Microsoft®

Deploying .NET Applications Lifecycle Guide

patterns & practices
proven practices for predictable results

Contents

1

Introducing Deployment of .NET Applications

Assuming you and your team have embraced the new development capabilities of the Microsoft® .NET Framework and Microsoft Visual Studio® .NET development system, you will have undergone something of a shift in how you develop modern distributed applications. You will have encountered new technologies, such as the .NET Framework class library and the common language runtime, and you will have become familiar with new concepts such as working with assemblies, manifests, configuration files, serviced components, and so on. Having made the move to developing for the .NET Framework, you will need to complete the transition by making your new applications available to the end users for whom you develop them. In short, you will need to deploy your solutions from the development environment, so that they can be used in production.

Understand the Deployment Process

The complete application development lifecycle includes many different processes, such as the actual coding of the solution, the management of source control in a team development environment, the build process, the test process, the final packaging and staging, the deployment process, and often the upgrade of applications after they are installed in the production environment. All of these processes are to some extent interrelated, but this guide focuses on some specific areas. Figure 1.1 on the next page illustrates the interrelated nature of the processes included in the application development lifecycle. The shaded areas depict the focus of this Deployment guide.

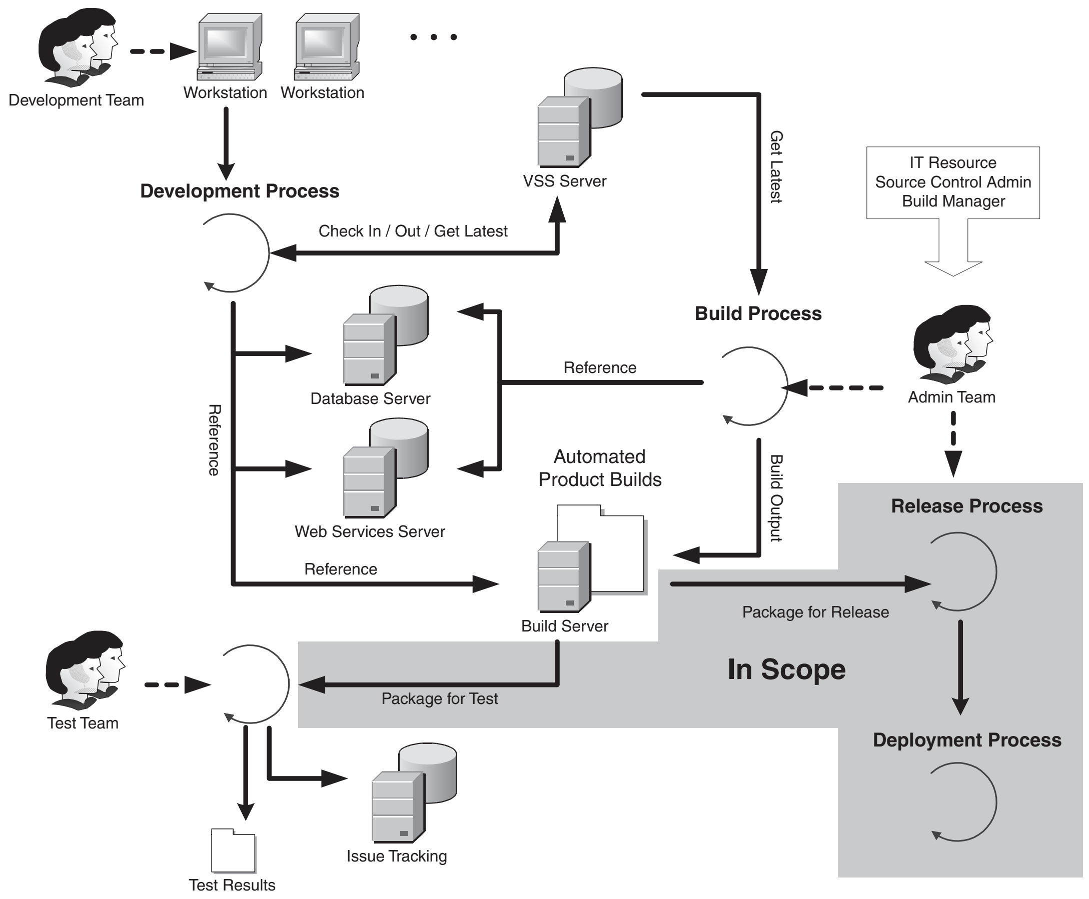


Figure 1.1
The Deployment Process

The main focus of this guide is, of course, the deployment of applications. For example, this guide does not delve into issues of managing source control for teams of developers. (For detailed guidance about setting up and working in a team environment, see "Team Development with Visual Studio .NET and Visual SourceSafe®" on MSDN at *http://msdn.microsoft.com/library/en-us/dnbda/html /tdlg_rm.asp*). However, this guide includes discussions other than the deployment of .NET applications into the production environment. For example, although this guide does not discuss testing methodology, it does include recommendations for deploying solutions from development computers to test environments. Similarly, this guide also provides recommendations for deploying to staging servers as part of the release process.

Manage the Different Deployment Environments

A successful deployment relies on more than development of the application and the final packaging process. For example, solutions of any substantial complexity need to be tested in a controlled environment, and the deployment process often benefits from the use of staging servers. You need to manage the deployment of your solutions between these environments, as well as to the final production environment. The following list briefly describes the characteristics of each environment, and explains how to use them to ensure a successful deployment of your solution:

