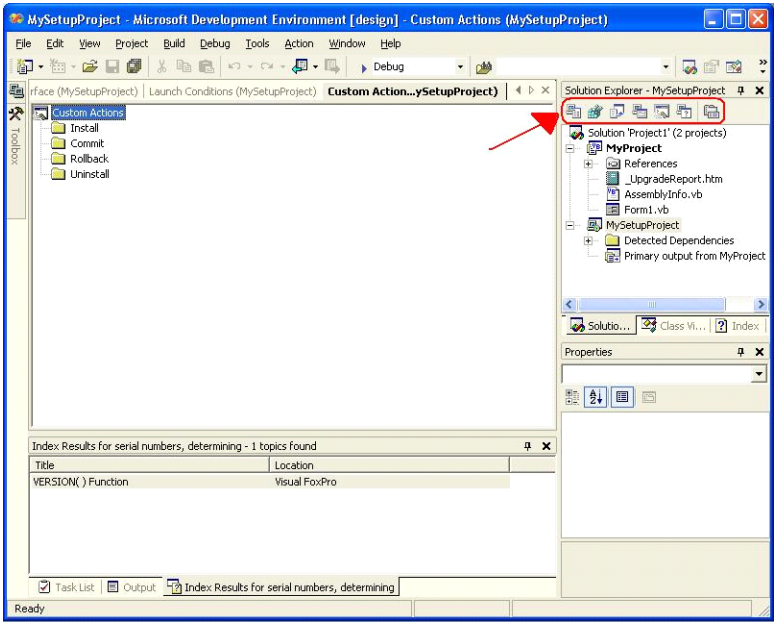

Figure 16.9
Deployment dialog box editors on the Solution Explorer pane

You can further customize your installer by using the setup project's properties. You can use the properties to set information such as the author's name, product name, manufacturer name, product description, and the version number. An example of the setup project properties page is shown in Figure 16.10.

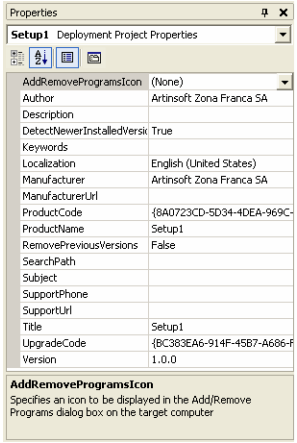

Figure 16.10
A typical setup project Properties page

Merge Modules

Another important step in the creation of your installer is the creation of the merge modules, which allow you to install components that are shared by multiple applications, such as the COM projects. A merge module (.msm) is a single package that includes all files, resources, registry entries, and setup logic to install shared files.

The merge modules are Setup and Deployment project options available in Visual Studio .NET that you can add to your solution. Anything that can only be used by a developer, such as .dll files, controls, resources, and components, should be packaged into a merge module. This module can later be included in a Windows Installer for distribution to the end user.

Although it is possible to put several components in a single merge module, as a general rule it is best to create a separate merge module for each component to avoid distributing unnecessary files. Note that if you need to add third-party shared code that is not distributed in a merge module file to your installer, you must set the **SharedLegacyFile** property appropriately.

A typical case where the merge modules are important is when your application uses COM components created in Visual Basic 6.0 (hybrid applications) because the deployment of this type of application requires that you include the Visual Basic 6.0 run-time merge module. This merge module is used to automatically install the Visual Basic 6.0 run-time files if they are not present on the target computer. When you create a setup project for this type of application, Visual Studio .NET automatically adds any COM components that are referred to as project output. The Windows Installer registers the COM components during the installation. However, the Visual Basic 6.0 run-time merge module is not added automatically. Therefore, you must include it manually. The process for doing so is straightforward, as described in the following procedure.

► **To manually add a Visual Basic 6.0 merge file**

1. In the **Solution Explorer**, select the **Setup** project.
2. On the **Project** menu, select the **Add** option, followed by the **Merge Module** option.
3. In the **Add Modules** dialog box, search for Visual Basic 6.0 merge module, Msvbvm6.msm. By default, this module should be installed in the \Program Files\ Common\Merge Modules directory. If you do not have the file on the system, you can download it from "Windows Installer Merge Modules" in the Microsoft Visual Studio Developer Center on MSDN. After you have selected the file, click the **OK** button.

There are some cases when Visual Studio .NET will not automatically detect the dependencies of an unmanaged component (for example, a COM .dll). In these situations, you must determine the types of dependencies in your project before you

create the installer. Based on the types of dependencies you identify, you can choose the proper solution when you create the deployment project.

An example of this type of situation occurs if you refer to a component that may only be installed as part of another product; for example, the Web Browser control (Shdocvw.dll), which is installed as a part of Internet Explorer. In this situation, you must exclude the component from the deployment project. You should add a launch condition that checks for the component on the target computer and prevents the installation if the component is not found. The end user must install the product that provides the component before installing your application.

Another example of this situation is when you add an unmanaged component that does not expose all of its dependencies, for example, the Microsoft Foundation Classes (MFC) which does not include localized satellite files as dependencies. In this situation, you must determine all of the possible dependencies and include them in the deployment project. You must check the documentation for the component or contact the component's author to get a complete list of dependencies.

For more information about merge modules, see the following MSDN articles:

- "Installer Package Files and Merge Modules"
- "Merge Module Properties"
- "Walkthrough: Creating and Consuming a Merge Module"

Web Deployment

You can deploy ASP.NET applications by using the Microsoft Visual Studio .NET Web Setup project, which allows you to create Windows installers for distribution through the Web rather than traditional media. Using deployment to install files on a Web server provides an advantage over simply copying files onto the server because deployment automatically handles any issues with registration and configuration.

This type of setup project uses the same editors described in the "Customizing Your Installer" section earlier in this chapter, with minor changes to include the necessary information about the installation Web site. The editor that has the most changes is the File System Editor, because it only shows the structure of the Web folder where your pages will reside and the properties that you can use to modify the attributes of the target Web site.

The main properties that the File System Editor provides are the following:

- **Virtual Directory**. This property maintains the name of the target Web site.
- **Port**. This property specifies the number of the port to which the Web site responds.
- **AllowDirectoryBrowsing**. This property specifies whether the target Web directory can be browsed.

- **ExecutePermissions**. This property specifies the execution mode of the target Web site.
- **IsApplication**. This property specifies whether it is necessary to create an application for the target Web site.

Additionally, Visual Studio .NET automatically adds a **Launch Condition** to check the availability of the Internet Information Services in the target computer in the Launch Conditions Editor.

COM+ Deployment

The Component Services administrative tool can automate many of the COM+ deployment tasks performed by system administrators, such as installing server applications on staging and production computers, installing application proxies on client computers, and removing or updating applications.

These are some COM+ application installation tasks you may need to perform using the Component Services administrative tool:

- **Create a new COM+ application**. To perform this procedure, you must be a member of the Administrator role on the target computer's system application.
- **Export COM+ server applications**. You can use the Component Services administrative tool to export a COM+ server application for installation on one or more other server computers. For example, you can use the Component Services administrative tool to export a COM+ application from a staging computer, on which the application was developed and tested, to a production computer at an individual site.
- **Install COM+ server applications**. You can install a COM+ application by using the COM+ Application Install Wizard, or by dragging an application file from Windows Explorer to the details pane of the Component Services administrative tool while the COM+ Applications folder of a computer is open.
- **Export COM+ application proxies**. The COM+ Application Export Wizard will export a COM+ proxy application. When the proxy is installed on client computers, client applications can use the proxy information to find a COM+ server application running on a production computer and access it by using DCOM.
- **Install COM+ application proxies**. The application proxy is created as a Windows Installer (.msi) file, and can be easily installed on a client computer. Installing the application proxy provides the registration information needed for client applications to access the COM+ application remotely from the client computer.
- **Remove COM+ applications**. As existing applications become outdated or are no longer being used, you may need to remove them. Deleting the application also deletes any components contained in the application. If these components depend on additional resources (database connections, data or text files, IIS virtual

root configuration, and so on), these resources must be removed by using tools or utilities other than the Component Services administrative tool.

- **Replicate COM+ applications**. You may need to replicate all COM+ applications and their configurations from one computer (master) to another computer (replication target). Replication is useful in the following situations:
  - Web farm
  - Failover cluster
  - Staging to production

You can replicate applications manually by repeatedly exporting and then installing each of the applications on the master computer. However, this procedure is rather tedious and error-prone, and you might find it simpler to use the replication utility, comrepl.exe, for COM+ applications. This utility is provided by the Microsoft Windows 2000 operating system and later. This replication utility copies all the COM+ applications from a master computer to a target computer, and replicates all computer-wide COM+ configurations.

Application Proxies

You can use the Component Services administrative tool to easily export a COM+ server application as an application proxy. For COM+ to generate an application proxy, it is important that all components in the server application were installed and not imported. This ensures that the application includes all the necessary registration information.

Application proxies generated by COM+ are Windows Installer installation packages. After installation, the application proxies appear in Add/Remove Programs in the Control Panel of the client computer (unless the .msi file is modified by using a Windows Installer authoring tool).

COM+ Installation Packages

You can use the Component Services administrative tool or the COM+ Administration Library to create Windows Installer installation packages for COM+ applications and application proxies, and then use these packages to deploy COM+ applications to server and client computers.

The Component Services administrative tool determines the classes that form the application to be exported, their attributes, and global-level attributes. The tool uses this information to generate a single .msi file with the following contents:

- Windows Installer tables with COM registration information
- An .apl file that contains the application's attributes
- The .dll files and type libraries that describe the interfaces implemented by the COM+ application's classes

Deployment Options

The packaging of components for deployment is an important aspect that you should consider carefully when you create component-based applications. For COM+ applications, your deployment strategy should consider the application tier in which a given distributed component will be deployed and executed. The decision regarding the type and organization of COM+ components will affect different aspects of the application, including performance, reliability, and fault tolerance. The type of COM+ component, that is, whether it is a server or library component, should be the determining factor when you select a deployment options.

Server components are isolated in their own process. If a server component is stopped, it will not affect other server applications. In contrast, library components are loaded in the immediate caller's process and become interdependent.

The first thing to consider when evaluating deployment options is the performance of the invocations. Calls between different processes are expensive compared with calls within the same process. Another aspect to consider is the security verification performed during component invocations. Security credentials are verified only for interprocess invocations. Library components use the security level of the host process.

In general terms, COM+ library applications should contain utilities that are used by the main COM+ components contained in server applications. In this way, the volume of interprocess invocations can be controlled.

The main application components should be COM+ server components that call library components as necessary. The quantity of application components should be in balance with the maintainability, security, and reliability considerations.

Deployment considerations can affect the architectural decisions for an application and vice versa. You may need to refine and change the architecture and deployment options during the life cycle of an application, and you should take this into account when you select a deployment schema.

Summary

Providing a user with a completely upgraded application requires not only upgrading the code, but also upgrading the Help and deployment features. It also means managing deployment features to decrease the likelihood of component version conflicts that lead to broken applications.

This chapter provided the information necessary to upgrade an application's Help system to Visual Basic .NET. It has also shown the various options available for deploying newly upgraded applications. Applying these procedures and techniques when upgrading your applications will help you to complete the upgrade process.

More Information

For more information about dialog boxes, see "Deployment Dialog Boxes" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconDeploymentDialogs.asp.

For more information about the **HelpProvider** class, see "HelpProvider Class" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemwindowsformshelpproviderclasstopic.asp.

For more information about editors, see "Editors Used in Deployment" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbcontheunifieddeploymenteditor.asp.

To download the Visual Basic 6.0 merge module, see "Windows Installer Merge Modules" in the Microsoft Visual Studio Developer Center on MSDN:
http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/mmoverview.asp.

For more information about merge modules, see the following MSDN articles:

- "Installer Package Files and Merge Modules":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsinstal/html/veconinstallerpackagefilesmergepackagefiles.asp.
- "Merge Module Properties":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsinstal/html/veovrmergepackageproperties.asp.
- "Walkthrough: Creating and Consuming a Merge Module":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxwlkwalkthroughcreatingconsumingmergemodule.asp.

17

Introduction to Application Advancement

Upgrading your applications does not have to end with functional equivalence. In fact, one of the key benefits of upgrading to Microsoft Visual Basic .NET is the ability to add new features that are difficult, if not impossible, to implement in earlier versions of Visual Basic. After you move an application to Visual Basic .NET, you open it up to new possibilities.

Application advancement is the process of adding new features, or improving the existing features, of an application. It is not always necessary to first upgrade to a new (or at least later version) language to advance an application, doing so often helps. Earlier versions of a language or framework may have too many limitations to enable them to embrace emerging new technologies.

Listing the possibilities for advancing a Visual Basic .NET application and providing the details on how to accomplish each is a book unto itself. As a result, Chapters 17 (this chapter), 18, 19, and 20 are provided to give you ideas about the kinds of advancements you can make to your applications and provide you with pointers to more detailed information if you want to pursue a particular possibility. They address common advancement scenarios without actually providing all the technical content. However, the information should provide a roadmap of the aspects of your application that you can advance.

You will have to decide which areas to advance and by how much. The trick is to make it a useful and realistic roadmap without it being a list of Visual Studio .NET or Visual Studio 2005 features. For some types of advancement, it may make sense to list out pre-requirements and caveats. The information provided in this and the following chapters should give you what you need to make decisions and determine courses of action. They will also give you references to the details you need to act on your decisions.

Target Audience

Application advancement requires the attention of all members of the product team.

Technical decision makers must determine if the benefits of a particular advancement justify the cost of the effort to achieve the advancement. In addition to a discussion of advancements that can be performed on an application, these chapters also discuss the advantages of the advancement. With this information, technical decision makers can weigh the benefit of the advancement to the business against the cost of performing it, and make an appropriate decision. The information provided in this part of the guide can also be used to make a business case for performing the advancement.

Solution architects responsible for the design of components, applications, and systems will find the discussions of software architecting and refactoring invaluable to improving their designs. Because these topics are too extensive to be fully discussed in these chapters, references to additional resources are provided.

Software developers will find introductory discussions of some of the many features and technologies available in the Microsoft .NET Framework and how they can be applied to improve the performance and scalability of applications. There are also references to additional resources for more detailed information about each feature and technology.

Advancing the Architecture, Design, and Implementation

Even the least ambitious advancement plans will require a new architecture, design, and implementation. Architecture topics will impact the overall system and what it depends on. It will also impact relationships to other systems. The design advancements will organize the code and help you determine which portions of your system can be reused and which must be recreated anew. Implementation advancements will improve the performance, readability, and maintainability of the code on at a fine granularity.

This section examines each of these advancement aspects in detail.

Advancing Architecture

The concept of architecture is an old and well-used concept in the software industry. It plays an important role in programming with new technologies, and architecting is a key practice when planning to develop scalable applications in the .NET Framework.

There are several opinions of what software architecture is and how it can be used or implemented. Some people consider it in terms of the highest level breakdown of a system into parts or a shared understanding of the system design by developers.

The most popular modern description considers architecture a collection of patterns. Regardless of your interpretation of the term "architecture," its role in software development is very important.

The .NET Framework provides a firm step toward the necessity to focus on architecture. Many portions of the framework have been designed to address specific patterns of software development. Understanding these patterns and their application help to utilize the framework properly and ease design and development cycles.

Object-oriented designs are built around classes. A single class (out of context from the application) is sometimes meaningless. Design patterns in architecture represent relationships, and when they are applied to objects, they give meaning to the overall application and the individual component.

Architecting is a key practice to apply to building complex solutions and enterprise applications. An application can be looked at as a building; it would be impractical and flawed to construct a building without architectural designs. Architecture also helps manage complexity; the larger the building, the more crucial the architecture. Architecture patterns provide a meaning for the application as a whole, and thus give us a better understanding of the problem to be solved. Having a plan in place built on strong architectural patterns provides quicker application development. A well built application is easier to maintain if it is architected properly, and it is also easier to extend functionality and implement changes. It is the job of the programmer to put together the parts based on known patterns. This opens the role of architect and programmer to be two distinct functions.

The .NET Framework creates a new focus on architecture. Applications are now more robust in nature and are built on frameworks that provide an enhanced approach to development and design. The .NET Framework enables development that is based on sound principles and methods and easily fits into complex situations while maintaining a true sense of structure and purpose.

Architecture was seen mostly in the early 1990s when client server applications were becoming prominent. These applications required much more management in the design, implementation, and deployment stages than today's applications.

The new millennium ushered in the new technologies such as .NET. These provided a rich framework for application development that augmented the Web development environment. So, now that these are in place, it is necessary to take a step back to client/server architecture and move in the right direction for future development of .NET Framework applications.

Taking Advantage of Object-Oriented Features

Visual Basic .NET is a fully object oriented language that supports most of the object-oriented features available in other languages such as C++ and C#. These features include inheritance, polymorphism, and member overloading. This section

will provide an overview of these features in the Visual Basic .NET language and some suggestions about their usage. A complete discussion about object-oriented design and development techniques is beyond the scope of this guide. For more information, see "Object-Oriented Programming in Visual Basic" on MSDN.

Object-oriented design and programming have been proposed as better ways to understand and model the world around us and to implement those models. Thus, they provide more natural representations and improve the quality and maintainability of code. These features include encapsulation, inheritance, and polymorphism.

Encapsulation

This feature allows a group of class members to be treated as a single conceptual unit. It enables developers to hide implementation details while exposing a well defined set of functionality to clients.

Inheritance

This feature is the ability to create new classes based on previously defined classes or interfaces. The new class (named the *derived* class) extends the functionality of the class it inherits from (named the *base* class) to solve a more specific problem. The derived class will have all existing members (including variables, methods, and properties) of the base class. If an inherited method's behavior is inadequate or incorrect for the derived class then the derived class can change the definition of an inherited method to make it more suitable. The name of this process is *overriding*.

A classic example of inheritance is a **Shape** base class with specific shape subclasses such as **Rectangle** and **Circle**. The **Shape** class might include a method **Area** to compute the area of a shape. Each subclass can override the definition of **Area** as appropriate for that particular type of shape.

Inheritance is demonstrated in the following example class definitions.

```
' Base class
Public Class Shape
    Public Overridable Function Area() As Double
        ' Generic shape area code goes here.
    End Function
End Class

Public Class Circle ' Derived class
    Inherits Shape  'Inheritance

    …
    Overrides Public Function Area() As Double
        ' Code to compute the area of a circle goes here.
    End Function
End Class
```

The preceding class definitions establish the inheritance relationship between the **Shape** class and the **Circle** class. The **Overridable** keyword in the **Shape** class definition indicates that this method can be modified in any class that inherits from **Shape**. In this example, the **Circle** class modifies the **Area** function as indicated by the use of the keyword **Overrides**. Derived classes do not have to override any functionality provided in the base class if the existing functionality is already sufficient.

Note that the inheritance property is transitive. If class A is the base class of class B, and class B is the base class of class C, then class C also inherits from class A. In such a situation, class A is said to be the *ancestor* class of class C. Objects of class C will have all the methods and variables of class A and can override any methods of class A tagged as **Overridable**.

Inheritance improves application design and allows developers to reuse and adapt existing code resources.

Interfaces

Interfaces specify the properties, methods, and events that classes must implement. They allow developers to separate the information that clients need to know (such as the names, return types, and parameter lists of methods) from the implementation details. This reduces compatibility risks because enhanced implementations can be developed for interfaces without jeopardizing existing code.

In Visual Basic 6.0, it is possible to implement existing interfaces, but it is not possible to define them yourself. The **Interface** statement has been introduced in Visual Basic .NET to allow developers to define interfaces as entities that are different from classes. The **Implements** keyword is used to implement interfaces in a class. It is important to note that .NET Framework classes can implement more than one interface to be applicable in different contexts. It is also important to note that implementing an interface establishes an inheritance relationship between the interface (which serves as the base) and the implementing class (which is derived from the interface).

Interfaces eliminate a possible problem of class inheritance: it eliminates the possibility of breaking code when you make post-implementation changes to your design. Design decisions need to be made when the class is first published. If there is a need to change a design assumption, it could be unsafe to change the application's code.

The following code example shows a Visual Basic .NET interface that corresponds to a class used to encrypt files.

```
Interface ICipher
    Sub SetKey(ByVal keyBase As String)
    Sub Encrypt(ByVal inputFile As String, ByVal outputFile As String)
    Sub Decrypt(ByVal inputFile As String, ByVal outputFile As String)
End Interface
```

For an example implementation of this class, see the "Using Cryptography" section in Chapter 20, "Common Technology Scenario Advancements." For more information about interfaces, see "Interfaces in Visual Basic .NET" on MSDN.

Polymorphism

This feature, in conjunction with inheritance, allows a program to invoke the proper version of a method depending on the object being used to invoke it. Conceptually, a base class provides some core set of functionality. Derived classes build on this core set by specializing some of the functionality. A base class reference variable can be used to refer to any object of the base class or any derived class. When an overridden method is invoked through this reference, the actual method invoked will be based on the type of the object, not the type of the variable. This is best demonstrated with an example, as shown here.

```
Public Function ShowArea(shape as Shape)
    Dim msg as String

    msg = "Area = " & shape.Area
    MsbBox(msg)
End Function

…
    Dim s as new Shape()
    Dim c as new Circle()
' Shape version of area will be invoked
    ShowArea(s)
' Circle version of area will be invoked
    ShowArea(c)
…
```

Polymorphism allows very generic programming. Developers can write solutions for problems based on what functionality will be provided without having to worry about how that functionality will be specialized in derived classes. In the preceding code example, all the developer needs to be aware of is that the **Area** method exists. The details of how it is computed for different shapes are unimportant. The runtime manages which version to invoke based on the object's type.

Overloading Functionality

A member is overloaded when it is declared more than one time with the same name but different arguments. This quality allows developers to program functionality in a generic way. For example, the **Encrypt** function can be designed to receive different data types and return an encrypted value. In this way, the **Encrypt** function can be used in all the places that an encrypted variable is required. In Visual Basic 6.0, you would have to define a function with a different name for each of the expected data types.

Overloading, overriding, and shadowing are similar concepts applicable to the declaration of members with the same name, but there are important differences that make them appropriate to certain circumstances, such as the following:

- Overloading is applicable when different members have the same name but accept a different number of arguments or arguments with different data types.
- Overriding is applicable when a member defined in a derived class must accept the same data type and number of arguments as a member in the ancestor.
- Shadowing is applicable when an inherited member has to be locally replaced.

Visual Inheritance

Another scenario where inheritance has been proven to be especially useful is in visual form inheritance. This type of inheritance allows you to apply the layout, controls, and code for existing forms to new forms. Through this type of inheritance, developers can build standard base forms. These forms can then be inherited by applications to give them a similar look and feel, but they still give the application developers freedom to expand on the form by adding new controls or changing a particular control's behavior. The result of visual inheritance is that developers can take existing forms and customize them for new applications, instead of having to recreate new forms for each application.

Layering Implementation

Layering is a common term used by developers when they try to present their applications as good *n*-tier model applications. Developers will often proudly discuss how an application is being built properly because it is in layers. This is a good start, but it does not imply a well architected solution.

This subsection discusses layering in detail, including how it can help you prepare your applications for advancement.

Layers represent a build-up of functionality. The higher layers are dependant on the lower layers to supply the required information and methods. A single layer represents a logical group of functionality that is contained in an application tier that usually corresponds to functionality that is bound to a physical resource such as a database server. In this way, application tiers are composed of one or more logical layers that build the functionality provided by the tier. For more information about application tiers and its classification, see the "Architecture Considerations" section in Chapter 4, "Common Application Types."

Development takes several approaches to this, such as the bottom-up and the top-down approaches. However, these approaches cannot necessarily be applied to all applications. Every application presents a solution to a specific problem, and what applies to one application cannot be applied to another. The key is to always observe

layers as building blocks for higher layers, and to also keep in mind that lower layers should almost never rely on higher layers for operation.

A group of layers usually present a coherent whole. For example, you may be familiar with the concept of dividing an application among various tiers that are composed of groups of layers, such as the presentation tier, the business logic tier, the data tier, and so on; each tier represents a coherent unit of the application and serves a single purpose, such as displaying information to the user or representing business logic.

A disadvantage of layers is that cascading interface changes may lead to more work. Adding a field to a screen or page may require a change not only in the layers of the presentation tier, but it may also require a change in the business and data tiers.

An advantage to layers is that substitution is possible. This means that you can pull out a layer and provide a different layer that adheres to the interface but provides a completely different level of functionality. This is very helpful for unit tests; it is also helpful for applications that may require different business logic or database access but no other changes. Keep in mind that careful layer design is important. Unnecessary layers can harm performance.

There are three tiers clearly defined for most business applications. These are:

- **The presentation tier**. This tier provides the application user interface (UI).
- **The domain (business) logic tier**. This tier implements the functionality of the application.
- **The data tier**. This tier is responsible for storage and retrieval of information to and from an external database.

There are other architectures with other tiers to consider, but the general outlook of an application provides these three distinct tiers of separation.

Note that the term *n*-tier application is one of the most misused terms in the field. Many people view the layered application as an *n*-tier application. Actually, the term is applicable only to applications whose business logic tier is divided in more than one layer that can run on different computers. In fact, the term "tier" denotes a physical separation and not a logical separation; the term "layer" denotes a logical separation.

Design Patterns

Design patterns are recurring solutions to software design problems you find again and again in real-world application development. Design patterns are about design and interaction of objects. They are intended to provide a communication platform centered on elegant, reusable solutions to commonly encountered programming challenges.

Use of Frameworks, Libraries, and Application Blocks

Some of the most useful patterns describe frameworks. Such patterns can be viewed as abstract descriptions of frameworks that facilitate widespread reuse of software architecture. Similarly, frameworks can be viewed as concrete realizations of patterns that facilitate direct reuse of design and code. One difference between patterns and frameworks is that patterns are described in a language-independent manner, whereas frameworks are generally implemented in a particular language. However, patterns and frameworks are highly synergistic concepts, with neither subordinate to the other. The next generation of object-oriented frameworks will explicitly embody many patterns, and patterns will be widely used to document the form and contents of frameworks.

The introduction and marketing of design patterns have created a small revolution in software development. Developers are encouraged to abstract their problems and recognize when a problem fits into one of the many defined design patterns. By thinking in terms of common patterns, developers are able to more easily apply knowledge and implementation experience gained from previous development efforts. Although thinking in terms of design patterns will not solve all the problems inherent to software development, it has proven to be a very powerful and effective way to develop software.

Some patterns are so commonly used, and often in exactly the same way, that they become more than just patterns. They evolve into common code libraries and are included in almost every project. This is nothing new: common code libraries have been around for almost as long as programming has existed. Every programmer has his or her own library of common code, as does every development team. After you figure out a good way to perform a common task, it makes sense to formalize it and put it somewhere where others can use it. That is what Microsoft application blocks are: code libraries that ease implementation of common tasks. Application blocks are focused, typically small abstraction layers that ease and standardize implementation of common tasks.

For example, accessing data is a very common task in today's applications. There are many different ways to access data, but over the years developers have learned that only a few of those ways are effective. The developers of the Data Access Application Block (DAAB) recognized the common database access patterns and abstracted them into a small, consistent interface that takes most of the drudgery out of database access. Compared to the amount of code required to directly access data through ADO.NET, using the DAAB is almost trivial. Because it is implemented in a separate class, it is very easy to use unmodified in many different applications.

The following application blocks are currently available for use in .NET:

- Asynchronous Invocation Application Block for .NET: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/PAIBlock.asp*

- Caching Application Block:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/CachingBlock.asp
- Configuration Application Block:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cmab.asp
- Data Access Application Block:
  http://msdn.microsoft.com/library/en-us/dnbda/html/daab-rm.asp
- Exception Management Application Block:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp
- Logging and Instrumentation Application Block:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/Logging.asp?frame=true
- Security Application Block:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/security1.asp
- Smart Client Offline Application Block:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/offline.asp
- Updater Application Block – Version 2.0:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/updater.asp
- User Interface Process (UIP) Application Block – Version 2.0:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/uipab.asp
- User Interface Process Application Block for .NET:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/uip.asp

All of these application blocks are supplied with source code and examples, and all are free to download and use.

For more information about application blocks, see the "Using Visual Basic .NET Application Blocks" section in Appendix B, "Application Blocks, Frameworks, and Other Developer Aids."

Refactoring to Patterns

Refactoring is an iterative process whose goal is to transform existing applications according to modern software engineering quality criteria. When refactoring is applied to an application, the characteristics of the application are improved in terms of clarity, maintainability, and redundancy reduction, amongst others. Refactoring may involve small changes, such as changing variable or subroutine names, or it may require large changes, such as consolidating functionality into a single component or breaking apart an existing component into multiple components that more logically isolate functionality.

The best approach to refactoring is to attempt to do so in small steps. This will help to ensure that the refactoring process does not introduce defects. Smaller refactoring processes, such as identifying poor variable names, may take only minutes. Larger refactoring processes, such as identifying functionality that can be consolidated (or separated) can take hours, days, or even weeks. However, even for extensive or intensive refactoring, it is still best to proceed in small steps.

There are several reasons to apply refactoring; the following motivations are among the most common:

- **To make it easier to modify and add new code**. The grouping and separation of common pieces of code allows for centralization of functionality. If functionality needs to be changed, it can be done in one single place.
- **To improve the design of existing code**. Refactoring tends to produce clearer code and eases the adoption of enterprise development standards. It also allows the developer to easily develop and test different system models and designs to identify which one best fits the enterprise needs.
- **To gain a better understanding of code**. The clarity and organization gain obtained with refactoring allows the human reader to identify and understand the key concepts of the functionality.
- **Improved growth and integration capabilities**. The code can grow in an organized and systematic way; this allows for the definition of standard interfaces that ease integration and reusability.

Refactoring to patterns implies the transformation of existing applications according to software design patterns. A pattern is based on an abstraction of a group of similar software designs that is further refined and at the end constitutes a model that contains the most relevant characteristics of the original applications. Patterns are obtained with a logical inductive process that has been applied to real-life application designs. Software design patterns are also enriched with good design principles and standards that will allow the user to obtain a better product when the pattern is applied.

Patterns can assist users with creating large-scale applications and with resolving recurring design problems with proven and highly effective guidance. When an application is refactored according to design patterns, the resulting structure of the application can be understood as a derivation of the pattern(s) applied; this reduces the future development and maintenance risks.

Keep in mind that the effort of refactoring existing code to a pattern must be weighed against the necessity of the code itself. It must be stressed that making an effort to use patterns will strengthen your overall design ability, but like your basic coding skills, it is something that is to be learned and cultivated.

For more information about design patterns, see “Microsoft Patterns” on MSDN.

Implementation

Eventually, advancing your application will require implementing the new features or improvements you have identified. This section discusses some of the implementation issues in application advancement.

Replacing API Calls with .NET Framework Intrinsic Functions

Because of some limitations in Visual Basic 6.0, many programmers have been forced to use the different functions that are available in the Windows API. For operations such as registry manipulation, finding window names, or finding the names of special folders, Visual Basic 6.0 developers were forced to use the Windows API because there was no other way to carry out these tasks.

In Microsoft Visual Basic .NET, all the barriers that held back Visual Basic 6.0 developers and forced them to resort to the Windows API are completely gone. For every function that accomplished a specific task in the Windows API, there is a way to achieve the same result in .NET using intrinsic functions.

You can still use Windows API calls by means of using interop services, but your .NET applications are usually be better off using the functions included in the .NET Framework. The amount of work required to get your .NET application working with the same API calls from Visual Basic 6.0 may be more than what you would expect. Calling these APIs in .NET is not as simple as it is in Visual Basic 6.0, and there may be some additional overhead making the necessary corrections so that changes in data types from Visual Basic 6.0 will adhere to the type specifications of the APIs. Furthermore, by using API calls in your .NET application, you are immediately limiting your application to be specific to a particular version of the Windows operating system.

Even though there is no harm in using Windows APIs in Visual Basic .NET, your code may be more portable and compatible if you replace your API calls by using .NET intrinsic functions. The end result may be the same, but if you ever need to move part of your code toward another .NET language or operating environment you know that you will not be held back by adhering to .NET functions instead of using the Windows API.

For a complete list of all the Windows API functions and possible replacements for you to use in .NET, see "Microsoft Win32 to Microsoft .NET Framework API Map" on MSDN.

Replacing Registry API with .NET Intrinsic Functions

Built-in registry manipulation from within Visual Basic 6.0 has been somewhat limited to the following four functions: **SaveSetting**, **GetSetting**, **GetAllSettings**, and **DeleteSettings**. Although these functions allow registry editing, they are limited in the subtrees they can access: **HKEY_CURRENT_USER\Software\VB** and **VBA**. Because of this limitation, many programmers use the Windows API to perform registry modification tasks from within Visual Basic 6.0.

Two new classes that offer the same registry manipulation functionality as the Windows Registry API have been added to the .NET Framework. These classes are the **Registry** class and the **RegistryKey** class; they are located in the **Microsoft.Win32** namespace.

> **For Visual Basic 2005:**
>
> Registry access has been simplified in Visual Studio 2005 with the addition of the **My.Computer.Registry** object and the addition of two new methods to the **Microsoft.Win32.Registry** object that **My.Computer.Registry** points to. These objects provide access to the registry on the local computer. The two new methods are **GetValue()** and **SetValue()**; they allow you to read from and write to arbitrary keys in the registry without having to first navigate the registry tree. For more information, see "My. Computer.Registry Object" in the Visual Basic Language Reference on MSDN.

The **Registry** class provides the set of standard base keys found in the registry. Using the **Registry** class, you can define the root key on which you will be working. After the base key is defined, you can use the **RegistryKey** methods to perform the necessary registry actions.

The **RegistryKey** class methods provide the means by which to add, delete, or change subkey values within the registry. There are equivalent .NET registry method calls for the most common operations performed using the Windows registry API functions. For example, if you used the **RegCreateKey**, **RegOpenKeyEx**, or **RegDeleteKey** API calls, you can now use the **RegistryKey** class methods **CreateSubKey**, **OpenSubKey**, or **DeleteSubKey** respectively. These .NET methods offer the same functionality as their API counterparts and are easier to use.

For information about the available properties and methods available in the registry and **RegistryKey** classes, their usage, and how to incorporate them into your application, see the following resources from the *.NET Framework Class Library* on MSDN:

- "Registry Class"
- "RegistryKey Class"

By using the **Registry** and **RegistryKey** classes, you will be able to manipulate the registry from within Visual Basic .NET. The end result will be exactly the same as when using the Windows API registry functions from within Visual Basic 6.0. In most cases, the necessary code to perform these operations will be smaller and more understandable than the API counterpart.

Serialization

When coding applications that do not have or need access to a database, there is frequently a need to store data used in the program for later retrieval. Before .NET, any attempt to serialize data required custom code. For example, if the programmer

needed to serialize the information from a class or data structure, a procedure could have been written that would write each instance of the class/structure to an XML document. Programming this code usually required a great deal of work and testing to make sure that the implemented serialization worked correctly. Fortunately, carrying out this task with .NET does not require writing, testing, or debugging custom code because it has been carefully planned and is very straightforward to implement.

Using .NET, there are different serialization schemes that can be used in your application. Each technique has its own positive and negative aspects; choosing the correct technique depends on the requirements that you may have for storing your data.

The first technique uses classes found in the **System.Xml.Serialization** namespace, specifically the **XmlSerializer** class. Using this class, you can easily serialize and de-serialize data in your code to XML format. You must first create an **XmlSerializer** instance and pass to the constructor the type of the class that you want to serialize. You can then create an object of type **FileStream** to store the data and then call the **Serialize** method of the **XmlSerializer** instance that was previously defined. Assuming you already have an **Employee** class and an instance of the class named **myEmployee**, the following code demonstrates how to serialize a class instance.

```
Imports System.Xml.Serialization
Imports System.IO
…
Dim mySerializer As New XmlSerializer(GetType(Employee))
Dim myData As New FileStream("myData.xml", FileMode.Create, FileAccess.Write)
mySerializer.Serialize(myData, myEmployee)
…
```

De-serializing data using the **XmlSerializer** class is very similar to serializing, except that you need to assign the retrieved information to an instance of the class of the same type that you originally serialized. The following code shows the de-serialization procedure using the **XmlSerializer** class.

```
Imports System.Xml.Serialization
Imports System.IO
…
Dim myDeserializer As New XmlSerializer(GetType(Employee))
Dim myData As New FileStream("myData.xml", FileMode.Open, FileAccess.Read)
Dim myEmployee As New Employee
myEmployee = myDeserializer.Deserialize(myData)
…
```

If the employee class has any private properties, these would not be serialized into the XML format. Only public properties are serialized to the XML document; this type of serialization is referred to as *shallow serialization*.

Unlike shallow serialization, the next serialization technique in .NET, binary serialization, allows the storage of both private and public properties. This technique is found in the **System.Runtime.Serialization** namespace and requires a bit more coding than the **XMLSerializer** method.

The class to be serialized using binary serialization must have the **<Serializable()>** attribute declared. This communicates to the serialization object that it can go ahead and try to serialize the object, as demonstrated in the following code example.

```
<Serializable()> _
Public Class Employee
    Public Name As String
    ' Other class properties here.
End Class
```

The class can now use binary serialization in a very similar manner as previously done with the **XmlSerializer** class. You need to first declare a **BinaryFormatter** object to store the data; then you are able to use a **FileStream** object and the **Serialize** method of the **BinaryFormatter** class to store this information to a file, as shown here.

```
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.IO
…
Dim mySerializer As New BinaryFormatter
Dim myData As New FileStream("myData.bin", FileMode.Create, FileAccess.Write)
mySerializer.Serialize(myData, myEmployee)
…
```

The process of deserializing the data using the **BinaryFormatter** class is very similar to the **XmlSerializer** and is left as an exercise for the reader.

The two classes discussed in this section make it very easy for serializing and deserializing data in .NET applications. The **XmlSerializer** technique requires less work, but it is more limited in what it can do. As previously mentioned, it will not serialize private properties from a class. Furthermore, there are some objects that it cannot serialize (for example, any class that implements the **IDictionary** interface). If you ever run into any limitations using the **XmlSerializer** class, be sure to check if the **BinaryFormatter** will help you overcome these hurdles.

Optimizing Performance with .NET Collections

In the **System.Collections** namespace, you can find a series of classes and interfaces that define a series of objects that can be used to implement sets of closely related objects in your classes. This namespace includes collections that can be used immediately, such as **ArrayList**, **Stack**, **Hashtable**, and **Queue**, but it also includes interfaces such as **IEnumerable** that allow you to create your own collection classes.

By having classes implement collections instead of dealing with groups of instances by means of arrays, you may be saving yourself a lot of trouble in the future. All the caveats that using arrays usually bring, such as (but not limited to) run-time errors, invalid indexes, or array length tracking, are no longer concerns when you use collections. By implementing collections, you are also following good object-oriented practice in the sense that you can easily employ other concepts such as inheritance and polymorphism in the future. Following this approach will also aid you in avoiding errors in the future by re-utilizing stable code.

The following examples demonstrate how to implement collections using the classes available in the **System.Collections** namespace. Assume that you are working on an inventory for a fish store. Because there would be plenty of fish to keep track off, a good idea would be to implement a **Fish** class and then create a **Fishes** collection class to keep track of all the different fish that would be available in the store. To offer this functionality, the **Fishes** class could inherit from **System.Collections.CollectionBase**.

```
Public Class Fishes
Inherits System.Collections.CollectionBase
```

It is then necessary to implement the **add** and **remove** methods from the **CollectionBase** abstract class in addition to the **Item** property, as shown here.

```
…
Sub add(ByVal aFish As Fish)
    List.Add(aFish)
End Sub

Sub remove(ByVal index As Integer)
    If (index > 0 & index < List.Count - 1) Then
        List.RemoveAt(index)
    End If
End Sub

ReadOnly Property Item(ByVal index As Integer) As Fish
    Get
        Return List.Item(index)
    End Get
End Property
…
```

After the **add** and **remove** methods and the **Item** property are implemented, the **Fishes** class makes it very easy to keep track of the fish by means of adding and removing elements at certain indexes.

```
…
Dim gerald As New Fish("Greg", "Guppy", "Orange")
Dim nemo As New Fish("Nemo", "Clown Fish", "Orange/White")
```

```
Dim myFishCollection As New Fishes
myFishCollection.Add(gerald)
myFishCollection.Add(nemo)
myFishCollection.remove(0)
MsgBox(myFishCollection.Item(0).getName)
…
```

As you can see, the procedure to build a class that can implement a collection is a simple one. The **Fishes** class now offers an easy way to add and remove items from the objects that you want to group together.

If you prefer to not implement a collection in your class, but you are still changing the sizes of your arrays, you might be better off using the **ArrayList** class in the **System.Collections** namespace. For insertions and deletions of sets of related objects, arrays do not support directly adding and removing the elements. If you are inserting or deleting elements at the end of an array, you have to use the **Redim** statement to carry out this operation, which usually degrades performance. To insert or remove elements anywhere else, be sure to use an **ArrayList** to carry out these tasks.

One thing to note is that classes such as **HashTables** or **ArrayLists** are not type-safe. Because these objects hold elements of type **System.Object**, they will accept any type regardless of what they are currently holding. For example, if you have an **ArrayList** that is currently holding Boolean values, the compiler will not identify a compilation error when you try to add an object of type **Integer**. This will go unnoticed until the application is executed at run time; at run time, it will throw a type-casting error.

COM Object Creation in Visual Basic .NET

The **CreateObject()** function in Visual Basic .NET is responsible for creating and returning an instance of a COM object. This function can only be used with classes that have been exposed as COM components.

The function has the following usage.

```
Public Shared Function CreateObject( _
   ByVal ProgId As String, _
   Optional ByVal ServerName As String = "" _
) As Object
```

The first parameter, **ProgId**, is the string representation of the program ID of the object that you would like to be created. The second optional parameter is the name of the network server where the object will be created. If an empty string is passed, the local computer will be used.

When using the **CreateObject()** function, you should be concerned about the way in which you are assigning the creation of the object to an instance of a variable. In the

following example, the **cnADO** variable will be late bound by the .NET common language runtime (CLR).

```
Dim cnADO as Object
Set cnADO = CreateObject("ADODB.Recordset")
```

The referenced variable in the first line of the preceding example can point to data of any type. Although flexible, this has the caveat of compromising performance because the **Object** variable will be late bound. After the application is running, the CLR will have to perform type checking and member lookup at run time. This clearly has a performance overhead that would not happen if the variable declaration is early bound because the compiler is able to perform the type checking and member lookup at compile time.

You can add a reference to the type library of the COM class that you are trying to instantiate instead of relying on late binding. In most cases, the use of the Dim statement and a primary interop assembly to instantiate the COM class will be more efficient and will yield code that is easier to read and maintain than code that uses late binding. The following example shows how to create an instance of a **Recordset** without using the **CreateObject**() method.

```
Dim cnADO as ADODB.Recordset
```

If you have to create objects on a remote server, the **CreateObject()** function will allow you to do this if you have made your application accessible from the remote server. This is achieved by passing the server name as the second parameter, and is demonstrated in the following code example.

```
Sub CreateRemoteExcelObj()
   Dim xlApp As Object
   xlApp = CreateObject("Excel.Application", "\\YourServerName")
   MsgBox(xlApp.Version)
End Sub
```

Summary

Taking advantage of the benefits available in Visual Basic .NET and the .NET Framework in general means advancing your upgraded application with new features and technologies. This chapter has presented some introductory material about how to advance the architecture, design, and implementation of your applications after you have upgraded them to Visual Basic .NET.

The possibilities discussed here and in the following chapters are only the beginning. By studying Visual Basic .NET and the .NET Framework in detail, you will discover even more technologies and features that you can add to your application

to increase their value to your business for years to come. To start you on your discovery of these possibilities, the next chapter will introduce advancements for typical application scenarios. The following chapter will introduce advancements for Web applications. Finally, a chapter about advancements based on technological needs, such as security and performance features, will round out this discussion.

More Information

A complete discussion about object-oriented design and development techniques is beyond the scope of this guide. For more information about object-oriented design, see "Object-Oriented Programming in Visual Basic" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vbconprogrammingwithobjects.asp.

For more information about interfaces, see "Interfaces in Visual Basic .NET" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vaconinterfaces.asp.

For more information about design patterns, see "Microsoft Patterns" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/MSpatterns.asp.

For a complete list of all the Windows API functions and possible replacements for you to use in .NET, see "Microsoft Win32 to Microsoft .NET Framework API Map" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/win32map.asp.

For more information on registry objects, see "My.Computer.Registry Object" in the Visual Basic Language Reference on MSDN:
http://msdn2.microsoft.com/library/sykcb9xf(en-us,vs.80).aspx.

For information about the available properties and methods available in the registry and RegistryKey classes, their usage, and how to incorporate them into your application, see the following resources from the *.NET Framework Class Library* on MSDN:

- "Registry Class":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfmicrosoftwin32registryclasstopic.asp.
- "RegistryKey Class":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfmicrosoftwin32registrykeyclasstopic.asp.

18

Advancements for Common Application Scenarios

Arguably, the best way to begin a discussion of possible application advancements is to start by examining typical application scenarios. From this viewpoint, you can identify advancements based on the type of application you want to advance.

There are many categories of applications possible, and each one has possible advancement potential. This chapter will focus on only two categories: Windows-based applications (form-based applications) and business applications (enterprise services).

Windows Applications and Windows Forms Smart Clients

Smart clients are easily deployed and managed client applications that provide an adaptive, responsive and rich interactive experience by leveraging local resources and intelligently connecting to distributed data sources. They are connected systems (usually through the Internet) that allow the user's local applications to work together with remote applications. For example, a smart client running an image recognition application can communicate with a remote database over the Internet to collect image patterns from the database to be used in the image recognition process. The following are characteristics that define smart clients:

- They have the ability to work offline; this means they can work with data even when they are not connected to the Internet.
- They have the capability to be deployed and updated in real time over the network from a centralized server.
- They support multiple platforms and languages because they are built on Web services.
- They can run on almost any device that has Internet connectivity, including desktop computers and portable computers.

Smart clients can be built to take maximum advantage of the features provided by the host device, and they can be tuned to provide the best user experience for the typical users of these devices.

Architecture Advancements

Smart clients provide the benefits of a rich client model with thin client manageability, but they also provide much more flexibility than traditional rich client applications. Smart client solutions are composed of functionality from more than one client application, with each application collaborating with the others to provide user functionality. Such composite applications integrate client-side software resources into a consistent solution or extend the functionality of an existing application to provide smart client features.

The following characteristics serve as a guide to the features provided by smart clients that are beyond the features provided by traditional rich client applications. If a client application displays these characteristics, it can be said to be a smart client:

- **Utilizes local resources**. A smart client always has code statements on the client that enable local resources to be utilized. They may take advantage of the local CPU or GPU, local memory or disk, or any local devices connected to the client, such as a telephone or bar-code/RFID reader. It may also take advantage of local software, such as Microsoft Office applications, or any installed applications that interact with it.
- **Connected**. Smart clients are never standalone and always form part of a larger distributed solution. This could mean that the application interacts with a number of Web services that provide access to data or a line-of-business (LOB) application. Very often, the application has access to specific services that help maintain the application and provide deployment and update services.
- **Offline capable**. Smart clients work whether or not they are connected to the Internet. Because they are running on the local computer, this is one of the key benefits smart clients offer: they can be made to work even when the user is not connected to the Internet.

Smart clients can take advantage of local caching and processing to enable operation during periods of unavailable or limited network connectivity. This functionality is especially valuable considering the cost, latency, and speed of mobile connections.

However, offline capabilities are not only useful in mobile scenarios; desktop solutions can take advantage of offline architecture to update server systems on background threads. This keeps the user interface responsive and improves the overall end-user experience. This architecture can also provide cost and performance benefits because the user interface does not have to be shuttled to the smart client from a server.

Because smart clients can exchange only the required data with other systems in the background, reductions in the volume of data exchanged with other systems are realized. Even on hard-wired client systems, this bandwidth reduction can realize huge benefits. This increases the responsiveness of the user interface (UI) because the UI is not rendered by a remote system.

Intelligent Install and Update

Smart clients manage their deployment and update in a much more intelligent way than traditional rich client applications. The Microsoft .NET Framework enables applications to be deployed using a variety of techniques, including simple file copy or download over HTTP. Applications can be updated while running and they can be deployed on demand by clicking a URL. The .NET Framework provides a powerful security mechanism that guarantees the integrity of the application and its related assemblies.

Self-Update and ClickOnce

ClickOnce is a new application deployment technology that makes deploying a Windows Forms application as easy as deploying a Web application. This technology is available in the .NET Framework 2.0, which is included in Visual Studio 2005.

With ClickOnce, running a Windows Forms application is as simple as clicking a link on a Web page. To deploy or update an application, administrators have to only update files on a server; there is no need to individually touch every client.

ClickOnce applications are fundamentally low impact for the following reasons:

- Applications are completely self-contained and install for each user; this means that no administrator rights are required.
- A ClickOnce application does not break other applications because it is self-contained. Nonetheless, if your application does have to do something risky at install time, for example installing drivers, you should use Microsoft Windows Installer.

ClickOnce applications can be deployed through Web servers, file servers, DVDs, and so on. They can also be installed. When they are installed, Start menu and Add/Remove program entries are created or they run and are stored in cache. Furthermore, ClickOnce has several ways that it can be configured to automatically check for application updates. Alternatively, applications can use the ClickOnce APIs (**System.Deployment**) to control when updates should happen.

Client Device Flexibility

The .NET Framework together with the .NET Compact Framework provides a common environment on which smart client applications can be built. Often, there will be multiple versions of smart clients, where each smart client targets a specific device type and takes advantage of the device's unique features and provides functionality that is appropriate to its usage.

Technology Updates

This section describes the technology updates that are provided in Visual Basic .NET with respect to Windows and smart clients.

Replacing Resize Logic with Docking and Anchoring

Visual Basic 6.0 requires you to write custom code for the **Resize** event to make sure that controls are correctly displayed on the resized form. This makes it possible for any resizable application to retain the original placement of controls on the form regardless of the size of the form itself. It is expensive to do this because you have to write custom code for each form. If you have only a few forms, this is not big deal, but on a larger scale application, this custom code can be very time consuming to implement.

In Visual Basic .NET, there is no need to write this code because you can accomplish the same result by working with the **Docking** and **Anchoring** properties of the controls in your forms. This section describes some examples about how to efficiently work with these properties to ensure that your .NET application will display controls correctly despite the form's dimensions.

A control's **Anchor** property can be modified so that it is dynamically positioned on screen. When a control is anchored to a form, and the form's size changes, the control's position on the form remains the same relative to the anchor position(s) it holds. By default, the anchoring property is set to the upper-left, as illustrated in Figure 18.1.

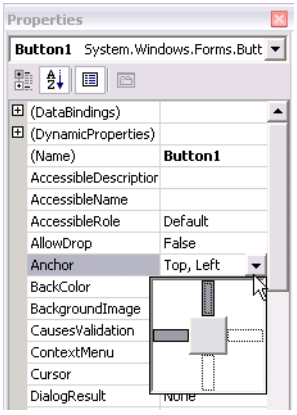

Figure 18.1
*The **Anchor** property of a button on a form*

The dark grey anchors are the ones that are currently set (top, left). To set or unset an anchor, click the target anchor bar.

Figure 18.2 illustrates the form with the button anchored in the bottom, right position before and after the form is resized.

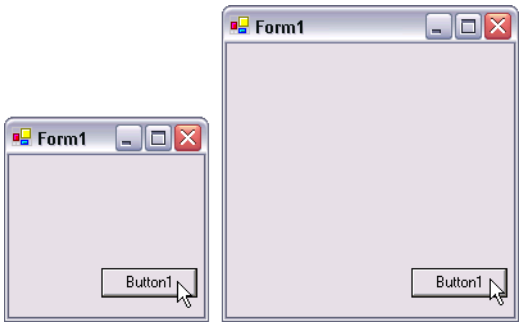

Figure 18.2
Resizing a form with an anchored button

In Figure 18.2, the form on the left displays the initial form, and the form on the right displays the resized form. Anchoring causes the button to automatically maintain a relative position on the form when the form is resized.

When opposing anchoring settings of a control are set, to maintain the same distances from both borders, the size of the control changes if the height or width of the container changes. For example, if a button's top and bottom anchoring attributes are set and the form's height is increased, the button's height also changes. A similar behavior can be observed if the control's anchor is set to left and right and the form's width changes: the width of the button changes to make sure that the distance from both borders is preserved. Figure 18.3 demonstrate three buttons with different anchoring properties and their state after the width and height of the form changes.

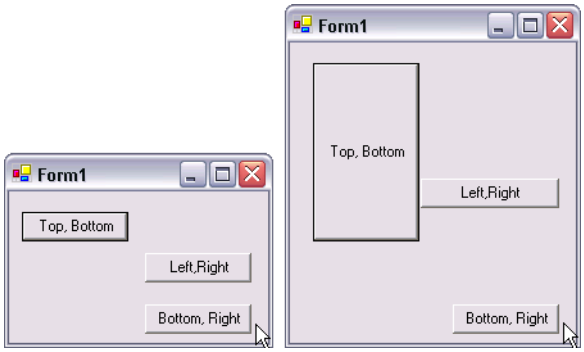

Figure 18.3
Opposing anchoring borders

In Figure 18.3, the form on the left illustrates the initial form, and the form on the right illustrates the form after it is resized.

The **Docking** property of a control allows the control to adhere to the edge of the form or the container in which it resides. The docking behavior in some controls has been available before .NET. In Visual Basic 6.0, the **Toolbar** control exhibited a docking behavior by attaching itself to the top edge of the form. In Visual Basic .NET, you can set the docking property of many controls so that they attach themselves to the side of the container when the size of the form changes.

The docking property of a control can be changed to set the adhering properties when a form is resized. Table 18.1 shows the different docking properties that can be set on controls.

Table 18.1: Docking Properties

Dock Property	Description
Top	The control's top edge is adhered to the top edge of its parent container.
Bottom	The control's bottom edge is adhered to the top edge of its parent container.
Left	The control's left edge is docked to left side of its parent container.
Right	The control's right edge is docked to right side of its parent container.
Fill	All of the control's edges are docked to the edges of its parent; the control will be resized accordingly.
None	The control is not docked.

When you dock a control to the left or right side of the container, the control's height will be the same as the height of its container. Likewise, when you dock a container to the top or bottom edges of the container, its width will be the same as the width of the container where it resides. When multiple docked controls exist at the same time, the second control will dock alongside the first. For example, in Figure 18.4, the button labeled "2-Dock = Right" is docked to the left and adheres itself to the top edge of the button labeled "1-Dock = Top" because this button's docking property was set first.

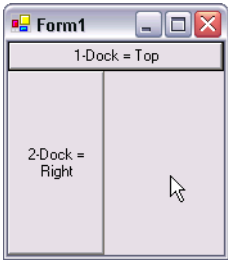

Figure 18.4
*The **Docking** property applied to buttons on a form*

When two (or more) controls are inside the same container and share the same docking properties, the way in which they are docked has a special behavior. They will not be placed on top on each other; instead, they will be placed next to each other. The container whose dock property was set first will be the one closest to the edge of its parent container.

Figure 18.5 illustrates some controls with their docking properties set to different borders on the form. Notice how they retain their docking properties when the form is resized.

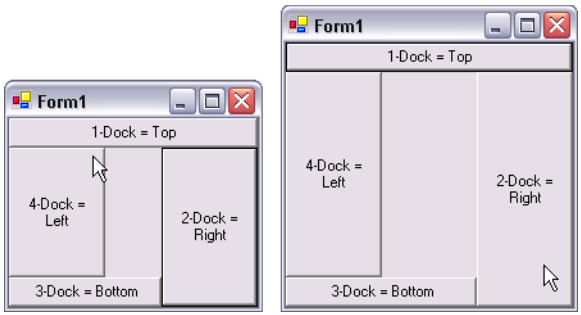

Figure 18.5
Docking applied to a form that is resized

By using the docking and anchoring properties of your controls, you can gain more control over what goes on when the form (or container) is resized. You do not have to write code for the placement of your controls when containers change size. The idea behind anchoring and docking may be a little perplexing at first, but with some practice and testing, you will see and learn how to best implement these properties without relying on resize events and custom code.

Intrinsic Controls

Many controls in Visual Basic 6.0 that enhanced the GUI of an application were available by means of importing ActiveX libraries. For example, if you wanted to use an **ImageList**, **Treeview**, or **Toolbar**, you could easily do this by adding a reference to the Microsoft Common Controls .ocx. After this reference is added, the power of these extremely useful controls was unlocked for you to use in your application.

Many of these components are automatically upgraded to .NET by the Microsoft Visual Basic Upgrade Wizard; however, they are not upgraded to their intrinsic counterparts in Visual Basic .NET. Instead, they are upgraded to ActiveX components. The only exceptions to this are the Microsoft Tabbed Dialog Control and your own custom user controls. For example, when a **Toolbar** control is upgraded to Visual Basic .NET, a wrapper reference to the ActiveX control is added to the references of the project, as illustrated in Figure 18.6 on the next page.

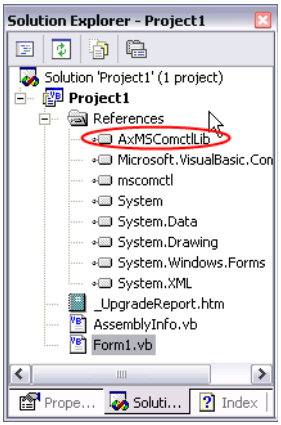

Figure 18.6

The reference to that ActiveX control automatically added by the Visual Basic Upgrade Wizard

Even though most of the ActiveX controls upgraded from Visual Basic 6.0 will work fine when upgraded to Visual Basic .NET, you might be better off substituting them for native Windows Forms controls when you have the opportunity to do so. You can make your project easier to maintain by removing references to ActiveX controls that are not imperative to use. Furthermore, by removing the references, you do not have to worry about their distribution and registration when they are installed on the user's machine. In most cases, the task of programming native Windows controls is easier than everything that is involved with ActiveX control programming.

The level of difficulty of upgrading your ActiveX controls depends on the ActiveX control that you want to upgrade. For instance, if you are using a **TreeView** or **ListView** control that adds some nodes using the **Add** method, this could implicate some work overhead. The **Add** method parameters signature may have changed in .NET, and you will probably have to check your code to obtain the same logic that you had implemented in your Visual Basic 6.0 code.

With ActiveX controls, such as **ImageList** or **Toolbar** controls, be aware that indexes for the collections that these items hold has changed in Visual Basic .NET. The lowest index number in Visual Basic started at 1; in Visual Basic .NET, the lowest index position now starts at position 0. You will need to verify and fix cases like this to avoid invalid index references at run time. For example, take a look at the following code from a Visual Basic 6.0 toolbar event.

```
Private Sub Toolbar1_ButtonClick(ByVal Button As MSComctlLib.Button)
    Select Case Button.Index
    Case 1:
        MsgBox "Button Click 1 Event"
    End Select
End Sub
```

To replicate the same behavior in Visual Basic .NET with the intrinsic counterpart, after removing the old **ToolBar** reference from the designer, the preceding code would have to be changed to address the correct index in Visual Basic .NET. Also note how the **ButtonClick** event signature has also changed and needed to be rewritten accordingly, as shown here.

```
Private Sub ToolBar1_ButtonClick(ByVal sender As Object, ByVal e As _
            System.Windows.Forms.ToolBarButtonClickEventArgs) _
            Handles ToolBar2.ButtonClick
    Select Case ToolBar2.Buttons.IndexOf(e.Button + 1)
        Case 1
            MsgBox("Button Click 2 Event")
    End Select
End Sub
```

You also have to consider that certain ActiveX signatures for events and methods differ from the ones that are used in Windows controls. To retain functional equivalence, you will have to revise all events; and in some cases, you will have to recreate the event to make sure things keep working as they are supposed to.

The required work to port all of your ActiveX components to native Windows Forms controls depends on which controls you used and the intricacy level of the operations you are performing on them. In essence, most of the basic operations are completely represented on the .NET side. More complex procedures might not be as easy to implement, but this does not mean that it is impossible to do — it just might require more work. In Microsoft Visual Studio 2005, the Visual Basic Upgrade Wizard will automatically upgrade most of the ActiveX components to Windows Forms controls.

Providing Visual Feedback with the ErrorProvider Control

Requesting input from a user in your application always involves the risk of a user entering invalid information. The reasons for this might range from a simple error to a malicious attempt to exploit vulnerability in your code. Regardless of the reasons for the error, the fact is that you must have input validation in your code.

If your code determines that a user has entered invalid data, the next step would be to alert the user so that the error is corrected. In Visual Basic 6.0, the most common way to perform this task was to use the **MsgBox** function. This approach worked well for many purposes, but there are some ways in which this input error validation can be improved.

The biggest drawback with using a message box to display an error is that in some occasions, a text message is insufficient. For example, if a form has many input fields and there are several errors made, chances are that the user will not remember all the errors after the message box is closed. This means that the user will have to be reminded of the errors more than once to correct the problems with the input.

In Visual Basic .NET, **ErrorProvider** control can be used to report errors and provide a better way to let the user know where the error is. The main difference between using this control and the message box approach is the means by which the user is alerted about the error. Instead of being presented with a text message, the input validation controls that require attention in your form will display an icon to let the user know exactly what and where the problem is. When the mouse is placed over the error notification icon, a ToolTip will be displayed to further assist with the error resolution procedure.

The **ErrorProvider** control is very easy to incorporate in your current controls. By using the **ErrorProvider**, you can be assured that utilizing this functionality will most likely be more efficient and error free than using a custom implementation to achieve the same results.

The next example demonstrates the use of the **ErrorProvider** control. Suppose you have a simple form that requests user information to be stored into a database, as shown in Figure 18.7.

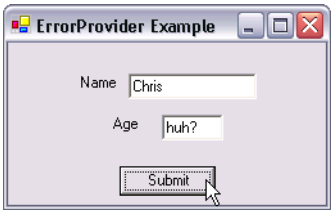

Figure 18.7
A sample input form with invalid input data

The form has a very obvious mistake that the user should be alerted about. This can be accomplished by using the **ErrorProvider** control to let the user know where and what the problem is. You can add the **ErrorProvider** control by accessing the **Tools** pane and selecting the corresponding tool icon, as shown in Figure 18.8.

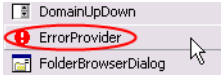

Figure 18.8
*Selecting the **ErrorProvider** control*

The following code shows all that needs to be done to alert the user where the problem is.

```
…
If Not validAge(txbAge.Text) Then
    textErrorProvider.SetError(txbAge, "Age should only be a number")
End If
…
```

The **SetError** function takes as parameters the control to which you would like to indicate an error and the text to be displayed when the user places the mouse over the error icon. Figure 18.9 illustrates the form after the invalid input has been entered and the **ErrorProvider** is notified to mark it.

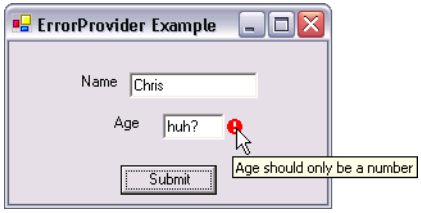

Figure 18.9
*The **ErrorProvider** indicates an invalid input in a text box control*

The **ErrorProvider** control is very easy to implement and is better than the message box approach to display errors. It clearly indicates where the problem lies and is fully customizable. For more information about this powerful feature, see "ErrorProvider Class" in the *.NET Framework Library* on MSDN.

FlowLayoutPanel and TableLayoutPanel

In Microsoft Visual Studio 2005, two new containers are available that facilitate control placement: the **FlowLayoutPanel** and the **TableLayoutPanel**. These controls differ from the **Panel** container control that was previously available in the way in which controls are placed within it. The two new controls offer full customization for how and where you would like your contained controls to be displayed. Both controls can be added to your forms by accessing the containers category in the Toolbox pane in Microsoft Visual Studio .NET, as shown in Figure 18.10.

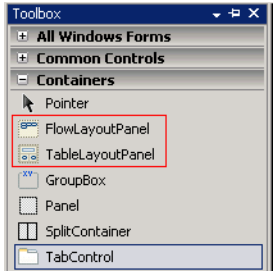

Figure 18.10
*The **FlowLayoutPanel** and **TableLayoutPanel** in the Toolbox pane*

The **FlowLayoutPanel** makes it easy to automatically position controls when they are added to the container. When controls are added to the panel, they will flow horizontally and vertically to fit within the container. This makes it quick in situations where you would like to deal with perfectly aligned tuples of several controls

(for example, several text boxes and labels). In Figure 18.11, the **Label1** and the bottom-level text box controls were added first. Subsequent addition of controls shifted each existing control downward.

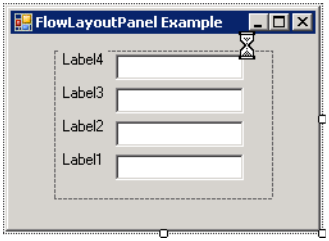

Figure 18.11
*An example **FlowLayoutPanel***

You can change the order in which controls flow within the **FlowLayoutPanel** by changing the values of the **FlowDirection** property. Table 18.2 and Figure 18.12 show the values and behavior of the different values that can be set in the **FlowLayout** property.

Table 18.2: FlowLayoutPanel Properties

FlowLayoutPanel Property	Description
LeftToRight (default)	When a new control is added, it will be placed to the right of any existing controls. If the **WrapContents** value is set to **true** (default), contents will also shift downward.
TopDown	New controls are placed below existing ones, resembling an HTML table with several rows and just one column.
RightToLeft	When a new control is added, it will be placed to the left of any existing controls in the panel. This is the opposite behavior of **LeftToRight**.
BottomUp	New controls are placed above existing ones.

If you would like to be in more control of the placement of the controls within a panel, you will definitely enjoy working with the **TableLayoutPanel**. Just like its HTML counterpart, this control offers a row-column approach by which to arrange your controls. A noticeable difference is that only one control can be placed in each of the cells in the panel.

The **TableLayoutPanel** offers a quick and efficient implementation to solve the problems that have long plagued programmers when dealing with the GUI section of their application. Aligning controls so that they are perfectly positioned with each other is very easy using this control. You need only to define the number of rows and columns and start placing them. Whenever it is necessary to add new controls between existing controls that have already been placed, you need only to add a new column to the existing table layout and shift items accordingly.

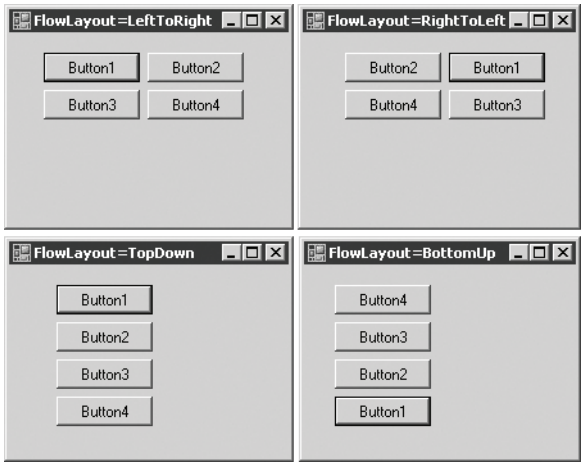

Figure 18.12
Variations of the FlowLayoutPanel automatic positioning

The **TableLayoutPanel** is fully customizable at design time. By accessing the contextual menu, you can carry out tasks such as adding columns and rows, removing columns and rows, and accessing the powerful **Columns** and **Rows** editor. Figure 18.13 illustrates the different options available to edit the **TableLayoutPanel** from the design-time contextual menu.

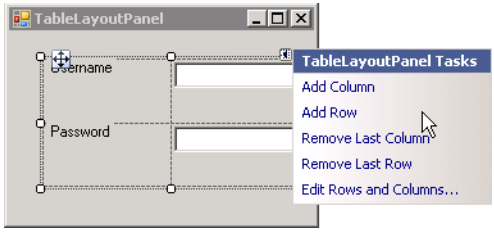

Figure 18.13
*The **TableLayoutPanel** shortcut menu*

Windows XP Look and Feel

If you are a regular Windows user, you may be familiar with the use of themes throughout the system. The option to use themes is turned on by default; thus users are encouraged to use the different available themes and variations on their computers.

Applications created with the .NET Framework version 1.0 will not support themes. Starting with the .NET Framework version 1.1, Windows XP visual styles are available. This behavior will be supported when Visual Studio 2005 ships because every form that is created will support themes by default. Figure 18.14 on the next page illustrates a Visual Basic .NET application with radio button and button controls.

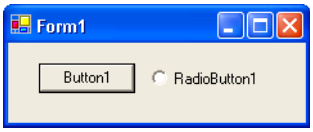

Figure 18.14
A .NET application without themes support

The buttons in Figure 18.14 look like they are using the Windows 2000 theme. The Windows XP style theme's buttons have rounded corners and offer a light blue border; the buttons in this graphic do not have these features.

Fortunately, when you are using the .NET Framework version 1.1 or later, changing your Visual Basic .NET application's code so that it supports themes is fairly straightforward. This can be done by following two approaches: invoking the **EnableVisualStyles** static method or using a manifest file.

The **EnableVisualStyles** method was introduced in the .NET Framework version 1.1 and allows your application to enable visual styles (if the operating system supports it). The method invocation causes all controls in your applications to be drawn using visual styles. Therefore, it is important to call this method before any controls are drawn in your application. To demonstrate this, the following code example can be added before invoking the **InitializeComponent()** method for the preceding example.

```
Application.EnableVisualStyles()
Application.DoEvents()
' This call is required by the Windows Form Designer.
InitializeComponent()
```

Furthermore, you will have to set the **FlatStyle** property of all controls that support it to **System**. Running the same application again yields a completely different look than the original example, as shown in Figure 18.15.

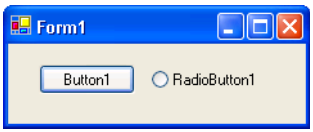

Figure 18.15
A .NET application with the standard Windows XP look and feel

Another option you can use to apply themes to your application is to produce a manifest file. The manifest file includes information that will let your application know which version of the common controls library to use. By specifying that the application should use version 6.0 of the common controls library, you can be assured that all your controls will be drawn according to the selected theme.

To create the manifest file, copy the following text to a file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="Executable Name"
    type="win32"
/>
<description>Application description</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
</assembly>
```

You will have to save the file with the same name of your application and append the extension ".manifest." For example, if the application is named WindowsApplication1.exe, you would save the manifest file as WindowsApplication1.exe.manifest and place it in the same directory where the executable resides. You also have to set the **Flatstyle** property for any control **System**. Running the application by using the manifest file shows the controls displaying the Windows XP styled theme the operating system currently has, as shown in Figure 18.15.

Each of the options previously mentioned have their own strengths and weaknesses. The manifest file does not involve writing code in your application to work, but using it risks having the user delete the manifest file and render your application themeless. The **EnableVisualStyles** procedure is easy to implement, but it requires the modification of your source code.

The question of whether you should support themes in your application depends on how complete you would like your application to look. As previously mentioned, themes in Windows XP are on by default. An application that does not support themes may send the wrong message to your users; it might appear as if it was unfinished or not fully supported.

Localization and Resources

Both Visual Basic 6.0 and Visual Basic .NET support localization of your applications. That is, they support mechanisms to produce different presentations of your application for different languages. Using a single code base, you can distribute versions of your application in English, Spanish, and any other language you want.

If you plan to dynamically determine what language your application will run in, you must call some Win32 functions in Visual Basic 6.0 to determine the current locale. The following code demonstrates this.

```
' Declare Win32 functions for getting locale information
Private Declare Function GetLocaleInfo Lib "kernel32" Alias "GetLocaleInfoA" _
(ByVal Locale As Long, ByVal LCType As Long, ByVal lpLCData As String, ByVal
cchData As Long) As Long
Private Declare Function GetUserDefaultLCID Lib "kernel32" () As Long

' Return a 3 letter abbreviation of the current locale
Function GetLocale() As String
Static locale_name As String
    If locale_name = "" Then
        Dim length As Long
        Dim locale_id As Long
        Dim buf As String * 1024
        locale_id = GetUserDefaultLCID()
        length = GetLocaleInfo(locale_id, &H3, buf, Len(buf))
        locale_name = Left(buf, length - 1)
    End If
    GetLocale = locale_name
End Function
```

In Visual Basic .NET, this same information can be read from the **CultureInfo** object in the current thread. This is demonstrated in the following code.

```
' Return a 3 letter abbreviation of the current locale
Function GetLocale() As String
    Return Thread.CurrentThread.CurrentUICulture.ThreeLetterWindowsLanguageName()
End Function
```

Resource Files

In Visual Basic 6.0, resource files are used to store the language specific data that your application uses. This data can be strings, images, or other types of resources. A Visual Basic 6.0 project can only contain a single resource file.

One way to add localization features to your application is to place versions of all your data for the different supported languages in a single resource file. If you do that, you can separate the localized data by placing them in separate string tables, for example, but because only one string table can be stored in an executable at a

time, it is more common to place all the strings in a single table and use suitable offsets to access them.

```
' Determine the offset to use for the localized resources
Private Function GetOffsetForLocale() As Long
    GetOffsetForLocale = 100
    If GetLocale() = "ENC" Then
        GetOffsetForLocale = 200
    ElseIf GetLocale() = "ESC" Then
        GetOffsetForLocale = 300
    End If
End Function

Private Sub Form_Load()
    Picture1.Picture = LoadResPicture(GetOffsetForLocale() + 1,
LoadResConstants.vbResBitmap)
    Label2.Caption = LoadResString(GetOffsetForLocale() + 1)
End Sub
```

Alternatively, you can store each language in different resource files; the resource files are compiled into independent DLLs. At run time, you dynamically load the DLL that contains the localized resources you need.

```
Dim objResoures As Object
Set objResoures = CreateObject("MyApp" & Hex$(GetLocale()) & ".Resources")
If objResoures Is Nothing Then
    objResoures c = CreateObject("MyAppENU.Resources") ' Default Locale
End If
' Load resources from objResoures
```

Visual Studio .NET provides a much more integrated level of support for localization than Visual Studio 6.0 did. If you set the **Localizable** property of your form to **True**, Visual Studio creates a resource file for that form and stores the text strings for different languages in separate resource files that are automatically compiled into satellite DLLs. These DLLs are loaded dynamically and automatically at run time, depending on the current locale of the user's computer. Alternatively, and for more control, you can do all this manually if you have to. For more information about how to do this, see "Walkthrough: Localizing Windows Forms" in *Visual Basic and Visual C# Concepts* on MSDN.

Visual Studio .NET does not provide any direct support for adding images to these resource files; however, the .NET Framework SDK provides the source code for a resource editor named ResEditor that does support images. Alternatively, you can embed all the images into the main assembly as resources. However, this could be overkill if you have a lot of images for different locales and you do not expect to need more than one locale per installation.

For more control over the assemblies that contain your resources, you have to use a class named **ResourceManager** from the **System.Resources** namespace. You create a separate instance of this class for each assembly that contains resources, and this class manages the loading of all resources from that assembly. To create a **ResourceManager** and retrieve resources from an assembly, you can use code similar to the following.

```
' You have to import the namespaces System.Reflection
' and System.Resources.

' Gets a reference to the assembly that contains the
' resources.
Dim myResourceAssembly As Assembly
myResourceAssembly = Assembly.Load("ResourceAssembly")

' Creates the ResourceManager.
Dim myManager As New ResourceManager("ResourceNamespace.myResources",
myResourceAssembly)

Dim myString As System.String
Dim myImage As System.Drawing.Image
myString = myManager.GetString("StringResource")
myImage = CType(myManager.GetObject("ImageResource"), System.Drawing.Image)
```

ToolTips

In Visual Basic 6.0, you could associate a ToolTip with a control by setting the **ToolTipText** property of that control. If a control had its **ToolTipText** property set to a non-empty string, it would display the text when the mouse was allowed on rest on that control. To disable the ToolTips for a form, it was necessary to go through all the controls in that form and set their **ToolTipText** properties to the empty string. Without using Win32 calls, this was about the extent of the functionality that was available to Visual Basic 6.0 programmers.

In Visual Basic .NET, ToolTips work a little differently. Each form has its own ToolTip component. ToolTips are attached to controls by calling the **SetToolTip()** method of the ToolTip component.

Additionally, in Visual Basic .NET, some new possibilities for easily customizing the behavior of your ToolTips are available.

Disabling ToolTips

You can disable all the ToolTips in a form with a single command in Visual Basic .NET. The **Active** property determines whether the ToolTips register in this ToolTip component are displayed. In Visual Basic 6.0, you could do this only by iterating through all the components in a form and learning the ToolTip for each control. Of course, you still do that in Visual Basic .NET, but it is more efficient to just set the ToolTip component's **Active** property to **False**.

```
' Visual Basic 6.0
For Each Control in Me.Controls
    Control.ToolTipText = ""
Next

' Visual Basic .NET
toolTip1.Active = False
```

Changing the ToolTip Delay

New to Visual Basic .NET is the possibility to change the delays associated with the ToolTips for a form. To change the amount of time that the mouse must remain still over a control before the ToolTip is displayed, you can modify the **ToolTip.InitialDelay** property. The default time for this property is half a second. To change the amount of time that the ToolTip remains visible after it displays, change the **AutoPopDelay** property, which is set to five seconds by default. However, the ToolTip will disappear immediately after the mouse leaves the region the ToolTip applies to. A property named **ReshowDelay** determines the length of time it takes for subsequent ToolTips to appear as the pointer moves around the form.

A separate property can be used to set all of the preceding properties appropriately. This property is named **AutomaticDelay** and when you set this property, the same value is automatically copied into the **InitialDelay** property, a value that is ten times as large is placed in the **AutoPopDelay** property, and finally, a value that is one fifth as large is copied into the **ReshowDelay** property. This is demonstrated in the following code example.

```
' Set the delay property as measured in milliseconds
toolTip1.AutomaticDelay = 500
' Specify the length of time before ToolTip appears
toolTip1.InitialDelay = 500
' Specify the length of time before ToolTip disappears
toolTip1.AutoPopDelay = 5000
' Specify the length of time before ToolTip appears after moving to a new control
toolTip1.ReshowDelay = 100
```

> **For Visual Basic 2005:**
>
> Visual Studio 2005 adds additional possibilities for customizing ToolTips, including the ability to change the color of the ToolTip or even to allow you to paint the entire ToolTip yourself. For more information, see “ToolTip Class” in the .NET Framework Class Library on MSDN.

Business Components (Enterprise Services)

The .NET Framework provides enterprise services for building highly scalable solutions. From the code itself, you can set properties required for enterprise services components or even deployment settings can be settled from the code. This is a great advantage for developers because you have the possibility of handling all those settings within the code.

This section provides guidelines to improve the architecture of an upgraded application to take advantage of the .NET Framework Enterprise Services.

Advancing the Application Architecture

It is important to know which properties are available in the Enterprise Services, so you first must know that Enterprise Services divides COM+ catalog properties into two categories:

- Properties the developer primarily determines, such as **Transactions**.
- Properties the administrator primarily determines, such as **Identity**.

Properties developers determine are exposed as attributes in the **System.EnterpriseServices** namespace and may be added to the assembly by applying these attributes to the assembly, classes, interfaces or methods.

Properties the administrator determines may be set through the Component Services administration tool after the assembly is registered or by using the COM+ Administrative interfaces.

Enterprise Services is a powerful technology that is very useful when building complex applications that require specific technologies such as transactions, pooling, or queuing.

One of the most confusing aspects of Enterprise Services is when to use them. If your existing systems use COM+ extensively, you should use Enterprise Services when upgrading your applications to .NET. However, if you do not currently use COM+, you may be wondering how to start using them and what the benefits are. The following sections detail these considerations.

Using Enterprise Services and Attributes

Before starting to work with enterprise applications, there are some basic attributes that an enterprise application must have. These are explained here.

To create a .NET class for an enterprise service component, the class must derive from the **System.ServicedComponent** class to exploit COM+ services. To do this, it is necessary to import the .NET Enterprise Services namespace. For some special-purpose .NET namespaces, it is necessary to add a new reference to the corresponding assembly because that they are not included by default. This is the case for the

System.EnterpriseServices namespace, so a reference to the associated assembly must be added.

► **To add a reference to this assembly**

1. In Solution Explorer, select the project where the reference is going to be added.
2. Right-click the **References** folder, and then click **Add Reference**.
3. Click the **.NET** tab. In the components list, look for **System.EnterpriseServices** and double-click it, as shown in Figure 18.17.

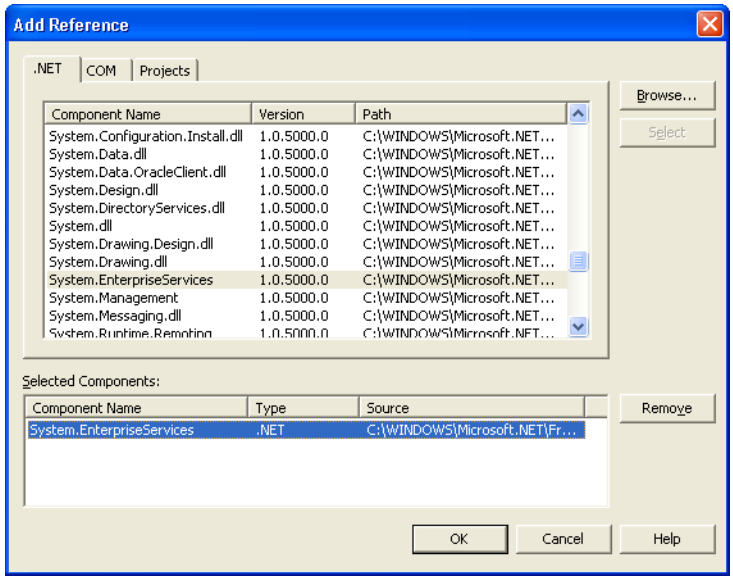

Figure 18.16
*Adding a reference to the **System.EnterpriseServices** assembly*

4. Click **OK**.

After the reference is added, the **System.EnterpriseServices** namespace can be imported in your application to allow your class to extend from **System.EnterpriseServices.ServicedComponent**. This is demonstrated in the following code.

```
Imports System.EnterpriseServices
Public Class _myClass
Inherits System.EnterpriseServices.ServicedComponent
```

It is possible to specify several attributes for the class, depending on the specific requirements for your COM+ application. The following code specifies how to declare attributes on the class.

```
<ObjectPooling()> Public Class _myClass
```

The next sections describe the characteristics and possibilities that you can use to take advantage of Enterprise Services using the Microsoft .NET Framework.

COM+ is, in large part, a merging of Microsoft Transaction Server (MTS) and Microsoft Message Queuing (also known as MSMQ) into the base framework of COM. It is basically the unification of COM, MTS, and Message Queuing, in addition to COM extensions, MTS extensions, and the addition of other services.

These services include self-describing components, queued components, events, security, transactions, and load balancing.

Self-Describing Components

The .NET Framework allows developers to specify attributes for COM components that will become part of the component's description. These class attributes are runtime traits that are applicable to components. System administrators can specify attributes using administrator tools. At design time, developers can use .NET Framework class attributes to specify these characteristics for a self-describing component. For example, a designer might stipulate that a component requires transaction, as shown here.

```
<Transaction>Public Class _myClass
    Inherits System.EnterpriseServices.ServicedComponent
    <AutoComplete()> Public Sub MyRoutine()

    End Sub
End Class
```

This characteristic allows the class to manage transactions without manually declaring any transaction management like commit or rollback transactions. This definition automatically detects the database operations to handle all the transactions in a transparent way with no additional code.

When a component is instantiated, an attribute can be passed to it. The attribute can be a configuration file that indicates to the component which system resources must be used, including database systems and application servers. For more information about the types of attribute classes that can be specified, see "System.EnterpriseServices Namespace" in the *.NET Framework Class Library* on MSDN.

Technology Updates

The following sections detail the technology updates available to your application with respect to Enterprise Services.

Working with Threads

COM+ manages threads for you. Every COM component has a **ThreadingModel** attribute that you can specify when you develop the component. This property determines how the component's objects are assigned to threads for method execution.

Threads can be associated with two possible types of apartments:

- Single Threaded Apartments (STA)
- Multi Threaded Apartments (MTA)

STAs specify that only one thread is being executed. They are implemented as hidden windows that receive messages and dispatch method calls to the objects residing in the apartment. There can be multiple objects in an STA and there can be multiple STAs in a process.

MTAs have a pool of executing threads. Synchronization must be considered in this scenario. Operating system synchronization mechanisms such as semaphores, critical sections, or mutexes can be used. MTAs are restricted to one per process.

All COM objects that reside in a multithreaded apartment can receive method invocations directly from any of the threads that belong to the same apartment. Threads in a multithreaded apartment use a model referred to as *free-threading*.

The different types of processes can be defined as follows:

- A process that consists of just one STA is referred to as a single-threaded process.
- A process that has two or more STA and no MTA is referred to as an apartment model process.
- A process that has a MTA and no STA is referred to as a free-threaded process.
- A process that has a MTA and one or more STA is referred to as a mixed model process.

In reality, all processes are apartment model processes, and some apartments have a single thread while others have multiple threads. The threading model applies to an apartment, not to a process. It can also apply to a class of objects, but not a component, such as a DLL; instead, it applies to the object classes within the DLL. Different classes in a DLL can have different threading models.

A process can use both the apartment and free-threaded models, as long as it uses only one free-threaded apartment. It can have more than one single threaded apartment. Invocations to objects in the STAs will be automatically synchronized by Win32 and data will have to be marshaled whenever an apartment barrier is crossed. For more information about threading models, see "Processes, Threads, and Apartments" on MSDN.

When the Thread Neutral Apartment model (TNA) is used, the components are automatically marked as Free Threaded or Apartment. When this model is used, components inherit the same thread type as the thread that made the invocation.

When a thread executes a method contained in a COM object, and the method creates a new object, MTS suspends the current thread and creates a new one to handle the new object. In the TNA model, apartments can have multiple threads.

The following code example demonstrates how to set the apartment state of a thread in Visual Basic 2005; the methods **SetApartmentState** and **GetApartmentState** are only supported in Visual Basic 2005.