- **Development environment**. This environment, as its name implies, is where your developers devote most of their coding and development effort. Because your developers require sophisticated development tools, such as Visual Studio .NET and various software development kits (SDKs), this environment does not usually resemble the one where your applications will eventually be installed. However, the development environment does affect the deployment of your applications. Your solutions are typically compiled and built in this environment, as well as packaged for distribution to the other environments. As such, you should be aware that solutions which run without problems on development computers do not necessarily run as smoothly in other environments. The most common causes of this problem include:
  - Dependencies required by your solution (such as assemblies, COM objects, the .NET Framework classes, and the common language runtime) are present on the developer computers, but they may not be included or installed successfully in other environments. Some of these dependencies are present on your development computers simply because they are provided by the development tools and SDKs, so you need to be aware of exactly what needs to be deployed to ensure successful installation into other environments.
  - Application resources (such as message queues, event logs, and performance counters) are created on developer workstations and servers, but they might not be successfully recreated or configured as part of the deployment to other environments. You need to thoroughly understand your application's architecture and any external resources that it uses if you are to deploy your solution successfully.
  - Applications are configured to use resources such as development database servers and Web services while they are being developed. When they are deployed to another environment, these configuration settings need to be updated so that they use the resources appropriate for that particular environment. Because many of these settings are stored in configuration files within the new .NET Framework, you need to manage the process of updating or replacing these configuration files as you deploy your solution from one environment to another.

These issues are among some of the most important ones that this guide addresses.

- **Test environment**. This environment should be used to test the following:
  - The actual deployment process, such as running Windows Installer files, copying and configuring application files and resources, or other setup routines. You (or your test team) need to verify that application files and resources are installed to the correct locations, and that all configuration requirements are met, before testing the solution's functionality.
  - After you are satisfied that the application installs correctly, you (or your test team) can then perform tests that verify the solution functions as expected.