```
Imports Microsoft.VisualBasic
Imports System
Imports System.Threading

Public Class ApartmentTest

    Shared Sub Main()

        Dim newThread As Thread = New Thread(AddressOf ThreadMethod)
        newThread.SetApartmentState(ApartmentState.MTA)

        ' The following line is ignored because
        ' ApartmentState can only be set once.
        newThread.SetApartmentState(ApartmentState.STA)

        Console.WriteLine("ThreadState: {0}, ApartmentState: {1}", _
            newThread.ThreadState, newThread.GetApartmentState())

        newThread.Start()

        ' Wait for newThread to start and go to sleep.
        Thread.Sleep(300)
        Try
            ' This causes an exception because newThread is sleeping.
            newThread.SetApartmentState(ApartmentState.STA)
        Catch stateException As ThreadStateException
            Console.WriteLine(vbCrLf & "{0} caught:" & vbCrLf & _
                "Thread is not In the Unstarted or Running state.", _
                stateException.GetType().Name)
            Console.WriteLine("ThreadState: {0}, ApartmentState: " & _
                "{1}", newThread.ThreadState, newThread.GetApartmentState())
        End Try

    End Sub

    Shared Sub ThreadMethod()
        Thread.Sleep(1000)
    End Sub

End Class
```

For more information about COM interoperability and thread programming in .NET, see "Interop Marshaling Overview" in the *.NET Framework Developer's Guide* on MSDN.

Working with Transactions

The transaction features of COM+ are a repackaging of MTS. COM+ provides a new interface with individual get/set methods for each bit: **SetComplete**, **EnableCommit**, **SetAbort**, and **DisableCommit**.

Distributed transactions can also be thought of as automatic or declarative transactions in .NET. Performance is only one aspect of the decision making process related to your strategy for transaction management in your application. The other considerations are ease of development and flexibility and future expansion.

If you use a **SqlTransaction** object or **OleDbTransaction** object to manage transactions, you are going to have to manage the transaction yourself. You must consider how you will flow that transaction between objects and which object will be the root of the transaction responsible for calling **Commit** or **Rollback**. At first, this may not seem like a difficult issue, but if you have several objects that require different transactions and could have database triggers, or dependencies between that object and complex operations on your code? This task will become a little messy, and transactions will help you to ensure that you do not require any special logic to determine when the object is the root of a transaction and when it is not.

> **For Visual Basic 2005:**
>
> The .NET Framework version 2.0 includes a new transaction programming paradigm. This functionality is included in the **System.Transactions** namespace and provides features such as asynchronous work, events, security, concurrency management, and interoperability.
>
> For more information about the **System.Transactions**, see "Introducing System.Transactions in the .NET Framework 2.0" on MSDN.

Another important point to consider is the flexibility of the application. What if in the future the inventory tables move to a separate database? What if the inventory object is moved to a remote server? What if one of the objects wants to dispatch to a transactional message queue? In these cases, a system built with manual transaction management will have problems. If the system was built from the ground up using Enterprise Services to provide transaction management, none of these contingencies will require you to change the code. In each case, the system will automatically manage and flow the transaction between all the resource managers whether they are SQL databases or other transactional resources such as message queues. Additionally, with Enterprise Services you can create a component that participates in a distributed transaction using the Compensating Resource Manager (CRM). This provides new opportunities for executing business logic when the transaction outcome is known.

Performance

The performance and robustness gains that can be obtained when using .NET Enterprise Services and COM+ are directly related to the scalability provided by these technologies. In a scalable system, the performance and the response time that a client perceives is not degraded when the quantity of client requests is increased. In COM+, this is achieved with features such as load balancing, object pooling, and improved transaction management.

For detailed information about .NET Enterprise Services performance, see ".NET Enterprise Services Performance" on MSDN.

Queuing

Message Queuing (also known as MSMQ) constitutes a platform that supports flexible and reliable communication between applications. A task is asynchronously invoked when it is called without waiting for it to terminate and produce a result. Using this type of execution allows applications to have better responsiveness and improved usage of the available resources. This mode of processing can be achieved using COM+ Queued Components and Message Queuing. For more information about Message Queuing, see the "Message Queuing and Queued Components" section in Chapter 15, "Upgrading MTS and COM+ Applications."

When using normal execution mode, the callers of a COM+ component method will wait until the method returns. Conversely, when a component is used in queued mode, the client-side stub will store interactions with the component and when the component is released, the interactions are delivered as Message Queuing messages with the appropriate properties for transaction management and recovery among others. When the destination queue receives the messages, the interactions are interpreted and executed.

If you want to create a queued component in your application, first declare the interface that you want to call asynchronously through queued components and put the **InterfaceQueuing** attribute on the interface. The methods of this interface must not use any return values or **ByRef** arguments.

```
Imports System.EnterpriseServices

<InterfaceQueuing()> Public Interface IClass1
    Sub MyProcess(ByVal i As Integer)
End Interface
```

Your queued interface must be declared as public. If it is not, when your assembly is registered with COM+, it will not include the queued interface in the registration.

You have to specify that this assembly will be a queued component. This is done in the assembly file using the **ApplicationQueuing** attribute.

When an attribute is declared, the assembly will specify this queued component must have a special behavior and it is applied to all the objects inside the component.

```
<Assembly: ApplicationQueuing()>
```

After that you must provide an implementation of this interface in a server application. The **ApplicationQueuing** attribute instructs the server application to listen on a queue. The following code demonstrates how to implement a queued interface.

```
Public Class Class1
    Inherits EnterpriseServices.ServicedComponent
    Implements IClass1

    Public Sub MyProcess(ByVal i As Integer) Implements IClass1.MyProcess
        ' Add the code.
    End Sub
End Class
```

It is important to consider that the queued components infrastructure does not support serializable objects. As a result you must consider the following conditions:

- Pass simple types (such as int, double, or string).
- Serialize the object(s) to XML using the XML serializer and pass them as strings.
- Serialize the object(s) to binary using the binary serializer and pass them as a byte array (a simple type).

Security in COM+

COM+ has security features similar to the features available in MTS, with the addition of the new security infrastructure provided by Windows XP. The application, component, and method levels can contain access controls. The user has control over the user accounts and groups associated to roles. These facilities allow developers to define logical security models based on the heterogeneous services offered by the underlying platform.

Microsoft Active Directory constitutes the main security component in Windows XP. Although Public Key Infrastructure (PKI) and NT LAN Manager (NTLM) are supported, Active Directory uses Kerberos as the primary security service. The PKI Kerberos extensions are implemented by Active Directory and allow users to be authenticated with X.509 certificates. Kerberos ticket-granting ticket (TGT) and session keys can be obtained when one of the X.509 certificates has been obtained.

Load Balancing

The COM+ load balancing architecture allows you to define a router on a server that forwards object creations to a computer that has low utilization. After the object is created, method calls go to the same object even when it is stateless. There are wizards for all common tasks and a single **Application Explorer** window for management.

COM+ Events

The COM+ Events service is an automated, loosely coupled event system that stores event information from different publishers in the COM+ catalog. Subscribers can query this store and select the events that they want to hear about. For more information about events, see the "COM+ Events" section in Chapter 15, "Upgrading MTS and COM+ Applications."

For complete information about the System.EnterpriseServices namespace in .NET, see "System.EnterpriseServices Namespace" in the *.NET Framework Class Library* on MSDN.

Summary

Taking advantage of the benefits available in Visual Basic .NET and the .NET Framework in general means advancing your upgraded application with new features and technologies. This chapter presented some suggestions for possible application advancements based on the type of application. Whether it is a Windows application, a business application, or some other category of application, the information provided should give you an idea of the types of advancements you can make from an application-category perspective.

Advancement is also possible for Web applications. The next chapter will provide an overview of possible advancements for Web applications.

More Information

For more information about the ErrorProvider control, see "ErrorProvider Class" in the *.NET Framework Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemwindowsformserrorproviderclasstopic.asp.

For more information on localizing, see "Walkthrough: Localizing Windows Forms" in *Visual Basic and Visual C# Concepts* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbwlkWalkthroughLocalizingWindowsForms.asp?frame=true.

For more information on ToolTips, see "ToolTip Class" in the .NET Framework Class Library on MSDN:
http://msdn2.microsoft.com/library/ecft989x(en-us,vs.80).aspx.

For more information about the types of attribute classes that can be specified, see "System.EnterpriseServices Namespace" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystementerpriseservices.asp.

For more information about threading models, see "Processes, Threads, and Apartments" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/html/cb62412a-d079-40f9-89dc-cce0bf3889af.asp.

For more information about COM interoperability and thread programming in .NET, see "Interop Marshaling Overview" in the ".NET Framework Developer's Guide" on MSDN:
http://msdn2.microsoft.com/library/eaw10et3(en-us,vs.80).aspx.

For more information about the **System.Transactions**, see "Introducing System.Transactions in the .NET Framework 2.0" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/introsystemtransact.asp.

For detailed information about .NET Enterprise Services performance, see ".NET Enterprise Services Performance" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomser/html/entsvcperf.asp.

For complete information about the **System.EnterpriseServices** namespace in .NET, see the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystementerpriseservices.asp.

19

Advancements for Common Web Scenarios

Web applications are commonplace in modern computing. They allow a business to maintain data and services in a central location while also permitting access from anywhere in the world. This makes them a popular type of application to create.

Visual Basic .NET provides many classes, objects, and language features that make it possible to rapidly build robust and powerful Web applications. This chapter provides an introduction to these features so that you can decide how to take advantage of them in your own applications.

Web Applications and ASP.NET

ASP.NET enables the rapid creation of Web applications and Web components that use Web Forms and XML Web services. Visual Basic .NET Web Forms expands the DHTML model to provide richer dynamic user-interface capabilities as well as client-side validation. In Visual Basic .NET, ASP.NET Web Application projects are used to create Web applications. ASP.NET offers significant improvements over ASP in the areas of performance, state management, scalability, configuration, deployment, security, output cache control, Web farm support, and Web service infrastructure.

Developers can use Web Forms or Web services when they create an ASP.NET application, or combine them in any way they see fit. Each is supported by the same infrastructure that allows you to use authentication schemes, cache frequently used data, or that customizes your application's configuration. In particular:

- Web Forms allow you to build powerful forms-based Web pages. When building these pages, you can use ASP.NET server controls to create common UI elements, and program them for common tasks. These controls allow you to rapidly build a Web Form out of reusable built-in or custom components to simplify the code for a page.

- An XML Web service provides the means to access server functionality remotely. Using Web services, businesses can expose programmatic interfaces to their data or business logic, which in turn can be obtained and manipulated by client and server applications. Web services enable the exchange of data in client-server or server-server scenarios, using standards such as HTTP and XML Messaging to move data across firewalls. Web services are not tied to a particular component technology or object-calling convention. As a result, programs written in any language, using any component model, and running on any operating system can access Web services.

The following descriptions highlight key features of ASP.NET:

- ASP.NET provides a model that enables Web developers to write logic that runs at the application level. Developers can write this code in the Global.asax text file or in a compiled class deployed as an assembly. This logic can include application-level events, but developers can easily extend this model to suit the needs of their Web application.
- ASP.NET provides application and session-state facilities.
- For advanced developers who want to use APIs as powerful as the ISAPI programming interfaces that were included with previous versions of ASP, ASP.NET offers the **IHttpHandler** and **IHttpModule** interfaces. Implementing the **IHttpHandler** interface gives you a means of interacting with the low-level request and response services of the IIS Web server and provides functionality much like ISAPI extensions, but with a simpler programming model. Implementing the **IHttpModule** interface allows you to include custom events that participate in every request that is made to your application. For more information about the **IHttpHandler** and **IHttpModule** interfaces, see the "HTTP Modules" section later in this chapter.
- All ASP.NET code is compiled instead of interpreted, which allows early binding, strong typing, and just-in-time (JIT) compilation to native code. ASP.NET is also easily factorable, which means that developers can remove modules (for example, a session module) that are not relevant to the application that they are developing. ASP.NET also provides extensive caching services, both built-in services and caching APIs. It also ships with performance counters that developers and system administrators can monitor to test new applications and gather metrics on existing applications.
- ASP.NET offers the **TraceContext** class, which allows you to write custom debug statements to your pages as you develop them. They appear only when you have enabled tracing for a page or an entire application. Enabling tracing also appends details about a request to the page, or, if you specify, to a custom trace viewer that is stored in the root directory of your application.
- The .NET Framework and ASP.NET provide default authorization and authentication schemes for Web applications. You can easily remove, add to, or replace these schemes, depending on the needs of your application.

- ASP.NET configuration settings are stored in XML-based files, which are human readable and writable. Each of your applications can have a distinct configuration file and you can extend the configuration scheme to suit your requirements.

Architectural Advancements

This section provides an overview of the ASP.NET subsystem and infrastructure. Figure 19.1 shows the relationships among the security systems in ASP.NET.

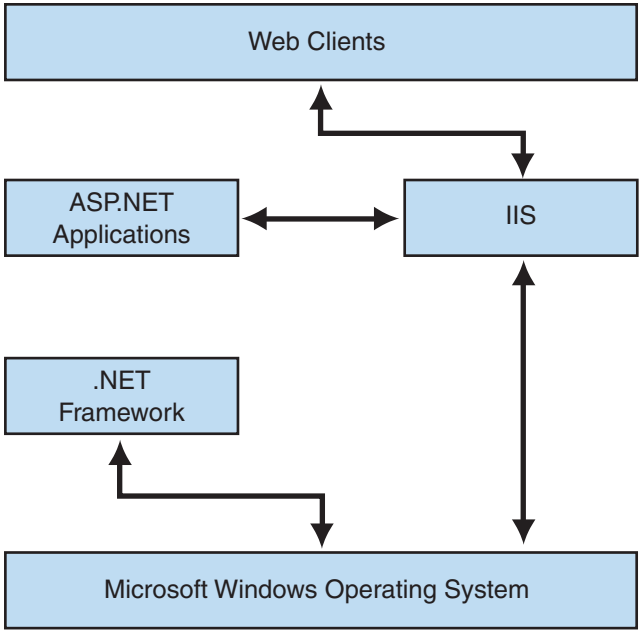

Figure 19.1
The ASP.NET security systems relationships

As shown in Figure 19.1, Web clients use the Microsoft Internet Information Services (IIS) to communicate with ASP.NET applications. IIS interprets and optionally authenticates the request. If **Allow Anonymous** access is set to true, no authentication occurs. IIS also finds the requested resource (such as an ASP.NET application), and, if the client is authorized, it returns the appropriate resource.

ASP.NET incorporates a variety of features and tools that allow you to design and implement high-performance Web applications. These features and tools include:

- A process model that has improvements over the model in ASP.
- A feature that automatically detects changes in ASP.NET pages, dynamically compiles changed pages, and stores the newly compiled files for reuse in subsequent requests.
- ASP.NET-specific performance counters.
- Web application testing tools.

The ASP.NET model provides a number of built-in performance enhancements. In particular, there are two enhancements involved in processing HTTP requests:

- When an ASP.NET page is requested for the first time, an instance of the **Page** class is dynamically compiled. In earlier versions of ASP, page code was interpreted for requests in the order that they appeared on the page. The common language runtime JIT compiles ASP.NET managed page code to the native code of the processing server at run time.
- When the **Page** instance has been compiled for the first request, it is cached on the server. For each subsequent request for that page, the cached instance of the class is executed. After the initial request, the **Page** class is recompiled only when the original source for the page or one of its dependencies has changed.

In addition, ASP.NET caches internal objects, such as server variables, to speed user code access. As a part of the .NET Framework, ASP.NET benefits from the performance enhancements provided by the common language runtime (CLR), including the previously mentioned JIT compiling, a fine-tuned common language runtime for both single and multiprocessor computers, and more.

These enhancements, unfortunately, cannot protect you from writing code that does not perform well when a large number of HTTP requests are processed simultaneously by your application. You must test your application to ensure that it meets the demands of its users.

Master Pages

Microsoft ASP.NET 2.0 master pages enable you to apply the same page layout to multiple content pages in a Web application. Master pages provide you with a method of creating a consistent look and style for a Web site.

The pages in most Web applications have standard elements, such as logos, navigation menus, and copyright notices. You can place all of these elements within a single master page. If you base the content pages in your application on this master page, all of the content pages will automatically include the same standard elements.

This feature represents a template mechanism that provides site-level page templates, a system for fine-grained content replacement, programmatic and declarative control over which template a page should use, and integrated designer support. It consists of two conceptual elements: master pages and content pages. Master pages acts as the templates for content pages, and content pages provides content to populate pieces of master pages that require "filling out." A master page is essentially a standard ASP.NET page except that it uses the extension of **.master** and a **<%@ master %>** directive instead of **<%@ page %>**. This master page file serves as the template for other pages, so typically it will contain the top-level HTML elements, the main form, headers, footers, and such. Within the master page, you add instances of the **ContentPlaceHolder** control at locations where you want content

pages to supply page-specific content. For more information about master pages, see "ASP.NET Master Pages Overview" on MSDN.

HTTP Modules

HTTP modules and HTTP handlers are an integral part of the ASP.NET architecture. Each request that is made is processed by multiple HTTP modules (for example, the authentication module and the session module) and is then processed by a single HTTP handler. After the handler has processed the request, the request flows back through the HTTP modules. Modules are called before and after the handler executes. They serve the following functions:

- Modules enable developers to intercept, participate in, or modify each individual request.
- Modules implement the **IHttpModule** interface, which is located in the **System.Web** namespace.

It is possible to use HTTP modules to extend ASP.NET applications by adding pre- and post-processing to each HTTP request that comes into an application. For example, if you want custom authentication facilities for your application, the best technique is to intercept the request when it comes in and then process the request in a custom HTTP module.

Configuring HTTP Modules

HTTP modules are configured in the ASP.NET configuration files, which are XML-based and can be edited in any text or XML editor. Like other .NET configuration files, the ASP.NET configuration file is divided into sections that are identified by various tags. The **<httpModules>** configuration section handler is responsible for configuring the HTTP modules within an application. It can be declared at the computer, site, or application level. The following example shows how a typical **<httpModules>** section handler is configured.

```
<configuration>
    <system.web>
        <httpModules>
            <add type="[COM+ Class], [Assembly]" name="[ModuleName]" />
            <remove type="[COM+ Class], [Assembly]" name="[ModuleName]" />
            <clear />
        </httpModules>
    </system.web>
</configuration>
```

The <**httpModules>** tag must be included in the main <**configuration>** section under the <**system.web>** subsection, as previously shown. The **<add>** tag indicates that an **HttpModule** is to be added to an application. The **<remove>** tag indicates that an **HttpModule** is to be removed from the application. The **<clear>** tag removes all of the **HttpModules** from an application.

Creating HTTP Modules

To create an HTTP module, you must implement the **IHttpModule** interface. The **IHttpModule** interface has two methods that include the following signatures.

```
void Init(HttpApplication);
void Dispose();
```

The **Init** method is called when the module attaches itself to the **HttpApplication** object and **Dispose** is called when the module is detached from **HttpApplication**. The **Init** and **Dispose** methods represent the module's opportunity to hook into a variety of events that are exposed by **HttpApplication**. These events include the beginning of a request, the end of a request, a request for authentication, and so forth. For more information about HTTP Modules, see "HTTP Modules" on MSDN.

Web Services

Web services represent an important evolutionary step in the building of distributed applications. At present, older systems are isolated and often incompatible blocks. This makes business-to-business or business-to-client integration almost impossible because there are a lot of technological walls between them. This is where the importance of a common technology arises. Web services can make older systems available on the Web, reusing existing software and opening the doors to systems integration. This section provides an overview of the Web services technology and explains how to incorporate it into your newly upgraded application.

There is not a universal definition of a Web service or a minimum standard for the requirements and services offered by one. What is commonly accepted is that a Web service is a software system that is designed to support interactions between different computers in a network.

Communication that takes place between Web service applications and components is encoded with an XML format. Because XML is an open standard that is supported on several platforms, Web services are not platform specific and can be used to communicate between applications that are based in different languages and platforms. For example Visual Basic .NET applications can communicate with Java applications and Windows applications can communicate with UNIX applications.

Additional features of a Web service include:

- A publicly exposed interface with all of the members that can be accessed by a client application or component. This interface can be described in a standard language by using an XML format. The Web Service Description Language (WSDL) is the most widely accepted language for this purpose.

- A mechanism to publish the services offered by a component that can be located by parties who are interested in the service provided. The most accepted directory of Web services that are currently available is Universal Description, Discovery, and Integration (UDDI).

One of the most important innovations in Web services is the use of XML as a transport medium for remote procedure calls. Previous technologies, such as traditional Remote Procedure Calls (RPC), used HTTP for such a purpose. By using XML as a standard, different applications can easily communicate data and service invocations. For an example of a publicly available Web service, see "Microsoft MapPoint Web Service" on the Microsoft Web site.

The .NET Framework Class Library includes the **WebService** class that provides access to ASP.NET Web service objects. This class can serve as the base class for XML Web services. For more information about the **WebService** class, see "WebService Class" on MSDN.

Web Services Benefits

Web services technology offers an excellent opportunity for the integration of systems at the internal corporate level. It can also improve customer-to-client communication and interaction. From a business point of view, other benefits include:

- Improved customer satisfaction because of new opportunities to access up-to-date information.
- Increased sales by improvements to the process of online purchases.
- Decreased costs by reducing the burden of excess calls on phone centers.
- Shared information between store locations and businesses.

Architecture Advancements

The following section explores the architecture advancements that Web services provide.

Service Interfaces

Offering a software service through the Internet is not new idea. Over the past decade, many companies have produced software systems that provide information to specific clients through the internet, such as virus scanner updates and virus definition files. Why then are the new Web services so important in the software industry? What other new technologies and protocols are involved with Web services?

The answer to both of these questions involves standardization. With new standards, such as XML and SOAP, you can set the foundation for several platforms and communicate between them. Today, software components can communicate with peers by using a structured language (XML) and through SOAP.

The following standards are the most common ones used in Web services today:

- **Emerging standards**. These include:
  - XML agreements for industry data exchange.
  - Security. This includes authentication (Microsoft WS-Security is an example).
  - Transactions. This includes management of a set of interdependent actions (XAML and OASIS BTP are examples).
  - Processes. This includes complex business interactions descriptions (Microsoft's WS-Routing is an example).
- **Existing standards**. These include:
  - SOAP. This protocol requests a Web service.
  - Web Service Description Language (WSDL). This is a standard for documenting what a Web service does and how to use it
  - Universal Description, Discovery, and Integration (UDDI). This is a standard for listing WSDL documents in a directory for automatic search.
- **Established standards**. These include:
  - Transmission Control Protocol/Internet Protocol (TCP/IP). This is an Internet network protocol.
  - Hypertext Transfer Protocol (HTTP). This protocol is used to transfer data between serves and clients in hypertext/hypermedia.
  - File Transfer Protocol (FTP). This method moves files between two Internet sites.
  - Simple Mail Transfer Protocol (SMTP). This protocol is used for sending electronic mail messages between computers.
  - Uniform Resource Identifier (URI). This method is used to identify any point of content on the web, whether it is a page of text, a video or sound clip, a still or animated image, or a program.
  - XML standard for exchanging data.

Web Service Discovery

In addition to the standards that are presented in the previous section, Microsoft introduces two new components that take an important role in the construction of Web services on .NET. These are .asmx and .disco files.

The .asmx file represents the addressable entry point for Web services that are created with managed code. The way you access this file through HTTP determines the type of response that you receive. For example, if you append "?WSDL" to the URL of your Web service, it will instantly create and return the appropriate WSDL file. The following code example is an HTML page that has a reference to methods that are exposed by a Web service.

```
<?xml version="1.0" encoding="utf-8" ?>
  <head>
    <link rel="alternate" type="text/xml" href="Service1.asmx?disco"/>
    <title>Service1 Web Service</title>
  </head>
  <body>
    <div id="content">
      <p class="heading1">Service1</p><br>
      <span>
          <p class="intro">The following operations are supported.
          For a formal definition, please review the
          <a href="Service1.asmx?WSDL">Service Description</a>.</p>
              <ul>
              <li>
                <a href="Service1.asmx?op=HelloWorld">HelloWorld</a>
              </li>
              <p>
              </ul>
      </span>
  </body>
```

Web service discovery is the process of locating and interrogating Web service descriptions. You do this as a preliminary step prior to accessing a Web service. It is through the discovery process that Web service clients learn that a Web service exists, what its capabilities are, and how to properly interact with it. A discovery file is an XML document with a .disco file name extension. You do not need to create a discovery file for each Web service. The following code is a sample discovery file for the securities Web service in the previous example.

```
<?xml version="1.0" encoding="utf-8"?>
<discovery xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef ref="http://tempuri.org/WebService2/Service1.asmx?wsdl"
docRef="http://tempuri.org/WebService2/Service1.asmx" xmlns="http://
schemas.xmlsoap.org/disco/scl/" />
  <soap address="http://tempuri.org/WebService2/Service1.asmx" xmlns:q1="http://
tempuri.org/WebService2/Service1" binding="q1:Service1Soap" xmlns="http://
schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

You can name this file Service1.disco and save it to the same directory as the Web service. For more information about the procedure to register, discover and access a Web service, see the "Consuming a Web Service" section later in this chapter.

If you create a Web service in the */webreference/machinename* directory, it is best to enable dynamic discovery. Dynamic discovery will automatically scan for all the *.disco files in all of the subdirectories of */webreference/machinename*.

```
<?xml version="1.0" ?>
    <dynamicDiscovery xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
</dynamicDiscovery>
```

By analyzing the discovery file you can find where the Web service resides on the system. Unfortunately, both the .disco file method and dynamic discovery method require that you know the exact URL of the Web service in advance. If you cannot find the discovery file, you will not be able to locate the Web service. One way to overcome this problem is to publish a Web service so that it can be located by potential users. Universal Description, Discovery, and Integration (UDDI) is a new specification that describes mechanisms that you can use to publish existing Web services. UDDI is an open, Internet-based specification that can be used as a building block to enable businesses to quickly, easily, and dynamically find and transact business by using their preferred application.

Creating a Web Service

Creating a Web service is a straightforward process. The most important part of it is being clear about what you want to publish.

When Visual Studio .NET creates a Web service, the code is a simple class with special attributes, as shown here.

```
Imports System.Web.Services

<System.Web.Services.WebService(Namespace := "http://tempuri.org/WebService2/
Service1")> _
Public Class Service1
    Inherits System.Web.Services.WebService
    <WebMethod()> _
    Public Function HelloWorld() As String
        Return "Hello World"
    End Function

    <WebMethod()> _
    Public Function HelloMe() As String
        Return "Hello Me"
    End Function

End Class
```

To create a Web service, you start by making an .asmx file, either through Visual Studio .NET or your favorite text editor. As is shown in the previous code example, a Web service is created as an ordinary class. The class inherits from the **System.Web.Services.WebServices** class. The methods that have the **<WebMethod()>** attribute before them indicate that the method should be accessible through the Web service.

Notice the following details in the prior code example:

- The two Web service-accessible methods are predicated with **<WebMethod()>**.
- The **System.Web.Services.WebService** attribute specifies the namespace where the Web service will be published. Note in the example that the URL

http://*tempuri.org/WebService2/Service1* is a placeholder that should be replaced with the actual URL of the Web service.

- The Web service is named **Service1**, as shown by the class name.

After you create the .asmx file and have stored it in a Web-accessible directory, you can view the methods by visiting the page directly through your Web browser.

As the creator of the Web service, your job is done — you have created the Web service and it is now ready to be published. But before you publish the Web service, be aware that the methods that are published on the Web service can not receive complex objects or non-serializable objects. If your code has any methods that receive these kinds of objects, you have to change the implementation of the methods or make the objects serializable so that they are accessible from the Web service.

The following section explains how to consume a Web service.

Consuming a Web Service

For a Web site to consume a Web service, a rather complicated and terse communication must occur between the client Web site and the Web site that provides the Web service.

To enable a Web site to consume a Web service you first need to add a Web reference to your application. The best way to do this is through the Visual Studio .NET IDE. The following example procedure adds a reference to the Web service created in the section "Creating a Web Service" earlier in this chapter.

► **Adding a Web reference to your application**

1. Open the client application.
2. On the **Project** menu, click **Add Web Reference**.
3. In the **Add Web Reference** dialog box, click **Web services on the local machine**.
4. In the list of available Web services, click **Service1**.
5. In the text box labeled **Web service name**, enter a name for this new reference. For this example enter **HelloWorldWS**. You will use this name to refer to the Web service from the Visual Basic .NET code.
6. Click on the **Add Reference** button.

For more information about adding Web references, see "Publishing and Discovering Web Services with DISCO and UDDI" on MSDN.

After the Web reference is added to the application, the Web service can be consumed by that application. The consumer application must decide which of the producer's methods it wants to call. If there are input parameters, these parameters must be converted into XML or some other serializable format to be passed along as was previously mentioned in the "Creating a Web Service" section earlier in this chapter. An HTTP request is made from the consumer to the producer to specify the

method they want to call and passes along the parameters in either a SOAP request, through a QueryString, or in POST headers.

The producer receives the incoming request, un-packages the input parameters, and calls the appropriate method of the specified class. This method, when complete, returns a value, which is packaged up and sent back to the consumer in an HTTP response. The consumer receives this response and un-packages the return value, completing the Web service call.

You do not want to worry about the HTTP message-passing semantics when you are using Web services. To avoid this issue, you can use a proxy class. Proxy classes serve as an intermediate step between the program (or Web page) on the consumer and the actual Web service on the producer. For each method in the producer's Web service, there is a method in the proxy class. It is the responsibility of the proxy class to do all of the complicated message-passing tasks. This essentially hides the complexity in the class, and your Web page or Windows application can simply call the methods of this proxy class and not concern itself with the underlying semantics that are involved in calling a Web service. For more information about Web services creation and usage, see "Consuming Web Services" on MSDN.

It is not necessary to create the proxy yourself; in fact, Visual Studio does all of this work for you automatically. When you click on the **Add Reference** button Visual Studio creates a proxy for you that is named **HelloWorldWS.Service1**. To perform an operation on this newly referenced Web service, you only need to create an instance of the proxy class and invoke its methods just as you would with a regular class. The following code invokes the **HelloWorld()** method of the Web service from the example and displays the results in a Windows Forms label named **Label1**.

```
Private Sub Button1_Click(ByVal sender As System.Object,_
        ByVal e As System.EventArgs) Handles Button2.Click
    Dim ws As HelloWorldWS.Service1
    ws = New HelloWorldWS.Service1
    Label1.Text = ws.HelloWorld()
End Sub
```

Technology Updates

The following sections detail the technology updates available through Web services.

Web Services Enhancements (WSE)

The Web Services Enhancements (WSE) is a .NET class library that augments the .NET Framework and related technologies to provide an implementation of various Web service specifications.

WSE is designed to simplify the development and deployment of secure Web services by helping Visual Studio .NET and .NET Framework developers apply a

security policy, establish long-running secure conversations, retrieve and validate security tokens, and more. The latest version of WSE at the time of this writing is 2.0 and includes a number of enhancements over earlier versions of WSE to more easily apply these features. Some of these enhancements, such as the WSE 2.0 support for the newly-approved OASIS standard version of WS-Security, mean WSE 2.0 cannot exchange secured SOAP messages with WSE 1.0. WSE 2.0 also introduces the use of WS-Addressing which deprecates WS-Routing used in WSE 1.0. The good news is that you can use the WSE 2.0 assemblies in the same applications as your WSE 1.0 assemblies so that a single application can communicate with both WSE 1.0 and WSE 2.0 Web services.

Web Services Security

WS-Security is not a complete security solution in and of itself. It is a protocol for exchanging security information between message senders and receivers. Developers need to design an appropriate security solution and deal with potential threats, such as replay attacks.

The Web Services Enhancements (WSE) supports security tokens based on the following credentials:

- Kerberos version 5 ticket
- User name and password
- X.509 certificate

Attachments

WSE 2.0 supports attaching files to SOAP messages outside the SOAP envelope; these files are not serialized as XML. This can be beneficial when sending large text or binary files because XML serialization is a very costly process and can result in files much larger than the originals. Direct Internet Message Encapsulation (DIME) is a lightweight, binary message format that WSE 2.0 uses to encapsulate SOAP messages and their attachments in a single message.

WS-Routing

WS-Routing is a specification used by participants in a service network to move messages from a source to a destination. WS-Routing is the heart of a SOAP router, which often acts as an intermediary, passing information between SOAP routers and to other service network participants.

In many cases, participants other than the router (such as a SOAP server) will also implement the WS-Routing specification. This enables SOAP services to pipe their output directly to another service as input.

WS-Routing uses routing tables to determine the path that the message should take. Another protocol, WS-Referral, is used to update those routing tables.

WS-Referral describes the format of messages that can be used to add or delete routes in these routers.

Summary

Web applications provide a means to exchange data and to remotely invoke services across the Web. They make it possible to provide access to data and services to significantly more users than a typical desktop application. The significance of this possibility has served as the catalyst for the increase in Web applications and Web services.

If your applications are not currently Web accessible, enhancing them with Web services, or even turning them into Web services, is not beyond possibility. This chapter has discussed Web technologies and language features available in Visual Basic .NET that will enable you to make your applications usable over the Web.

Yet another perspective on application advancement is to consider technological features that are desirable in any application. The next chapter will look at advancements that will help make your applications more secure, manageable, and scaleable.

More Information

For more information about Master Pages see "ASP.NET Master Pages Overview" on MSDN:
http://msdn2.microsoft.com/library/wtxbf3hh.aspx.

For more information about HTTP Modules, see "HTTP Modules" in *MSDN Magazine*:
http://msdn.microsoft.com/msdnmag/issues/02/05/asp/default.aspx.

For an example of a publicly available Web service see "Microsoft MapPoint Web Service" on Microsoft.com:
http://www.microsoft.com/mappoint/products/webservice/default.mspx.

For more information about the **WebService** class, see "WebService Class" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWebServicesWebServiceClassTopic.asp.

For more information about adding Web references, see "Publishing and Discovering Web Services with DISCO and UDDI" in *MSDN Magazine* at
http://msdn.microsoft.com/msdnmag/issues/02/02/xml/default.aspx.

For more information about Web Service creation and usage see "Consuming Web Services" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sdk/htm/ebiz_prog_webservices_yydh.asp.

20

Common Technology Scenarios

Many technology scenarios apply to all applications, regardless of the application's purpose. These scenarios include requirements for security, manageability, and performance. Ensuring that an application protects the system from accidental or malicious actions depends on providing good security. Managing an application goes beyond simply installing it on a user's computer. As computer hardware performance continues to increase, the demands made on applications that run on them also increase. This means that application performance and scalability are serious considerations for modern applications.

This chapter examines each of these scenarios and provides information about how each is addressed in Microsoft Visual Basic .NET.

Application Security

The security features in the Microsoft .NET Framework were designed, in part, to make it much easier for developers to write secure code. This section examines some of the features available in Visual Basic .NET to enhance the security of your applications.

Using Identities and Authentication

Security is one of the most important features to consider when you build .NET applications. Application developers must consider several situations that may occur when an application is used in a production environment. Security attacks are very common, and nobody wants to have an unsecured application running in their organization. The .NET Framework provides classes that help to manage security situations. They make it easy to implement role-based security in your application.

Implementing and enforcing security consists of two parts: authentication and authorization. This is achieved by implementing custom principal and identity classes based on the **IPrincipal** and **IIdentity** interfaces.

Identities represent users and contain properties that hold information (such as the user name) about that user. The classes for working with identities reside in the **System.Security.Principal** namespace. This namespace contains two classes: **GenericIdentity** and **WindowsIdentity**. By using these classes, you can determine the properties of a user. The namespace also includes the **IIdentity** interface that you can use to create custom identities, which extend the base identity type to suit the specific needs of your application.

The following is an example of an implementation of the **IIdentity** interface.

```
Import System.Threading
Import System.Security.Permissions
Public Class MyIdentity
    Implements System.Security.Principal.IIdentity

    Public Shared strAuthenticationTypeString As String = "NTLM"
    Public strUserName As String
    Private _userInfo As Hashtable

    Public ReadOnly Property AuthenticationType() As String _
           Implements System.Security.Principal.IIdentity.AuthenticationType
        Get
            Return strAuthenticationTypeString
        End Get
    End Property

    Public ReadOnly Property IsAuthenticated() As Boolean _
           Implements System.Security.Principal.IIdentity.IsAuthenticated
        Get
            Return True
        End Get
    End Property

    Public ReadOnly Property Name() As String _
           Implements System.Security.Principal.IIdentity.Name
        Get
            Return _userInfo(strUserName)
        End Get
    End Property

    Private Sub New(ByVal userInfo As Hashtable)
        Me._userInfo = userInfo
    End Sub

    Public Shared Function strCreateMyIdentity(ByVal userInfo As Hashtable) _
                  As MyIdentity
        Return New MyIdentity(userInfo)
    End Function

End Class
```

The **UserIdentity** type implements the **IIdentity** interface which requires you to implement three properties:

- **Name**. This returns the name of the identity. You need to call the shared function **strCreateMyIdentity()** and pass it a hash table with all of the user information. This method then returns an instance of your identity.
- **IsAuthenticated**. This returns a value, whether or not the user has been authenticated. If you allow anonymous access, you set it to **false** for anonymous users.
- **AuthenticationType**. This returns the type of authentication. **WindowsIdentity** returns NTLM, while **GenericIdentity** returns an empty string or the type specified when you instantiated **GenericIdentity**.

A **Principal** object is a holder for all the roles the user belongs to (according to the active authentication mechanism). Any .NET class that implements the **IPrincipal** interface is a valid **Principal** object. The **IPrincipal** interface exposes the **Identity** property (which returns the underlying **Identity** object) and the **IsInRole** method.

The following code example shows how to implement an **IPrincipal** interface. This class works with the **IIdentity** class previous defined.

```
Import System.Threading
Import System.Security.Permissions
Public Class MyPrincipal
    Implements System.Security.Principal.IPrincipal

    Private Groups As Hashtable
    Private Rights As Hashtable
    Private MyIdentity1 As MyIdentity

    Public ReadOnly Property Identity() As System.Security.Principal.IIdentity _
                   Implements System.Security.Principal.IPrincipal.Identity
        Get
            Return MyIdentity1
        End Get
    End Property

    Public Function IsInRole(ByVal role As String) As Boolean _
           Implements System.Security.Principal.IPrincipal.IsInRole
        Return Groups.ContainsValue(role)
    End Function

    Public Function HasPermissions(ByVal Permission As String) As Boolean
        Return Rights.ContainsValue(Permission)
    End Function

    Private Sub New(ByVal SecGroups As Hashtable, ByVal SecRights As Hashtable, _
                ByVal userInfo As Hashtable)
        Me.Groups = SecGroups
        Me.Rights = SecRights
        MyIdentity1 = MyIdentity.strCreateMyIdentity(userInfo)
    End Sub
```

```
    Public Shared Function CreateSecPrincipal(ByVal SecGroups As Hashtable, _
                  ByVal SecRights As Hashtable, ByVal userInfo As Hashtable) _
                  As MyPrincipal
        Return New MyPrincipal(SecGroups, SecRights, userInfo)
    End Function

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
    End Sub
End Class
```

The **Security.Principal** type implements the **IPrincipal** interface, which requires that you implement the **Identity** property and the **IsInRole()** method. Also, this type has a private constructor to prevent you from instantiating an instance of this type. You have to call the shared function **CreateSecPrincipal()**, pass a hash table with the user information and the roles that the user belongs to, and the security rights (permissions) that the user has in your implementation.

The constructor for this type calls **CreateUserIdentity()** from your custom identity class, passing along the user information, and then holds on to the identity it receives. **CreateSecPrincipal()** returns an instance of your security principal to you. The **Identity** property returns the identity object associated with the security principal.

After you define these classes, you can use them as shown in the following code example.

```
Import System.Threading
Import System.Security.Permissions
Public Class Example
    Public Shared Sub Main()

        Dim MyID As New System.Security.Principal.GenericIdentity("hvalverde")
        Dim roles As String() = {"Manager", "Administrator"}
        Dim GP As New System.Security.Principal.GenericPrincipal(MyID, roles)


        Thread.CurrentPrincipal = GP
        Try
            doSomething()
        Catch e As Exception
            Console.WriteLine("Error:")
            Console.WriteLine(e.ToString())
        End Try
        GP = New System.Security.Principal.GenericPrincipal(MyID, _
            New String() {"Administrator"})
        Thread.CurrentPrincipal = GP
        doSomething()
    End Sub
```

```
    <PrincipalPermissionAttribute(SecurityAction.Demand, Role:="Administrator")> _
    Shared Sub doSomething()
        Console.WriteLine("You are Adm")
    End Sub
End Class
```

The .NET Framework class library contains two **Principal** objects: **WindowsPrincipal** and **GenericPrincipal**. When you use a **WindowsIdentity**, **WindowsPrincipal** pair, the **Principal** role list is built using the Windows groups to which the Windows user belongs.

With principal and identity objects, you can authenticate users and authorize them to (or prevent them from) accessing application functionality as appropriate for each user's role.

Using Cryptography

The .NET Framework provides a set of cryptographic objects that support well-known algorithms and their common uses, including hashing, encryption, and generating digital signatures. These objects are designed to allow you to easily incorporate these basic capabilities into more complex operations, such as signing and encrypting a document.

The .NET Framework uses cryptographic objects to support internal services, but these objects are also available to developers who need cryptographic support. The .NET Framework provides implementations of many such standard cryptographic algorithms and objects.

The **System.Security.Cryptography** namespace in the .NET Framework provides several cryptographic services. The cryptographic algorithms supported include:

- **Rivest Shamir Adelman (RSA) and Digital Signature Algorithm (DSA) public key (asymmetric) encryption**. Asymmetric algorithms operate on fixed buffers. They use a public key algorithm for encryption and decryption.
- **Data Encryption Standard (DES), TripleDES, and RC2 private key (symmetric) encryption**. Symmetric algorithms are used to modify variable length buffers and perform one operation for periodical data input.
- **Message Digest 5 (MD5) and Secure Hash Algorithm (SHA1) hashing**. MD5 is a one-way hash algorithm. It always produces a 128-bit hash value for variable length input data.

The .NET Framework offers an extensive collection of encryption algorithms that can be used in different situations. Symmetric algorithms can be used to encrypt streams of data, such as files or communication channels, and have relatively good performance. Asymmetric encryption provides the strongest protection, but it is generally slower than symmetric encryption.

The following example shows a **Cipher** class that contains methods for file encryption and decryption. The example uses the DES encryption algorithm. This algorithm uses a single key for encryption and decryption; therefore, it belongs to the category of symmetric encryption algorithms. (For more information about the DES encryption algorithm, see "The DES Encyption Algorithm" on the Ius Mentis Web site.) The **Cipher** class has a method named **SetKey** that specifies the key for the algorithm to use. Users of the class must keep the original key in a secret and safe place because it is needed to decrypt the encrypted file. Notice that the **Decrypt** method differs from **Encrypt** only by the invocation of **cipherObj.CreateDecryptor** instead of **cipherObj.CreateEncryptor**. The **chainVec** variable is used as an additional parameter for the encryptor/decryptor. It provides additional protection by introducing part of the information in the previous encrypted block into the current block. This feature is known as *chaining ciphering*.

```
Imports System.IO
Imports System.Text
Imports System.Security.Cryptography
Public Class Cipher
    Public encryptKey(7) As Byte
    Private chainVec() As Byte = {&HAB, &HCD, &HEF, &H12, &H34, &H56, &H78, &H90}

    Sub SetKey(ByVal keyBase As String)
        Dim encoder As New ASCIIEncoding
        Dim arrByte(7) As Byte
        Dim i As Integer = 0
        Dim res() As Byte
        encoder.GetBytes(keyBase, i, keyBase.Length, arrByte, i)

        Dim SHAHash As New SHA1CryptoServiceProvider
        res = SHAHash.ComputeHash(arrByte)

        'Update the encryption key
        For i = 0 To 7
            encryptKey(i) = res(i)
        Next i
    End Sub

    Sub Encrypt(ByVal inputFile As String, ByVal outputFile As String)
        Try
            Dim dataBuffer(4096) As Byte
            Dim totFileLen As Long
            Dim BytesWrittenTot As Long = 8
            Dim blockSize As Integer
            Dim fileIn As New FileStream(inputFile, _
            FileMode.Open, FileAccess.Read)
            Dim fileOut As New FileStream(outputFile, _
            FileMode.OpenOrCreate, FileAccess.Write)
            Dim cipherObj As New DESCryptoServiceProvider
            Dim encryptedStream As New CryptoStream(fileOut, _
                cipherObj.CreateEncryptor(encryptKey, _
                    chainVec), CryptoStreamMode.Write)
```

```
            fileOut.SetLength(0)
            totFileLen = fileIn.Length

            While BytesWrittenTot < totFileLen
                blockSize = fileIn.Read(dataBuffer, 0, 4096)
                encryptedStream.Write(dataBuffer, 0, blockSize)
                BytesWrittenTot = Convert.ToInt32 _
                (BytesWrittenTot + blockSize / _
                cipherObj.BlockSize * cipherObj.BlockSize)
            End While

            encryptedStream.Close()
        Catch e As Exception
            'Handle exception
        End Try
    End Sub

    Sub Decrypt(ByVal inputFile As String, ByVal outputFile As String)
        Try
            Dim dataBuffer(4096) As Byte
            Dim totFileLen As Long
            Dim BytesWrittenTot As Long = 8
            Dim blockSize As Integer
            Dim fileIn As New FileStream(inputFile, _
            FileMode.Open, FileAccess.Read)
            Dim fileOut As New FileStream(outputFile, _
            FileMode.OpenOrCreate, FileAccess.Write)
            Dim cipherObj As New DESCryptoServiceProvider
            Dim encryptedStream As New CryptoStream(fileOut, _
            cipherObj.CreateDecryptor(encryptKey, _
            chainVec), CryptoStreamMode.Write)

            fileOut.SetLength(0)
            totFileLen = fileIn.Length

            While BytesWrittenTot < totFileLen
                blockSize = fileIn.Read(dataBuffer, 0, 4096)
                encryptedStream.Write(dataBuffer, 0, blockSize)
                BytesWrittenTot = Convert.ToInt32( _
                BytesWrittenTot + blockSize / _
                cipherObj.BlockSize * cipherObj.BlockSize)
            End While

            encryptedStream.Close()
        Catch e As Exception
            'Handle exception
        End Try
    End Sub
End Class
```

When you incorporate cryptography in your application, you may want to consider using the Microsoft patterns & practices Security Application Block. This application block allows you to choose a security option based on your system requirements.

You can use the Security Application Block in a variety of situations, such as using a database to authenticate and authorize users, retrieving role and profile information, and caching user profile information. The Security Application Block has the following features:

- It reduces the requirement to write boilerplate code to perform standard tasks.
- It helps maintain consistent security practices, both within an application and across the enterprise.
- It eases the learning curve for developers by using a consistent architectural model across the various areas of functionality provided.
- It provides implementations that you can use to solve common application security problems.
- It is extensible; it supports custom implementations of security providers.

For more information about security and cryptography, see "Security Application Block," "Cryptography Application Block," and "Cryptography Simplified in Microsoft .NET" on MSDN.

Application Manageability

The .NET Framework provides enhancements that allow you to improve the manageability of your application. These enhancements are described in the following subsections.

Using Configuration Files

You can use configuration files to configure .NET applications without recompiling them. These configuration files help you to store application variables and configuration values, just as you could with earlier versions of Visual Basic (by using .ini files and later, registry settings). In addition, Visual Basic .NET configuration files allow you to keep properties for visual components so they can be dynamically changed by just editing a text file.

Configuration files can contain different types of information, depending on the type of application that will use them. A configuration file for a .NET client application will be significantly different from those for a .NET server application: the configuration file will contain different sections, attributes, and values.

Typically, you need to perform the following tasks when you configure a client application that communicates with a remote server application:

- **Configure the client application security policy**. By default, applications that run in an intranet are not allowed to access other servers in the intranet. You must set the required permissions to allow the client to access other servers.

- **Configure assembly binding**. If the application uses shared third-party components, you might need to indicate specific versions of the components to be used by the client application.
- **Configure remoting**. You must indicate the server address to the client application. You can use the .NET Framework Configuration tool to complete this task. For more information about using this tool, see ".NET Framework Configuration Tool (Mscorcfg.msc)" on MSDN.

After you apply the configuration settings, you must deploy the resulting configuration file to the computers that will use the application. In the example, these are client computers. The following XML sample specifies the configuration settings discussed in this section.

```
<configuration>
    <runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <dependentAssembly>
                <assemblyIdentity name="MySpecificVersionComponent"
                                  publicKeyToken="e9b4c4996039ede8"
                                  culture="en-us"/>
                <publisherPolicy apply="no"/>
            </dependentAssembly>
        </assemblyBinding>
    </runtime>

   <system.runtime.remoting>
      <application name = "myApplication">
         <client url = "tcp://server/clientApplication"
                  displayName="Client Application">
            <activated type = "ClientClassInstance,ClientClass"/>
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

Notice that the **dependentAssembly** node indicates the component that the client application depends on.

The configuration file for a server application usually includes settings for the objects to be exposed to remote clients, life times for client-side objects, and the polling frequency for objects that are going to be released.

Configuration (.config) files are plain XML files with a defined structure. Typically, they reside in the same folder that contains your application's executable (.exe) file. You can add a configuration file to your application from Visual Studio .NET by selecting **Add New Item** on the **File** menu, and then selecting **Application Configuration File**.

Configuration files contain XML elements, which are logical data structures that set configuration information. For example, the **<runtime>** element is specified in the format **<runtime>***child elements*</runtime>**. Configuration settings are specified using predefined attributes, which are name/value pairs inside the corresponding element.

The following example specifies the version and href attributes for the **<codeBase>** element, which indicates an assembly's location. Note that the configuration file's content is case-sensitive.

```
<codeBase version="2.0.0.0"
          href="http://www.litwareinc.com/myAssembly.dll"/>
```

You can add new entries to the **<appSettings>** section of the configuration file, to define the following:

- **User-defined settings**. These are configuration settings that your application uses. The application reads these settings from the **System.Configuration.AppSettingsReader** class.
- **Dynamic properties**. These are Windows Forms control properties that are assigned a value from a configuration file entry. You can change the values for these properties by modifying the .config file. You do not need to recompile the application in order to apply the changes.

The following example shows how the **AppSettingsReader** is used to read data contained in the application's configuration file.

```
…
Dim myReader As System.Configuration.AppSettingsReader()
Dim myInstance As New MyClass()
myInstance.StringProp = CType(myReader.GetValue("DynamicClass.StringProp", _
GetType(System.String)), String)
…
```

Notice that the **AppSettingsReader.GetValue** method receives the key of an entry in the configuration file and the type used to parse the data. The content of the configuration file for this example is shown in the following example.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <appSettings>
      <add key="DynamicClass.StringProp" value="This is a test">
   </appSettings>
</configuration>
```

In general, you need to follow these steps to retrieve data from the configuration file manually:

1. Create an instance of **System.Configuration.AppSettingsReader**.
2. Use the **AppSettingsReader.GetValue** method to retrieve the value for the specified key.
3. Convert the object returned by the **GetValue** method to the appropriate data type.

For more information about application configuration files for machine-level and security settings, see "Application Configuration Files," "Application Configuration Scenarios," "System.Configuration Namespace," and "Configuration Application Block" on MSDN.

Using Deployment and Update Features

After you upgrade your application to Visual Basic .NET, you can take advantage of the deployment process features provided in the .NET Framework. The .NET Framework offers different ways to deploy your application, including:

- **XCOPY deployment**. This is the easiest way to deploy an application built on the .NET Framework. To use **XCOPY**, you copy the assemblies to the installation path. This is a good option if your application does not require additional files to be installed or any customization to be performed on the client computer.
- **Setup projects**. The .NET Framework provides a more powerful way to deploy your application than **XCOPY** deployment. Setup projects allow you to deploy the application and any necessary files. They also provide a detailed structure of the target file system. In these projects, you can also specify the following:
  - Custom actions to be executed on the target computer
  - Launch conditions
  - Registry settings
  - Messages and information displayed during the installation process
- **No-touch deployment**. This option allows you to make the assemblies of your application available online on a Web server. Target (client) computers can connect to the server and download the latest version of those assemblies each time the application is executed. The assemblies are copied to the server or Web directory that is trusted by the .NET runtime and by the Internet browser on the client computer. Client computers then access that Web directory by using its URL in the browser, and the .NET Framework takes charge of downloading the assemblies as they are required. No-touch deployment requires client computers to have a permanent Web connection to the server.

For more information about setup projects and .NET deployment, see "Setup Projects" and "No-Touch Deployment in the .NET Framework" on MSDN.

Using Performance Counters

The .NET Framework provides a series of performance counters that are quite useful for optimization and diagnostic purposes. A predefined set of the counters is available in Visual Studio .NET Server Explorer. To use them, you must drag and drop them on your application. Among other information, the predefined performance counters provide valuable data about the following aspects of your application:

- State of the .NET common language runtime (CLR), including memory, security, exceptions, and other state data
- System, including processes, threads, and other information
- Physical devices, including memory, processor, and other devices
- SQL Server, including databases, locks, statistics, and other information

You can use performance counters for your application after you upgrade it to Visual Basic .NET. For more information, see "Performance Counters" in the *.NET Framework General Reference* on MSDN.

Using Tracing and Logging

The .NET Framework tracing and logging mechanisms are available through two classes in the **System.Diagnostics** namespace: **Trace** and **Debug**. Both classes contain the same properties and methods. The difference between them is that **Debug** is available only when the application runs in **Debug** mode, while **Trace** is available in both **Debug** and **Release** modes. This means that the code you write using the **Debug** class cannot be executed in the **Release** configuration.

Using **Trace** and **Debug**, you can output logging information about system events, errors, and failures. Output can also be conditional: **Trace** and **Debug** generate messages only if certain conditions are met or if an expression evaluates to **true**.

You can redirect output generated by **Trace** and **Debug** calls to the **Output** window, the system's **Event Log**, or to a text file by using the **TraceListener** class. You can disable or inhibit the generated messages (depending on their severity) by using the **BooleanSwitch** and **TraceSwitch** classes available in **System.Diagnostics**.

To learn more about **Trace** and **Debug**, see "Trace Class" and "Debug Class" in the *.NET Framework Class Library* on MSDN.

Application Performance and Scalability

Performance and scalability are very important in real world applications. Often, one of these factors will impact the other. You must decide which is more important to your needs.

There are sections of an application that are critical to the overall performance. These sections can be contained in frequently invoked functions or in loops. When the quantity of tasks performed in these sections increases, the overall performance of the application is noticeably degraded. It is vital that you identify these performance bottlenecks before you can improve the application's overall performance. During application development, you should measure and identify potential performance issues so that you can avoid resource performance tuning tasks at the end of the development process.

For more information about performance considerations in software development, see *Improving .NET Application Performance and Scalability* on MSDN.

There are several common performance considerations that you should consider when you build .NET applications. Some of the most important are discussed in this chapter.

Exception Handling Considerations

Exception handling is a key aspect of developing applications on the .NET Framework. It allows you to easily and gracefully recover from errors that could easily crash an application. However, exception handling can be expensive in terms of system resources. For this reason, you should exercise restraint when using exceptions in your applications.

String Handling Considerations

When a string is altered, the original string is garbage collected, and a new object is created to hold the changed string. This may not be an issue for a small number of changes, but an excessive number can tax the system.

The .NET Framework includes the **StringBuilder** class, which is a special class designed to be used when you manipulate string objects. The **StringBuilder** class includes methods for altering the contents of a string. It is contained in the **System.Text** namespace. For more information, see the "Replacing Strings with StringBuilders" section in Chapter 11, "Upgrading String and File Operations."

Database Access Considerations

The .NET Framework recommends that you tune for database access by using only the functionality that you need, and that you design for a disconnected approach. With this approach, you make several connections in sequence, instead of holding a single connection open for a long time. In addition, Microsoft recommends an *n*-tier strategy for maximum performance, as opposed to a direct client-to-database connection strategy. Consider this recommendation as part of your design philosophy, because many of the technologies in the .NET Framework are optimized to take advantage of a multi-tiered architecture.

You should also consider the following recommendations:

- Use the optimal managed provider, such as the SQL-specific provider, instead of generic providers.
- Use a **DataReader** object instead of a **DataSet** object whenever you do not need to store data for later navigation. A **DataReader** offers improved performance because it has less overhead than a **DataSet**.
- Use Mscorsvr.dll for multiprocessor computers. This library offers optimizations that scale when more processors are available.
- Use stored procedures whenever possible. They offer increased performance because they are optimized and do not need to interpreted, compiled, or transmitted to client computers.
- Be careful about using dynamic connection strings. Connection pooling (discussed later in this chapter) creates a connection pool based on unique connection strings. When dynamic connection strings are used, they may result in different strings for the same connection which will not be recognized by the connection pool. Thus, any potential performance increase will be lost.
- Avoid auto-generated commands, which must frequently connect with the server to obtain metadata to carry out the command. Each time a connection occurs performance is reduced.
- Beware of ADO legacy design. Whenever you execute a command, every record specified in the query is returned.
- Keep your datasets compact. Keep only the records that are needed, because the more records you have in the dataset, the more memory is required to store them, even if they are not used.
- Use sequential access whenever possible. This is especially important for large data types. Sequential access will only read a small chunk of the data does not have the latency that reading in a large data type does.

For more information about database access and performance, see "Performance Tips and Tricks in .NET Applications" on MSDN.

Multithreading and the BackgroundWorker Component

Multithreading, or free-threading, refers to the ability of a program to execute multiple threads of operation simultaneously.

Multithreading can be a powerful tool to use in component programming. By writing multithreaded components, you can create components that perform complex calculations in the background while leaving the user interface free to respond to user input.

The .NET Framework offers more than one option for multithreading in components. The functionality in the **System.Threading** namespace is one option. The

Asynchronous Pattern for Components is another option. The **BackgroundWorker** component is an implementation of the Asynchronous Pattern. It provides that advanced functionality encapsulated in a component for ease of use.

For more information about multithreading and the **BackgroundWorker** component, see "BackgroundWorker Component Overview" and "Asynchronous Pattern for Components" on MSDN.

Caching

One of the new features in ASP.NET is a system for caching page and application data, so that the application does not need to perform the same expensive process each time a user views a page. Caching can be performed on a per-page, or on a per-user control basis.

The performance, primarily in terms of speed, of a Web page drives user satisfaction. If your goal is to increase traffic on a Web site, you cannot afford to have slow Web pages.

One of the ways to improve throughput is to use caching for Web pages. Simply defined, caching means storing frequently used items in memory. Caching is a well-tested and successful technique for performance improvement. HTML pages can be cached to improve their loading speed. A more indirect example of caching is connection pooling, which provides an efficient way to manage connections and share them across different service requests. In connection pooling, whenever a connection request is received, the connection pool checks if there is an existing idle connection that can be used to fulfill the request. Thus, new connections are not created unless there are no existing connections available. The .NET Framework supports caching of the ASP.NET response page that is generated by requests and supports storing individual objects in memory.

The following are some of the key concepts for caching ASP.NET pages:

- Know how to set the expiration policy.
- Know where to set the location of the cache.
- Know when you should cache multiple versions of a page.
- Know when you should cache portions of ASP.NET pages.
- Determine when you should cache application requests.
- Always notify an application when an item is removed from the cache.

For more information about caching, see "Caching ASP.NET Pages" in the *.NET Framework Developer's Guide* on MSDN.

Communication and State Management

When you build an application for deployment on the Internet or an intranet, it is important that the application is both open and scalable. To achieve this goal, you must design your application so that it can communicate and connect with different platforms and with other applications deployed in the environment.

Protocols and communication technologies play a critically important role in achieving this connectivity. The major protocols used for inter-machine communication are Distributed COM (DCOM) and Remote Method Invocation (RMI), but these protocols are not easy to use when you have to work between different networks or over the Internet. For these reason, other protocols — such as SOAP — are becoming more reliable and efficient.

Applications are not designed to be deployed on one computer only. A good application will be deployed on many computers and on many networks. To allow these computers to communicate, the protocol and infrastructure configuration are significant, both in terms of their security implications and the complexities of facilitating cross-computer, cross-network interaction.

The following subsections discuss some of the best known and most used communication protocols and technologies, and focus on the security implications inherent in each.

Moving From DCOM to HTTP

DCOM is a technology based on remote procedure call (RPC). This section provides an overview of the upgrade process necessary to move applications based on DCOM to HTTP.

When you use DCOM, you may encounter problems when it has to work through a firewall. This occurs because DCOM needs to use some ports that are normally disabled. However, enabling the RPC port mapper (port 135) can open the door to possible disclosure attacks.

In addition, using DCOM in tunneling mode (DCOM over HTTP) can have negative security implications because it can provide a covert channel exploitable by intruders.

The following sections explain some techniques for working around these security implications, including the use of marshaling, Web proxies, and service agent technologies.

Using Marshaling

The process of packaging data to be sent from one object to another is known as *marshaling*. Marshaling is the process of converting information for communication between threads. In COM, marshaling is used when data moves across context boundaries such as operating system processes or computers.

The most common practice for marshaling is to precede marshaled objects with data type information. You can do this in several different ways. The following list summarizes three approaches:

- **XML marshaling**. This approach uses XML nodes and attributes to represent type information and XML text to hold values. The XML result can be sent as plain text or as a structure that allows the reconstruction of the transmitted objects.
- **Text marshaling**. This uses plain text to represent data. Simple parsing on the receiving application can be used to reconstruct the object.
- **Binary marshaling**. This uses a binary header that contains data type and size information.

XML and text marshaling are readable by humans, and they generally have poor performance. The process of constructing an object from XML or text can be costly for large or numerous objects. In contrast, binary marshaling is very efficient but not readable by humans. You must weigh the performance needs against readability needs to help you decide which approach is best for your situation.

Creating Web Proxies

Web proxies are components that reside on the client computer. They facilitate the client's communication with a Web service and are responsible for transparently issuing requests to the Web service and interpreting the results for the client.

Web proxies communicate with XML Web services by using the SOAP protocol. You can use Visual Studio.NET or the Web Services Descriptive Language Tool (WSDL or Wsdl.exe) to automatically create Web proxies for the .NET Framework. These tools can query the desired Web service to discover how to interact with it. Based on the results of the query, the tools build the proxy class, which can be used in client applications to facilitate interaction with the Web service. You can specify the language used for creating the class at creation time; for example, you can specify Visual Basic .NET as the target language. You can then view the resulting class file in Visual Studio or any other source code editor, which gives you the ability to modify it if necessary. You can then include the proxy class in your client application.

Using Service Agents

A service agent is a service that helps your application work with other services. Service agents are frequently supplied by the provider of the target service. The agent runs topologically close to the application consuming the service. It helps both to prepare requests sent to a service and to interpret responses from the service.

Service agents offer advantages, such as ease of integration because of the simplified service interface, which makes consuming Web services as simple as using a local in-process object. Although you can easily achieve this same effect by using WSDL-generated proxies, service agents have advantages that you cannot achieve with a proxy from a WSDL file. These advantages include the following:

- **Error handling**. Service agents can be designed to recognize the errors that a service can produce, which can greatly simplify integration efforts.
- **Data handling**. A service agent can cache data from the service in a correct and knowledgeable manner. This can greatly improve response times from a service, reduce load on the service, and permit applications to work when disconnected (offline).
- **Request validation**. Service agents can check the input document of a service request to ensure correctness prior to submission, allowing obvious errors to be caught without the lag time and server load of a roundtrip. On the other hand, the service still requires validating all requests on receipt.
- **Intelligent routing**. Some services can use agents to send requests to a specific service instance based on the content of the request.

Service agents also have support for operating in both online and offline modes. The agents can detect when a remote service is unreachable, and switch to offline mode without any intervention from the applications that are consuming the agents. As stated in the "Windows Applications and Windows Forms Smart Clients" section in Chapter 18, "Advancements for Common Scenarios," this offers a huge advantage to smart client users.

Replacing Message Queuing with System.Messaging

The **System.Messaging** namespace provides classes that allow you to connect to, monitor, and administer message queues on the network and send, receive, or peek at messages. This namespace provides developers with the ability to send and receive messages primarily through two classes: **Message** and **MessageQueue**. The **Message** class encapsulates any data you are sending, as well as properties to control the routing, formatting, and security of the message. **MessageQueue** classes are used to represent the individual queues and contain the methods for working with the queues, such as **Send()** and **Receive()**.

By using Message Queuing and the **System.Messaging** namespace, .NET developers can create highly scalable, reliable applications that are loosely coupled and share only the name of a common queue.

For more information about the upgrade of Message Queuing API access to the **System.Messaging** namespace, see the "MSMQ and Queued Components" section in Chapter 15, "Upgrading MTS and COM+ Applications."

Note: It is important to note that MSMQ 3.0 provides functionality not available in the **System.Messaging** namespace. For example, Message Queuing can provide information about the quantity of messages contained in a queue. If you need this functionality, you will need to implement it by using other .NET Framework functionality (for example a .NET Framework **ServicedComponent** that traces the quantity of messages in a queue).

For more information about the **System.Messaging** namespace, see "System.Messaging Namespace" in the *.NET Framework Class Library* on MSDN.

Upgrading ODBC and OLE DB Data Access Components

By using Microsoft Data Access Components (MDAC), developers can connect and access data from different sources, including relational and XML data sources. The components available in MDAC include ActiveX Data Objects (ADO), Open Database Connectivity (ODBC), and OLE DB. You can use these components in combination with appropriate providers and drivers to access many different types of data sources.

This section provides an overview of ODBC and OLE DB data access components in the .NET Framework. ADO.NET is covered in detail in the next section.

The ODBC and OLE DB components can be described as follows:

- **ODBC**. The Microsoft Open Database Connectivity (ODBC) interface is a C programming language interface that allows applications to access data from a variety of database management systems. Applications that use this API can access relational data sources only.
- **OLE DB**. OLE DB is a comprehensive set of low-level COM interfaces for accessing a diverse range of data in a variety of data stores. OLE DB providers exist for accessing data in database systems, file systems, message stores, directory services, workflow, and document stores.

The .NET Framework data providers offer numerous advantages over the extensive low- level data access components. The data access architecture has been simplified to provide improved performance without loss of functionality. Additionally, the .NET Framework data providers are executed inside the managed environment provided by the CLR. Interaction with COM components is not necessary.

Visual Basic 6.0 offers a variety of techniques for accessing ODBC data sources. For example, you can use Data Access Objects (DAO) with ODBC connections. You could use the ODBCDirect DAO option to access remote ODBC data sources. Alternatively, you could use Remote Data Objects (RDO) to access ODBC data sources. RDO implements a code layer on top of the ODBC API. Yet another possibility is to directly interact with the ODBC API through the ODBC handles provided by RDO.

In Visual Basic 6.0, you can access OLE DB data sources by using ADO, which is the programmer interface to OLE DB. This interface exposes virtually all OLE DB functionality.

To upgrade a Visual Basic 6.0 application that accesses ODBC or OLE DB data sources to Visual Basic .NET, you must upgrade the programmer interface used to interact with the data source. Depending on the data source, this interface will be DAO, RDO, or ADO. Information on how to upgrade DAO, RDO, and DAO code is provided in Chapter 12, "Upgrading Data Access." However, if the resources are available to do so, it is recommended that you upgrade any data access interfaces to ADO.NET.

To upgrade the underlying data provider component, you must use one of the .NET framework data providers. This remainder of this section explains this process.

The ODBC .NET Data Provider

.NET Framework data providers are used for connecting to a database, executing commands, and retrieving results. The results obtained can be directly processed, stored in an ADO.NET **DataSet**, or transmitted through the different application tiers.

The ODBC .NET data provider is an additional component that can be referenced by .NET applications. It provides access to native ODBC drivers and was designed to be compliant with all ODBC drivers including the following:

- Microsoft SQL ODBC driver
- Microsoft ODBC driver for Oracle
- Microsoft Jet ODBC driver

Each of these drivers has been tested by Microsoft and is known to work with the ODBC.NET data provider.

The .NET Framework data provider for ODBC is recommended for middle-tier applications that use ODBC data sources. Note that this data provider is not included in .NET Framework version 1.0. However, to download this component, see "ODBC .NET Data Provider" in the Microsoft Download Center.

This data provider uses the **Microsoft.Data.Odbc** namespace. You need to add the corresponding assembly as a reference in your Visual Basic .NET project so that the component will be accessible from your application.

The following code example shows how a connection string is setup for an ODBC SQL Server data provider and how the connection is used to create a simple command.

```
…
Dim cn As OdbcConnection = New OdbcConnection _
    ("DRIVER={SQL Server};SERVER=MySQLServer;UID=sa;" & _
    "PWD=mypassword;DATABASE=northwind;")
Dim mystring As String = "select * from MyTable"
Dim cmd As OdbcCommand = New OdbcCommand(mystring)
cn.Open()
…
cn.Close()
…
```

The OLE DB .NET Data Provider

The OLE DB .NET data provider has native access to OLE DB through COM interop, and supports both local and distributed transactions.

The following list presents the data providers that have been tested with ADO.NET:

- Microsoft OLE DB provider for SQL Server
- Microsoft OLE DB provider for Oracle
- OLE DB provider for Microsoft Jet

This data provider is recommended for middle-tier applications that use Microsoft SQL Server version 6.5 or earlier. All other OLE DB providers that support the OLE DB interfaces used by the .NET Framework data provider for OLE DB are also supported. This data provider is also recommended for single-tier applications that use the Microsoft Access database. For more information about OLE DB interfaces, see "OLE DB Interfaces Used by the .NET Framework Data Provider for OLE DB" in the *.NET Framework Developer's Guide* on MSDN.

The following code example shows how a connection string is setup for an OLE DB SQL Server data provider and how the connection is used to create a simple command.

```
…
Dim cn As OleDbConnection = New OleDbConnection( _
    "Provider=SQLOLEDB;Data Source=localhost;" & _
    "Integrated Security=SSPI;Initial Catalog=northwind")
Dim mystring As String = "select * from MyTable"
Dim cmd As OleDbCommand = New OleDbCommand(mystring)
cn.Open()
…
cn.Close()
…
```

Accessing Oracle Databases from the .NET Framework

Microsoft Data Access Components (MDAC) provides components that use ODBC and OLE DB interfaces to access Oracle databases. Although the Oracle ODBC data provider is still supported in the current release of MDAC, it is recommended that you use another .NET-supported Oracle data provider for new and upgraded applications.

By using the appropriate connection string, you can access an Oracle data source with the ODBC .NET data provider, as shown in the following code example.

```
Dim cn As OdbcConnection cn= New OdbcConnection ( _
    "Driver = {Microsoft ODBC for Oracle};" & _
    "Server=myOracleserver;uid=myuid;pwd=mypwd")
```

Another alternative is to use the OLE DB .NET data provider. The following code shows a sample connection string for this upgrade option.

```
Dim cn As OleDbConnection = New OleDbConnection( _
    "Provider=MSDAORA.1;User ID=myUID;password=myPWD; " & _
    "Data Source=myOracleServer;Persist Security Info=False")
```

The .NET Framework Data Provider for Oracle

When you upgrade a middle-tier or single-tier application that uses an Oracle database version 8.1.7 or later, it is recommended that you use the .NET Framework data provider for Oracle. This data provider allows developers to access Oracle data sources through Oracle client connectivity software and supports local and distributed transactions.

The classes that correspond to this data provider are located in the **System.Data.OracleClient** namespace and are contained in the **System.Data.OracleClient.dll** assembly. You must include this assembly as a reference to use the data provider functionality. The .NET Framework data provider for Oracle is not included in .NET Framework version 1.0. However, you can download the data provider from the Download & Code Center on MSDN.

Upgrading ADO to ADO.NET

ADO is a set of classes that allows users to access and manipulate data from a variety of sources through an OLE DB provider. With ADO, you can build both client/server and Web-based applications.

Some of the core services ADO provides are the following:

- Connecting to a data provider

- Executing commands or calling stored procedures
- Searching and navigating data
- Retrieving data

Visual Studio .NET offers a new and completely redesigned collection of classes for data access that considers modern application requirements of distribution, reliability, and scalability. This new data access model is ADO.NET and, in addition to the ADO features, provides the following advantages:

- **Interoperability**. All data in ADO.NET is transported in XML format. The data is provided as a structured text document that can be read by anyone on any platform.
- **Scalability**. ADO.NET promotes the use of disconnected datasets, with automatic connection pooling bundled as part of the package.
- **Productivity**. ADO.NET can improve overall development time. For example, typed **DataSets** help you work more quickly and allow you to produce more bug-free code.
- **Performance**. Because ADO.NET provides disconnected datasets, the database server is no longer a bottleneck and application performance is improved.

ADO.NET includes the **DataSet** and **DataTable** classes, which implement database-independent data containers. It also provides database-oriented tools, such as SQL and OLE DB commands, managed data providers and connections, data readers, and data adapters. ADO.NET allows your application to work with data, regardless of the data source, data format, or physical location. It provides a new object model and features a data-centric design that differs from the ADO database-centric approach.

To achieve a smooth transition from ADO to ADO.NET, you must first understand that ADO.NET is significantly different from ADO. You must also have an understanding of the ADO.NET philosophy.

ADO.NET Overview

ADO.NET allows developers to access data sources in a uniform way. ADO.NET can access data from sources such as Microsoft SQL Server, Oracle, and other data sources exposed through OLE DB or XML. Applications can use ADO.NET to access these data sources and retrieve, manipulate, and update data.

The ADO.NET components have been designed to factor data access from data manipulation. There are two central groups of functionality in ADO.NET that accomplish this: the **DataSet** component, and the .NET Framework data provider, which is a set of components including the **Connection**, **Command**, **DataReader**, and **DataAdapter** objects.

The ADO.NET **DataSet** is the core component of the disconnected architecture of ADO.NET. The **DataSet** is explicitly designed for data access independent of any data source. The **DataSet** contains a collection of one or more **DataTable** objects made up of rows and columns of data, as well as primary key, foreign key, constraint, and relation information about the data in the **DataTable** objects.

The other core element of the ADO.NET architecture is the .NET Framework data provider. The data provider components are explicitly designed for data manipulation and fast, forward-only, read-only access to data. The **Connection** object provides connectivity to a data source. The **Command** object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information. The **DataReader** provides a high-performance stream of data from the data source. Finally, the **DataAdapter** provides the bridge between the **DataSet** object and the data source. The **DataAdapter** uses **Command** objects to execute SQL commands at the data source to both load the **DataSet** with data and update changes made to the data in the **DataSet** back to the data source.

Differences Between ADO and ADO.NET

In ADO, the in-memory representation of data is the **Recordset**. In ADO.NET, it is the **DataSet**. A **Recordset** works as a single table. In contrast, a **DataSet** is a collection of one or more tables. In this way, a dataset can mimic the structure of the underlying database. Because the **DataSet** can hold multiple, separate tables and maintain information about relationships between them, it can hold much richer data structures than a recordset, including self-relating tables and tables with many-to-many relationships. A more direct option is to replace **Recordsets** with **DataTable** objects. The **DataTable** class corresponds to a single table, as does **Recordset** does, and should be considered the first upgrade option for a **Recordset**. This guide focuses on using **DataSet** because it is a more powerful collection.

Transmitting an ADO.NET **DataSet** between applications is much easier than transmitting an ADO disconnected **Recordset**. To transmit an ADO disconnected **Recordset** from one component to another, you use COM marshaling. To transmit data in ADO.NET, you use a **DataSet**, which can transmit an XML stream.

In ADO, you scan sequentially through the rows of the recordset using the ADO **MoveNext** method. In ADO.NET, rows are represented as collections, so you can loop through a table as you would through any collection or access particular rows by using an ordinal or primary key index. **DataRelation** objects maintain information about master and detail records and provide a method that allows you to get records related to the one you are working with.

In ADO.NET, you open connections only long enough to perform a database operation, such as a **Select** or **Update**. You can read rows into a dataset and then work with them without staying connected to the data source. In ADO, the **Recordset** can provide disconnected access, but ADO is designed primarily for connected access.

There is one significant difference between disconnected processing in ADO and ADO.NET. In ADO, you communicate with the database by making calls to an OLE DB provider. In ADO.NET, you communicate with the database through a data adapter, which makes calls to an OLE DB provider or the APIs provided by the underlying data source.

ADO vs. ADO.NET Components

The following sections map some of the most used ADO objects and their counterparts in ADO.NET. Keep in mind that the recommendations described should be used only as guidelines for most code transformations when upgrading from ADO to ADO.NET. There may be differences in the behaviors between these components; therefore, if you choose to consider using any of these suggestions, it is recommended that you review the documentation for the particular component to determine if the new behavior will meet your needs.

Upgrading ADODB.Connection

This class represents an open connection to a SQL Server database. The equivalent in ADO.NET is **SQLConnection** in the **SqlClient** context and **OleDbConnection** in the **OleDb** context.

Upgrading ADODB.Command

The **Command** object is used to execute general commands on the data within the database. ADODB.Command has a direct equivalent in **SqlCommand** or **OleDbCommand**. This is one of the simplest objects in ADO; therefore, the equivalent members between both classes are easily identifiable.

Upgrading ADODB.Parameter

This class represents a parameter to **ADODB.Command**. The behavior in ADO.NET is basically the same, and the class works as a parameter for SQL commands and OLEDB commands.

Upgrading ADODB.Recordset

The functionality of **ADODB.Recordset** is spread across more than one class in ADO.NET. ADO.NET has three classes to manage information from the database:

- **DataReader**. This provides a fast, forward-only, read-only stream of data from a database.
- **DataTable**. This represents one table of in-memory data. It has the ability to store and manage a group of homogeneous rows.
- **DataSet**. This provides an in-memory relational representation of data: a complete set of data that works completely disconnected from the database. It is

necessary to have a **DataAdapter** to create a **DataSet**. With this adapter, you can update the data source if the **DataSet** is updated.

In most cases, a **DataTable** is a more direct upgrade for a **Recordset**. When a single and homogeneous group of data is stored in a **Recordset**, the **DataTable** is the most appropriate upgrade option. However, when your application works with multiple tables or hierarchies of results, the recommended upgrade option is a **DataSet**. Regardless of which option you choose, you must manually upgrade the **ADODB.Recordset** objects to objects of the new **DataSet** or **DataTable** classes.

Note: The Visual Basic Upgrade Wizard Companion includes support for the automated migration of **Recordsets** to **DataSets**. It also supports the most commonly used ADO components in typical usage patterns. For information about obtaining the Visual Basic Upgrade Wizard Companion, as well as additional migration tools and services, go to the ArtinSoft Web site.

If you have been working with ADO for any length of time, you have probably memorized all of the **Recordset** facts and now assume that this is how data access is supposed to work. Although there are changes in ADO.NET, there are some advantages and improvements to using the **DataSet**, which is the core data object in ADO.NET. The following summarizes some of the important basic information about the **DataSet**:

- A **DataSet** can represent an entire relational database in memory, complete with tables, relations, and views.
- A **DataSet** is designed to work without any continuing connection to the original data source.
- Data in a **DataSet** is loaded all at one time, instead of being loaded on demand.
- There is no concept of cursor types in a **DataSet**.
- **DataSets** have no current record pointer. You can use **For Each** loops to move through the data.
- You can store many edits in a **DataSet**, and write them to the original data source in a single operation.
- Although the **DataSet** is universal, other objects in ADO.NET come in different versions for different data sources.

Note: The **DataTable** class has been greatly extended in .NET Framework version 2.0. The extensions are aimed at making this class capable of standalone functionality. As an example, the **DataTable** class includes the ability to serialize a **DataTable** object.

As shown in the previous list, the most significant changes between ADO and ADO.NET are related to the differences between **Recordset** and **DataSet**. If you plan to pursue this upgrade path, it is recommended that you first learn more about the

differences between ADO and ADO.NET. For more references and information about ADO.NET, see:

- "Overview of ADO.NET" in the *.NET Framework Developer's Guide* on MSDN.
- "Comparison of ADO.NET and ADO" in *Visual Basic and C# Concepts* on MSDN.
- "Microsoft ADO.NET (Core Reference): Sample Code" in the Microsoft Download Center.
- "MSDN TV: First Look at ADO.NET 2.0" in the Microsoft Download Center.

Summary

Developing applications goes beyond just solving a particular business problem. To build robust and powerful applications, you must include features that are separate from the actual business requirements, but are nonetheless vital. Ensuring that users are properly identified and authenticated for sensitive operations is important. Creating applications with high performance and that are easy to scale as needs increase will keep them useful over a long period of time. Other key considerations are the need to keep applications up-to-date and easy to configure. This chapter has provided an introduction to these and similar scenarios that you should consider when upgrading your applications. Many new features available in Visual Basic .NET make it easier to address these situations, making this an ideal time to consider adding them to (or expanding them in) your applications.

More Information

For more information about the DES encryption algorithm, see "The DES Encyption Algorithm" on the Ius Mentis Web site:
http://www.iusmentis.com/technology/encryption/des/.

For more information about security and cryptography, see the following articles on MSDN:

- "Security Application Block":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/security1.asp.
- "Cryptography Application Block":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/crypto1.asp.
- "Cryptography Simplified in Microsoft .NET":
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/cryptosimplified.asp.

For more information about using the .NET Framework Configuration Tool, see ".NET Framework Configuration Tool (Mscorcfg.msc)" in *.NET Framework Tools* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconNETFrameworkAdministrationToolMscorcfgmsc.asp.

To learn more about performance counters for your application after you upgrade it to Visual Basic .NET, see "Performance Counters" in the *.NET Framework General Reference* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gngrfperformancecounters.asp.

For more information about using the .NET Framework configuration tool, see ".NET Framework Configuration Tool (Mscorcfg.msc)" in *.NET Framework Tools* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconNETFrameworkAdministrationToolMscorcfgmsc.asp.

For information about application configuration files, see "Application Configuration Files" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconapplicationconfigurationfiles.asp.

For more information about the **System.Configuration** namespace, see "System.Configuration Namespace" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemconfiguration.asp.

For more information about application configuration scenarios, see "Application Configuration Scenarios" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetapplicationconfigurationscenarios.asp.

For information about the Configuration Application Block, see "Configuration Application Block" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/config.asp.

For more information about setup projects, see "Setup Projects" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vbconSetupProjects.asp.

For more information about .NET deployment, see "No-Touch Deployment in the .NET Framework" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbtchNo-TouchDeploymentInNETFramework.asp.

To learn more about **Trace** and **Debug** classes, see "Trace Class" and "Debug Class" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemdiagnosticstraceclasstopic.asp
and:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDiagnosticsDebugClassTopic.asp.

For more information about performance considerations in software development, see *Improving .NET Application Performance and Scalability*, on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenet.asp.

For more information about database access and performance, see "Performance Tips and Tricks" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftips.asp.

For more information about multithreading and the **BackgroundWorker** component, see "BackgroundWorker Component Overview" on MSDN:
http://msdn2.microsoft.com/library/8xs8549b(en-us,vs.80).aspx.

For more information about multithreaded programming see "Asynchronous Pattern for Components" on MSDN:
http://winfx.msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_fxmclicc/html/792aa8da-918b-458e-b154-9836b97735f3.asp.

For more information about caching, see "Caching ASP.NET Pages" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconaspoutputcache.asp.

For more information about the **System.Messaging** namespace, see "System.Messaging Namespace" in the *.NET Framework Class Library* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemmessaging.asp.

To download the .NET Framework data provider for ODBC component, go to "ODBC .NET Data Provider" in the Microsoft Download Center:
http://www.microsoft.com/downloads/details.aspx?FamilyID=6ccd8427-1017-4f33-a062-d165078e32b1&displaylang=en.

For more information about OLE DB interfaces, see "OLE DB Interfaces Used by the .NET Framework Data Provider for OLE DB" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoledbinterfacessupportedbyoledbnetdataprovider.asp.

To download the .NET Framework data provider for Oracle, go to the Download & Code Center on MSDN:
http://msdn.microsoft.com/downloads.

For information about obtaining the Visual Basic Upgrade Wizard Companion, as well as additional migration tools and services, visit the ArtinSoft Web site:
http://www.artinsoft.com.

For more information about ADO.NET, see:

- "Overview of ADO.NET" in the *.NET Framework Developer's Guide* on MSDN:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoverviewofadonet.asp.
- "Comparison of ADO.NET and ADO" in *Visual Basic and C# Concepts* on MSDN:
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconadopreviousversionsofado.asp.
- "Microsoft ADO.NET (Core Reference): Sample Code" in the Microsoft Download Center:
  http://www.microsoft.com/downloads/details.aspx?FamilyID=bae2de67-0062-4bf5-b120-f970865be92e&DisplayLang=en.
- "MSDN TV: First Look at ADO.NET 2.0" in the Microsoft Download Center:
  http://www.microsoft.com/downloads/details.aspx?FamilyID=df7f7dc1-512f-4d07-b04f-17dde0fd318a&DisplayLang=en.

21

Testing Upgraded Applications

Testing an upgrade project is just as critically important as testing a development project, although the test effort required for testing an upgraded application may be significantly less than that required for testing a new application with similar requirements developed from the beginning.

To test an upgraded application, testers can take two different approaches, depending on their level of experience in testing upgraded applications. The first approach requires relatively more experience and less test effort than the second option.

In the first approach, it is assumed that the code automatically upgraded by the upgrade wizard works properly except in places where errors, warnings, and issues (EWIs) are present. In this approach, the test effort is focused on testing the fixes for the EWIs. However, even when EWIs are fixed, there can be functional holes whose pattern can be very different from those present in a normal development project. A tester who has considerable experience in testing upgraded applications can identify the functional holes by reviewing the code and the design of the application. There are no standard processes or guidelines for this, and all the steps are based on experience.

In the second approach, a more traditional test strategy and test process is followed, although this will likely require more test effort than the first approach. This approach is recommended for testers with limited experience testing upgrade projects. This chapter emphasizes this second approach.

The bigger the application to be upgraded, the smaller the savings in cost and effort for testing will be. In addition to the size of the application, the test effort and cost depend on the upgrade strategy, the upgrade plan, the test strategies applied and the count and complexity of upgrade issues listed by the Visual Basic Upgrade Wizard and/or the ASP to ASP.NET Migration Assistant. There are a handful of test strategies to choose for testing an upgraded application. The test strategies are based on test practices followed in different software development methodologies. They

all have a common core testing process, but they are different in how they integrate the testing process into the upgrade process. Each upgrade strategy can be a "best-fit" with only one of the test strategies.

This chapter explains which test strategy is the "best-fit" for each upgrade strategy. It deals with the test strategies, test process, test planning, and different tasks in testing.

Fitch & Mather Stocks 2000

The Fitch & Mather Stocks 2000 application will be used to illustrate the concepts presented in this chapter. Fitch & Mather Stocks 2000 (FMStocks 2000) is an ASP/Visual Basic 6.0 application developed for Microsoft Windows 2000 that simulates a public stock trading Web site on the Internet. The online stock trading scenario has been extended by adding an online bookstore. FMStocks 2000 consists of the following use cases:

1. Logon for existing user
2. Logon for new user
3. Logout
4. Symbol or company lookup
5. Display of account balance
6. Display of portfolio
7. Chart portfolio
8. Buy a stock
9. Sell a stock
10. View shopping cart
11. Browse products
12. Checkout

The projects that make up the business logic layer of FMStocks 2000 are:

- **FMStocks_Bus.vbp**. This project contains the business code for the online stock trading functionality. It includes four class modules: Account, Broker, Ticker, and Version. It references the **FMStocks_DB** component.
- **FMSStore_Bus.vbp**. This project contains the business code for the online bookstore functionality. It includes two class modules: Product and ShoppingCart. It references the **FMSStore_DB** and **FMSStore_EvtSub2** components.
- **FMSStore_Events.vbp**. This project defines the interface for creating events for external order processing for the online bookstore functionality.

- **FMSStore_EvtSub2.vbp**. This project implements the event for external order processing for the online bookstore functionality. It references the **FMSStore_Events** component defined in the FMSStore_Events.vbp project.

The projects that make up the data access layer of FMStocks 2000 are:

- **FMSStore_DB.vbp**. This project contains the data access code for the online bookstore functionality. It includes two class modules: Product and ShoppingCart. It references the **FMSStore_DB** component defined in the FMStocks_DB.vbp project.
- **FMStocks_DB.vbp**. This project contains the data access code for the online stock trading functionality. It includes seven class modules: Account, Broker, DBHelper, Position, Ticker, Product, and ShoppingCart.

Test Objectives

The objective of testing is to ensure that the upgraded application meets the upgrade objectives. If the upgrade objective is only to achieve functional equivalence, the test objective would be to test for functional equivalence. However, if the application is upgraded for performance and/or security enhancement, testing must include performance and security. Furthermore, if the application is being advanced beyond functional equivalence with new features, the test effort must include functional, performance, and security testing of these new features. Globalization testing should also be a part of test effort if it is one of the upgrade objectives.

Because the primary focus of this guidance has been how to reach functional equivalence in upgrading from Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET, this chapter focuses on testing for functional equivalence.

Testing Process

The testing process includes a basic set of steps that are to be followed regardless of the test or upgrade strategies used. These steps form the core of testing and are listed here:

- Create a test plan and test code
- Create a test environment
- Review the design
- Review the code
- Modifying existing test cases and code, or create additional test cases and code
- Perform unit testing – white box testing
- Perform black box testing
- Perform profiling – white box testing

These steps are performed on the upgraded application to validate that it conforms to functional specifications or functional equivalence with the original Visual Basic 6.0 version of the application.