  To be certain that your tests have meaningful implications for how your solution will install and function in the live production environment, you should ensure that test computers resemble your production computers as closely as possible. They should not have the development tools installed that you used to produce the solution, because that could mask potential problems that will then only become apparent when you roll out your solution to the end users.
- **Staging environment**. This environment should be used to house your solution after it has been fully tested in the test environment. It provides a convenient location from which to deploy to the final production environment. Because the staging environment is often used to perform final tests and checks on application functionality, it should resemble the production environment as closely as possible. For example, the staging environment should not only have the same operating systems and applications installed as the production computers, it should also have a similar network topology (which your testing environment might not have). Usually, the staging network mimics the production environment in all respects, except that it is a scaled-down version (for example, it may have fewer cluster members or fewer processors than your server computers).
- **Production environment**. This is the "live" environment where your solutions are put to work to provide benefits and functionality for the enterprise that uses them. This environment is undoubtedly the most complex, usually with many installed productivity, line-of-business and decision support systems used by different users in the organization. Consequently, it is the most difficult to manage and control; therefore, teams of administrators and information technology professionals are employed to ensure that users can take advantage of your applications along with solutions from other vendors. The main goal of this guide is to provide guidance for allowing you to ensure that your solutions are installed successfully in this environment. It also describes various approaches for upgrading applications after they are installed in this live environment.

Practice Deployment and Create Backup Plans

Some of the biggest problems during the lifecycle of a distributed application occur when your solution is rolled out to the production environment. The lack of rehearsal and planning for this event often leads to all of the involved parties attempting to breathe air into a lifeless application that does not work in the environment for which it was designed. Before your "going live" day, you need to rehearse the deployment in a clean environment with all of the participants present. You should take this opportunity to work out the little problems in a controlled environment without the added pressure of impacting production users. You can avoid the turmoil and embarrassment of a failed deployment by planning and documenting the production installation process and the contingency plans.

Create an environment that is a clean system. It should not know about your application and test deployment. Starting from a well-known configuration (a clean install of Microsoft Windows® 2000 Advanced Server, or whatever operating system maps to your production environment), begin to clearly document every step involved in the installation of the distributed application. Note the location of the application files and the installation points. Pretend that you and your developers will not be there to troubleshoot during this process. Ensure that you have included small test scenarios that the installers can use as checkpoints to verify that the installation is going well. Lastly, document how to uninstall your application, including registered components, registry keys, temporary files, and any user accounts created specifically for your solution, and include details of services installed, started, or stopped as part of the deployment.

Choose Appropriate Deployment Tools and Mechanisms

This guide provides introductions to setup tools and deployment mechanisms that you can use to deploy your .NET applications. The tools include Visual Studio .NET Setup and Deployment projects for building Windows Installer files, the Windows Installer Platform SDK, and other utilities that ship with Visual Studio .NET. The deployment mechanisms include Microsoft Systems Management Server, Active Directory® directory service group policies for software distribution, and simple copy mechanisms, such as XCOPY and Visual Studio .NET copy operations, as well as Microsoft Application Center for deploying to server environments. The main focus of this guide is to provide guidance for when using these tools and mechanisms is appropriate, although technical information and instructions are included in many places. This guide also provides many useful links to more information and step-by-step instructions to help you to implement your chosen deployment strategies.

Moving Forward with Your Deployment

Before you choose any specific deployment packaging, tools, or distribution mechanisms, you should read through this guide to identify issues that you will encounter as you implement your deployment strategies. As previously noted, this guide provides guidance on choosing specific tools and mechanisms and outlines the advantages and disadvantages of many different approaches. You should thoroughly understand the many deployment issues this guide describes, and take special note of the advantages and disadvantages of specific approaches to solving these issues. You will then be able to match appropriate solutions with your own specific deployment requirements.

2

Deploying the .NET Framework with Your .NET Applications

Your primary concern when deciding whether your applications require the Microsoft® .NET Framework is to determine which parts of your application target the common language runtime and the .NET Framework class library, and which do not.

Understand Your Application Dependencies on the .NET Framework

Different parts of your .NET application rely on the .NET Framework to function. You must understand which parts of your solution rely on the .NET Framework common language runtime and the .NET Framework class library.