The testing process may also include the optional design review, globalization testing, performance testing, and security testing. These are only required if they are included in the upgrade objectives.

The next sections describe each of these steps.

Create Test Plan and Test Code

The test plan documents the test cases that will be used to test the upgraded application. It also helps in planning and estimating the actual test effort that is required to prepare the test code and to execute the tests. Finally, the test plan helps in tracking the tests and their results, especially when there are multiple iterations or changes in specifications. Test plans are usually prepared very early in the upgrade life cycle. The information required to prepare a test plan include:

- The upgrade project plan.
- Use cases.
- Functional specifications.
- The original Visual Basic 6.0 source code or the upgraded Visual Basic .NET source code.

The methodology for creating test cases depends on the type of testing that will be performed. Test cases for design review, white box testing, and code review are prepared by reviewing the source code. Test cases for black box testing are prepared based on the use cases and the functional requirements. In the case of the upgrade project, if test cases are already available for the Visual Basic 6.0 version of the application, these test cases can be upgraded for the Visual Basic .NET version.

The guidance on how to identify the test cases for each step of the testing process is explained in the respective topics later in this chapter. However, you should keep in mind that as the upgrade proceeds, the test plan may be updated to modify existing test cases or to include new test cases. The test plan may also be modified to account for changes in functional specifications that occur during the upgrade.

The test plan usually consists of two documents; the detailed test plan (DTP) and the detailed test case (DTC) documents. The DTP is a list of the test cases that are indexed and prioritized based on the criticality of each test case. It contains summaries about each test case. This information includes a brief description of the feature, project, and/or component that is to be tested using the test case. The DTC contains detailed information about the test cases and is mapped to the DTP index. It includes detailed information about the test cases, such as steps to be followed to execute a test case, the expected result, the actual result, status of the test case execution (pass/fail), and the data/resources required to execute a test case. The DTC has to be updated with the actual result and the status of test execution each time a test case documented in the DTC is executed. The DTP and DTC documents are also updated whenever changes are made to any test cases.

Table 1 shows the test plan for the black box testing of the login use case of FMStocks 2000.

Table 1: Test Plan for the Black Box Testing of the Login Use Case for FMStocks 2000

Scenario 1		To test the functionality of Login Procedure For Existing User
Priority		High
Comments		
1.1	High	To test that both Login and Home page are displayed in correct format. That is, all links, the font, and the contents are same as those of the existing application in Visual Basic 6.0.
1.2	High	To test that if user enters a valid e-mail name and password, the user is redirected to the home page.
1.3	High	To test that if SQL Server is not running, the following message is displays: “Cannot open database connection.”
1.4	High	To test if the entered e-mail name and/or password are invalid, the following message displays: “Invalid e-mail and password combination. Please try again.”

Table 2 on the next page shows a sample test case for black box testing the login functionality of FMStocks 2000. Note that the test case also includes the following columns: Data Required, Actual Result, and Test OK (Y/N). They are not included in this table because they contained no information in this case.

Table 2: Sample Test Case for testing the Login Functionality for FMStocks 2000

Test case	Priority	Condition to be tested	Execution details	Expected results
1.1	High	To test that both login and home page are displayed in the correct format. That is, all links, the font, and the contents are the same as those of the existing application in Visual Basic 6.0.	Compare the login and home page with the corresponding pages in the Visual Basic 6.0 version of FMStocks 2000 and verify the following points: Content General look and feel of the Web page Location of input fields Font Links to other pages Functionality	Users should not feel any difference in the look and feel, fonts, functionality, content, and so on, between the login and home pages in the upgraded version and the corresponding pages in the Visual Basic 6.0 version of FMStocks.
1.2	High	To test that if user enters a valid e-mail name and password, the user is redirected to home page	Enter the following e-mail name and password on the login page E-mail: ta450 Password: ta	The user should be able to enter the site and should be redirected to the home page.

Test code to actually perform testing is created after the test cases are finalized. The test code usually consists of code to automate the execution of the test cases, sample test projects, test stubs, profiling and monitoring code, and other code to support the testing process. Test code is required for both white box and black box testing.

Create Test Environment

The test environment is generally comprised of the upgraded component or application to be tested, the basic software required to execute the application or component, the testing tools, and the test code. For testing an upgraded Visual Basic .NET application, the test environment should include the following items:

- The upgraded application or component to be tested
- Basic software for execution of the application, which consists of the following items:
  - The .NET Framework 1.1 and Visual Studio .NET 2003
  - Visual Basic 6.0 (in the case of iteration based test strategy or test driven upgrade strategy)
- Design review and code review tools (such as FxCop) and documents

- Unit testing tools (such as NUnit)
- Profiling tools, which may include the following tools:
  - CLR Profiler
  - Enterprise Instrumentation Framework (EIF)
  - Windows Management Instrumentation (WMI)
- Performance testing tools, which may include the following tools:
  - Application Center Test (ACT) for ASP.NET applications
  - Performance counters

For more information about testing tools, see the "Tools for Testing Visual Basic .NET Applications" section later in this chapter.

The test environment should be similar to the target/production environment, especially for black box testing, because the behavior of the application can change, depending on the environment the application will be deployed in.

Review the Design

This is an optional step that is not carried out unless design optimization is an upgrade objective or if the application is being upgraded to take advantage of features that are specific to the Microsoft .NET Framework, such as performance or security enhancements. Design review can be performed for several reasons, including:

- To verify that the design satisfies the functional specifications. This verification is performed only for the new features that are being added and advancements being made to the application. This verification is performed by verifying whether each functional feature in the functional specifications can be satisfied by the design of the application.
- To verify that the design includes the best practices from a performance and scalability point of view. This verification is performed only if performance enhancement is one of the upgrade objectives or if the application is being upgraded to take advantage of features that are specific to .NET. For more information, see Chapter 4, "Architecture and Design Review of .NET Application for Performance and Scalability," of *Improving .NET Application Performance and Scalability* on MSDN.
- To verify that the design includes the best practices from a security point of view. This verification is performed only if security enhancement is among the upgrade objectives. For more information about design review for security enhancement, see "Architecture and Design Review for Security" on MSDN.
- To verify that the design uses best practices for globalization: This verification is performed only if globalization is an upgrade objective.
- To verify that the design uses best practices for code maintainability.

Review of Code

Code review of the upgraded application is actually a part of white box testing. However, there are so many aspects associated it that it merits mentioning as a separate step instead of grouping it with white box testing.

Test cases for code review are created at the same time as the test cases for unit testing. To prepare the test cases for code review, either design documents are required or the Visual Basic 6.0 code must be analyzed to document the design. The assessment tool can be used to provide analytical information about the Visual Basic 6.0 code.

Review of the upgraded code includes:

- Validating that the implementation conforms to the design. This validation is performed only for the features that are advancements in the upgraded application or for design changes in the upgraded application. The tester must have the design documents to validate that the implementation adheres to the design.
- Validating that the naming standards and commenting standards have been followed in the code. There are several standard coding conventions recommended for Visual Basic and .NET Framework code that provide various guidelines such as capitalization styles, case sensitivity, indentation, acronyms and abbreviations. Your organization may also have its own coding standards. Adhering to coding standards makes code easier to understand and maintain.

  For information about recommended coding standards in your Visual Basic .NET code, see "Program Structure and Code Conventions" on MSDN.
- Validating that performance and scalability guidelines have been followed in the code. This validation is performed only if performance enhancement is an upgrade objective or if advancements have been applied the upgraded application that might impact performance.

  For more information about code review for performance and scalability, see Chapter 13, "Code Review: .NET Application Performance," of *Improving .NET Application Performance and Scalability* on MSDN.
- Validating that the guidelines for writing secure code have been followed. This validation is performed only if security enhancement is an upgrade objective or if the upgrade application has been advanced with .NET Framework technologies that might impact security. Security code reviews help in identifying areas in code that have security vulnerabilities because of failure to adhere to security guidelines. Some of the areas where the tester pays special attention to during code review are the following:
  - Review for hard coded information such as user names, passwords, and connection strings.

- Review whether proper permissions are in place for privileged operations and secure resources so that any other code accessing it must request permission to access them.
- Review for buffer overflows.
- Review for cross-site scripting attack vulnerabilities.

For more information about security code review, see Chapter 21, "Code Review," of *Improving Web Application Security: Threats and Countermeasures* on MSDN.

- Validating that globalization-related guidelines have been followed. This validation is performed only if globalization is a part of the upgrade objectives. For more information about best practices for globalization, see "Best Practices for Developing World Ready Applications" in the *.NET Framework Developer's Guide* or "Best Practices for Globalization and Localization," both of which are available on MSDN.
- Validating that the implementation has followed guidelines for code maintainability. The implementation is reviewed to verify that object-oriented principles have been followed. Unreachable or obfuscated code is detected and identified as a defect during code review.
- Validating that proper exception handling has been implemented in the code. Some of the exception handling guidelines that need to be verified are the following:
  - The exception handling code control should not control application logic.
  - The exception handlers should add some value such as logging the error message, cleaning up resources, customizing the exception message, or wrapping the exception in a custom exception. Exceptions should not be handled without a justifiable, documented reason.
  - Exceptions should not be re-thrown unless absolutely necessary because throwing exception is expensive.
  - Exceptions should not be swallowed. Exceptions should be allowed to traverse back through the call stack to the outer layer unless the functionality requires otherwise.
  - Exception handler **Finally** blocks should clean resources as appropriate.
  - Logging the error message in exception handlers is recommended.
  - Specific exceptions should be handled before general exceptions.
  - Custom application exceptions should be preferred over standard exceptions.

  For more information about the exception management architecture, see the *Exception Management Architecture Guide* on MSDN.

- Identifying failure scenarios or additional test scenarios. During the code review, the code is examined for failure scenarios, especially those related to resource contention, increased resource utilization, invalid inputs, and so on. For example, a loop in a method may work properly with only positive numbers as its upper limit and fail with negative numbers. If the upper limit number for the loop is provided as an argument to the method, the method may fail if a negative number is provided as the argument. If this is detected during code review, a defect is logged and a test case for unit testing is created for this scenario.

  In general, if areas in code are suspected to result in issues such as resource management contention, deadlocks, or invalid inputs, test cases are created around them to facilitate unit testing, profiling, or black box testing.

  The upgraded code is also reviewed to find additional test scenarios for the following conditions:

  - Boundary conditions
  - Invalid input
  - Thread safety and deadlock conditions

Performing Unit Testing – White Box Testing

White box testing is testing based on analysis of the implementation details and the program logic of the component or application tested to identify potential failure scenarios. White box testing includes unit testing and profiling the upgraded application. Profiling will be explained later in this chapter.

Unit testing involves testing a single component. The components of the upgraded application are unit tested to detect failure scenarios in loops, conditional constructs, and internal subroutines. Unit testing also focuses on testing if upgrade EWIs have been properly fixed without affecting functionality or introducing new defects. Unit testing is also performed to make sure that the behavior of a method in an upgraded component is the same as that of the corresponding method in the original Visual Basic 6.0 version of the component.

The test cases for unit testing are obtained by reviewing either the Visual Basic 6.0 source code or the upgraded Visual Basic .NET source code, depending on the test strategy and by reviewing the assessment tool report for the upgrade EWIs. The test cases and the test code are prepared during the test planning stage, but they may be modified during the code review stage. Additional test cases and test code may also be added during the code review stage.

In an upgrade project, if test cases for unit testing are present for the Visual Basic 6.0 version of the application, they are upgraded and reused for the Visual Basic .NET version of the application. Similarly, the existing Visual Basic 6.0 test code is upgraded to Visual Basic .NET.

Unit testing is usually automated with the help of developer tools. One example of a tool that helps to automate unit testing in the .NET Framework is NUnits, which is discussed in the "Tools for Testing Visual Basic .NET Applications" section later in this chapter.

Table 3 shows a sample test plan and test cases for unit testing for FMStocks 2000.

Table 3: Test Plan for Unit Testing the FMSStore_Cart.ShoppingCart Class

Scenario 1		Class FMSStore_Cart.ShoppingCart
Priority		High
Comments		
1.1	High	Public Function GetByKey(ByVal AccountID As Integer, ByVal SKU As Integer) As ADODB.Recordset
1.2	High	Public Sub Add(ByVal AccountID As Integer, ByVal SKU As Integer)
1.3	High	Public Sub Buy(ByVal AccountID As Integer)
1.4	High	Public Function ListByAccount(ByVal AccountID As Integer) As ADODB.Recordset
1.5	High	Public Sub SetQuantity(ByVal AccountID As Integer, ByVal SKU As Integer, ByVal Quantity As Short)
1.6	High	Public Function TotalByAccount(ByVal AccountID As Integer) As ADODB.Recordset
1.7	High	Public Sub EmptyShoppingCart(ByVal AccountID As Integer)

Table 4 on the next page shows a sample test case for unit testing the **FMSStore_Cart.ShoppingCart** class. Note that there is an additional column, named Test OK (Y/N), which is not included in this table because it has no entries in this case.

Table 4: Sample Test Case for the FMSStore_Cart.ShoppingCart Class

Test case	Priority	Condition to be tested	Execution details	Data required
1.1a	High	Public Function GetByKey(ByVal AccountID As Integer, ByVal SKU As Integer) As ADODB.Recordset - All parameters within valid range	All input parameters in the valid range. **Input Parameters:** AccountID = 5249 SKU = 1004009 **Expected Output:** RecordSet with the following values in the fields Quantity = 1 SKU = 1004009 Price = 29.95 Description = The Secrets of Investing in Technology Stocks **Actual Output:**	NUnit Test Case: GetByKey_Valid
1.1b	High	Public Function GetByKey(ByVal AccountID As Integer, ByVal SKU As Integer) As ADODB.Recordset - AccountID parameter higher the valid range	AccountID parameter higher than the valid range. **Input Parameters:** AccountID = 5250000 SKU = 1004009 **Expected Output:** Empty RecordSet **Actual Output:**	NUnit Test Case: GetByKey_ AccountInValid

Black Box Testing

In black box testing, the implementation details of the component or application to be tested are unknown. Black box testing simulates the end-user/client experience and tests the response of the upgraded applications to end-user or client actions. The objectives of black box testing are:

- To test for functional equivalence of the upgraded application with the original Visual Basic 6.0 version of the application.
- To test that the upgraded application has achieved the upgrade objectives.
- To test that the upgraded application can respond properly to normal and exceptional usage scenarios. For invalid inputs, the upgraded application should

provide error messages that are clear and present sufficient information to the user to enable diagnosis of the error.

- To test performance and security features, if required.

Black box testing includes all of the following:

- **Validation testing**. This includes testing that the component or application satisfies the documented specifications and customer/client needs.
- **External interface testing**. This includes testing that publicly-accessible methods and properties of a component exhibit the expected behavior.
- **Boundary condition testing**. This includes testing boundary values for parameters. For example, if a method requires that an integer value greater than 0 be provided as a parameter, boundary condition testing would test values near this boundary, such as -1, 0, and 1.
- **Destructive testing**. This includes testing extreme scenarios where failure of the application is guaranteed, but the application should be able to gracefully handle the failure. This testing involves removing or deleting aspects of the execution environment of the application, such as removing permissions to access resources, deleting the database accessed by the component, and disabling the network. The application is expected to handle these scenarios without losing data or permanently changing the state of the data. It should also have the capacity to rebound after the proper execution environment is completely restored.
- **Load testing**. This includes tests to verify that the upgraded application meets the desired performance objectives without overshooting its allocated budget of resources such as memory, I/O, and processor and network utilization. Load testing is performed only for the application advancements or if performance enhancement is an upgrade objective.
- **Stress testing**. This evaluates the upgraded application's behavior when the load on the application is pushed beyond normal values. The objective is to verify that the application is able to properly handle failures under heavy load conditions. Stress testing is performed only if performance enhancement is an upgrade objective or if the application has been advanced with new .NET Framework technologies.
- **Security testing**. This is used to test whether the upgraded application has security vulnerabilities. Test cases for the security testing are based on the threat models and countermeasures that were developed based on the design review of the upgraded application. Security testing verifies that the countermeasures effectively counter the threats to the application. It also determines whether all the threats have been identified. All potential types of input that may break the security, such as inputs that may cause buffer overflows and SQL Injection inputs, are used for security testing. Security testing is performed if security is an upgrade objective.

- **Globalization testing**. This includes tests to verify if the upgraded application is capable of globalization. Globalization testing is performed if globalization is an upgrade objective.

In black box testing, the test environment should be as close to the target environment as possible because the behavior of the upgraded application may change based on the environment the application is deployed in. Thus, it is necessary for the test environment to match the deployment environment of the application to avoid differences in behavior.

Black box testing is performed based on the requirements, functional specifications, and use cases. The test cases for black box testing should be documented during the creation of the test plan.

Table 5 shows the test plan for the black box testing of the "buy a stock" use case for FMStocks 2000.

Table 5: Test Plan for Black Box Testing of the "Buy a Stock" Use Case for FMStocks 2000

Scenario 1		To test the functionality of buying stock
Priority		High
Comments		
8.1	High	To test that the Buy A Stock page is displayed when the "buy a stock" link on the left side of the screen is clicked.
8.2	High	To verify that the Buy A Stock page's appearance, look and feel, fonts, and all other user interface features are similar to the corresponding page in ASP.
8.3	High	To verify that after the user enters the ticker and the valid number of shares, the shares are allocated to the user, and the market price of the shares is debited from the user account.
8.4	High	To verify that if an invalid ticker is entered, the following message displays: "The ticker symbol you entered is not valid. Please try again."
8.5	High	To verify that if an invalid number of shares is entered, such as negative values or decimal numbers, no transaction takes place and the following message displays: "Invalid number of shares."
8.6	High	To verify that if there is not enough money in the user's account to buy the entered number of shares, no transaction takes place and the following message is displayed: "Don't have enough cash in your account to purchase the requested shares."
8.7	High	To test that if SQL server is not running, the following message displays: "Cannot open database connection."
8.8	High	To test that if the SQL connection is lost during in the midst of the transaction, the transaction is still atomic.

Table 6 shows a sample test case for black box testing the "buy a stock" use case of FMStocks 2000. Note that the test case also includes the following columns: Data Required, Actual Result, and Test OK (Y/N). They are not included in this table because they contained no information in this case.

Table 6: Sample Black Box Test Case for the "Buy a Stock" Use Case for FMStocks 2000

Test case	Priority	Condition to be tested	Execution details	Expected results
1.1	High	To test that both login and home page are displayed in the correct format. That is, all links, the font, and the contents are the same as those of the existing application in Visual Basic 6.0.	Compare the login and home page with the corresponding pages in the Visual Basic 6.0 version of FMStocks 2000 and verify the following points: Content General look and feel of the Web page Location of input fields Font Links to other pages Functionality	Users should not feel any difference in the look and feel, fonts, functionality, content, and so on, between the login and home pages in the upgraded version and the corresponding pages in the Visual Basic 6.0 version of FMStocks.
1.2	High	To test that if user enters a valid e-mail name and password, the user is redirected to home page	Enter the following e-mail name and password on the login page E-mail: ta450 Password: ta	The user should be able to enter the site and should be redirected to the home page.

White Box Testing – Profiling

Profiling is the process of gathering statistical information about an application during run time, such as the time and resource usage of an application and the execution tree exercised. This information can be analyzed for different purposes, such as identifying components that can be optimized to improve speed or resource use.

In the context of white box testing, the upgraded application is profiled to identify failure scenarios and issues based on resource management, memory allocation, resource contentions, code coverage, and deadlocks. In the cases resource management, deadlocks, and resource contention issues, profiling is not performed on the entire application; it is performed only on those parts of the application where it is

suspected that these issues occur as identified during black box testing or code review.

Profiling an upgraded application monitors the application at run time. In conjunction with code coverage testing, profiling can be used to detect dead code (code that never gets called or executed). Code coverage testing executes test cases that exercise the entire execution tree. These test cases are designed to cause every possible path of execution to occur at least one time. Note that code coverage testing is not critical from the functional equivalence perspective because extra functionality in a component or application will not affect the behavior of the required functionality. During code coverage testing, the application can be profiled to identify the methods that are called. Because the code coverage testing exercises all possible execution paths, any methods that are never called during this testing can be identified as dead code.

Profiling is also performed to identify excessive utilization of a resource such as network I/O, disk I/O, memory usage, and CPU. Excessive resource use indicates that the code is blocking resources disproportionately to other applications. Profiling helps to detect such code. This type of profiling is performed if the upgraded application consumes too much memory or takes longer than expected to complete execution during black box testing. In such instances, profiling can be limited to isolated components that are suspected of causing the issue to identify the cause.

The code can also be profiled to ensure that there no memory leaks or fragmentation of the heap. This type of profiling is beyond the scope of this chapter. For more information about memory profiling using the .NET Common Language Runtime profiler, see "How To: Use CLR Profiler" in *Improving .Net Application Performance and Scalability* on MSDN.

An Overview of Test Strategies

Test strategies are based on testing practices followed in different software development methodologies such as the waterfall model and iterative development model. These strategies can also be applied to different types of upgrade projects. The test process is fairly consistent across the different strategies, but the test planning and the test effort varies for these strategies. This section examines several different test strategies that can be applied in upgrade projects.

Test Strategy Based on the Waterfall Methodology

One possible testing strategy is based on the waterfall development methodology. In this strategy, after a single project of the entire application is upgraded, the upgraded project is unit tested. Then after all projects of the application are upgraded, all the upgraded projects/components are integrated to get a working version of the application, which is then system tested for functional equivalence. This test strategy is illustrated in Figure 21.1.

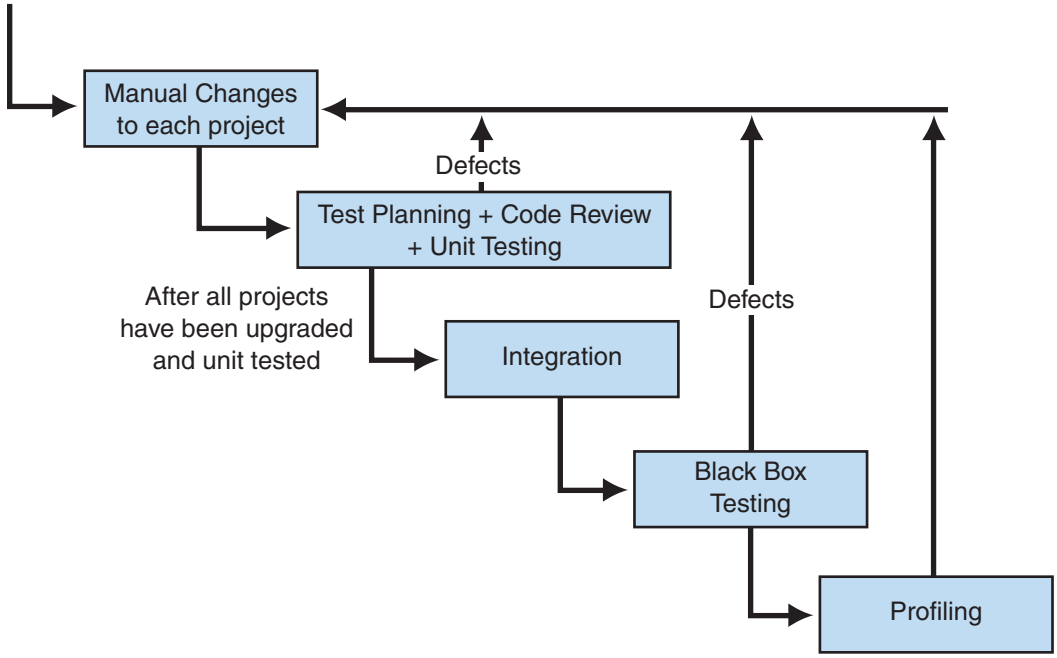

Figure 21.1
Graphical representation of the waterfall upgrade process

This test strategy suits the complete upgrade strategy best. In the complete upgrade strategy, all the components/projects of the application are upgraded before they are integrated to get a complete working version. This complete application is then system tested for functional equivalence. For information about the complete upgrade strategy, see the "Complete Upgrade" section of Chapter 2, "Practices for Successful Upgrades."

In the complete upgrade strategy, a working version of the application is not created until late in the upgrade life cycle. For this reason, the complete upgrade strategy fits with the test strategy that is based on the waterfall model. However, this is also a disadvantage because problems in the system testing phase are more costly to fix because they are harder to isolate at the system level than at the unit level. Because of this, it is recommended that you restrict the usage of this test strategy to relatively simple applications with a small number of components.

If an upgraded application has been advanced using .NET features such as replacing ADO with ADO.NET, the advancement would have required a design phase. Any flaw related to the design during the system testing phase is costly to fix, so applications with advancement features are not suitable for this test strategy. Furthermore, this test strategy does not accommodate uncertainties in the application that may have been passed down to the testing phase. In this test strategy, it is required that the previous upgrade phase is completely frozen with no uncertainties lingering

until the test phase. If the application includes advancements that may have some uncertainties, this strategy is not the best strategy to use.

This test strategy is the best strategy for upgrading FMStocks 2000. Based on the source code metrics and the project files overview generated by the assessment tool, FMStocks 2000 is a medium-sized application that can be upgraded completely. FMStocks 2000 has 4,943 lines of code divided between 22 files, 3 project groups, and 6 projects. For more information about the upgrade of the FMStocks 2000 application, see Appendix D, "Fitch and Mather Stocks 2000 Upgrade Case Study." This case study also details the testing of this application, which follows the strategy described in this section.

The general test process for this strategy is specified here:

1. During upgrade planning and after the upgrade strategy is decided, create test plans and test cases for black box testing based on the functional specifications or the original Visual Basic 6.0 version of the application. If test plans and test cases already exist for the Visual Basic 6.0 version of the application, upgrade them for testing the Visual Basic .NET version of the application. From the assessment reports, set tentative effort estimate for unit testing because the test cases and test plan for unit testing are not developed at this stage (although if unit test cases exist for Visual Basic 6.0 version of the application, they can serve as the basis for the estimates).
2. Perform the automated upgrade for each upgrade project.
3. Apply manual changes to each upgrade project, concentrating on fixing the EWIs.
4. Create test plans, test cases, and test code for unit testing the upgraded project.
5. Perform a design review if required.
6. Perform a code review.
7. Modify or add additional test cases and test code for unit testing or black box testing based on the code review.
8. Perform unit testing, focusing on the EWIs.
9. After all the projects are upgraded and unit tested, integrate them and create a working application.
10. Perform black box testing.
11. If any resource management or deadlock issues are suspected based on the black box testing, perform profiling to identify the issues.
12. Fixes for defects found during code review, unit testing, black box testing, or profiling must undergo a complete regression testing cycle of code review, unit testing, black box testing, and profiling.

Test Strategy Based on the Iterative Methodology

Another possible test strategy for upgrade projects is based on the iterative development methodology. This test strategy overcomes the shortcomings of the waterfall-based test strategy and is best used in complete upgrade projects for complex applications and for staged upgrade projects for reasons explained later in this chapter.

In a staged upgrade strategy, each project/component is upgraded and is integrated with the system. The system consists of the .NET upgraded projects and the Visual Basic 6.0 projects communicating through interoperability techniques. After each project is upgraded, it is unit tested. After it passes testing, it is integrated with the system and the entire system is tested for functional equivalence. This ensures that a working system is present at all times.

In this strategy, the waterfall-based test strategy is repeated over a number of iterations. One iteration is performed after each project is upgraded and integrated with the system. Then the system is tested for functional equivalence. This test strategy is best for the staged upgrade strategy and for the complete upgrade strategy for complex projects with many components. This test strategy requires more coordination between the test team and the development team than the previous test strategy. In particular, the testers must be knowledgeable about the upgrade plan to know which component is to be upgraded at each point in the upgrade process.

As previously stated, in a complete upgrade strategy, a working system is developed only during the last few phases of the project. Any bug detected in the last phases of the upgrade cycle, including the system testing phase, will be costly to fix because bugs that appear in an integrated application are more difficult to isolate than those found in unit testing. This test strategy makes it possible to find bugs and take corrective actions at the end of each iteration. Bugs detected at the end of each iteration will be less costly to fix than those that are not identified until after the upgrade is complete because each iteration will have isolated changes. Therefore, it is recommended to upgrade large and complex projects using the iterative model based test strategy. Here, the test team receives an upgraded project unit and tests it, integrates it with the system, and then tests the complete system for functional equivalence just as in the case of the staged upgrade strategy. Though the task of integration is an overhead, it is required to ensure that the system is being thoroughly tested.

This strategy mitigates the risk of finding major flaws in the last few phases of the upgrade cycle, but it introduces the overhead of system integration after each project is upgraded in a complete upgrade strategy. It may also introduce an overhead if there is any change in the projects that have already been upgraded and integrated. Therefore, it is required that the test team implement an efficient change control process to manage changes made to the system in between iterations.

FMStocks 2000 upgrade is not an appropriate candidate for this test strategy, but if it was applied, the upgrade and testing steps would be as follows:

1. The ASP pages are upgraded to ASPX pages and integrated with the business components.
2. The ASPX pages are tested for functional equivalence with the ASP pages.
3. FMStocks_Bus.vbp project is upgraded and integrated with the system. The upgraded Visual Basic .NET version of FMStocks_Bus component replaces the Visual Basic 6.0 version of FMStocks_Bus in the working system and is made to interoperate with the FMStocks_DB component that has yet to be upgraded.
4. The upgraded FMStocks_Bus project is then unit tested, followed by the testing of the integrated system for functional equivalence.
5. The FMSStore_Bus.vbp project is upgraded. The upgraded Visual Basic .NET version replaces the original Visual Basic 6.0 version in the system. The upgraded version of the component will then be made to interoperate with FMSStore_DB and FMSStore_EvtSub2 components.
6. The upgrade process continues for another project in the FMStocks 2000 application. After each project is upgraded, it is unit tested, integrated with the entire system (using interoperability techniques if there are still Visual Basic 6.0 components to be upgraded), and the entire system would be tested for functional equivalence. This would be repeated for each project of the application until all projects have been upgraded.

The test process for this strategy is specified here:

1. During upgrade planning and after the upgrade strategy is decided, create test plans and test cases for black box testing based on the functional specifications or the original Visual Basic 6.0 version of the application. Create test cases and test code for unit testing based on the analysis and review of the Visual Basic 6.0 version of the application. If test plans, test cases, and test code already exist for the Visual Basic 6.0 version of the application, upgrade them for testing the Visual Basic .NET version of the application.
2. Perform automated upgrade for an upgrade project in an iteration cycle.
3. Perform manual changes to the upgraded project concentrated on fixing the EWIs.
4. Integrate the upgraded project with the system.
5. Perform design review if required.
6. Perform code review.
7. Modify or add additional test cases and test code for unit testing or black box testing based on the code review.
8. Perform unit testing, focusing on the fixes for EWIs.
9. Perform black box testing.

10. If any resource management or deadlock issues are suspected based on the black box testing, perform profiling to identify the issues.
11. Repeat the steps 2 through 10 for the remaining iteration cycles.
12. Fixes for defects found during code review, unit testing, black box testing, or profiling must undergo a complete regression testing cycle of code review, unit testing, black box testing, and profiling.

Test Strategy Based on the Agile Methodology

If the applications to be upgraded are large and very complex — such as if they are composed of several projects or project groups, contain multiple cross-references between projects or components, or have dependencies on external systems— the test strategies mentioned earlier become less effective and reliable. This is also true if the Visual Basic 6.0 application has to be advanced with .NET technologies, such as replacing all business COM components with Web services. This situation becomes more complicated if the requirements and specifications for the advancements are not clearly defined.

A test strategy that is robust enough to handle the complexity and that absorbs changes quickly and smoothly is required in these situations. The test strategy that is based on the agile development methodology is the best choice in these situations.

The agile methodology is similar to the iterative methodology in that it progresses in iterations, but the time periods for iterations are shorter. These time periods are also typically considered strict delivery deadlines instead of planned goals (as is the case in the iterative methodology). It incorporates the principles of the iterative methodology of development, test, and feedback. Test-driven development is a core practice of the agile methodology. Test-driven development is the practice of developing a test case and then building the functionality to satisfy the test case, which is followed by refactoring of the implementation. This results in very short development cycles. Extending the principles of the test-driven development to the upgrade process requires using test cases to test a single feature and then fixing code until the test cases for that feature are satisfied. This is followed by refactoring. This process is the test-driven upgrade process.

In the test-driven upgrade process, the test cases reflect the functional specifications of the system. These test cases do not change unless the requirements or specifications of the application change. The upgrade is performed to satisfy test cases, one feature at a time. The test-driven upgrade process highlights the feedback concept. The upgrade phase is periodically punctuated by test iterations, that is, each upgrade iteration is followed by a test iteration. The test iteration provides a feedback on the preceding upgrade iteration. Each iteration upgrades the functionality in an incremental fashion, and the test iteration that follows tests the incremented functionality and the complete system's functionality. No other portion of the application

is upgraded until after only those sets of test cases not related to the targeted upgrade feature fail.

In the test-driven upgrade process, the test cases are prepared in the early phases of the project before the actual upgrade or the advancement work begins. The upgraded code changes according to the test cases. The test cases include cases for unit testing and system testing. They cover the entire functionality of the application and are categorized into subsets based on the different use cases or functionalities of the application. As mentioned earlier, the entire upgrade phase is divided into iterations. In an upgrade iteration, only a small portion of the application is upgraded to satisfy a subset of test cases and to upgrade a subset of entire the application's functionality. Feedback is then obtained about the upgraded code by testing it with the complete set of test cases. If any test case in the subset of test cases to be satisfied fails, the upgraded code is modified to make the failed test case pass. The iteration is not complete until all the test cases in the target subset of test cases pass. If all the test cases associated with the upgraded functionality pass, the code is refactored to clean any duplication of design, dead code, identifiers and names, and comments. Then the next upgrade iteration starts. In the next iteration of the upgrade phase, another portion of the application is upgraded that builds on the functionality that has been already upgraded. This iteration is again followed by a test iteration and then code refactoring. Subsequent iterations increment the functionality and test the incremented functionality. If the upgraded code breaks during a test iteration, the code is modified to fix only those test cases associated with the current and previous iterations. The upgrade of the feature is not considered complete until these test cases pass. Further upgrade iterations are allowed only after that. If any change occurs in the requirements in the middle of the upgrade process, the test cases are updated accordingly and the upgraded code runs against the test cases. Because of changes in the test cases, some of the test cases may fail and the upgraded code will have to be modified to pass the failed test cases. This is how the test-driven upgrade process operates.

This alternating periodic cycle of upgrade and test ensures that a change in the requirements or the specifications in the middle of the upgrade phase has little impact and can be easily absorbed. This makes the test strategy flexible to changes. Furthermore, the requirements and specifications, especially for the application advancements, can be evolved along with the progress of the upgrade. Because test-driven upgrade can absorb changes effectively, it reduces the overhead of extensive planning before upgrade. When the application is being advanced, there is a scope for architectural change, and it is not required to sketch in detail the way the architecture is going to be modified. The practice of not allowing further upgrade iterations until all defects or pending issues with the upgraded code are resolved ensures robustness of the code even in complex applications.

In addition to the similarity of the practice of having multiple iterations of upgrade and test, the other similarity between the test-driven upgrade and the iterative

model based test strategy is that a working version of the upgraded application is maintained throughout the upgrade and test phase. However, the difference between the test-driven upgrade strategy and the iterative model-based test strategy is the way the upgraded system is kept functional throughout the upgrade and test phase. In the iterative model-based test strategy, the system is made functional by interoperating with components that have not been upgraded. This introduced an overhead of writing interoperability code to keep the system functional. In contrast, the test-driven upgrade strategy does not have that overhead. Instead, the system is divided into multiple functional units that can stand alone and each functional unit is upgraded and integrated with existing upgraded functional units. To divide the application into multiple functional units, it is necessary to categorize the components or projects according to the different use cases or functionalities of the application. For example, the FMStocks 2000 projects can be mapped with the different use cases as follows:

- Login for existing user:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- Login for new user:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- Logout:
  - FMStocks_Bus.vbp
- Symbol or company lookup:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- Display of account balance:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- Display of portfolio:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- Chart portfolio:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- Buy a stock:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp

- Sell a stock:
  - FMStocks_Bus.vbp
  - FMStocks_DB.vbp
- View shopping cart:
  - FMSStore_Bus.vbp
  - FMSStore_DB.vbp
- Browse products:
  - FMSStore_Bus.vbp
  - FMSStore_DB.vbp
- Checkout:
  - FMSStore_Bus.vbp
  - FMSStore_DB.vbp
  - FMSStore_EvtSub2.vbp
  - FMSStore_Events.vbp

Although FMStocks 2000 is an unsuitable candidate for test-driven upgrade, if it is upgraded using the test-driven upgrade strategy, the following steps will be followed:

1. ASP pages will be upgraded to ASPX pages. The projects FMStocks_Bus.vbp and FMStocks_DB.vbp will be upgraded.
2. The upgraded FMStocks_DB and FMStocks_Bus projects will be unit tested.
3. The following functionalities/use cases will be tested for functional equivalence:
   - Login for existing user
   - Login for new user
   - Logout
   - Symbol or company lookup
   - Display of account balance
   - Display of portfolio
   - Chart portfolio
   - Buy a stock
   - Sell a stock
4. The projects FMSStore_Bus.vbp, FMSStore_EvtSub2.vbp, FMSStore_Events.vbp, and FMSStore_DB.vbp will be upgraded.
5. The upgraded projects FMSStore_Bus.vbp, FMSStore_EvtSub2.vbp, FMSStore_Events.vbp, and FMSStore_DB.vbp will then be unit tested.

6. The following use cases will then be system tested:
   - View shopping cart
   - Browse products
   - Checkout
   - Login for existing user
   - Login for new user
   - Logout
   - Symbol or company lookup
   - Display of account balance
   - Display of portfolio
   - Chart portfolio
   - Buy a stock
   - Sell a stock

Tools for Testing Visual Basic .NET Applications

The current version of Visual Studio .NET does not provide tools specifically for automating the testing of Visual Basic .NET applications. However, other tools do exist for automating tests. This section examines some of the tools available.

NUnit

A popular tool for performing unit testing on applications in the .NET Framework is named NUnit. NUnit provides a unit-testing framework that can be applied to any .NET Framework language, including Visual Basic .NET. For more information and to download NUnit, see the NUnit Web site.

FxCop

The FxCop tool is used to automate code review. The FxCop tool has various sets of pre-defined rules that are similar to the guidelines provide in the "Review of Code" section earlier in this chapter. The tool analyzes the application binaries to determine whether the implementation adheres to the predefined rules. These rules are customizable and can be extended. For more information about FxCop, see the FxCop Team Page on the .NET Framework Community Website.

Application Center Test (ACT)

Application Center Test is a performance testing tool. It is designed to stress Web servers and to test the performance of ASP.NET applications. ACT simulates a large number of users by opening multiple connections to the server so that a real-world environment with thousands of simultaneous users accessing a Web site can be simulated. Using ACT, a Web application can be load tested and stress tested. Furthermore, functional testing can be performed using programmable dynamic tests. ACT is included with the Enterprise Edition of Visual Studio .NET. For more information about ACT, see "Microsoft Application Center Test 1.0, Visual Studio .NET Edition" on MSDN.

Visual Studio Analyzer

Visual Studio Analyzer is a performance analysis tool that is used to analyze and debug distributed applications. This tool analyzes components at a high level and presents information that can be used to identify the components that fail in a distributed environment. For more information about Visual Studio Analyzer see "Visual Studio Analyzer" on MSDN.

Trace and Debug Classes

The .NET Framework comes with a pair of classes that can be used to print useful debugging messages during execution to allow you to see the flow of your application and to help you spot unusual behavior. These classes are **Debug** and **Trace** and both are defined in the **System.Diagnostics** namespace.

Both classes are very similar and contain static methods to print arbitrary text messages, print messages only when tested expressions evaluate to true, pause and ask the user what to do when certain conditions do not evaluate to **true**, and format the text messages using simple indentation commands.

The only difference between these classes is that by default the **Debug** method calls are not compiled into the **Release** builds of your application, and **Trace** method calls are compiled into the **Release** builds. You can change this behavior by defining compiler directives on the compiler command line (or in your code — though this is less likely) to include the **Trace** and **Debug** method calls whenever you want. This debugging mechanism uses a technique named conditional compilation, which is borrowed from the C programming language. When the compiler directives are not defined, the use of the **Trace** and **Debug** classes is not even compiled into your final code, so there is no performance hit in your final release. This is very helpful because it allows you to put the same amount of tracing information into your code without having to be concerned about performance in the final release version.

An additional advantage of these classes is that they can be configured to increase and decrease the amount of information that they produce using external configuration files. This allows you to adjust the information that is generated. You can even release the application with the tracing information included, but you should turn off the tracing so that it is not visible until you need it.

TraceContext Class

The **TraceContext** class provides functionality similar to the **Trace** and **Debug** classes, but it provides this functionality for ASP.NET applications. This class contains fewer methods but allows you to add conditional text to your HTML pages that are generated by ASP.NET and formats them for viewing in your browser. The page also automatically includes information that is related to your request and that can help debug problems in your Web applications.

CLR Profiler

CLR Profiler is a tool used for profiling memory utilization by an application. CLR Profiler profiles the interaction between the managed applications, the managed heap, and the garbage collector. CLR Profiler is used to identify memory leaks. For more information about CLR Profiler, see "How To: Use CLR Profiler" in *Improving .NET Application Performance and Scalability* on MSDN.

Enterprise Instrumentation Framework (EIF)

EIF is used to instrument a .NET application so that it can be profiled and traced. After it is profiled and traced, application metrics can be captured. EIF encapsulates the functionality of event logging, Event Tracing for Windows (ETW), and Windows Management Instrumentation (WMI). EIF provides the means to publish errors, warnings and informational events. Even business-specific events can be published. EIF is used to profile for code coverage testing. It is also used to profile for identifying functional loopholes or specific failure scenarios that are caused by resource contention or invalid inputs.

For more information about how to use EIF, see "How To: Use EIF" in *Improving .NET Application Performance and Scalability* on MSDN.

Performance Counters

Another Windows service that the .NET Framework provides access to and that can help meet your testing requirements is performance counters. Windows monitors many different aspects of the .NET runtime and the rest of the operating system. You can even define your own performance counters and then monitor those counters from the Performance Monitoring tool (Perfmon.exe) or from within your own applications.

For more information about performance counters in .NET, see "Performance Counters" in the *.NET Framework General Reference* on MSDN.

For more information about how to create and use custom performance counters, see the following How Tos in the patterns & practices guide *Improving .NET Application Performance and Scalability* on MSDN (the links for the How Tos are in the left pane):

- "How To: Monitor the ASP.NET Thread Pool Using Custom Counters"
- "How To: Time Managed Code Using QueryPerformanceCounter and QueryPerformanceFrequency"
- "How To: Use Custom Performance Counters from ASP.NET"

Summary

Testing is arguably the most important part of any software development project, including upgrade projects. Carefully planned and executed testing processes help to identify bugs in and verify the correctness of the software.

The options available for testing upgrade development are the same as those available for any type of software development. In fact, the test strategies themselves are based on the various software development methodologies. The particular test strategy you choose to apply when upgrading applications will depend on the characteristics of the application you are upgrading. Whether you perform testing at the end of the upgrade or in iterative stages throughout the upgrade process will depend on the size and complexity of the application itself. For small or simple applications, waiting until the upgrade has been completed will require the least amount overhead and will, therefore, be more efficient. For larger and more complex applications, a better approach is to perform testing on individual components as each is upgraded, or to perform iterative testing. Although these approaches have increased overhead because of increased test passes, the benefit is that they better isolate bugs and other issues earlier in the upgrade when they are easier to fix.

Having a clear understanding of the steps of the test process will make testing more efficient and increase the accuracy of the results.

This chapter has provided the details you need to make the best decisions about how to test your upgraded applications and the steps to needed to perform those tests.

More Information

For more information about upgrading performance, see Chapter 4, "Architecture and Design Review of .NET Application for Performance and Scalability," of *Improving .NET Application Performance and Scalability* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt04.asp.

For more information about design review for security enhancement, see "Architecture and Design Review for Security" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secmod/html/secmod78.asp.

For information about recommended coding standards in your Visual Basic .NET code, see "Program Structure and Code Conventions" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vbconProgrammingGuidelinesOverview.asp.

For more information about code review for performance and scalability, see Chapter 13 "Code Review: .NET Application Performance" of *Improving .NET Application Performance and Scalability* on MSDN:
http://msdn.microsoft.com/library/en-us/dnpag/html/ScaleNetChapt13.asp.

For more information about security code review, see Chapter 21, "Code Review," of *Improving Web Application Security: Threats and Countermeasures* on MSDN:
http://msdn.microsoft.com/library/en-us/dnnetsec/html/THCMCh21.asp.

For more information about best practices for globalization, see "Best Practices for Developing World Ready Applications" in the *.NET Framework Developer's Guide* or "Best Practices for Globalization and Localization," both of which are available on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconbestpracticesforglobalapplicationdesign.asp

or:
http://sdn.microsoft.com/library/en-us/vsent7/html/vxconBestGlobalizationPractices.asp.

For ore information about the exception management architecture, see the *Exception Management Architecture Guide* on MSDN:
http://msdn.microsoft.com/library/en-us/dnbda/html/exceptdotnet.asp.

For more information about memory profiling using the .NET Common Language Runtime profiler, see "How To: Use CLR Profiler" in *Improving .Net Application Performance and Scalability* on MSDN:
http://sdn.microsoft.com/library/en-us/dnpag/html/scalenethowto13.asp.

For mre information on NUnit, and to download the tool, see the NUnit Web site:
http://ww.nunit.org.

For mre information about FxCop, see the FxCop Team Page on the .NET Framework Community Web site:
http://www.gotdotnet.com/team/fxcop/.

For more information about ACT, see "Microsoft Application Center Test 1.0, Visual Studio .NET Edition" on MSDN:
http://msdn.microsoft.com/library/en-us/dnbda/html/exceptdotnet.asp.

For more information about Visual Studio Analyzer see "Visual Studio Analyzer" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsavs70/html/veoriVisualStudioAnalyzerInBetaPreview.asp.

For more information about CLR Profiler, see "How To: Use CLR Profiler" in *Improving .NET Application Performance and Scalability* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto13.asp.

For more information about how to use EIF, see "How To: Use EIF" in *Improving .NET Application Performance and Scalability* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenethowto14.asp.

For more information about performance counters in .NET, see "Performance Counters" in the *.NET Framework General Reference* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/gngrfperformancecounters.asp.

For more information about how to create and use custom performance counters, see the following How Tos in the patterns & practices guide *Improving .NET Application Performance and Scalability* on MSDN (the links for the How Tos are in the left pane):
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenet.asp

- "How To: Monitor the ASP.NET Thread Pool Using Custom Counters"
- "How To: Time Managed Code Using QueryPerformanceCounter and QueryPerformanceFrequency"
- "How To: Use Custom Performance Counters from ASP.NET"

Appendix

A

References to Related Topics

In addition to the technical details involved in upgrade projects, there are some topics that are important but secondary that you should keep in mind as you perform your upgrade. The topics included here are beyond the scope of this guide, but they are important enough to be mentioned. Because the topics included here could not be directly included in this guide, links to available online information is provided.

Visual Basic 6.0 Resource Center

Although Visual Basic .NET represents the future for the language, there are still many developers that prefer to work in Visual Basic 6.0. The Microsoft Visual Basic 6.0 Resource Center is for developers in this category.

The resource center contains articles, links, and downloads explicitly targeted at Visual Basic 6.0 developers. You can find tutorials, discussions, downloadable components, and more.

The content available is not limited to pure Visual Basic 6.0 applications and components. The site also contains information about how to use Visual Basic .NET components and features in Visual Basic 6.0. It includes details about how to use your Visual Basic 6.0 components in Visual Basic .NET. It even provides resources to help you move from Visual Basic 6.0 to Visual Basic .NET.

Coding Standards

Code is much easier to understand and maintain when a set of standards are defined and consistently applied to all code. Such standards typically include:

- Naming conventions for variables, subroutines, and other programmer-defined identifiers.
- Indentation style.
- Module, class, and program structure.
- Comment and documentation styles.

Typically, a company will have its own standards defined and distributed to developers. If your company does not currently have standards in place, you should give careful consideration to defining and using them. For information about defining coding standards for Visual Basic .NET code, see "Program Structure and Code Conventions" in *Visual Basic Language Concepts* on MSDN.

Choosing File I/O Options

Chapter 11, "Upgrading String and File Operations," provided detailed information about how to upgrade flat file access. Different options, including using the Microsoft Visual Basic Compatibility library and the use of file streams, were discussed.

It may be difficult to choose between the different options. A discussion of when to choose streams over the compatibility library is beyond the scope of this guide. For more information, see "Choosing Among File I/O Options in Visual Basic .NET" in *Visual Basic Language Concepts* on MSDN.

More Information

The Microsoft Visual Basic 6.0 Resource Center, VBRun, is on MSDN: *http://msdn.microsoft.com/vbrun/default.aspx*.

For information about defining coding standards for Visual Basic .NET code, see "Program Structure and Code Conventions" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vbconProgrammingGuidelinesOverview.asp*.

For more information about choosing I/O options, see "Choosing Among File I/O Options in Visual Basic .NET" on MSDN: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcn7/html/vatskchoosingamongfileiooptionsinvisualbasicnet.asp*.

Appendix

B

Application Blocks, Frameworks, and Other Development Aids

Microsoft Visual Basic .NET and the Microsoft .NET Framework offer numerous new features that help developers make new applications or enhance applications that already exist. This appendix explores some of the application blocks, frameworks, and other development aids that are built into Visual Basic .NET.

Using Visual Basic .NET Application Blocks

Microsoft offers several application blocks for the .NET Framework. Application blocks encapsulate certain key areas by adding a layer over the .NET Framework. They also aid faster application development and promote certain favored practices. Application blocks encapsulate an optimal way of doing things in the .NET Framework. You can reuse the application blocks to reduce the amount of redundant code.

Application blocks are C# and Visual Basic .NET classes that are distributed as Visual Studio projects. You can use them in any .NET application, including ASP.NET Web applications. Application blocks are useful and powerful tools that include source code and sample applications. You can use them to make applications more maintainable, scaleable, and efficient.

The following list includes the available application blocks and toolkits:

- **Caching Application Block**. The Caching Application Block is a component of Enterprise Library that provides a flexible and extensible caching mechanism. You can use it in client-side and server-side .NET development projects.
- **Configuration Application Block**. The Configuration Application Block is a component of Enterprise Library that makes it easier to build applications that

read and write configuration information from a variety of data sources. This application block also includes a graphical tool to assist you in editing and viewing application configuration data.

- **Cryptography Application Block**. The Cryptography Application Block is a component of Enterprise Library that makes it easier to include cryptographic functionality in .NET applications. This application block provides a simple interface to the Data Protection API (DPAPI) and symmetric encryption and hashing. It also uses the Enterprise Library configuration tool to simplify key management.
- **Data Access Application Block**. The Data Access Application Block is a component of Enterprise Library that reduces the amount of custom code that you need to create, test, and maintain when you build data access layers in .NET applications.
- **Exception Handling Application Block**. The Exception Handling Application Block is a component of Enterprise Library that makes it easier to implement consistent exception handling policies at logical tiers in an application. You can configure exception policies to perform tasks, such as logging exceptions, and wrapping or replacing exception types.
- **Guidance Automation Toolkit**. The Guidance Automation Toolkit is an extension to Visual Studio 2005 that allows architects to author rich, integrated user experiences for reusable assets, including frameworks, components and patterns. The resulting guidance packages are composed of templates, wizards and recipes that help developers build solutions in a way that is consistent with the architecture guidance.
- **Logging and Instrumentation Application Block**. The Logging and Instrumentation Application Block is a component of Enterprise Library that allows developers to instrument their applications with logging and tracing calls. Log and trace messages can be routed to a choice of data sinks, including the Event Log, text files or Windows Management Instrumentation (WMI). This application block is the successor of the Enterprise Instrumentation Framework (EIF).
- **Security Application Block**. The Security Application Block is a component of Enterprise Library that builds on the capabilities of the Microsoft .NET Framework to help you perform authentication and authorization, check role membership, and access profile information.
- **Smart Client Offline Application Block**. The Offline Application Block enables the smart client user to enjoy a seamless experience even when he or she works offline. It provides approaches to detect the presence or absence of a network. It also caches the required data so that the application can function while it is offline, and synchronizes the client application state and the data with the server when the application goes online again.

- **Updater Application Block Version 2.0**. The Updater Application Block is a .NET Framework component that you can use to detect, download, and apply client application updates that are deployed in a central location. By using the Updater Application Block, you can keep smart client applications up to date with little or without any user intervention. You can also extend the Updater Application Block to use custom classes for downloading files, and for performing post-deployment configuration tasks.
- **User Interface Process Application Block - Version 2.0**. The User Interface Process Application Block provides a simple, yet extensible framework for developing user interface processes. It abstracts the control flow and state management out of the user interface layer and into a user interface process layer.

For more information about the application blocks and to download them, see "patterns & practices Guidance: Application Blocks" on MSDN.

Building "My" Facades

The Visual Basic .NET **My** feature provides access to information and default object instances that relate to the application and its run-time environment. This information is organized in a format that is discoverable through IntelliSense and is logically delineated according to its use.

Note: The **My** facades feature is available only in Visual Studio 2005.

The top-level members of **My** are exposed as objects. Each object behaves similarly to a namespace or a class with shared members, and it exposes a set of related members.

The central objects that provide access to information and commonly used functionality are:

- **My.Application**. This object provides properties, methods, and events that relate to the current application or DLL. This object is available only in Console and Windows Forms applications.
- **My.Computer**. This object provides properties for manipulating computer components, such as audio, the clock, the keyboard, the file system, and so on. This object is available in all types of Visual Basic .NET applications.
- **My.User**. This object provides access to information about the current user. It is available in all types of Visual Basic .NET applications.

The following code example shows how you can retrieve information by using **My**.

```
…
' Displays a message box that shows the full command line for the application.
MsgBox(My.Application.CommandLineArgs)

' Get a list of subfolders in a folder
My.Computer.FileSystem.GetDirectories _
    (My.Computer.FileSystem.SpecialDirectories.MyDocuments, True,"*Logs*")
…
```

The **My**-feature objects also provide the following functionality:

1. **My.Forms**. This object provides properties that you can use to access an instance of each Windows form that is declared in the current project. It is available only in Windows Forms applications.
2. **My.Log**. This object provides a property and methods that you can use to write event and exception information to the application's log listeners. It is available only in Web applications.
3. **My.Request**. This object gets the **System.Web.HttpRequest** object for the requested page. It is available only in Web applications.
4. **My.Response**. This object gets the **System.Web.HttpResponse** object that is associated with the **System.Web.UI.Page**. You can use this object to send HTTP response data to a client. It also contains information about that response. It is available only in Web applications.
5. **My.Resources**. This object provides properties and classes that you can use to access the application's resources. It is not available in Web applications.
6. **My.Settings**. This object provides properties and methods for accessing the application's settings. It is not available in Web applications.
7. **My.WebServices**. This object provides properties that you can use to create and access a single instance of each XML Web service that is referenced by the current project. It is not available in Windows Forms applications.

For more information about the **My** feature, see "Development with My" on MSDN.

Building Visual Studio .NET Snippets

In Visual Studio 2005, Visual Basic .NET includes a code library of approximately five hundred pieces of IntelliSense code snippets. These code snippets are ready for you to insert into your application. Each snippet performs a complete programming task, such as creating a custom exception, sending an e-mail message, or drawing a circle. You can insert IntelliSence code snippets into the source code with a few mouse clicks.

In addition to using the predefined code snippets, you can also create user-defined snippets that suit your business needs. You can add these snippets to the library, and then use them when you want.

Increase your productivity by using the IntelliSense code snippets to:

- Reduce the amount of time that you spend looking for code samples.
- Reduce the time that is required to learn to use unfamiliar features.
- Reuse code that you have already written.

Increase your productivity by using the code library to:

- Insert one of 500 prewritten snippets, or *tasks*, into the Code Editor.
- Create new tasks that you can reuse in your projects.
- Create new tasks to share with your workgroup and colleagues.
- Edit the tasks.
- Download more tasks from third parties.

These code blocks are available throughout Visual Studio. You can add them through the shortcut menu of the Code Editor, or you can drag the XML code files from Windows Explorer to your source-code file. For more information about Visual Studio code snippets, see "Introduction to IntelliSense Code Snippets" on MSDN.

Mobile Applications and the .NET Compact Framework

The growing popularity of mobile computing devices, such as Personal Digital Assistants (PDAs) and handsets, increases the need for developers to write applications that run across a range of mobile devices. This section provides an overview of Microsoft's mobile technologies and introduces the .NET Compact Framework as an important environment to create mobile applications.

Microsoft Mobile Technology Overview

The market for mobile devices has expanded at a high rate over the last few years. This trend is expected to continue. Industry analysts predict that by 2008 there will be over 100 million converged devices and 2 billion mobile phone subscribers[1]. These statistics motivate businesses to create new applications for mobile devices and to adapt new versions of existing desktop applications to the mobile market.

Microsoft offers an extensive set of technologies that are aimed at mobile computing. You can integrate and adapt these technologies to provide innovative solutions for the mobile market. The following list includes the key technologies that are part of the Microsoft Mobile effort[2]:

- **Windows Mobile® 5.0**. This development kit offers a consistent platform for different devices. It includes tools, a common set of APIs, uniform data services,

common installers, and application security models for developing mobile applications.

- **Windows CE .NET**. This is a full-featured operating system that is specially tailored for mobile devices.
- **The .NET Compact Framework**. This is derived from the .NET Framework and provides a managed code infrastructure and a general class library that provide the most common functionality that is necessary to develop mobile applications for devices, such as personal digital assistants (PDAs) and mobile phones.
- **SQL Server CE**. This is a compact relational database that is used to rapidly develop applications that extend enterprise data management facilities to mobile devices. It has a development model and an API that are consistent with SQL Server.
- **ASP.NET Mobile controls**. This technology, formerly known as Microsoft Mobile Internet Toolkit (MMIT), extends the functionality of the .NET Framework and Visual Studio .NET to build mobile Web applications by enabling ASP.NET to deliver markup content to a wide variety of mobile devices.
- **Windows XP Embedded**. Windows XP Embedded is the successor to Windows NT Embedded 4.0. Based on the same binary files as Windows XP Professional, Windows XP Embedded enables you to rapidly develop reliable and full-featured connected devices.
- **Microsoft Windows XP Tablet PC Edition**. Microsoft Windows XP Tablet PC Edition operating system runs on the Tablet PC, which is considered to be the next step in the evolution of the PC. A Tablet PC provides all the performance and features of today's notebook computer in an ultra-light form, including the ability to run full versions of standard Windows applications.

Overview of the .NET Compact Framework

The .NET Compact Framework is a redesigned version of the .NET Framework that has been scaled down and optimized for resource-constrained devices. It provides a subset of the features and classes that are available in the full .NET Framework. The most important features of the .NET Compact Framework are its rich class library, automatic memory management, and CPU independence.

The .NET Compact Framework allows developers to produce applications that display, gather, process, and forward information. The data used by these applications can be local, remote, or a combination of both.

Mobile application development is highly simplified by the .NET Compact Framework. It provides a uniform infrastructure that allows developers to create applications in a way that is similar to standard Windows applications and deploys them in different types of devices. This allows Visual Studio .NET developers to port mobile

applications to a wide variety of devices. These applications can be built with Visual C# .NET or Visual Basic .NET.

The .NET Compact Framework has two main components: the common language runtime (CLR) and the .NET Compact Framework class library. The CLR is responsible for managing the execution of code and provides core services, such as memory, thread, and safety management. The .NET Compact Framework is a collection of classes that are used as the foundation for new applications.

The next few subsections highlight a few of the features of this framework. For more detailed information about the .NET Compact Framework, see ".NET Compact Framework" in the Smart Client Developer Center on MSDN.

Included Components

The .NET Compact Framework includes a subset of the classes that are defined in the **System.Windows.Forms** and **System.Drawing** namespaces. These classes allow developers to rapidly build user interfaces for mobile applications.

It is important to note that when you create a mobile version of a desktop application, some of the user interface behavior is changed and restricted. However, when you upgrade applications that were created in previous mobile application development environments, such as eMbedded Visual Basic, most of the original functionality is supported and improved with the new .NET Compact Framework.

The compact version of Windows Forms includes support for forms and most of the controls found in the .NET Framework. Third-party components, bitmaps, and menus can also be included. The following list presents the controls included in the .NET Compact Framework:

- Button
- CheckBox
- ComboBox
- Control
- ContextMenu
- Cursor/Cursors
- DataGrid
- DomainUpDown
- Form
- HScrollBar
- ImageList
- InputPanel
- Label
- ListBox

- ListView
- MainMenu
- Message Window (this is unique to the compact framework)
- NumericUpDown
- OpenFileDialog
- Panel
- PictureBox
- ProgressBar
- RadioButton
- SaveFileDialog
- StatusBar
- TabControl
- TabPage
- TextBox
- Timer
- ToolBar
- TrackBar
- TreeView
- VScrollBar

To adapt the .NET Framework to the size and performance restrictions that are present in mobile devices, some of the properties, methods, and events have been removed from the compact version of the framework. Additional functionality can be defined by inheriting from the existing controls.

The .NET Compact Framework provides namespaces that contain additional functionality that is equivalent to the .NET Framework or is specially designed for mobile devices. These namespaces provide support for data access, XML services, Web services, GDI, IrDA, and Bluetooth.

Removed Features

The .NET Framework was adapted to the resource limitations of mobile devices to create the .NET Compact Framework. When you port an application to a mobile platform, it is possible to extend the .NET Compact Framework to build additional functionality that was removed from the .NET Framework. When you extend the .NET Compact Framework classes according to the needs of specific applications, you optimize the use of the limited resources that are available in mobile devices. Additionally, you can use other components from different development communities in ported applications. For more information about third-party components, see the IntelliProg Web site.

The following list shows the primary features that are not included in the .NET Compact Framework:

- **Method overloads**. Many overloaded method definitions that are available in the .NET Framework were omitted to reduce the size of the .NET Compact Framework.
- **Controls**. Some controls were omitted from the .NET Compact Framework. The functionality of some of these controls is not typically needed on mobile devices. You can replace missing controls with system components that are accessible through the Windows CE API.
- **XML functionality**. The parsing and searching functionality that is provided by the **XPath** namespace and the Extensible Stylesheet Language Transformation (XSLT) components was omitted.
- **Database support**. The data access components were reduced and replaced by the SQL Server CE components. Unlike eMbedded Visual Basic, the .NET Compact Framework does not support access to the local data store (sometimes called CEDB or Pocket Access).
- **Binary serialization**. The **BinaryFormatter** and **SoapFormatter** classes were omitted.
- **Access to the Windows registry**. The **Microsoft.Win32.Registry** namespace was removed and must be replaced with the corresponding functionality in the Windows CE API.
- **Security**. Functionality for role-based security was removed from the .NET Compact Framework. Additionally, secure access verification to unmanaged code was omitted.
- **XML Web services**. Cookie support was omitted and cryptographic functionality has been limited.
- **Printing**. Support for printing was eliminated. You can use IR port communication and interaction with a desktop application as alternative ways to print data.
- **GDI+**. This functionality was omitted because Windows CE does not natively support GDI+.
- **Remoting**. This functionality is not included in the .NET Compact Framework.

Default Project Settings

When you create a new mobile application, or port one application from another mobile or desktop environment, it is important to know the default project settings that are defined for mobile Visual Basic .NET applications.

Visual Studio .NET provides a consistent and uniform development environment for building all types of applications. This includes mobile applications.

Within the Visual Studio .NET IDE, mobile Visual Basic .NET applications are almost indistinguishable from desktop Visual Basic .NET applications. The general project settings (accessible by selecting **Project \ Properties** in the main menu) are organized in the same way and they have similar values. The only new section that is included in mobile Visual Basic .NET project settings is the **Device** section (which you can access in the **Common Properties \ Device** tab in the Project Settings window). This new section indicates the target platform for your new or ported application, and it allows you to port your mobile application to different devices. It also allows you to test your application in different device emulators.

Another important difference between mobile and desktop applications is the group of references that is added by default to every new project. More references are included in desktop applications to provide the minimum assemblies necessary to make a basic application work. Because mobile applications are more limited in terms of available resources, fewer components are added by default to a new mobile application project. Also, fewer components are available in general. For a list of available components, see the previous subsection, "Included Components."

You can include additional references for mobile applications in your project.

► **To add references for mobile applications**

1. In Solution Explorer, right-click the **Reference** folder, and then click **Add Reference**. The **Add Reference** dialog box appears.
2. In the **Add Reference** dialog box, add the reference for the mobile application you want in your project.

Porting from eMbedded Visual Basic

Microsoft eMbedded Visual Basic is a reduced version of Visual Basic. Its design allows Visual Basic developers to apply their knowledge of the language toward building Windows CE-based applications. eMbedded Visual Basic is not supported by more recent mobile devices, such as Windows Mobile-based Smartphones or Pocket PC 2003-based devices. Instead, Microsoft recommends building mobile and embedded applications using the .NET Compact Framework for these devices. This includes porting eMbedded Visual Basic applications to Visual Basic .NET for the .NET Compact Framework.

Porting applications from eMbedded Visual Basic to Visual Basic .NET for the .NET Compact Framework can yield significant advantages, such as the following:

- **Better performance**. Visual Basic .NET code is compiled instead of interpreted like eMbedded Visual Basic code. This results in improved application performance.
- **Extensive class library**. The .NET Compact Framework offers classes for the most common tasks that are performed in mobile applications. You will find more components and classes here than in eMbedded Visual Basic.

- **Improved development tools**. Visual Studio .NET offers an unprecedented development environment with tools and facilities that increase developer productivity. The same IDE is used to develop both desktop and mobile applications, so developers do not have to be familiar with multiple IDEs to develop different types of applications.
- **Improved language**. Visual Basic .NET is a language that was redesigned to take advantage of the advancements in programming languages, such as object oriented programming, exception handling, strong typing, and more.

When porting applications that are based on other eMbedded languages, such as eMbedded Visual C++, more analysis and better understanding of the application is required. When high performance, a minimal working set, and low-level device control are a top priority, developers should continue using eMbedded Visual C++. When time-to-market and a consistent programming model are your primary concerns, the best choice is to use Visual Studio .NET and the .NET Compact Framework.

Even though no tool exists to automatically upgrade mobile applications, other tools can be used to accelerate the process of porting an eMbedded Visual Basic application to the .NET Compact Framework. Considering that the eMbedded Visual Basic language was derived from Visual Basic 6.0 and is comparable to VBScript (the language used by ASP pages), you can use the automatic Visual Basic upgrade tools that are discussed in this guide to upgrade fragments of mobile applications. Use the following general procedure as a guideline for this approach. You can apply it to each part of the application that you want to port.

► **To upgrade fragments of a mobile application**

1. Modify the application, or parts of it, to make it work in Visual Basic 6.0. Be aware that because eMbedded Visual Basic uses a subset of the Visual Basic features, the Visual Basic 6.0 version of the application uses a reduced subset of all the functionality that is normally available.
2. Upgrade the Visual Basic 6.0 application to Visual Basic .NET using the Visual Basic Upgrade Wizard.
3. Create a new Visual Basic .NET mobile application in Visual Studio .NET and include the Visual Basic .NET files that you obtained in the previous step.
4. Correct all of the pending upgrade issues and make the necessary adjustments to make the application work.
5. Test the new application.

For more information about manually porting eMbedded Visual Basic-based applications to Visual Basic .NET, see "Moving from eMbedded Visual Basic to Visual Basic .NET" on MSDN.

Creating a Mobile Version of a Desktop Application

When you decide to create a new mobile application that is based on a desktop application, it is necessary to port and review several aspects of the original application. These aspects range from the user interface, to data access, and even to low-level manipulation of the target device. This section provides an overview of the steps that are necessary to port a desktop application to a mobile application. It also discusses some of the tools that are available to port the application, and it details the procedures to facilitate the port.

The application that you want to port can be either of these two application types:

- **Mobile application**. If the target platform of the original application was a mobile device, the mobile environment constraints have already been taken into account. This facilitates the upgrade to the .NET Compact Framework because most of the work is concentrated in specific conversion issues. Otherwise, you have to redesign the application to make it suitable for use on a mobile device.
- **Desktop application**. If your intention is to create a mobile version of your desktop application, you should first perform a preliminary suitability study. More than likely, a considerable part of the original application's functionality will require deep modifications or will even need to be removed from the mobile version. In most cases this kind of mobile application works as a complement to a desktop application. It might include a subset of the original functionality that is sufficient for mobile computing purposes.

As in any Visual Basic porting project, the porting of an application to a mobile platform requires that you execute of a series of steps that begin with preparing the application and end in testing and debugging. For more information about the suggested porting procedure, see Chapter 5, "The Visual Basic Upgrade Process." Although the chapter discusses the upgrade process, the discussions apply equally to porting projects.

The initial steps that are necessary to port an application so that it will compile and run on a mobile platform are presented here. In-depth information about low-level upgrades, and the testing and debugging of a mobile application, is beyond the scope of this book, but pointers to more information about these processes is provided later in this chapter.

The following sections provide guidelines that you can use to port a Visual Basic .NET application to the .NET Compact Framework.

Porting the User Interface

The user interface that is a likely aspect of your application that will require significant change when you upgrade it to a mobile platform is the user interface. The components that are available to build visual interfaces for mobile devices are much smaller in comparison to those that are available for desktop applications.

The user interface of your application is likely to require significant change when you upgrade it to a mobile platform. The components that you can use to build visual interfaces for mobile devices are much smaller in size than those that are available for desktop applications. This implies that a mobile application will be visually different from its desktop counterpart. Its behavior will also likely be different because part of the application's functionality must be implemented using different visual elements. Yet, another visual difference is the size of the display on mobile devices. Considering the reduced size of Pocket PCs or Windows CE devices, you must review and adjust the user interface to efficiently use the available space. In general, the look and style of the mobile application will be different with respect to the original desktop application.

The Porting Process

The following process presents the steps that are necessary to port a Visual Basic .NET application form to work on a mobile platform. This process shows the typical adjustments that you need to make to convert the application's user interface and basic functionality.

Figure 1 shows an example of a form before it is converted.

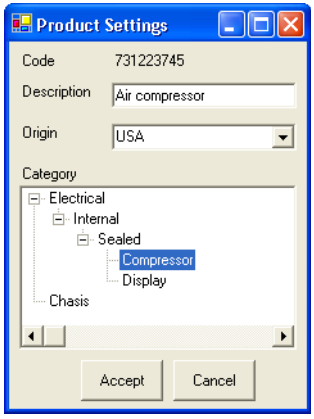

Figure 1

A sample Visual Basic .NET form to be ported for use on a mobile device

This Form is contained in a file named Form2.vb and presents some sample product settings that can be changed by the user and updated in a database. You can use the following steps to create an initial version of the application. The resulting form is executable on a mobile platform.

For this example, the original application files are located in the C:\MobileAppTest directory. You should replace this directory with the actual location of files on your system when it is appropriate to do so.

► **To convert the form to a compact framework version**

1. Copy the application that you want to port to a working directory. Some of the new files require modifications to make them work with the .NET Compact Framework, so it is best to work with a copy to preserve the original code. For this example, the original application files have been copied to the directory C:\MobileAppTest.
2. Open Visual Studio .NET.
3. Create a new Visual Basic .NET mobile application in Visual Studio .NET by following these steps:
   a. On the **File** menu, click **New**, and then click **Project**.
   b. In the **Templates** list, click **Smart Device Application**. This displays the **Smart Device Application Wizard** window. Use the wizard to select the target platform and the type of project you want to create. For this example, select **Windows CE** as the platform and **Windows Application** as the project type.
4. The new project appears in Visual Studio .NET. By default, it includes a new **Form** that is named **Form1**. You do not need **Form1** because the original application forms will be imported into the mobile project. Use the following steps to remove **Form1**:
   a. In Solution Explorer, right-click **Form1**, and then click **Delete**.
   b. In the **Delete Confirmation** dialog box, click **Yes** to remove **Form1**.
5. Add the file that you want to convert to the mobile application project. Follow these steps to convert the file:
   a. In Solution Explorer, right-click the project folder and on the pop-up menu, click **Add**, and then click **Add Existing Item**. The **Add Existing Item** dialog box appears.
   b. Browse to find the file that you want to convert, and then select the file. For this example, select the **Form2.vb** file in **C:\MobileAppTest**.
6. A list of tasks appears in the **Task List**. They include tasks that relate to problems that are caused by nonexistent properties, methods, and events. The general strategy to fix these problems is to select each of the tasks and examine the source code that it refers to. You can then either fix the code or remove it, depending on whether the property, method, or event is needed. Table 1 presents common problems and solutions that relate to user interface functionality.

Table 1: User Interface Issues and Solutions When Converting a .NET Form to a Compact Framework Form

Error Description	Solution
SuspendLayout is not a member of **SmartDeviceApplication1.Form2**	Remove the statement.
Name is not a member of **System.Windows.Forms.*SpecificControl***. Applies for all controls derived from **System.Windows.Forms.Control**	Remove the statement. If it is strictly necessary to identify dynamically created controls, you can keep a variable to refer to the desired control and to use in comparisons to find specific controls.
TabIndex is not a member of **System.Windows.Forms. SpecificControl**. Applies to all controls that are derived from **System.Windows.Forms.Control**	Remove the statement.
Overload resolution failed because there is not an accessible **New** to accept this number of arguments.	Change the corresponding control constructor call by using one of the available method overloads. Typically, you can replicate the original functionality by accessing the appropriate control methods and properties in the form **Load** event handler or other initialization code.
Method / Property is not a member of **System.Windows.Forms. SpecificControl**.	Control initialization included in the **InitializeComponent** method cannot access unsupported methods or properties. In most of the cases, you can replicate the functionality of unsupported members by accessing other available methods after the controls have been initialized.
ResumeLayout is not a member of **SmartDeviceApplication1.Form2**.	Remove the statement.

7. Change the project **Startup** object according to your needs. To do this, follow these steps:
   a. On the Visual Studio **Main** menu, click **Project**, and then click **SmartDeviceApplication1** properties.
   b. In the **Common Properties** folder select the **General** tab.
   c. In the **Startup** object field, select the name of the startup form or method. For this example, select **Form2**.

8. Correct the resource files that you want to use in the .NET Compact Framework. The .NET Compact Framework uses a different class to read the resources file. Correct the resource file by modifying the **Form2.resx** file to replace the class name with the appropriate .NET Compact Framework resource reader class. You can make the modifications with a text editor or with the Visual Studio .NET IDE. The original resource file is shown here.

```
<resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms,
    Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</
value>
</resheader>
```

   Make the following changes to the content of the file.

```
<resheader name="reader">
    <value>System.Windows.Forms.Design.CFResXResourceReader, System.CF.Design,
    Version=7.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a</
value>
</resheader>
```

9. Make an analogous change to the **Form2.resx** file to correct the resources writer class. The following code shows the original file.

```
<resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms, Ver-
sion=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
</resheader>
```

   Make the following changes to the file.

```
<resheader name="writer">
    <value>System.Windows.Forms.Design.CFResXResourceWriter, System.CF.Design,
Version=7.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a</value>
</resheader>
```

10. Because of other class differences between the .NET Framework and the .NET Compact Framework, it may be necessary to correct other types that are specified in the resources file. You must change these classes to the corresponding class in the .NET Compact Framework. For example, you have to modify the **System.Drawing.Size** class, which is frequently used in .resx files. The following code shows the original .resx file.

```
"System.Drawing.Size, System.Drawing, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
```

You must modify this line as is shown in the following code.

```
"System.Drawing.Size, System.CF.Drawing, Version=7.0.5000.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a"
```

11. Compile the application. To do this, from the **Main** menu, click **Build**, and then click **Build Solution**. Use the previous two steps to correct any other resource file errors.

You can now deploy and execute the application on the corresponding mobile device or an appropriate emulator. Figure 2 shows the converted form running on a Windows CE emulator.

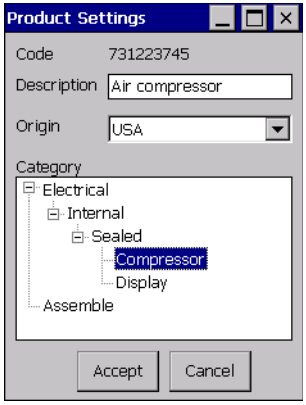

Figure 2
The ported application executing on a Windows CE emulator

Because the source and target languages are Visual Basic .NET, most of your application's source code should compile and run without problems. Most of the difficulties arise when the application uses unsupported library components, such as data access or GDI+. The following section provides suggestions to implement data access functionality. You can comment unsupported functionality in the application to produce a working base that you can later use as the starting point to redesign or re-implement the functionality.

Synchronizing with Server Applications

Integrate mobile applications with other applications by using one of the following patterns:

- **Device Data to Server Data**. With this pattern, the database on the mobile device is synchronized directly with the enterprise database server. This option is feasible when the data that you want to synchronize does not require additional verification or transformations. You can achieve this kind of synchronization with SQL Server CE .NET Remote Data Access and Merge Replication.

- **Device Logic to Server Logic**. With this pattern, a mobile application connects to an application on the server computer. Because most synchronization requires business logic, this is the most frequent option. You can achieve this kind of synchronization with Web Services.
- **Device Logic to Server Data**. With this pattern, a mobile application can connect directly to a data repository and obtain data as it is needed based on the application logic. You can achieve this kind of synchronization with the SQL Server .NET Data Provider.

When you port an eMbedded Visual Basic or Visual Basic .NET desktop application to the .NET Compact Framework you should be aware of the extended communication capabilities that are offered by the new platform and additional options that are offered by third parties.

When you implement synchronization functionality, one of the most common choices is to use SQL Server CE Merge Replication. This technology allows your application to synchronize data stored in a SQL Server CE subscription database with its publisher. Use this option when mobile devices are not connected to the enterprise network permanently and must perform synchronization when it is convenient. In most of the cases, you can meet these conditions when you use eMbedded Visual Basic applications.

The .NET Compact Framework **SqlCeReplication** class (which is available in the **System.Data.SqlServerCe** namespace) is the data provider component that corresponds to SQL Server CE. It allows the developer to programmatically activate merge replication with a published database. For more information about publishing databases, see "Replication with SQL Server 2000 Windows CE Edition" on MSDN.

The following example demonstrates the use of the **SqlCeReplication** class to synchronize data in a mobile application with a published database.

```
…
Sub SynchronizeDB()
    Dim rep As New SqlCeReplication( _
        "http://myserver/pubsmr/sscesa20.dll", _
        "CEDBTest", "ceDBtest", _
        "MYSERVER", _
        "pubs", _
        "pubs", _
        "Testing", _
        " Data Source=\My Documents\PubsMR.sdf")
    Try
        ' Check if the database file already exists
        If (System.IO.File.Exists("\My Documents\PubsMR.sdf")) Then
            ' Add a new subscripton and create the local database file
            rep.AddSubscription(AddOption.CreateDatabase)
        End If
        rep.Synchronize()
    Catch ex As SqlCeException
```

```
            MessageBox.Show(ex.Message)
        Finally
            rep.Dispose()
            MessageBox.Show("Replication complete")
        End Try
    End Sub
    …
```

The **Synchronize** method of the **SqlCeReplication** class retrieves the initial snapshot of data for your local SQL Server CE database if it is the first time the method is invoked. After the data is initialized and the subscription is set up, new invocations of **Synchronize** send and receive only the changes that have been applied to the data.

After a mobile database and subscription are created you can use the classes that are contained in the **System.Data.SqlServerCe** namespace to update, insert, and delete data. All of these changes are applied the next time the **Synchronize** method is invoked.

If your eMbedded Visual Basic application is targeted at the Pocket PC platform, it is important to consider third-party components that allow access to the Pocket Access database that is part of the Pocket PC platform. Applications that are written in eMbedded Visual Basic are able to access the Pocket Access database by using a Windows CE implementation of ADO (ADOCE). However, this COM component is not easily accessible from the .NET Compact Framework and third-party components must be used as an alternative. For more information about additional data access components, see "Pocket Access and the .NET Compact Framework" on MSDN.

More Information

For more information about the application blocks and to download them, see "patterns & practices Guidance: Application Blocks" on MSDN: *http://msdn.microsoft.com/practices/guidetype/AppBlocks/default.aspx.*

For more information about the **My** feature, see "Development with My" on MSDN: *http://msdn2.microsoft.com/library/5btzf5yk(en-us,vs.80).aspx.*

For more information about Visual Studio code snippets, see "Introduction to IntelliSense Code Snippets" on MSDN: *http://msdn2.microsoft.com/library/18yz4be4(en-us,vs.80).aspx.*

For more information about the .NET Compact Framework, see ".NET Compact Framework" in the Smart Client Developer Center on MSDN: *http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/.*

For more information about third-party components, see the IntelliProg Web site: *http://www.intelliprog.com/.*

For more information about manually porting eMbedded Visual Basic-based applications to Visual Basic .NET, see "Moving from eMbedded Visual Basic to Visual Basic .NET" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnppcgen/html/fromemb.asp.

For more information about publishing databases, see "Replication with SQL Server 2000 Windows CE Edition" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/replsql/replimpl_6pet.asp.

For more information about additional data access components, see "Pocket Access and the .NET Compact Framework" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnroad/html/road10222003.asp.

Footnotes

1. Enterprise Wireless Email Market, 2004-2008
   http://www.gii.co.jp/english/rd25765_wireless_email.html
2. Understanding Mobile & Embedded
   http://msdn.microsoft.com/mobility/understanding/

Appendix

C

Introduction to Upgrading ASP

Web developers have used Active Server Pages (ASP) technology for several years to build Web sites and Web applications. The Microsoft .NET Framework provides the infrastructure for a new way for developers to build sophisticated Web applications. This new infrastructure is referred to as ASP.NET.

ASP.NET is not simply a new version of ASP; it has been entirely re-architected to provide a robust infrastructure for building reliable and scalable Web applications based on the .NET Framework.

When porting an ASP application to ASP.NET, you will need to decide how much time you want to spend incorporating the new features of ASP.NET into the existing ASP application. One option is to begin by converting the ASP pages to ASP.NET automatically with the ASP to ASP.NET Migration Assistant, and then to complete the work by applying some manual changes. The other option is to perform a full re-architecture of the application and take advantage of many of the new features of .NET including ASP.NET Web controls, Microsoft ADO.NET, the Microsoft .NET Framework classes, and more. Although the latter option can make your ASP.NET pages more readable, maintainable, and feature-rich, it requires more time and effort than simply using the migration assistant.

Converting existing ASP pages to ASP.NET pages is not as simple as changing the file name extension from .asp to .aspx. Some important areas to consider when upgrading your ASP applications include:

- **Core API changes**. ASP's core APIs consist of a few intrinsic objects (such as **Request**, **Response**, and **Server**) and their associated methods. With the exception of a few changes, these APIs continue to function correctly under ASP.NET.

- **Structural changes**. Structural changes affect the layout and coding style of Active Server Pages. You need to be aware of several of these changes and address them to ensure that your code will work in ASP.NET.
- **Visual Basic language changes**. There are some changes between Visual Basic Scripting Edition VBScript and Microsoft Visual Basic .NET script to take into account during the upgrade.
- **COM-related changes**. COM has not changed with the introduction of the .NET Framework and ASP.NET; however, this does not mean that you do not need to worry about COM objects and how they behave when you are using them from ASP.NET. There are a few fundamental issues to consider.
- **Application configuration changes**. In ASP, all Web application configuration information is stored in the system registry and the IIS metabase. In ASP.NET, each application has its own configuration file.
- **State management issues**. You can continue to use the **Session** or **Application** intrinsic object to store state information in ASP.NET. As an added benefit, ASP.NET offers a few more options for storing state.
- **Data access**. Another key area you may need to focus on for your upgrade efforts is data access. ADO.NET offers a powerful new way to store and access your data.

ASP and ASP.NET can both coexist on the same Web server; that is, a Web site or Web application within a site can contain both ASP.NET pages and ASP pages. This can be very beneficial if you have to move a large, rapidly changing site to ASP.NET one piece at a time. Additionally, if you are going to put forth the effort to move to ASP.NET as a long-term investment, you may want to take advantage of the opportunity to make as many architectural and design improvements as you can. For more information, see "Migrating to ASP.NET: Key Considerations" on MSDN.

Note: ASP.NET is supported in the following operating systems: Microsoft Windows 2000, Microsoft Windows 2003, and Microsoft Windows XP Professional. ASP.NET applications are not supported in Microsoft Windows NT. For more information, see "System Requirements for Visual Studio .NET 2003" in the Microsoft Visual Studio Developer Center on MSDN.

Because both ASP and ASP.NET pages can be accessed from the same Web server, you do not have to convert your existing ASP pages to ASP.NET. However, converting to ASP.NET offers many advantages, some of which include:

- **Increased performance**. Microsoft tests have shown that ASP.NET applications can handle two to three times as many requests per second as classic ASP applications.
- **Increased stability**. Processes are closely monitored and managed by the ASP.NET runtime. If a process leaks, deadlocks, or otherwise "misbehaves," the

runtime can create a new process in its place, which helps keep your application constantly available to handle requests.

- **Increased developer productivity**. New features in ASP.NET, such as server controls and event handling, help developers build applications more rapidly and in fewer lines of code. It is also easier than ever to separate code from HTML content.
- **Ease of deployment**. The installation of ASP.NET applications is dramatically simplified with "no touch" deployment. Applications can be deployed just by copying files to the server; they can also be updated without restarting the Web server. When updating a DLL, ASP.NET will automatically detect the change and start using the new component.
- **Improved security**. ASP.NET works in harmony with the .NET Framework and Microsoft Internet Information Services (IIS) to cooperate in the overall Web application security. The main ASP.NET security functions, authentication and authorization, allow applications to verify the identity of the users and manage permissions given an authenticated identity. Additionally, ASP.NET has access to all the built-in security features of the .NET Framework, including code access security, role-based security, cryptographic services, and security policy management.

For more information, see "Converting ASP to ASP.NET" on MSDN.

This appendix provides an introductory overview of the process for upgrading from ASP to ASP.NET with the help of the ASP to ASP.NET Migration Assistant. In particular, it discusses aspects such as preparation, automatic upgrade, manual changes, testing, and common issues and their resolutions. A full discussion of the complete process for upgrading ASP to ASP.NET is beyond the scope of this guide, but the information provided here should help get you started.

Process Overview

The steps involved in an upgrade project can be divided into three phases:

1. Preparing the application
2. Upgrading the application
3. Testing and debugging the upgraded application

The execution of these task groups is usually done by different teams, each with different skills and experience. The tasks included in each group can be executed in parallel to some degree, depending on the availability and specialization of the required resources. The task groups do not define a linear order of execution; there may be groups of tasks that are simultaneously executed for different parts of the application.

The next sections summarize the upgrade procedure explained in this chapter and list the inputs and outputs for each.

Preparing the Application

1. Preparing the development environment:

   Input: Software development installers, third party component installers.

   Output: A fully configured application development environment that corresponds to the application to be upgraded.
2. Checking Internet Information Services (IIS) and its integration with the .NET Framework:

   Input: Internet Information installers, Visual Studio .NET installers.

   Output: IIS must be running normally and be completely integrated with Microsoft .NET Framework for ASP.NET to work.
3. Obtaining the application resource inventory:

   Input: Available information about the specifications and design of the application.

   Output: Catalog of documentation useful for upgrade purposes.
4. Verifying compilation:

   Input: The original ASP application in a correctly configured system.

   Output: System where the original application can be compiled, debugged and executed.
5. Defining the project upgrade order:

   Input: Original source code.

   Output: An analysis of the application component dependencies based on analysis tools that can be used to plan the upgrade order of the different components.
6. Preparing the upgrade tool:

   Input: Upgrade wizard installer, installers for the analysis tools.