Common Language Runtime

The common language runtime is the foundation of the .NET Framework. The common language runtime manages your code and objects at execution time, providing core services such as:

- Memory management.
- Thread management.
- Remoting.

Code that uses these core services by targeting the runtime is known as “managed” code, while code that does not target the runtime is known as “unmanaged” code.

In other words, managed code requires the .NET Framework be installed on the computer on which it runs.

.NET Framework Class Library

The .NET Framework class library is a comprehensive, object-oriented collection of reusable classes that you can use to develop applications. They provide access to system functionality such as:

- Security features.
- Graphical user interface (GUI) elements.
- Enterprise services (such as message queuing, load balancing, and transacted components).
- File input and output.
- Data access.

The .NET Framework types are the foundation on which .NET applications, components, and controls are built. The .NET Framework includes types that:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Invoke .NET Framework security checks.
- Provide data access.
- Provide rich client-side GUIs.
- Provide server-controlled, client-side GUIs (such as in the case of ASP.NET Web Forms).

Again, code that uses any of the classes from the .NET Framework class library requires the .NET Framework be installed on the computer on which it runs.

Deploy the .NET Framework for Windows Forms-based Applications

Microsoft Windows® Forms-based applications require that the client computer have the .NET Framework installed before they can operate. In Windows Forms-based applications, the logic and code used to build the GUI runs on the client computer. Windows Forms controls and menus are actually .NET Framework classes, as are the Windows Forms themselves. In addition, display effects, such as colors, borders, sizes, and positions are all controlled by accessing classes and enumerations from the .NET Framework class library.

Determine Whether the .NET Framework Is Required for Browser-based Applications

If your GUI is solely browser-based, and does not include .NET managed controls, you do not have to distribute the .NET Framework to your client computers.

Much of the logic and code used to build the user interface for a browser-based .NET application resides (and runs) on a Web server, rather than on the computer at which the user is sitting. The browser-based GUI is dynamically built by your ASP.NET application, which transmits Hypertext Markup Language (HTML) to the browser. The GUI may be supplemented with some client-side script to provide ease of usability, increased responsiveness, data-entry validation, and dynamic display effects, but the bulk of the interface is controlled by server-based code to which the client Web-browser does not have direct access. In effect, the client uses HTML, supplemented by dynamic HTML (DHTML), to present the user interface. Although the server-side code depends on the .NET Framework to build the GUI, it produces browser-agnostic HTML. Consequently, the client computer may not need to have the .NET Framework installed.

One important exception that you should be aware of is if your browser-based application includes managed controls. Managed controls are assemblies referenced from Web pages that are downloaded to the user's computer and executed upon demand. These controls can provide extra functionality to your application, much like Microsoft ActiveX® controls hosted in a Web-browser did in the past. Because managed controls rely on the .NET Framework to manage their execution, you need to ensure that the framework is installed on any client computers where this special type of control is used.

Note: Using managed controls introduces a number of other issues, such as browser and code security, with which you need to be familiar. A full discussion is outside the scope of this deployment guide. For more information, see "Writing Secure Managed Controls" on MSDN (*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingsecuremanagedcontrols.asp*).

Deploy the .NET Framework to Web Servers

If you use managed code or the .NET Framework class libraries in your server-side code, then you must ensure that the target Web servers for your solution have the .NET Framework installed before your application will function properly. The distributed nature of modern Internet or intranet-based applications means that

Web servers often perform a number of different roles in your .NET solutions. You might use Web servers to provide any (or all) of the following features for your distributed .NET application:

- ASP.NET Web Forms for the GUI
- Web Services for re-usable, Web-based functionality
- Transfer of Extensible Markup Language (XML) or text-based data
- Intermediate XML or text-based processing
- Connectivity and communication across computer and network boundaries

The actual deployment of Web applications is discussed in Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for Your .NET Application," and how to manage the deployment of Web Services is discussed in Chapter 4, "Deployment Issues for .NET Applications."

Deploy the .NET Framework to Business Logic Servers

Most modern .NET applications implement a physically separate tier (or tiers) for encapsulating business logic. The advantages of doing so include ease of manageability and scalability, among others. Traditionally, business logic was encapsulated in COM objects and accessed by way of local automation, remote automation, distributed COM (DCOM), or more recently by way of the COM+ services of Windows 2000. Naturally, as you develop your presentation services and GUI within the .NET Framework, you will also want to take advantage of the many features of the framework for your business objects.

In effect, your application will include your own .NET classes and objects that encapsulate business logic.

Note: This guide is not saying that you won't use traditional COM objects that already provide you with tried and tested functionality, but rather that you will almost certainly want to migrate your objects to make use of the .NET Framework at some point. You might decide to migrate all of your objects at the same time, as you develop your .NET application, or alternatively, you might want to roll out your application while still using legacy COM objects with a plan to replace those objects over time. In fact, the .NET Framework provides comprehensive support that enables managed code to interoperate with COM (known as COM Interop). For more information, see "Microsoft .NET/COM Migration and Interoperability" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cominterop.asp*).

Deployment issues specific to COM Interop are discussed in Chapter 4, "Deployment Issues for .NET Applications," but whether you encapsulate your business logic in purely .NET objects or use a mixture of .NET and COM objects, you need to ensure that your business servers have the .NET Framework installed if .NET is being used in your business tier.

Determine Whether You Need to Deploy the .NET Framework to Database Servers

For scalability, manageability, stability, and fault-tolerance, data tiers, in modern distributed applications, are often physically separated from the other tiers in a typical solution.

Although products such as Microsoft SQL Server™ 2000 and Microsoft Analysis Services are often described as constituent parts of the Microsoft .NET Server family, they do not actually require the .NET Framework to be installed. In other words, if your application architecture includes a physically separate data tier, .NET Framework does not need to be installed on your database servers and analysis servers. However, if you need .NET assemblies to execute on your database servers, .NET Framework needs to be installed on those computers.

Deploy the Redistributable .NET Framework When Required

After determining which tiers in your distributed .NET application require the .NET Framework, you need to decide how you are going to ensure that the framework is installed in your production environment.

Obtain the Redistributable .NET Framework

To ensure that the .NET Framework is installed as required by the various tiers in your application, you should obtain and deploy the redistributable .NET Framework, Dotnetfx.exe. This redistributable package is freely available from Microsoft, subject to certain license agreements. You must have a validly licensed copy of Microsoft Visual Studio® .NET development system or the Microsoft .NET Framework software development kit (SDK) if you want to redistribute the .NET Framework.

For version 1.0 of the .NET Framework, the redistributable package consists of a single executable file, named Dotnetfx.exe. This installer file contains the common language runtime and .NET Framework class libraries necessary to run .NET Framework applications.

You can download Dotnetfx.exe from the Microsoft Download Center (*http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/829/msdncompositedoc.xml*) or the Windows Update Web site (*http://www.windowsupdate.com*). Alternately, you can obtain the redistributable package on a product CD or DVD. Dotnetfx.exe is available on the .NET Framework SDK CD in the \dotNETRedist directory. It is also available on the Microsoft Visual Studio .NET Windows Component Update CD in the \dotNetFramework directory, and on the Microsoft Visual Studio .NET DVD in the \wcu\dotNetFramework directory.

Note: If you download the Microsoft .NET Redistributable Package from MSDN, you receive a file named Dotnetredist.exe. This file contains Dotnetfx.exe. In order to extract Dotnetfx.exe, double-click Dotnetredist.exe. You are prompted to save the extracted files on your computer. The extracted file is Dotnetfx.exe.