   Output: A fully configured system where the original application can run.

Upgrading the Application

1. Executing the ASP to ASP.NET Migration Assistant:

   Input: Original ASP pages and migration assistant executable file.

   Output: Initial upgraded application code base in ASP .NET. This code base will likely contain upgrade issues that will later need to be addressed.
2. Verifying the progress of the upgrade:

   This is a control step that verifies the correct execution of the migration assistant.

3. Completing the conversion with manual changes:

   Input: Initial upgraded application code base in ASP.NET.

   Output: An upgraded ASP .NET application that can be compiled.

Testing and Debugging the Upgraded Application

1. Executing the original test cases:

   Input: An upgraded ASP.NET application that can be compiled.

   Output: List of broken test cases and run-time bugs detected in the application.

2. Fixing run-time errors:

   Input: An upgraded ASP.NET application that can be compiled.

   Output: An upgraded ASP.NET application that can be correctly run.

The rest of this chapter will explore each of these steps in detail.

Understanding the ASP to ASP.NET Migration Assistant

The ASP to ASP.NET Migration Assistant is designed to help you convert ASP pages and applications to ASP.NET. It does not automate the complete process, but it speeds up your project by automating some of the steps required for upgrade. The migration assistant is not related to the Visual Basic 6.0 Upgrade Wizard, and it is not packaged with Visual Studio .NET; however, it is freely available. To download the ASP to ASP.NET Migration Assistant, see the Microsoft ASP.NET Developer Center on MSDN.

Tasks Performed by the Migration Assistant

When you upgrade an ASP project with the ASP to ASP.NET Migration Assistant, the tool performs three general tasks:

- The tool automatically generates the necessary ASP.NET files, using file names that end with the .aspx file name extension. At the same time, the tool inspects any ASP or HTML files in the project that include references to the file or files being upgraded and updates them so that they refer to the newly created files instead. There are several places in a file where a reference to another file can be found: in **include** directives, in code surrounded by **<script>** tags, in **href** attributes, in **Response.Redirect** and **Application.execute** instructions, and in the **action** attribute of **<form>** tags.

  ASP allows files included using the **<!— #include ... —>** and **<script src="***filename***">** tags to contain HTML code, scripting code, or a mix of both. The migration assistant updates all of these files to use the ASP.NET syntax, regardless of their extensions, and resolves any variables and functions declared inside

these files that are being used by the ASP file that contains the reference. However, in ASP.NET you can only use **<script src="***filename***">** to import files containing pure script code. If there is any HTML code or script code (enclosed in **<% … %>)** in these files, the compiler raises an error.

- VBScript does not allow type declaration, but ASP.NET requires it, so the tool inspects the code and analyzes the way variables are used to infer their data types in simple cases, such as the creation of objects by means of the **CreateObject()** method, and includes the corresponding declarations. This increases the quality of the code generated by the migration assistant and eases the expansion of the default properties.
- ASP allows you to declare subroutines and variables inside the render tags (**<% … %>**) and execute statements in server scripts, but ASP.NET requires that all declarations be made within script tags (**<script** … **>** … **</script>**) and that statement execution be restricted to render tags. Therefore, the tool will relocate VBScript code to the correct code blocks, as follows:
  - Functions and subroutines declared inside render tags **<%** … **%>** are moved into a global **<script>** tag with its **language** attribute is set to "VBScript" and **runat** attribute set to "server".
  - Statements and comments located within **<script runat="server">** tags are moved into a render tag.
  - Variable and constant declarations in render tags are moved to a **<script>** tag at the beginning of the file, as are all variables that are not declared and used in the file.

Limitations of the Migration Assistant

There are some areas in which the migration assistant cannot upgrade your ASP code. The limitations include:

- The migration assistant will not convert the security model, an important component of ASP projects. It will be your responsibility to re-architect the application to take advantage of the security features of ASP.NET and IIS. For more information about ASP.NET security, see "ASP.NET Web Application Security" in the *.NET Framework Developer's Guide* on MSDN.
- ASP.NET does not allow the use of multiple languages on a single page for server-side processing. The ASP to ASP.NET Migration Assistant will not convert or analyze any server-side code written in scripting languages other than VBScript, and it will add an error notice to your converted files before each occurrence of server-side code in other languages.

Note: This limitation does not affect client-side code written in multiple languages. The migration assistant can convert valid client-side code regardless of language.

- The ASP to ASP.NET Migration Assistant will not modify complex rendering functions. This affects the Visual Studio .NET IDE's ability to render pages in Designer view; it will produce compilation errors. This limitation does not affect simpler rendering functions.

Preparing the Application

Upgrading an ASP application to ASP.NET can be a complex and error-prone process without proper preparation, even with the ASP to ASP.NET Migration Assistant. Errors introduced in the initial stages of the process can easily propagate and consume an extended amount of work, and some parts of the application may need to be upgraded again, possibly delaying other processes involved in the upgrade. By taking certain steps to prepare your ASP application for upgrade before using the ASP to ASP.NET Migration Assistant to upgrade it, you can save a considerable amount of work in a number of areas, including management, development, and testing.

Your first step should be to prepare the related components of your system, such as Internet Information Services (IIS) and any third-party components, for upgrade. It is usually inexpensive to prepare these components, determine the implications they have for the rest of the upgrade, and correct any problems that arise. After these external aspects are prepared, you can concentrate on the internal aspects of the application, such as source code, application compilation, and common upgrade issues.

Preparing the Environment

If possible, execute the upgrade process using a computer that has been configured similarly to the development environment that was used to create the original application. This will facilitate analysis of the original application and make it easier to execute the initial preparation tests and identify dependencies and other issues.

There are two main environment aspect types that affect the upgrade process. The first of these affects the speed of the upgrade process; this includes computer resources. The second type affects the normal execution and termination of the migration assistant; an example is when external dependencies are not present on the computer where the process is executed, which results in exceptions during the execution of the migration assistant.

System Resources

The migration assistant works by loading all of the application's VBScript code into memory, converting it, and then creating all of the necessary .aspx files. Therefore, it can place significant demands on system memory and processor resources for storing language structures and executing transformation rules.

It is recommended that the migration assistant be run on a computer with at least 512 MB of system RAM. If you are upgrading a complex ASP application, which has more than 100,000 lines of code or that includes a large amount of code that will have to be moved between render tags and **<script>** tags, a minimum of 1 GB of RAM is recommended. On average, an application with these characteristics can take from three to five hours to process. You may also want to separate the application into smaller components to upgrade more manageable pieces.

The system's hard drive should have enough free space to store at least three times the size of the original application to accommodate the resulting ASP.NET files and all of the temporary files that will be generated. Ensure that the developer performing the upgrade has the appropriate user rights for the file locations that will be used.

Third-Party Components

Many ASP applications use third-party components in some way. Applications can explicitly reference external DLLs through the project settings and can dynamically create objects based on the components currently installed by using instructions such as **CreateObject**.

When upgrading applications that depend on external components, you must ensure that they are installed and available on the computer you are using for the upgrade. If the ASP to ASP.NET Migration Assistant cannot access a component, it generates an issue in the upgraded code in all instructions where the component is used. For example, if your application references the TRE component and the component is not installed on the computer you are using to upgrade the application, the migration assistant will add the following note to the ASP.NET code.

```
' UPGRADE_NOTE: The 'TRE.DLL' object is not registered in the
' migration machine.
obj = CreateObject("TRE.DLL")
```

Note: Some trial versions of third-party components display a license information dialog box every time the component is instantiated. This dialog box will suspend the execution of the host application, which is the ASP to ASP.NET Migration Assistant in this case. If you are using a trial version of any components, pay attention to the messages shown by the tool in response to any dialog boxes that display during the upgrade process.

As part of the application preparation, you should identify any complex and unusual components and perform some limited testing. Create reduced-size applications to instantiate and access the core functionality of these components. After these applications are written, run the migration assistant and evaluate the results, paying special attention to design time and run time visualization and behavior of the upgraded components.

IIS and Virtual Directories

To ensure that IIS can correctly create the virtual directories you will use to run your new ASP.NET applications, verify that IIS 5.0 or later and the Microsoft FrontPage® Server Extensions are installed on the computer that you will be using for the upgrade process. Testing your ASP applications before upgrading helps you confirm that the computer meets all of your application's requirements.

Tools

The ASP to ASP.NET Migration Assistant is an add-in to Microsoft Visual Studio .NET, so ensure that Visual Studio is installed on the computer you will be using for upgrade.

Additionally, the migration assistant uses a different version of the Visual Basic upgrade tool, named Visual Basic Upgrade Wizard Companion, which parses and converts all VBScript code that is used in your ASP pages. For more information about Visual Basic Upgrade Wizard Companion, go to the ArtinSoft Web site.

Preparing Your Code for the Migration Assistant

There are some actions that you can take to modify your code to help the migration assistant improve its automatic conversion and reduce the number of manual changes you have to make after upgrade. The most important of these changes are described in the next sections.

Register All Referenced COM Components

The first step is to analyze your code to identify all the COM components that your application uses. After compiling the list, ensure that all of the components are correctly installed and registered on the computer that you will be using for the upgrade, and resolve any problems with licenses or permissions.

Verify Compilation

It is very important to verify that the code is syntactically correct. If there are syntax errors in your files, the migration assistant might encounter problems when it tries to parse the code, and you may get some unexpected results.

Reviewing your code can also help you ensure that you have all the files your project needs. Before converting an ASP project, the migration assistant analyzes all the code and uses the information it gathers to determine how to proceed with the upgrade. If there are some missing files, it may negatively affect some of the decisions the tool makes.

Some common mistakes to look for include:

- **Lost files**. The **include** instruction is a very useful feature that allows ASP developers to reference other files using a virtual or physical path. One mistake people frequently make when upgrading to ASP.NET is failing to copy these referenced files into the proper locations on the computer being used for upgrade. For example, assume that your ASP page includes the file Variables.asp, as shown here.

```
<!– #Include Virtual="http://ServerName/myVirtualRoot/Lib/Variables.asp" –>
```

  If **Variables.asp** is not copied into the virtual directory http://ServerName/myVirtualRoot/Lib/ before you upgrade the ASP page, the migration assistant will add an upgrade note to the page before the **include** instruction, as shown here.

```
<% 'UPGRADE_NOTE: The
    'http://ServerName/myVirtualRoot/Lib/Variables.asp' file was
    not found in the migration directory. Copy this link in your
    browser for more:
    ms-help://MS.MSDNVS/aspcon/html/aspup1003.htm %>
<!– #Include Virtual="http://ServerName/myVirtualRoot/Lib/Variables.asp" –>
```

  To resolve this, copy the referenced file into the correct virtual directory or change the path to the valid local path name or Uniform Resource Identifier (URI) where the file actually resides, and delete the upgrade note.

- **Incorrect HTML tags**. If there are any invalid HTML tags in a page, the migration assistant will be unable to parse it fully and will generate an upgrade note. This can happen if the closing quotation marks have been inadvertently omitted from an instruction tag, as shown here.

```
<!–#include file="./Include/Table.asp –>
```

  When this happens, the migration assistant cannot identify and analyze all of the page's included files, which can cause problems. If you apply the migration assistant to a page that contains this tag, you should obtain the following note.

```
<% 'UPGRADE_NOTE: '#INCLUDE' tag is malformed. Copy this link in your browser
for
    more: ms-help://MS.MSDNVS/aspcon/html/aspup1004.htm %>
<!–#include file="./Include/Table.asp –>
```

  To resolve this, fix the HTML tag and remove the upgrade note.

```
<!–#include file="./Include/Table.asp" –>
```

After fixing the file, run the migration assistant again so that it will correctly recognize all the included files.

- **Unregistered COM objects**. When you convert files with the migration assistant, be sure to correctly install all of the referenced COM objects on the computer where you will be performing the conversion. When the migration assistant cannot access a COM object referenced by an ASP file it is upgrading, it will generate an upgrade note. For example, consider an ASP page that contains the following instruction.

```
<%
     Set myObject = CreateObject("MyObjectCOM.MyClass")
%>
```

If the **MyObjectCOM** component is not registered on the computer when you run the migration assistant, it will generate the following note.

```
<%
    ' UPGRADE_NOTE: The 'MyObjectCOM' object is not registered in the migration
        machine. Copy this link in your browser for more:
        ms-help://MS.MSDNVS/aspcon/html/aspup1016.htm
    myObject = New MyObjectCOM.MyClass
%>
```

To prevent run-time errors in your ASP.NET page, correctly install the **MyObjectCOM** component and run the migration assistant again to fix the code.

Remove Circular References

The key to the flexibility and reusability of ASP (and ASP.NET) is its inclusion functionality, which lets you easily add code from other files to a page by using the **#include** instruction or the **scr** attribute of the **<script>** tag. However, when it used carelessly, this can lead to circular references, which happens when two or more files mutually include each other. Circular references do not affect the execution of your ASP pages, but they are not legal in ASP.NET. If you run the migration assistant on a page involved in a circular reference, it will affect the performance of the tool and may prevent it from converting all of the code. Try to identify possible circular references in your code and eliminate them by moving the shared code into a new file that you can then import into the original files.

Avoid Mixing Scripting Languages on the Server Side

Another important aspect that must be taken into consideration before you convert your ASP pages is that ASP allows you to use scripts with different languages like VBScript or JavaScript; however, ASP.NET does not support this combination in those scripts that will run on the server side in the same page. Therefore, you must ensure that your pages use only a single language in these scripts.

Because the migration assistant can only parse VBScript language, it is recommend that you change all code that is executed on the server to VBScript because your upgraded code will use all new features that Visual Basic .NET offers to you. Instead, you can unify to another script language. However, if you do this, the code will not be upgraded and will only be copied in the new pages. Because of differences in ASP and ASP.NET, this may result in different behavior in your upgraded ASP.NET page.

Upgrading the Application

After the correct preparation steps are performed, it is time to begin the actual upgrade process. This section provides information about the steps to achieve the upgrade.

Upgrade Options

Upgrading an enterprise application that uses both ASP and Visual Basic 6.0 components could require using both the ASP to ASP.NET Migration Assistant and the Visual Basic 6.0 to Visual Basic .NET Upgrade Wizard. If the application has a business logic layer, user defined classes in the data access layer, or additional components written in Visual Basic 6.0 that run on an application server, you will have to upgrade these fragments of the application using the Visual Basic 6.0 to Visual Basic .NET Upgrade Wizard. After these foundation components are upgraded, you can process the ASP code with the ASP to ASP.NET Migration Assistant.

Using the ASP to ASP.NET Migration Assistant

You can use the ASP to ASP.NET Migration Assistant as a wizard through the Visual Studio .NET IDE or as a command-line tool. The next sections describe each of these approaches.

Using the Wizard

To convert your ASP applications using Visual Studio .NET, point to **Open** on the **File** menu, and then click **Convert**. Next, click **ASP to ASP.NET Migration Assistant** from the list of available converters. If you have a solution open in Visual Studio .NET, you can click **Add to current solution** to add the result to your current solution. Alternatively, you can click **Create new solution** if you want the result to be stored as a separate solution.

The wizard contains seven pages that walk you though the procedure of upgrading your ASP files:

1. The Welcome page appears first. Click **Next** to begin the procedure.
2. Page 2 allows you to specify the directory that contains the files to be upgraded. Type the path to the directory that contains the files you want to convert, or click **Browse** and use the **File Open** dialog box to locate it. After the desired directory is located, click **Next**.
3. Page 3 allows you to specify a name for the project. Type a name for your new ASP.NET application, and then click **Next**.
4. Page 4 allows you to specify the target directory where the upgraded project will reside. Type the path to the directory where you want to create your new ASP.NET project, or click **Browse** and use the dialog box to locate it. After the target directory is specified, click **Next**.
5. Page 5 allows you to specify any absolute references or Uniform Resource Identifier (URI) addresses that you want to upgrade from .asp to .aspx. In general, the ASP to ASP.NET Migration Assistant converts relative addresses to make them refer to the new upgraded files; for example, <A href="../../OtherDirectory/Default.asp"> ... will be converted to <A href="../../OtherDirectory/Default.aspx"> ... . It is not convenient to apply this conversion to all URI addresses because some of them can refer to external resources that will not be upgraded and therefore require no changes. To control this situation for URIs and absolute addresses, the user has to specify on this page of the wizard which addresses will be upgraded.

   After you specify these references, click **Next** to continue.
6. Page 6, the **Ready to Upgrade** page, is provided as an indicator that the upgrade procedure can begin. Click **Next** to begin the upgrade.
7. Page 7 provides a progress indicator that displays the progress of the migration assistant as it upgrades your files. When the progress indicator is full, the upgrade procedure is complete.

Using the Command Line Tool

The ASPUpgrade.exe command-line tool has the same functionality as the wizard. The command line syntax for ASPUpgrade.exe is shown here.

```
ASPUpgrade[.exe] DirectoryName [/out DirectoryName] [/nolog | /logfile filename ]
[/Verbose] [/ProjectName] [/ForceOverwrite]
```

Table 1 lists the meanings for the options.

Table 1: ASPUpgrade.exe Options

Argument	Option	Description
DirectoryName		Required. The path of the ASP file(s) to be upgraded
	/out *DirectoryName*	Specify the path for the folder where the ASP.NET project will be created. If this option is not specified, the default path is \OutDir.
	/verbose	Display all output to the Command Prompt window during the upgrade.
	/nolog	Do not create a log file during the upgrade.
	/logfile *filename*	Create a log file using the *filename* specified (may include full path name if desired). If no path and file name are specified, a log file will be created in the same folder as the ASP.NET project with the default file name *ProjectName*.log, where *ProjectName* is the name of the project file.
	/? or **/help**	Display a list of command syntax options.

When using the command-line version of the tool, keep the following in mind:

- Arguments are case insensitive.
- Paths with spaces must be enclosed in double quotation marks.
- If the target directory does not exist, the migration assistant will create it.
- If the target directory already exists, the migration assistant will first verify whether you want to overwrite the contents of this directory.
- You must have write privileges in the target directory.
- You cannot use both the **/NoLog** and **/LogFile** options together.

Completing the Upgrade with Manual Changes

After your applications are upgraded, the most difficult errors to detect are the run-time errors. The migration assistant can detect many potential run-time errors and mark them in the code so you can fix them. However, there may still be some problems that the migration assistant does not detect; those problems will become apparent only when the upgraded application runs and is thoroughly tested. The original application test cases and design documentation, if you have them, will be extremely valuable for the testing phase.

Some of the common upgrade issues that the ASP to ASP.NET Migration Assistant detects that have to be manually corrected include the following:

- **Features with new behavior**. Some ASP functions have been changed for ASP.NET. When the migration assistant upgrades your code to use these ASP.NET functions, it adds upgrade warnings to alert you to the changes so you can properly test the new methods. For example, the following code uses the ASP **Date** function to obtain the current date and display it in the browser.

```
<script Language=VBScript runat=server>
Sub PrintDate()
    Dim myVar
    myVar = Date
    Response.Write("<strong>Current Date</strong> = " & CStr(myVar))
End Sub

PrintDate
</script>
```

When you convert this code to ASP.NET, you will get the following code.

```
<script language="VB" runat="server">
Sub PrintDate()
    Dim myVar As Date
    ' UPGRADE_WARNING: Date was upgraded to Today and has a
    ' new behavior.
    myVar = Today
    Response.Write(("<STRONG>Current Date</STRONG> = " & CStr(myVar)))
End Sub
</script>

<%
    PrintDate()
%>
```

Note: In the preceding code listing and throughout this chapter, some of the longer code lines and comments that are automatically produced by the ASP to ASP.NET Migration Assistant have been reformatted to make them easier to read and understand. Your code may be slightly different.

In most cases, you can simply test the application to verify that it still works with the new function; if the ASP.NET application exhibits the same behavior as the ASP version, you should remove the warning. In the example, the new function **Today** returns the current date in the same format as the ASP **Date** function, so you only need to remove the warning, as shown here.

```
<script language="VB"  runat="server">
Sub PrintDate()
    Dim myVar As Date
```

```
    myVar = Today
    Response.Write((“<STRONG>Current Date</STRONG> = “ & CStr(myVar)))
End Sub
</script>

<%
    PrintDate()
%>
```

- **Unsupported features**. Some ASP properties, methods, and events do not exist in ASP.NET, and their behavior cannot be simulated with any new feature. When the migration assistant encounters these features, it leaves them untouched and adds upgrade notifications to the code to alert you to the problem. For example, this ASP code uses the **Calendar** property, which is not supported in ASP.NET.

```
<script Language=VBScript runat=”server”>

Sub ChangeCalendar()
    if (Calendar = vbCalGreg) then
        Calendar = vbCalHijri
    else
        Calendar = vbCalGreg
    end if
End Sub

ChangeCalendar
</script>
```

Converting this code results in the following ASP.NET code.

```
<script language=”VB” runat=”server”>

Sub ChangeCalendar()
    ' CONVERSION_TODO: The 'vbCalGreg' identifier was not declared,
    ' because it is a constant in 'VBA.VbCalendar' library. Copy
    ' this link in your browser for more:
    ' ms-its:C:\Program Files\ASP to ASP.NET Migration Assistant\
    ' AspToAspNet.chm::/1020.htm
    ' UPGRADE_ISSUE: Calendar property is not supported. Copy this
    ' link in your browser for more:
    ' 'ms-help://MS.VSCC.2003/commoner/redir/redirect.htm?keyword=”vbup1039"'
    If (Calendar = vbCalGreg) Then
        ' UPGRADE_ISSUE: Calendar property is not supported.
        ' CONVERSION_TODO: The 'vbCalHijri' identifier was not
        ' declared, because it is a constant in 'VBA.VbCalendar'
        ' library. Copy this link in your browser for more:
        ' ms-its:C:\Program Files\ASP to ASP.NET Migration Assistant\
        ' AspToAspNet.chm::/1020.htm
        Calendar = vbCalHijri
    Else
```

```
        ' UPGRADE_ISSUE: Calendar property is not supported.
        ' CONVERSION_TODO: The 'vbCalGreg' identifier was not declared,
        ' because it is a constant in 'VBA.VbCalendar' library.
        ' Copy this link in your browser for more:
        ' ms-its:C:\Program Files\
        ' ASP to ASP.NET Migration Assistant\AspToAspNet.chm::/1020.htm
        Calendar = vbCalGreg
    End If
End Sub

</script>
<%
ChangeCalendar()
%>
```

When the migration assistant reports that a property, method, or event cannot be converted to an ASP.NET equivalent, you have two options: you can remove the instruction if doing so will not have a significant impact on your application, or you can look for a workaround in the new language that can provide your application with similar functionality. For example, in place of the ASP **Calendar** property, you can use theASP.NET **System.Globalization** namespace. This namespace contains classes that define culture-related information, including language; country or region; calendars in use; format patterns for dates, currency, and numbers; and sort order for strings.

- **Rendering functions**. In ASP, you can define methods using multiple render tags, with ordinary HTML in between. However, in ASP.NET, you can only declare methods between opening and closing **<script>** tags. For example, the **helloWorld** method in this ASP code begins and ends in two separate render tags, with HTML in between, as shown here.

```
<%Sub helloWorld()%>
    <h1>Hello World</h1>
<%End Sub %>
```

When the code is converted, the migration assistant does not modify the rendering functions, generating an error instead.

```
<% 'UPGRADE_NOTE: Rendering Functions are not supported. Copy this
    link in your browser for more:
    ms-help://MS.MSDNVS/aspcon/html/aspup1008.htm %>
<%Sub helloWorld()%>
    <h1>Hello World</h1>
<%End Sub %>
```

To resolve the error, you must move the entire method between a single set of opening and closing **<script>** tags and use the **Response.Write** method to send the HTML information to the page. Rewriting the previous example results in the

following legal ASP code, which the migration assistant can process with no problems.

```
<script language=VBScript runat=Server>
Sub helloWorld()
    Response.Write("<h1>Hello World</h1>")
End Sub
</script>
<% helloWorld %>
```

- **Multiple method declarations**. ASP allows you to declare the same method more than once because it uses only the last declaration that was written and ignores the others. However, in ASP.NET, multiple declarations of a method with identical signatures are not supported. If the migration assistant encounters more than one method declaration with the same signature, it will comment out all declarations except the last. For example, in this ASP code, the **MySub** function is declared twice.

```
<%
    Sub MySub()
        Dim y
        y = "String Value"
        Response.Write y
    End Sub

    Sub MySub()
        Dim y
        y = 1
        Response.Write y
    End Sub

    MySub
%>
```

When the code is converted, the migration assistant will comment out the first method declaration and add an upgrade note, resulting in the following ASP.NET code.

```
' UPGRADE_NOTE: All function, subroutine and variable declarations
' were moved into a script tag global. Copy this link in your browser
' for more:
' ms-help://MS.MSDNVS/aspcon/html/aspup1007.htm
<script language="VB" runat="Server">
    ' UPGRADE_NOTE: Subroutine 'MySub' was commented. Copy this
    ' link in your browser for more:
    ' ms-help://MS.MSDNVS/aspcon/html/aspup1009.htm
    ' Sub MySub()
    '     Dim y
    '     y = "String Value"
    '     response. write y
```

```
    ' End Sub

    Sub MySub()
        Dim y As Integer
        y = 1
        response. write(y)
    End Sub

</script>

<%
    MySub()
%>
```

To clear the upgrade note and the commented code from the project, delete it from the page.

- **Unreachable code inside a script tag**. You can use the **<script>** tag to add code to a page in one of two ways: by entering code directly on the page between opening and closing **<script>** tags, or by using the **src** attribute of the opening tag to specify an external file that contains the code that you want to use. If a pair of **<script>** tags includes both a reference to an external file and code of its own, ASP uses only the code in the file specified by the **src** attribute and ignores the code between the opening and closing tags. For example, when IIS processes the following ASP code, it will import the code from the file FileName.vb and ignore the **MySub** declaration on the page.

```
<script Language=VBScript src="FileName.vb" runat=server>
Sub MySub()
    dim myVar
    myVar = myVar + 1
    Response.Write("SUM = " & CStr(myVar))
End Sub
</script>
```

When you convert this page with the migration assistant, it will generate the following upgrade notes.

```
<%'UPGRADE_NOTE: Code inside the Script tag was ignored by the upgrade tool.
Copy this link in your browser for more: ms-its:C:\Program Files\ASP to ASP.NET
Migration Assistant\AspToAspNet.chm::/1006.htm  %>
<%'UPGRADE_NOTE: Language element 'SCRIPT' was migrated to the same language
element but still may have a different behavior. Copy this link in your browser
for more: ms-its:C:\Program Files\ASP to ASP.NET Migration
Assistant\AspToAspNet.chm::/1011.htm  %>
<script language="VB" src="FileName.vb" runat=server>
Sub MySub()
    Dim myVar
    myVar = myVar + 1
    Response.Write("SUM = " & CStr(myVar))
End Sub
</script>
```

Like ASP, ASP.NET ignores the code between the opening and **closing <script>** tags because the **src** attribute specifies an external file to use. If you need to use this code, you must move it to another pair of **<script>** tags.

```
<script language="VB" src="FileName.vb" runat=server></script>
<script language="VB" runat=server>
Sub MySub()
    Dim myVar
    myVar = myVar + 1
    Response.Write("SUM = " & CStr(myVar))
End Sub
</script>
```

If you do not need the code, remove it from the page, leaving only the empty pair of **<script>** tags that references the external file.

```
<script language="VB" src="FileName.vb" runat=server></script>
```

Testing and Debugging the Upgraded Application

The objective of the testing and debugging phase is to identify and correct problems that occur after you converted the application to ASP.NET. Some problems will be easy to identify and fix immediately; others will only become apparent when you run your application.

The ASP to ASP.NET Migration Assistant generates a conversion report that contains a list of issues for you to address after you convert your application. The report can include the following types of issues:

- **Conversion issues**. These are generated when there are problems with unsupported class members or language elements. You may need to manually convert some of the affected expressions to similar .NET elements. For each issue, review the functionality of the affected expressions and perform some limited testing.
- **ToDo**. These are generated when code requires user intervention. The instructions included in the item should be followed and tested after it is integrated in the upgraded source code.
- **Run-time warnings**. These are generated by code that will compile but may generate a run-time error. This code needs to be thoroughly tested.
- **Global warnings**. Global issues such as the lack of a deterministic object lifetime because of the garbage collector are included here. Most of these warnings are caused by setup issues that can be easily fixed and tested. Other warnings require more work to fix, such as source code lines that the migration assistant does not recognize and that require manual conversion.

- **Notes**. These are generated when the source code has been significantly altered, or the resulting ASP.NET code behaves differently from the original Visual Basic 6.0 version. The code needs to be tested to determine if the behavior differences affect the core application functionality.

Unit and system testing should be performed for all the upgraded application functionality, using the original application specifications and test cases to generate new test cases for the ASP.NET version of the application.

For large size applications, a recommended approach is to divide the application into smaller modules for individual testing. In a first stage, you can test the most basic components of your ASP.NET application, that is, components that do not depend on other user-defined components. After the basic components are tested and adjusted if necessary, components that depend on them can be verified. This approach ensures that work is performed in a progressive way and makes it easier to isolate and correct any problems that are detected. The main disadvantages are that more planning and coordination is required and the degree of parallelism is diminished. It is important to consider that ASP can be run side-by-side with ASP.NET on the same Web server; this allows a progressive testing of the application features based on known and tested features. For example, if you are testing a Web application that receives order inquiries and processes them to produce a detailed report when requested, you can test the ASP.NET detailed report generation components while having the rest of the functionality running on ASP. This approach will be useful as a preliminary test of functionality and workload. It can be performed as long as there is no direct communication between the ASP and ASP.NET components because they are executed in separate server processes.

Deployment

After your application is tested and is running according to your standards, the next step is to create a new installer to distribute it to other computers automatically. The Visual Studio .NET deployment feature automatically builds a solution for deployment and handles any registration and configuration issues.

► To deploy an ASP.NET application

1. Create a Web setup project.
2. Create the structure of the web site in the setup project.
3. Include your ASP.NET project as project output to the deployment project.
4. Build the solution.

The resulting ASP.NET solution must be deployed by someone with Administrator user rights on the server to which the application is deployed. For more information about creating installers, see Chapter 16, “Application Completion.”

More Information

For more information about upgrading from ASP to ASP.NET, see "Migrating to ASP.NET: Key Considerations" and "Converting ASP to ASP.NET" on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/aspnetmigrissues.asp
and:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/convertasptoaspnet.asp.

For more information about system requires for ASP.NET, see "System Requirements for Visual Studio .NET 2003" in the Microsoft Visual Studio Developer Center on MSDN:
http://msdn.microsoft.com/vstudio/productinfo/sysreqs/default.aspx.

To download the ASP to ASP.NET Migration Assistant, see the Microsoft ASP.NET Developer Center on MSDN:
http://msdn.microsoft.com/asp.net/migration/aspmig/aspmigasst/default.aspx.

For more information about ASP.NET security, see "ASP.NET Web Application Security" in the *.NET Framework Developer's Guide* on MSDN:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconASPNETWebApplicationSecurity.asp.

For more information about VB Companion, see the ArtinSoft Web site:
http://www.artinsoft.com/pr_vbcompanion.aspx.

Appendix D

Upgrading FMStocks 2000 — A Case Study

This case study showcases the process of upgrading a Microsoft Visual Basic 6.0 application to a functionally equivalent Visual Basic .NET application using the Visual Basic 6.0 Upgrade Assessment Tool, the Visual Basic Upgrade Wizard, and the guidance chapters using the Fitch & Mather Stocks 2000 (FMStocks 2000) application as a reference application. It describes the issues encountered while upgrading the FMStocks 2000 application to Visual Basic .NET, identifies their resolutions, and explains how these resolutions were determined. The aim of this case study is not to identify all the issues involved in upgrading to Visual Basic .NET; the aim is to demonstrate the approach taken to identify issues and resolve them.

This case study will also serve as a starting guide for those who would like to test the assessment tool and upgrade wizard by using FMStocks 2000 as a sample application. This will give developers some familiarity with these tools and with the upgrade process before starting a real upgrade project.

To demonstrate the upgrade process, FMStocks 2000 was upgraded to Visual Basic .NET with the name FMStocks_AutomatedUpgrade using the upgrade wizard. FMStocks_AutomatedUpgrade was then manually changed to a functionally equivalent .NET version and given the name FMStocks_NET to distinguish it from the auto-generated code. Both FMStocks_AutomatedUpgrade and FMStocks_NET are available as FMStocks_AutomatedUpgrade.msi and FMStocks_NET.msi on the companion CD or from the GotDotNet community site for this guide.

About FMStocks 2000

Fitch & Mather Stocks 2000 (FMStocks 2000) is a Microsoft Windows Distributed interNet Applications Architecture (Windows DNA) sample application developed for Microsoft Windows 2000 that simulates a public stock trading Web site on the Internet. The online stock-trading scenario has been extended by adding an online bookstore.

FMStocks 2000 is comprised of the presentation layer, the business logic layer, and the data layer. The presentation layer uses ASP and interacts with the business logic layer through distributed COM. The prominent components in the business logic layer are the FMStocks_Bus and FMSStore_Bus components that contain the services necessary for online stock trading and bookstore, respectively. The data layer consists of the data access components that exist in the middle tier and the database. The data access components are FMStocks_DB and FMStore_DB components, which handle data exchange between the database (Microsoft SQL Server 2000) and the business logic components.

The following are the features of FMStocks 2000:

- It demonstrates performance and scalability gains.
- It simplifies component development by taking advantage of easier COM+ programming idioms.
- It extends the online stock-trading scenario by adding a bookstore. This allows handling a more complicated state and shopping cart feature and ensures the high availability of the site during large-grain transactions such as purchasing.
- It expands the client experience by adding Microsoft Office 2000 connectivity to the site by way of Excel and wireless access using Windows CE.
- It provides additional security for the application by eliminating clear-text cookies and separating key business components from the Web servers.

Reasons FMStocks 2000 Is Used for the Case Study

FMStocks 2000 includes a host of features that makes it an ideal subject for a case study on upgrading from Visual Basic 6.0 to Visual Basic .NET. These features include:

- COM+ interoperability and MTS.
- Office 2000 integration.
- Error logging.
- Use of interfaces.
- Calling Windows APIs.
- Use of ActiveX Data Objects (ADO).

FMStocks 2000 Setup

The Fitch & Mather Stocks 2000 (FMStocks 2000) application, including source code, can be obtained from the download section of the FMStocks 2000 Web site. To install the application, download the installation setup program and execute it. Follow the instructions in the installation wizard to specify where you would like to install the application.

For more information about building and executing the application, see "FMStocks2000_Setup.doc" in the <installation directory>\FM Stocks 2000\documentation folder.

FMStocks_AutomatedUpgrade Setup

The green code produced by the upgrade wizard can be found in FMStocks_AutomatedUpgrade. If you want to compare the green code you obtain with the green code obtained during this case study, you can download the case study version from the community site. The green code is found in the file FMStocks_AutomatedUpgrade.msi. When the contents of this file are installed, the installation folder is C:\Program Files\ FMStocks_AutomatedUpgrade. The installation file installs the upgrade wizard reports and the green code so that you can compare the progress of your upgrade as you proceed through this case study.

FMStocks_NET Setup

The final functionally equivalent code produced during this case study can be found in FMStocks_NET. You can obtain this code from the GotDotNet community site. The file is FMStocks_NET.msi. This file installs FMStocks_NET, which is the functionally equivalent working version of FMStocks 2000. When the contents of this file are installed, the installation folder is C:\Program Files\FMStocks_NET. The assessment tool reports and the final functionally equivalent code are installed so that you can compare your results against the results produced while preparing this case study. The folder also contains the FMStocks_NET_Setup.doc, which provides detailed instructions about setting up FMStocks_NET.

FMStocks 2000 Assessment and Analysis

The main objectives for upgrading FMStocks were to achieve functional equivalence and to accelerate the upgrade process. To achieve functional equivalence, an analysis of the functionality of the Visual Basic 6.0 version of the FMStocks was required. To fulfill the second objective of accelerating the upgrade process, apart from the automated upgrade, a concrete upgrade strategy and plan that included the estimates, tasks, risks and issues involved in the upgrade were required. To make a detailed plan of the upgrade, a complete analysis of the content of the Visual Basic

6.0 version of FMStocks was required. This included analysis of the structure and design of FMStocks, including the prominent subsystems in the application and how they interact, the features offered by the application, the technologies that it used, the complete source code, and the dependencies on external systems and third-party libraries.

The use case analysis was helpful for analyzing and documenting the functionality of the Visual Basic 6.0 version of FMStocks for upgrade planning and test planning to achieve functional equivalence. The Visual Basic 6.0 Upgrade Assessment Tool was used to perform a complete analysis of the application. It helped to fulfill the objective of accelerating the upgrade process by providing the data for devising an upgrade plan and for doing so on-demand.

Overview of the Structure of FMStocks

The structure of FMStocks 2000 was determined from the documentation provided with it, the analysis produced by the assessment tool, and a high level code review of the application. FMStocks 2000 is comprised of three layers: the presentation layer, the business logic layer, and the database layer. The projects that make up the business logic layer are as follows.

- FMStocks_Bus.vbp. This project contains the business code for the online stock trading functionality. It includes four class modules: Account, Broker, Ticker, and Version.
- FMSStore_Bus.vbp. This project contains the business code for the online bookstore functionality. It includes two class modules: Product and ShoppingCart.
- FMSStore_Events.vbp. This is the interface for defining events for external order processing for online bookstore functionality.
- FMSStore_EvtSub2.vbp. This implements the event for external order processing for online bookstore functionality.

The projects that make up the data access layer are:

1. FMSStore_DB.vbp. This project contains the data access code for the online bookstore functionality. It includes two class modules: Product and ShoppingCart.
2. FMStocks_DB.vbp. This project contains the data access code for the online stock trading functionality. It includes eight class modules: Account, Broker, DBHelper, Position, Ticker, Product and ShoppingCart.

The preceding projects are grouped under the following project groups:

1. FMS2000_Core.vbg. This project group includes FMStocks_Bus.vbp and FMStocks_DB.vbp.
2. FMS2000_Store.vbg. This project group includes FMSStore_Bus.vbp and FMSStore_DB.vbp.
3. FMS2000_Events.vbg. This project group includes FMSStore_Events.vbp and FMSStore_EvtSub2.vbp.

Overview of the Use Cases

The use case analysis was required to analyze and document the functional specifications of the FMStocks application to achieve functional equivalence during upgrading and testing. In the case of FMStocks, the use cases were derived from the working Visual Basic 6.0 version of the FMStocks application. Because this case study does not involve advancing the application, no new use cases were created. FMStocks 2000 consists of the following use cases:

- Login for existing user
- Login for new user
- Logout
- Symbol or company lookup
- Display of account balance
- Display of portfolio
- Chart portfolio
- Buy a stock
- Sell a stock
- View shopping cart
- Browse products
- Checkout

Table 1 on the next page illustrates a sample use case from FMStocks 2000 for the functionality of selling stocks.

Table 1: Sample Use Case for FMStocks 2000

Use case ID	UC-09
Name	Sell a stock
Actors	Application user
Description	It displays the current portfolio and asks the user to select which shares and number of stock shares to sell. It orders a sell request.
Trigger	None
Pre-conditions	User must have logged on to the system.
Post-conditions	None
Normal flow	1. User clicks **Sell a stock** link. 2. Application displays a list of the user's current stocks. 3. The user selects which shares to sell and the quantity. 4. The application creates a new transaction and updates the relative database fields. 5. A sell receipt is displayed after the sell is confirmed.
Alternative flows	If the input number of shares is more than the existing shares, the following message is displayed:"Unable to place order. You tried to sell more shares than you own."
Exceptions	None
Includes	None
Priority	High
Frequency of use	Medium
Business rules	None
Special requirements	None
Assumptions	None
Notes and issues	None
Use case ID	UC-09

The use cases for FMStocks 2000 have been documented and are available on the companion CD.

Obtaining the Upgrade Inventory

The most important step of the FMStocks application analysis was getting the statistics of how many components, classes, modules, and user controls the application contained. Getting to know the inventory helped assess the complexity of the FMStocks structure which in turn contributed to the planning of the upgrade and estimation of time and effort. The assessment tool was used to analyze the inventory

for each project group. The Project Files Overview worksheet of the DetailedReport.xls it produced listed the files in each project and categorized them into classes, modules, and so forth. Because the FMStocks projects that were analyzed were business layer components or data access layer components, they did not contain any forms or designers. Tables 2, 3, and 4 list the data collected for each project group from the Project Files Overview worksheet. Note that the actual worksheet includes the following columns: UserControls, Non-upgradable, Designers, and Forms. They are not included in the tables here because there are no values in those columns in this particular worksheet.

Table 2: FMS2000_Core Project Group

Project	Total files	Classes	Modules
FMStocks_Bus	5	4	1
FMStocks_DB	9	8	1
Total	14	12	2

Table 3: FMS2000_Store Project Group

Project	Total files	Classes	Modules
FMStocks_Bus	3	2	1
FMStocks_DB	3	2	1
Total	6	4	2

Table 4: FMS2000_Events Project Group

Project	Total files	Classes	Modules
FMStocks_Events	1	1	0
FMStocks_EvtSub_OrderProc	1	1	0
Total	2	2	0

The data from the Project Files Overview worksheet helped determine that the FMS2000_Store project group was the prominent project group for migration. This was taken into account when finalizing the upgrade and the test plan.

The upgrade of third-party components accounted for a significant amount of upgrade effort. The upgrade wizard can upgrade only one project at a time. Because of this, when upgrading one project, the ActiveX components that were referenced by the original version of the project were wrapped with an interoperability DLL. Because the referenced ActiveX components had also been upgraded to .NET, the interoperability DLL was unnecessary. Thus for each project that was upgraded, the references to ActiveX components had to be manually upgraded to references to

.NET components. Because the .NET components had names and progIDs different from that of the ActiveX components, it was necessary to drill down to each line of code and change the call from the ActiveX component to the .NET component. This is where the Third Party Components Summary and Members report produced by the assessment tool came in handy. The Third Party Components Summary report lists the third-party components instantiations with the count of their instances. The information in Table 5 is provided by the Third Party Components Summary report for the FMS2000_Store project group.

Table 5: Third Party Components Summary Report for FMS2000_Store

Component	Count
FMSStore_Events.ShoppingCart	1
FMStocks_DB.DBHelper	2
ADODB.Recordset	2

This report shows that the FMS2000_Store project group depends on the class module ShoppingCart of the FMSStore_Events component, the class module DBHelper of the FMStocks_DB component, and the recordset type of the ADODB library. The Third Party Components Members report lists the members of the third party components that are being used in the code along with the count. Shown below is the Third Party Components Members report for the FMS2000_Store project group.

Table 6: Third Party Components Members Report for FMS2000_Store

Member	Count
FMSStore_Events.ShoppingCart.ExecuteBuy	1
FMStocks_DB.DBHelper.RunSP	4
ADODB.Recordset.BOF	2
FMStocks_DB.DBHelper.RunSPReturnRS	6
ADODB.Recordset.EOF	2

These reports made it easy to locate third-party components and make the appropriate changes to their instantiations.

The other reports the FMStocks 2000 inventory information was collected from were the Data Access Components report, API Calls report, User Components Members report, Intrinsic Components Members report, and the User Com Objects Members report. Tables 7, 8, 9, and 10 show the information these reports produce.