Install Localized Versions of the .NET Framework When Required

When installing Dotnetfx.exe on a computer running the Windows 98 operating system, you must install the localized version of Dotnetfx.exe that corresponds to the localized version of Windows 98 running on the computer. For example, you must install the German version of Dotnetfx.exe on a computer running the German version of Windows 98. This limitation applies only to Windows 98. You can install any language version of Dotnetfx.exe on any language version of Windows Millennium Edition (Windows Me), Windows 2000, Windows NT 4.0, Windows XP, or the Windows .NET Server family.

Localized versions of the redistributable Dotnetfx.exe file are available from the Download Center on MSDN (*http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/829/msdncompositedoc.xml*).

Ensure that You Deploy the .NET Framework to Supported Platforms

To install Dotnetfx.exe, you must have one of the following operating systems with Microsoft Internet Explorer 5.01 or later installed on the target computer:

- Microsoft Windows 98
- Microsoft Windows 98 Second Edition
- Microsoft Windows Me
- Microsoft Windows NT 4.0 (Workstation or Server) with Service Pack 6a
- Microsoft Windows 2000 (Professional, Server, or Advanced Server) with the latest Windows service pack and critical updates available from the Microsoft Security Web site (*http://msdn.microsoft.com/isapi/gomscom.asp?Target=/security*).
- Microsoft Windows XP (Home or Professional)
- Microsoft Windows .NET Server family

Note: ASP.NET applications are not supported on Windows 98, Windows Me, or Windows NT 4.0.

The minimum hardware requirements and recommendations for your computers vary, depending upon the role each computer plays in your solution. Table 2.1 describes the minimum and the recommended RAM and processor specifications for both client and server machines.

Table 2.1: Hardware Specifications for Dotnetfx.exe

Computer	RAM (Min/Recommended)	Processor (Min)	Disk Space (Required/For Installation)
Client	32 MB / 96 MB or more	Pentium 90 MHz	70 MB / 160 MB
Server	128 MB / 256 MB or more	Pentium 133 MHz	370 MB / 600 MB

Determine Other Dependencies for the .NET Framework and Your Application

As you plan the deployment of your application and the .NET Framework, you need to be aware of other software dependencies that your target computers require for your solution to function properly. Typical examples of such dependencies include:

- **Microsoft Data Access Components (MDAC)**. Depending on your data access strategy, you may need to ensure that MDAC is installed on the client computers, the business servers, the Web servers, or the database servers. MDAC 2.6 or later is required by the .NET Framework, and MDAC 2.7 is recommended.

  You should ensure that you deploy the same version of MDAC to the production environment that you use in development. Visual Studio .NET ships with version 2.7, so it is likely that your developers will have developed and tested your solution with that version. If you subsequently deploy the application to computers where MDAC 2.6 is installed rather than MDAC 2.7, you run the risk of encountering unforeseen problems.

  The latest version of MDAC is available for download at Microsoft's Universal Data Access Web Site (*http://www.microsoft.com/data*).
- **Microsoft Internet Information Services (IIS)**. If your solution includes a Web tier, you need to ensure that IIS and the latest security patches are installed on your target computer(s). IIS can be installed from the operating system installation CD or DVD, and the latest security patches can be downloaded and installed from the Windows Update site (*http://www.windowsupdate.com*).
- **Microsoft Windows Management Instrumentation (WMI)**. If your solution uses WMI features, you need to ensure that the core WMI is installed on your target computers. WMI is included as a standard feature in Windows XP, Windows 2000, and Windows Me. Additionally, it is automatically installed by Visual Studio .NET on Windows 95, Windows 98, and Windows NT 4.0. However, if

your target computers do not have the core WMI installed, a setup package can be downloaded and installed from MSDN Code Center (*http://msdn.microsoft.com/downloads/?url=/downloads/sample.asp?url=/msdn-files/027/001/576/msdncompositedoc.xml*).

Note: Although it is not really a deployment issue, it is worth mentioning that you can download and install the Visual Studio .NET Server Explorer Management Extensions if you are developing WMI-enabled applications. The Server Explorer Management Extensions are a set of development tools that provide browsing and rapid application development (RAD) capabilities for building WMI-enabled applications. For more information, and to download these management extensions, see "Managing Windows Management Instrumentation (WMI) Data and Events with Server Explorer" on MSDN (*http://msdn.microsoft.com/library/en-us/WMISE/wmioriManagingWindowsManagementInstrumentationWMIDataEventsWithServerExplorer.asp*).

Rather than manually ensuring dependencies such as MDAC and WMI are correctly installed before you roll out your solution to the production environment, you might want to build checks into your setup routines to detect whether these dependencies are present. If they are not installed, you can automatically install them along with your application, using Windows Installer. Windows Installer is discussed in further detail in Chapter 3, "Visual Studio .NET Deployment Projects."

Ensure that You Adhere to the Redistributable .NET Framework License Agreements

If you choose to distribute Dotnetfx.exe with your application, you must have a validly licensed copy of the Microsoft .NET Framework SDK or Visual Studio .NET, and you must agree that any further use or distribution of Dotnetfx.exe associated with your application is defined in the Microsoft .NET Framework SDK end-user license agreement. If you do not have a validly licensed copy of the Microsoft .NET Framework SDK or you do not agree to these terms and conditions, you are not authorized to distribute Dotnetfx.exe and your use of Dotnetfx.exe is subject to the terms of the end-user license agreement contained in the Dotnetredist.exe.

Choose a Mechanism for Deploying the .NET Framework

After you identify the target computers in your production environment, and ensure that they meet the minimum hardware and software requirements for installing your application and the .NET Framework, you need to decide upon a mechanism for actually ensuring that the framework is present as you roll out your solution.

Must be Administrator to Install the Framework

Dotnetfx.exe installs significant run-time components onto a target computer; therefore, the user initializing the .NET Framework setup must have administrator privileges for that computer (or be able to perform the installation with elevated privileges). This requirement has implications for how you choose to deploy the .NET Framework. For example, using Microsoft Active Directory® directory service software deployment to install the package over the network allows you to ensure that the package installs with elevated privileges. Similarly, using Microsoft Systems Management Server (SMS) allows you to install the framework with the required permissions. In contrast, if you expect your users to initialize the setup of the .NET Framework, either locally or over the network, then you (or your systems administrator) must ensure that the user has administrator privileges on his or her local computer or can run the installation with elevated privileges.

Distributing the .NET Framework with Your Application

If you are developing and packaging your .NET application for general sale, or perhaps for installation onto a number of different customer networks, you might not have any knowledge of the infrastructure where your application is to be deployed. A typical solution for this type of scenario is to include the redistributable .NET Framework with your application's setup files. Although you cannot include the .NET Framework in your Windows Installer files, you can create a bootstrapping setup program that includes checks to determine whether the .NET Framework is already installed—if it is not present, you can automatically launch Dotnetfx.exe to install the framework before continuing with your application setup. A sample application (Setup.exe) that performs these tasks can be downloaded from MSDN (*http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/001/830/msdncompositedoc.xml*).

The Setup.exe bootstrapping application sample demonstrates how to create a setup program that verifies whether the .NET Framework is installed. If the sample determines that the .NET Framework is not present, it performs a silent install of the framework included with your application, and then runs Windows Installer.

You can use the sample for your own applications by downloading the precompiled application and simply modifying the [Bootstrap] section of the Settings.ini file that ships with the sample. Alternatively, you can download the source code for the sample and modify it to meet your specific needs. For more information about how to work with this sample, see "Redistributing the .NET Framework" on MSDN (*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp*).

Distributing the .NET Framework Separately from Your Application

If you are rolling out your application to your own corporation, or if you have good knowledge of your customer's infrastructure, you might choose to not include the .NET Framework with your application's setup files. It might be more convenient or appropriate to deploy the .NET Framework onto target computers before installing your application. This is a particularly good approach if you know that more than one .NET application will be installed for your users—you don't want to redistribute the framework with every application.