Table 7: Data Access Components Report

ADO data bindings	
Control name	**Control type**
No data found.	
ADO data members used	
Member	**Count**
ADODB.Command.CommandText	9
ADODB.Recordset.BOF	1
ADODB.Recordset.Open	5
ADODB.Recordset.Close	1
ADODB.Parameters.Append	8
ADODB.Recordset.AddNew	1
ADODB.Recordset.ActiveConnection	2
ADODB.Command.Execute	5
ADODB.Recordset.MoveNext	2
ADODB.Fields.Append	5
ADODB.Command.CreateParameter	8
ADODB.Recordset.Delete	1
ADODB.Recordset.Fields	5
ADODB.Recordset.MoveFirst	1
ADODB.Recordset.CursorLocation	5
ADODB.Command.Parameters	8
ADODB.Recordset.UpdateBatch	2
ADODB.Command.ActiveConnection	23
ADODB.Recordset.EOF	3
ADODB.Recordset.Value	5
ADODB.Command.CommandType	9

Table 8: API Calls Report

Function name	Parameter list	Parameter count	Library	Uses
RegisterEventSource			Advapi32.dll	2
DeregisterEventSource			Advapi32.dll	2
ReportEvent			Advapi32.dll	2
GetLastError			kernel32	0
CopyMemory			kernel32	2
GlobalAlloc			kernel32	2
GlobalFree			kernel32	2
GetComputerNameAPI			kernel32	2

Table 9: User Components Members Report

Member	Count
FMStocks_DB.Version.Version	1
FMStocks_Bus.Helpers.NullsToZero	19
FMStocks_DB.Helpers.NullsToZero	2
FMStocks_DB.Position.ListForSale	1
FMStocks_Bus.FMStocks_TransactionType.Insufficient_Funds	2
FMStocks_DB.Version.ConnectionString	1
FMStocks_DB.Ticker.GetPrice	3
FMStocks_DB.DBHelper.RunSP	1
FMStocks_DB.DBHelper.RunSPReturnInteger	7
FMStocks_DB.Helpers.GetKey	1
FMStocks_DB.Helpers.GetVersionNumber	1
FMStocks_DB.DBHelper.GetConnectionString	2
FMStocks_DB.Helpers.GetValue	2
FMStocks_DB.Ticker.ListByTicker	1
FMStocks_DB.Account.GetAccountInfo	1
FMStocks_Bus.Helpers.RaiseError	13
FMStocks_DB.Helpers.mp	34
FMStocks_DB.Account.Add	1
FMStocks_DB.Position.ListSummary	1

Member	Count
FMStocks_DB.TxNew.SetTxType	4
FMStocks_Bus.FMStocks_TransactionType.Not_Enough_Shares	2
FMStocks_DB.Account.Summary	1
FMStocks_DB.Position.ListForAdjustment	1
FMStocks_DB.Account.VerifyUser	1
FMStocks_DB.Version.ComputerName	1
FMStocks_DB.Broker.Buy	2
FMStocks_DB.Tx.AddBuyOrder	1
FMStocks_DB.Ticker.GetFundamentals	1
FMStocks_DB.Ticker.VerifySymbol	1
FMStocks_Bus.Helpers.GetComputerName	1
FMStocks_DB.DBHelper.RunSPReturnRS	8
FMStocks_DB.DBHelper.RunSPReturnRS_RW	1
FMStocks_DB.Helpers.RaiseError	30
FMStocks_DB.Broker.Sell	2
FMStocks_DB.Tx.AddSellOrder	1
FMStocks_DB.Tx.GetByID	1
FMStocks_Bus.Helpers.GetVersionNumber	1
FMStocks_DB.Ticker.ListByCompany	1
FMStocks_DB.Helpers.GetComputerName	1

Table 10: User Com Objects Members Report

Member	Count
FMStocks_DB.DBHelper.RunSP	1
FMStocks_DB.DBHelper.RunSPReturnInteger	7
FMStocks_DB.DBHelper.GetConnectionString	2
FMStocks_DB.DBHelper.RunSPReturnRS	8
FMStocks_DB.DBHelper.RunSPReturnRS_RW	1

Obtaining Source Code Metrics

FMStocks 2000 upgrade planning and time and cost estimation required knowledge of the number of lines of source code present in each project. The assessment tool was used to obtain these numbers. The Lines of Code worksheet in the DetailedReport.xls provided the number of lines for each project and it categorized the lines of code into visual lines, comment lines, blank lines, and code lines. Because the components being upgraded did not have user interfaces, the number of visual lines was negligible. The categorization of the lines of code proved useful because the comment lines and the blank lines were ignored for the estimation of upgrading cost and effort. Table 11 includes the information from the report that gave the number of lines of code for the FMS2000_Core project group.

Table 11: Lines of Code Report for the FMS2000_Core Project Group

Project	Total lines	Visual lines	Code lines	Comment lines	Blank lines
FMStocks_Bus	900	32	556	94	218
FMStocks_DB	1118	64	715	82	257
Total	2,018	96	1,271	176	475

The total number of lines in FMStocks 2000 is 4,943.

Handling of Unsupported Features

The upgrade wizard does not upgrade all the language features and technologies of Visual Basic 6.0 to Visual Basic.NET because some of these language features and technologies are not supported in .NET. The Visual Basic 6.0 features that are not upgraded by the upgrade wizard have to be manually upgraded. The manual effort that would be required to achieve functional equivalence for FMStocks 2000 after the automated upgrade would have been a difficult task if not for the assessment tool. The assessment tool was helpful when the issues related to automated upgrade had to be located and assessed. The assessment tool's DetailedReport.xls has an Upgrade Issues worksheet that lists the Visual Basic 6.0 features that will not be upgraded to Visual Basic .NET by the upgrade wizard along with the count of occurrences of these features in the application and the cost of upgrade associated to each one of them. This helped in assessing how much manual effort would be required to achieve functional equivalence.

Table 12: Upgrade Issues Table for the FMS2000_Core Project Group

Name	MSDN ID	Occurrences
Declaring a parameter "As Any" is not supported	1016	8
Setting an object variable to Nothing does not destroy the object. Object may not be destroyed until it is garbage collected	1029	99
Couldn't resolve default property of object	1037	46
Function has a new behavior in .NET	1041	22
Use of Null/IsNull() detected	1049	5
Due to differences between Visual Basic 6.0 and Visual Basic.NET, some objects or collections cannot be upgraded	2068	4
Some objects cannot be upgraded. Properties or methods of those objects have been copied into the upgraded code, but they will not compile because the underlying objects were not upgraded	2069	6

Note: The "Guidance topic" values in the actual report will be linked to the online version of this guide. Each link will take you to the chapter that explains how to address the specified issue.

Determining Application Dependencies

The knowledge of relationships between components and projects in FMStocks 2000 was required to define the upgrade order and to prepare the upgrade and test plans. The analysis of dependencies and relationships between projects and components was made easy by the assessment tool. The assessment tool generates a call graph and dependency graph for each project group analyzed. Figure 1 and Figure 2 on the next pages illustrate the file dependency graph and the call graph that are generated for the FMS2000_Core project group.

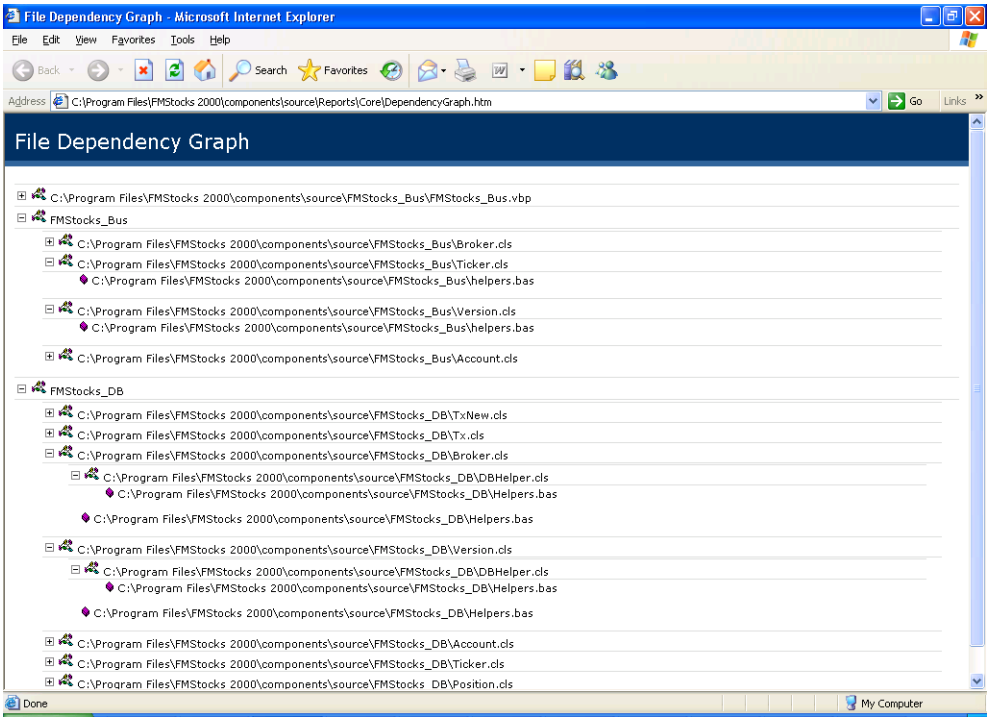

Figure 1

File Dependency Graph for the FMS2000_Core project group

The file dependency graph shows the files that belong to the projects FMStocks_Bus and FMStocks_DB. The files in the project FMStocks_Bus are dependent on the Helpers.bas file that belongs to the same project. The files in the project FMStocks_DB are dependent on the Helpers.bas file and the DBHelper.cls file, both belonging to the FMStocks_DB project.

The call graph shows the methods of the class modules that belong to the projects FMStocks_Bus and FMStocks_DB. In this call graph, the **GetSummary** method of the **Account** class module of the FMStocks_Bus project calls the **RaiseError** and **NullsToZero** methods of the **Helpers** module. Similarly, in the FMStocks_DB project, the **Buy** method of the **Broker** class module is dependent on the **RaiseError**, **mp**, and **NullsToZero** methods of the **Helpers** module.

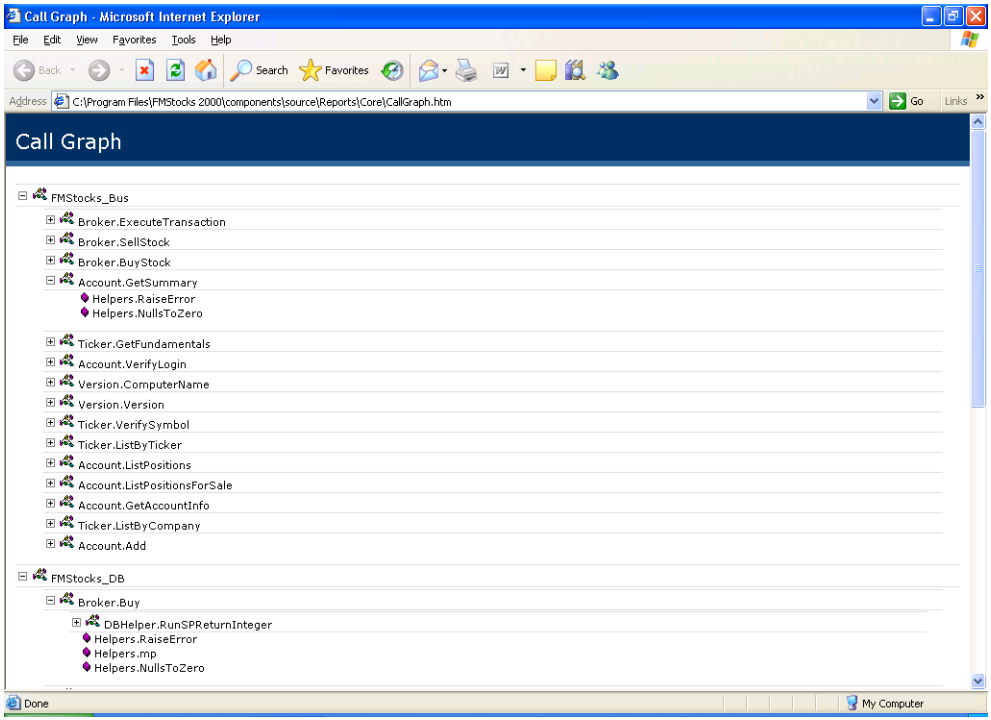

Figure 2
Call graph for the FMStocks_DB and FMStocks_Bus projects

The file dependencies are also identified by the assessment tool. The assessment tool generates the Upgrade Order report found in the DetailedReport.xls. Table 13 lists information from the Upgrade Order report for the project group FMS2000_Core.

Table 13: Upgrade Order Report for FMS2000_Core

Project	Dependency group	File
FMStocks_Bus	Dependency group 1	C:\Program Files\FMStocks 2000\components \source\FMStocks_Bus\Helpers.bas
FMStocks_DB	Dependency group 1	C:\Program Files\FMStocks 2000\components \source\FMStocks_DB\Helpers.bas

The information in the Upgrade Order report reflects the same data that was shown in the file dependency graph. The FMStocks_Bus project files depend on the Helpers.bas file. The same is true for the project FMStocks_DB.

The typical and direct approach is to first upgrade the core or basic components/files that have no dependencies on any other components or files, and then to proceed to upgrade the next set of components or files that are dependent on the earlier set of components/files. Based on the dependencies discussed earlier, the upgrade order defined for the FMStocks_Db and FMStocks_Core projects will have the **Helpers** module being upgraded before upgrading the rest of the class modules or the files.

The analysis and the assessment process of the FMStocks 2000 application took approximately 12 work hours of effort. That was the effort for analyzing and assessing 32 ASP pages and 4,943 lines of Visual Basic code in 22 Visual Basic files.

Upgrading FMStocks 2000

After the preliminary assessment is performed, the actual work in upgrading can begin. This section details the steps performed in upgrading FMStocks 2000 from Visual Basic 6.0 to Visual Basic .NET.

Planning the Upgrade

The upgrade planning stage for FMStocks 2000 consisted of the following substages: deciding the upgrade strategy, defining the upgrade order, and creating an upgrade schedule with estimates of cost and effort. The data from the assessment and analysis stage of FMStocks 2000 was the input for the upgrade planning stage of FMStocks 2000. The upgrade planning stage required approximately 4 work hours of effort.

Deciding the Upgrade Strategy

The upgrade strategy adopted for the FMStocks 2000 upgrade was the complete upgrade strategy. The reasoning behind adopting the complete strategy is as follows.

- A complete upgrade is fast and inexpensive.
- FMStocks 2000 has 4,943 lines of code divided between 22 files, 3 project groups, and 6 projects. From the source code metrics and the project files overview, it is apparent that FMStocks is a medium-sized application that can be upgraded completely before deploying with a low probability of any complications occurring because of choosing a complete upgrade over staged upgrade.

- The Upgrade Issues report lists 261 issues that are unsupported in Visual Basic .NET, but the issues identified are not related to deprecated technologies (such as DAO, RDO, and DDE). Because of this, it is unlikely that there will complications when deploying the complete upgraded application in .NET. Thus, this upgrade project does not require system testing whenever an individual component is upgraded, and as a result, this application is an unsuitable candidate for staged upgrade.
- A staged upgrade requires the implementation of wrappers or interfaces to provide the interoperability mechanisms between the Visual Basic 6.0 and .NET code. These wrappers will often be temporary and will be discarded later on. The overhead of creating the wrappers, especially with a dependency on COM components, makes the staged upgrade strategy unattractive for FMStocks 2000. Furthermore, the deployment takes more effort and time when interoperation is required between legacy COM components and .NET assemblies.
- The test strategy that was adopted was a test-driven upgrade, which meant that every component would be unit tested after being upgraded. This reduced the need for system testing after the upgrade of each component.

Defining the Upgrade Order

Although the complete upgrade strategy was adopted, the upgrade order still had to be defined because each component had to be united tested after upgrade. To define the upgrade order, the call graph, file dependency graphs, and upgrade order report in the DetailedReport.xls were taken into account, and the project references were also reviewed. The data that was obtained is as follows:

- FMSStore_EvtSub_OrderProc project depends on the FMSStore_Events project.
- FMSStore_DB project depends on the DBHelper class module of the FMStocks_DB project.
- FMSStore_Bus project depends on FMSStore_DB and FMSStore_EvtSub_OrderProc projects.
- FMStocks_Bus project depends on FMStocks_DB project.

The basic components/projects that did not depend on any other components/projects came first in the upgrade order list. The next set of components and projects to be upgraded were the ones that depended on the basic components/projects. The greater the number of components/projects a project depended on, the later it was in the upgrade list.

Based on the preceding data and upgrade strategy, the following upgrade order at a project level was defined:

1. FMSStore_Events project and FMStocks_DB project
2. FMSStore_EvtSub_OrderProc project and FMStocks_Bus project
3. FMSStore_DB project
4. FMSStore_Bus project

Creating the Upgrade Schedule

To create the upgrade schedule for FMStocks 2000, the following steps had to be executed:

- Identifying the different upgrade tasks
- Gathering estimation reports generated by the assessment tool
- Modifying the effort and cost estimation reports according to the project requirements

As discussed in Chapter 5, "The Visual Basic Upgrade Process," the different upgrade tasks identified for the FMStocks 2000 upgrade are:

- **Preparing the application**. This includes:
  - Preparing the development environment.
  - Verifying the compilation.
- **Upgrading FMStocks 2000**. This includes:
  - Executing the upgrade wizard.
  - Analyzing the upgrade wizard report and manually upgrading unsupported features.
  - Performing upgrade administrative tasks.
- **Testing the upgraded application**. This includes:
  - Creating test cases.
  - Creating automated tests.
  - Executing test cases for unit testing.
  - System testing for functional equivalence.
  - Managing bugs.
  - Regression testing.
  - Fixing bugs.
  - Performing test administrative tasks.

The next step was to gather the data from the estimation reports generated by the assessment tool. Table 14 lists information that is provided by the estimation report for the FMStocks2000_Core project group.

Table 14: Estimation Report for FMStocks2000_Core

Task	Effort (Hours)	Cost	Resource
Application preparation			
Development environment	4	$200	DEV
Application resource inventory	0	$7	DEV
Compilation verification	2	$100	DEV
Migration order definition	0	$3	DEV
Upgrade wizard report review	0	$5	DEV
Total	6	$315	
Application conversion			
Upgrade wizard execution	0	$3	DEV
Manual code adjustment	18	$1,032	
System integration and smoke test	4	$221	DEV
Administrative Tasks	1	$60	DLE
Total	23	$1,316	
Testing and debugging			
Test case creation	3	$210	STE
Test case execution	4	$160	TES
Resolution of Defects	5	$234	TES
Administrative Tasks	1	$77	STE
Total	13	$681	
Project management	2	$165	**PM**
Configuration management	1	$70	**CM**
Totals	45	$2,547	

The estimation report was customized for FMStocks 2000. The following configuration values were changed to customize this report:

- **In Config – Fixed Tasks**. The number of hours for compilation verification of the original application was changed from 8 hours to 2 hours. Because this is just a project group with 2,018 lines of code, the compilation verification time was reduced. Furthermore, FMStocks 2000 was a working application with all the required references. There was only one third-party component referenced by this project group and there were no missing components or missing files according to the data collected from the DetailedReport.xls generated by the assessment tool. Because of this, the chance of compilation errors occurring because of missing references was very slim. The result of this analysis was the reduction in estimate for compilation verification of the original application from 8 hours to 2 hours.
- **In Config – Fixed Tasks**. The number of hours for configuration management was changed from 8 hours to 1 hour. The size of the project group (2,018 lines of code), the choice of complete upgrade strategy, and the number of upgrade issues identified (191) indicated that the churn in the source code would require only a configuration management effort of 1 hour. This resulted in modifying the effort and cost estimation reports according to the project requirements.
- **In Config – Dependent Tasks**. The percentage for test case execution was increased from 1 to 10. This is because the test execution effort included feature testing for functional equivalence apart from manual and automated unit testing along with the analysis of the results.

Table 15 lists the cumulative effort and cost estimation for the three project groups, FMS2000_Core, FMS2000_Events, and FMS2000_Store, that make up the FMStocks 2000 application. Some of the important points to be noted in how this cumulative report was arrived at are the following:

- The entire effort of 4 hours for the development environment under preparing the application was included in the report generated for the project group FMS2000_Core, so the number of hours for preparing the development environment for rest of the project groups was made 0 hours. The assumption here was that the remaining project groups would not require any additional development environment preparation.
- The project group FMS2000_Events is significantly smaller (52 lines of code) than the rest of the project groups, so the configuration values were changed significantly for this project group. For the fixed task of verifying compilation, an effort of 1 hour had been allocated for FMS2000_Events project group. Similarly, the percentage for executing test cases was modified to 5.

Table 15: Modified Estimation Report

Task	Effort (Hours)	Cost	Resource
Application preparation			
Development environment	4	$200	**DEV**
Application resource inventory	0	$7	**DEV**
Compilation verification	3	$150	**DEV**
Migration order definition	0	$3	**DEV**
Upgrade wizard report review	0	$5	**DEV**
Total	7	$365	
Application conversion			
Upgrade wizard execution	0	$3	**DEV**
Manual code adjustment	24	$1,443	
System integration and smoke test	6	$293	**DEV**
Administrative Tasks	1	$81	**DLE**
Total	31	$1,820	
Testing and debugging			
Test case creation	4	$280	**STE**
Test case execution	5	$200	**TES**
Resolution of Defects	7	$350	**TES**
Administrative Tasks	2	$140	**STE**
Total	18	$970	
Project management	3	$226	**PM**
Configuration management	3	$210	**CM**
Totals	66	$3,381	

One of the most important points that came up at this juncture was the estimation for the upgrade of the presentation layer from ASP pages to ASP.NET. There were 32 ASP pages that had to be converted to ASP.NET. The strategy was to use the ASP to ASP.NET Migration Assistant for an automated upgrade. However, some preliminary investigations and estimations were made as follows:

- Because there was no way to predetermine the features that the ASP to ASP.NET Migration Assistant might not upgrade, the complete ASP code was analyzed for contentious issues. Based on the analysis, five sample ASP pages that contained a major part of the FMStocks functionality were selected and upgraded using the migration assistant. Then by analyzing the conversion report generated by the migration assistant, the upgrade issues that are likely to be thrown by the migration assistant were selected and their complexity was analyzed. The final estimated number of upgrade issues was 15.
- After analyzing the complexity of each issue that was likely to be thrown by the migration assistant, it was determined that an average complexity of 15 minutes of effort per occurrence would be needed. One of the reasons for the relatively high effort per occurrence when compared to the Visual Basic upgrade issues is that unlike the Visual Basic Upgrade Wizard Report the ASP to ASP.NET Migration Assistant Conversion report does not specify the line where the upgrade issue occurs. Thus, extra effort is required to locate the issue.

Based on this analysis and past experience with upgrading ASP pages to ASP.NET, the estimates in Table 16 were determined.

The points to be noted for the ASP to ASP.NET migration estimate are as follows:

- The effort for run-time error execution was set at a higher value because the ASP to ASP.NET migration assistant is an error-prone tool as indicated in Appendix C.
- The ASP pages execution verification task was included instead of the compilation verification task for Visual Basic code. This task included validating the execution of the ASP pages, the links/URLs within the ASP pages, and the calls to the proper COM components.

The final cost and effort estimates for the entire FMStocks 2000 application were as follows:

- Total Estimated Effort: 90.5 work hours
- Total Estimated Cost: $4,723

Table 16: Estimated Cost and Effort to Upgrade FMStocks 2000 ASP Pages to ASP.NET

Task	Effort (Hours)	Cost	Resource
Application preparation			
Development environment	0	$0	**DEV**
Application resource inventory	1	$50	**DEV**
ASP Pages execution verification	2	$100	**DEV**
Total	3	$150	
Application conversion			
Migration Assistant execution	0	$3	**DEV**
Manual code adjustment	4	$200	
System integration and smoke test	2	$100	**DEV**
Administrative Tasks	0.5	$45	**DLE**
Total	6.5	$345	
Testing and debugging			
Test case creation	3	$140	**STE**
Test case execution	4	$80	**TES**
Bug management	1	$50	**TES**
Regression test	1	$50	**TES**
Run-time error correction	7	$307	**DEV**
Administrative Tasks	1	$70	**STE**
Total	13	$697	
Project management	1	$80	**PM**
Configuration management		1	$70
Totals	24.5	$1,342	

Preparing the Application

Preparing the FMStocks 2000 application for upgrade involved the following tasks

1. Preparing the development environment: Necessary software that was required for FMStocks 2000, such as IIS 5.0 and SQL Server 2000, was installed according to the setup documentation for FMStocks 2000. Then FMStocks 2000 was installed. Visual Studio .NET 2003, the .NET Framework 1.1, and the Visual Basic 6.0 Upgrade Assessment Tool were also installed. The software needed for the assessment tool to run, such as Microsoft Excel 2003, was also installed.
2. Compilation verification/ASP pages execution verification: After FMStocks 2000 was installed, the source code for the business components was also copied to the installation folder. The project groups for the business components were then compiled and verified if they were being registered. The ASP pages were then executed against the registered COM components to validate the ASP pages.

The preparation stage for FMStocks 2000 required a total effort of approximately 7 work hours.

Executing Automated Upgrade

The first step in the actual upgrade process itself is to begin with the automated upgrade procedure. This section describes this experience for the FMStocks 2000 upgrade.

Visual Basic 6.0 to Visual Basic .NET Using the Upgrade Wizard

The Visual Basic Upgrade Wizard in Visual Studio .NET 2003 was used for to perform the automated upgrade. The project files were upgraded according to the upgrade order specified earlier in this case study. After a project was upgraded using the upgrade wizard, the project was manually adjusted and unit tested before the process continued with the next project.

Figure 3 illustrates the upgrade wizard report generated for the FMSStore_Bus project.

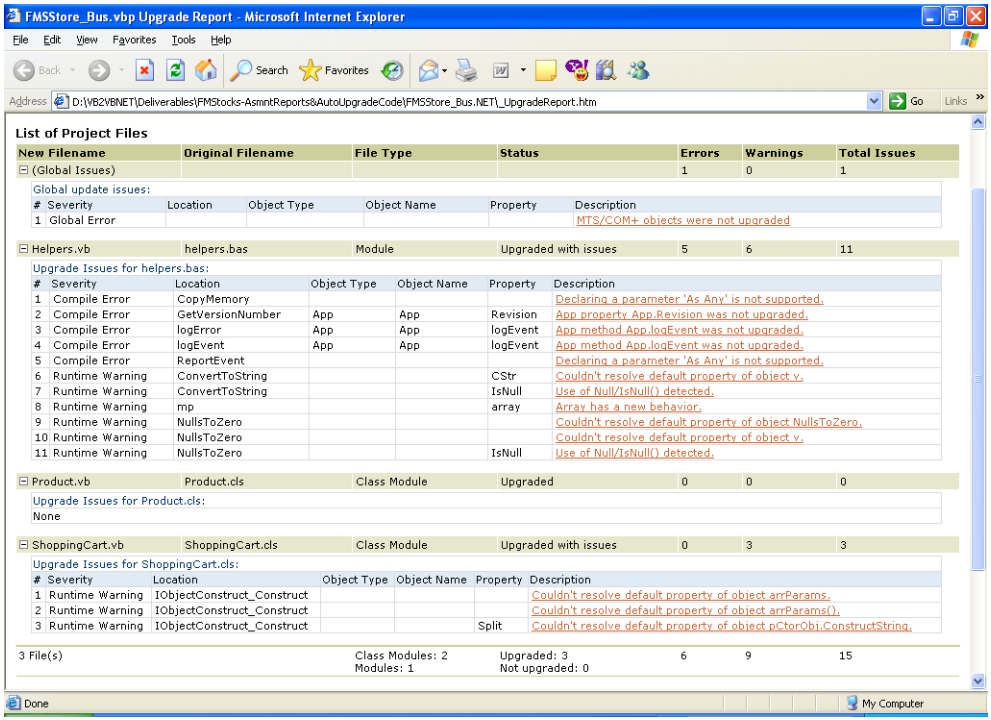

Figure 3

The upgrade wizard report for the FMSStore_Bus project

ASP to ASP.NET Using the Migration Assistant

The last stage in the FMStocks 2000 automated upgrade was to upgrade the ASP pages project to ASPX pages using the ASP to ASP.NET Migration Assistant.

Figure 4 on the next page illustrates the conversion report generated by the ASP to ASP.NET Migration Assistant.

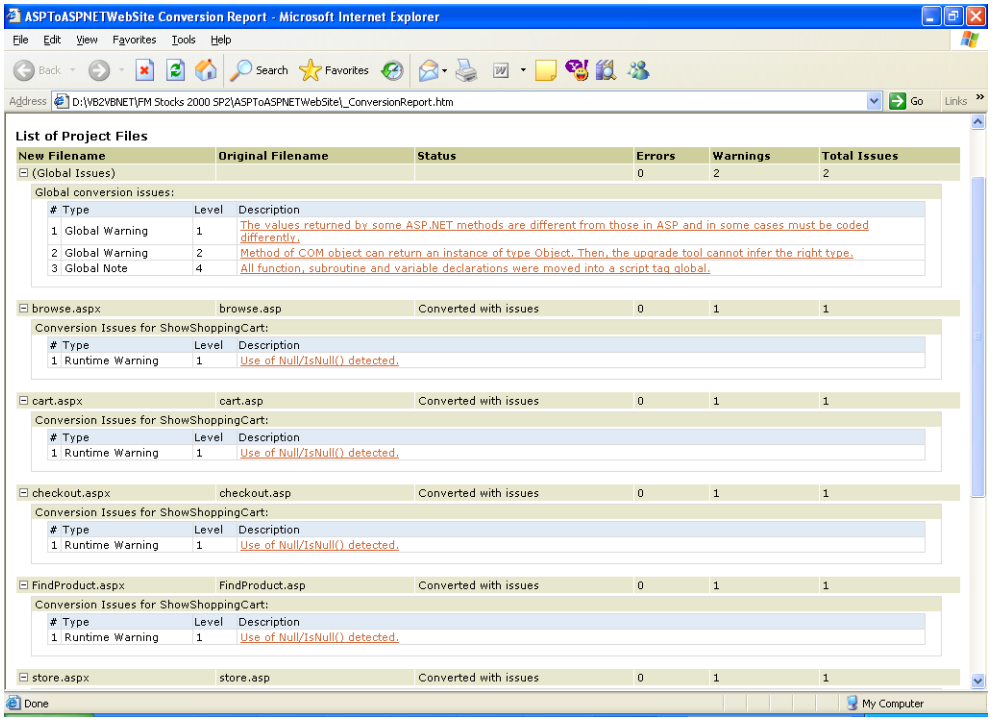

Figure 4

The ASP to ASP.NET Migration Tool Conversion Report for FMStocks 2000 ASP pages

The automated upgrade, including the application of both the Visual Basic Upgrade Wizard and the ASP to ASP.NET Migration Tool, required an effort of approximately one work hour.

Applying Manual Upgrade Adjustments

After performing the automated upgrade procedure, each project underwent a manual upgrade stage before proceeding to the next project. The upgrade wizard report generated in the automated upgrade stage was used to identify the issues. The resolutions for the upgrade issues were based on the *Upgrading Visual Basic 6.0 Applications to Visual Basic .NET and Visual Basic 2005* guide. Locating the appropriate resolution was also helped by the assessment tool's Upgrade Issues report in the DetailedReport.xls. This report lists most of the issues identified by the upgrade wizard and it provides pointers to the sections in the guide where the resolutions for these issues are detailed. Thus, it proved easy to detect and resolve the upgrade issues. After the issues were resolved, the upgraded code was reviewed and cleaned up; comments were changed or removed (for comments inserted by the upgrade wizard during the automated upgrade stage), and identifiers (such as variable names) were changed to make them suitable for the upgraded version of FMStocks 2000.

For manually upgrading the ASP pages, the report generated by the ASP to ASP.NET Migration Assistant was not very helpful as a guide to detect and resolve upgrade issues. Because of this, the ASPX code that had been upgraded by the migration assistant had to be completely reviewed line by line in order to detect and resolve upgrade issues.

Note: When the business COM+ components of FMStocks 2000 were upgraded from Visual Basic 6.0 to .NET, the ProgID for the COM+ components was changed during automated upgrade. For example, the FMStocks_Bus.Account component in Visual Basic 6.0 was upgraded to Account_NET.Account in Visual Basic .NET. However, these changes had to be reflected in the ASPX pages. During the manual upgrade, each ProgID had to be changed manually in the ASPX pages.

The manual upgrade of FMStocks 2000 required an effort of approximately 38 work hours. Extra effort was required to completely review the ASPX pages manually to identify and resolve upgrade issues.

Functional Testing

The objective for testing the upgraded .NET version of FMStocks 2000 was to test for functional equivalence. The following test process was followed for testing FMStocks 2000:

- Test planning and creating of test cases
- Unit testing
- Black box testing
- White box testing

Test plans and test cases were created for unit testing and black box testing before the upgrade of FMStocks began. The data required for the creation of the test plans was taken from the Use Case Analysis and by reviewing the original Visual Basic 6.0 code for FMStocks 2000.

In unit testing, each public method in each class of the upgraded project was tested to validate that the behavior of each method in the upgraded project matched the behavior of the corresponding method in the original version of the application. The test cases for each public method included passing different values for the input parameters of the method starting from values in the valid range, to boundary values, and on to values outside the boundaries. All the test cases for unit testing were automated using NUnit. (For more information about unit testing with NUnit, see the NUnit Web site.) Table 17 on the next page shows the test plan for the **FMSStore_Cart.ShoppingCart** class.

Table 17: Test Plan for Unit Testing the FMSStore_Cart.ShoppingCart Class

Scenario 1		Class FMSStore_Cart.ShoppingCart
Priority		High
Comments		
1.1	High	Public Function GetByKey(ByVal AccountID As Integer, ByVal SKU As Integer) As ADODB.Recordset
1.2	High	Public Sub Add(ByVal AccountID As Integer, ByVal SKU As Integer)
1.3	High	Public Sub Buy(ByVal AccountID As Integer)
1.4	High	Public Function ListByAccount(ByVal AccountID As Integer) As ADODB.Recordset
1.5	High	Public Sub SetQuantity(ByVal AccountID As Integer, ByVal SKU As Integer, ByVal Quantity As Short)
1.6	High	Public Function TotalByAccount(ByVal AccountID As Integer) As ADODB.Recordset
1.7	High	Public Sub EmptyShoppingCart(ByVal AccountID As Integer)

Table 18 shows a sample test case for unit testing the class **FMSStore_Cart.ShoppingCart**. Note that there is an additional column, named Test OK (Y/N), which is not included in this table because it has no entries.

Table 18: Sample Test Case for the FMSStore_Cart.ShoppingCart Class

Test case	Priority	Condition to be tested	Execution details	Datarequired
1.1a	High	Public Function GetByKey(ByVal AccountID As Integer, ByVal SKU As Integer) As ADODB.Recordset - All parameters within valid range	All input parameters in the valid range. **Input Parameters:** AccountID = 5249 SKU = 1004009 **Expected Output:** Recordset with the following values in the fields Quantity = 1 SKU = 1004009 Price = 29.95 Description = The Secrets of Investing in Technology Stocks **Actual Output:**	NUnit Test Case: GetByKey _Valid

Test case	Priority	Condition to be tested	Execution details	Datarequired
1.1b	High	Public Function GetByKey(ByVal AccountID As Integer, ByVal SKU As Integer) As ADODB.Recordset - AccountID parameter higher the valid range	AccountID parameter higher than the valid range. **Input Parameters:** AccountID = 5250000 SKU = 1004009 **Expected Output:** Empty Recordset **Actual Output:**	NUnit Test Case: GetByKey _Account InValid

The objective of black box testing is to test for functional equivalence at the application level. Therefore, the test plans and test cases for black box testing were taken from the use case analysis of the FMStocks 2000 application. Table 19 shows the test plan for black box testing of the login use case.

Table 19: Test Plan for the Black Box Testing of the Login Use Case

Scenario 1		To test the functionality of Login Procedure For Existing User
Priority		High
Comments		
1.1	High	To test that both Login and Home page are displayed in correct format. That is, all links, the font, and the contents are same as those of the existing application in Visual Basic 6.0.
1.2	High	To test that if user enters a valid e-mail name and password, the user is redirected to the home page.
1.3	High	To test that if SQL Server is not running, the following message is displays: “Cannot open database connection.”
1.4	High	To test if the entered e-mail name and/or password are invalid, the following message displays: “Invalid e-mail and password combination. Please try again.”

Table 20 on the next page shows a sample test case for black box testing of the login functionality/use case. Note that the test case also includes the following columns: Data Required, Actual Result, and Test OK (Y/N). They are not included in this table because they contained no information in this case.

Table 20: Sample Test Case for Black Box Testing the Login Functionality

Test case	Priority	Condition to be tested	Execution details	Expected results
1.1	High	To test that both login and home page are displayed in the correct format. That is, all links, the font, and the contents are the same as those of the existing application in Visual Basic 6.0.	Compare the login and home page with the corresponding pages in the Visual Basic 6.0 version of FMStocks 2000 and verify the following points: Content General look and feel of the Web page Location of input fields Font Links to other pages Functionality	Users should not feel any difference in the look and feel, fonts, functionality, content, and so on, between the login and home pages in the upgraded version and the corresponding pages in the Visual Basic 6.0 version of FMStocks.
1.2	High	To test that if user enters a valid e-mail name and password, the user is redirected to home page	Enter the following e-mail name and password on the login page E-mail: ta450 Password: ta	The user should be able to enter the site and should be redirected to the home page.

The objective of white box testing was to find the failure scenarios in the code that may have been missed in black box testing and to make sure that the upgrade issues have been properly resolved. White box testing consisted of the following tasks:

- Code review for detecting failure scenarios, especially in the cases of loops, conditional statements, and interoperability with the COM components that may have been missed in black box testing.
- Code review to verify whether exception handling/logging was implemented correctly.
- Code review to find additional test scenarios for unit testing.
- Review whether the upgrade issues were resolved correctly.
- Profiling and testing the code for intermediate results that may be critical.

The following subsections explain the errors/issues that arose during unit testing and black box testing.

Interface Not Upgraded Correctly

In the Visual Basic 6.0 code for FMStocks 2000, the **ShoppingCart** class module of the **FMStore_Events** project defines an interface as follows.

```
' This is the event interface and has no implementation because it exists
' only to define the interface.
Public Sub ExecuteBuy(ByVal AccountID As Long)
End Sub
```

This is how an interface is defined in Visual Basic 6.0, as an empty class module with method definitions that have no body. In .NET, an interface is defined by using the interface keyword. However, when the code was upgraded by the upgrade wizard, it could not differentiate the **ShoppingCart** class module from other class modules and upgraded it as follows.

```
Option Strict Off
Option Explicit On

<System.Runtime.InteropServices.ProgId(_
    "ShoppingCart_NET.ShoppingCart")> Public Class ShoppingCart

 ' This is the event interface and has no implementation because
 ' it exists only to define the interface.

 Public Sub ExecuteBuy(ByVal AccountID As Integer)
 End Sub
End Class
```

In the original FMStocks 2000 code, the class module **ShoppingCart.cls** of the project **FMStore_EvtSub2** implements the interface as follows.

```
Option Explicit

Implements FMSStore_Events.ShoppingCart

Private Sub ShoppingCart_ExecuteBuy(ByVal AccountID As Long)
    Dim objSC As FMSStore_Bus.ShoppingCart

    Set objSC = New FMSStore_Bus.ShoppingCart
    objSC.EmptyShoppingCart AccountID

End Sub
```

However, the upgraded .NET code defines and implements the interface in the same class module as follows.

```
Option Strict Off
Option Explicit On

Public Interface _ShoppingCart
End Interface

<System.Runtime.InteropServices.ProgId( _
    "ShoppingCart_NET.ShoppingCart")> _
Public Class ShoppingCart
    Implements _ShoppingCart
    Implements _ShoppingCart

    Private Sub ShoppingCart_ExecuteBuy(ByVal AccountID As Integer)
        Dim objSC As _ShoppingCart

        objSC = New FMSStore_Bus.ShoppingCart
        objSC.EmptyShoppingCart(AccountID)
    End Sub
End Class
```

Request Changed to Response During Upgrade

The original FMStocks code contains the following ASP code in the file t_head.asp.

```
if Request.Cookies("Account") <> "" then
g_AccountID = Request.Cookies("Account")
else
g_AccountID = 0
end if
```

This code was upgraded to the following in t_head.aspx by the ASP to ASP.NET Migration Assistant.

```
If Not IsNothing(Request.Cookies.Item("Account")) Then
g_AccountID = IIf(IsNothing(Response.Cookies.Item("Account").Value), "",
Request.Cookies.Item("Account").Value)
Else
g_AccountID = 0
End If
```

This upgraded code generated the "Input String was not in the Correct Format" error because the **Account** value is taken from the **Response** object instead of the **Request** object. Thus, the preceding code had to be modified to the following.

```
If Not IsNothing(Request.Cookies.Item("Account")) Then
g_AccountID = IIf(IsNothing(Request.Cookies.Item("Account").Value), "",
Request.Cookies.Item("Account").Value)
Else
g_AccountID = 0
End If
```

Exception While Passing Parameter Using ByRef

In code-behind files found in ASPX pages, an exception for some function is thrown if the parameters are being passed by reference. For example, the **Item(ByRef text As String, ByRef align As String)** function in PortFolio.aspx throws an exception that it is either not supported by its provider or is read-only.

The resolution for this problem was to pass these parameters by value and not reference.

Type Format Exception for Recordset

In the upgraded ASPX pages, when **recordset("<fieldname>")** was used with **Response.Write** or in an assignment statement a "Type Format Exception" resulted. This issue was resolved by changing **recordset("<fieldname>")** to **recordset("<fieldname>").Value** in **Response.Write** or assignment statements.

Response.Cookies.Item("Account").Value

The following ASP code appears in the t_head.asp file.

```
<% if Request.Cookies("Account") <> "" then %>
```

This code was converted to ASP.NET by the ASP to ASP.NET Migration Assistant as follows.

```
<%If Not IsNothing(Request.Cookies.Item("Account")) Then%>
```

There was a change in the behavior because of this change. The Default.aspx page includes the page t_head.aspx. In the Logout.aspx page, the value for the **Account** item was set as an empty string ("") as it was in the corresponding ASP page.

```
Response.Cookies.Item("Account").Value = ""
```

After setting the value of the **Account** item to "", the Logout.aspx page redirects the user to the Default.aspx page where the value "" of the **Account** item is validated using the condition statement **IsNothing()**. However, during testing the value "" of the **Account** item was passing the comparison condition and as a result, an exception was thrown. This issue was resolved by setting the value of the **Account** item to zero instead of "" in Logout.aspx as shown here.

```
Response.Cookies.Item("Account").Value = 0
```

Err.Number 5 Occurred When Redirecting to Another ASPX Page

When an ASPX page in the upgraded FMStocks 2000 redirected the user to another ASPX page, an error (Err.Number=5, Err.Description="Thread was being aborted.") occurred.

The error was caught by the error handler block and an error message was displayed instead of displaying the new ASPX page FMStocks was redirecting to.

The error originated in the following line of the converted ASPX page.

```
If Err.Number <> 0 Then
```

To resolve the issue, the line was modified to the following.

```
If Err.Number <> 0 and Err.Number <> 5 Then
```

The functional testing of FMStocks 2000 required an effort of 46 work hours. Extra effort was required for run-time error corrections for ASPX pages.

Summary

The complete upgrade of FMStocks 2000 required an effort of approximately 95 work hours. The final cost of the upgrade was $5,023.

The reason for the increase in the cost was that the upgrade of ASP to ASPX pages exceeded the estimated costs by $300 because of the extra effort required to upgrade the ASP pages to ASP.NET using the migration assistant. This effort could not be estimated by the Visual Basic Upgrade Assessment Tool; therefore, it was not taken into consideration in the reports it produced. Despite the extra manual effort required upgrade the ASP pages, the migration assistant still saved time compared to the time it would have taken to completely rewrite the ASP pages to ASPX pages.

Ultimately, using the combination of the assessment tool, the upgrade wizard, and the Visual Basic 6.0 to Visual Basic.NET guide, upgrading the Visual Basic 6.0 version of the FMStocks 2000 application to Visual Basic .NET and ASP.NET proved to be successful and efficient.

More Information

For more information about unit testing with NUnit, see the NUnit Web site: *http://www.nunit.org*.

Index

A

B

C

D

E

F

G

H

N

S

T

U

V

W

X