Distributing the framework separately from your application is also a good idea if you know that the end users will not have the required permissions to complete the installation — in this case, you can deploy Dotnetfx.exe using Active Directory or SMS to ensure that the appropriate privileges are in place for a successful installation.

For more information about using Active Directory software deployment to install the .NET Framework and make modifications to the Settings.ini file, see the "Deploying the .NET Framework using Active Directory" topic in "Redistributing the .NET Framework" on MSDN (*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp*).

For more information about using SMS to deploy the .NET Framework, see the "Deploying the .NET Framework using Systems Management Server" topic in "Redistributing the .NET Framework" on MSDN (*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp*).

If your corporation or your customers do not want to use an electronic software distribution tool, and you know that the end users have the required privileges to install the .NET Framework, you can have your users run Dotnetfx.exe locally or from a network share prior to installing your application. Again, this may be an appropriate choice if you know that more than one .NET application will be installed with your application or in the future.

Deploying Visual Studio .NET

Depending on how you manage the software development cycle, your environment may include any (or all) of the following:

- Development computers (workstations and servers)
- Test computers (workstations and servers)
- Staging servers
- Production computers (servers and end-user workstations)

You must decide whether to install the Visual Studio .NET suite of applications and utilities on these computers.

Table 2.2 provides guidance on whether installing Visual Studio .NET is required or recommended on the different computers that participate in the software development process.

Table 2.2: Visual Studio .NET Deployment Targets

Computer type	Visual Studio .NET Required/Recommended?
Development Computers	Your developers will certainly need to use Visual Studio .NET on their development workstations. You can actually have your developers run Visual Studio .NET across the network with Terminal Services, but it is more typical to have the suite of applications installed locally. Whether you should include Visual Studio .NET on your development servers depends on how you write and debug code. For example, if you develop Web applications on a development server, but you actually connect to that Web server from a development workstation, then you might not need Visual Studio .NET installed on the server — you can install the .NET Framework on the servers and use the remote debugging tools provided by your local copy of Visual Studio .NET. On the other hand, if you typically develop your solutions by coding on the development server itself, then you, of course, need Visual Studio .NET installed there.
Test Computers	Because the test environment should mimic the production environment as closely as possible, it is not usually advisable to install Visual Studio .NET on test computers — having Visual Studio .NET installed on test servers can mask errors that would otherwise appear in a production environment. However, it can be appropriate in some circumstances to allow testers to have Visual Studio .NET on their test *workstations* — they will often want to use the debugging tools provided with Visual Studio .NET to report more fully on errors that they encounter. As with the development workstations, they can use the remote debugging tools from their local copy of Visual Studio .NET to analyze the code on a test server, without introducing the possibility that Visual Studio .NET masks problems on those test servers.
Staging Servers	Staging servers are used as sources for deploying applications to the live production environment. Because final tests are often carried out just before an application is deployed from staging servers, they should resemble the production computers as closely as possible. Therefore, it is not usually recommended that staging servers have Visual Studio .NET installed.
Production Servers	Most corporate networks do not have Visual Studio .NET installed on production servers and end-user workstations. There should not be any source code deployed on these computers and no development should occur on these computers. You should install only the .NET Framework without the Visual Studio .NET suite of development applications.

After you identify which computers should have Visual Studio .NET installed, you need to plan for how to deploy it for your developers. Like any other software, you can deploy Visual Studio .NET in a number of different ways, including:

- Allowing developers to install locally from the Visual Studio .NET product CDs or DVD.
- Creating an administrative install for Visual Studio .NET on a network share, and have developers connect and run the setup over the network.
- Deploying Visual Studio .NET with an electronic software distribution tool, such as SMS.

Consider Using SMS to Deploy Visual Studio .NET

You should consider deploying Visual Studio .NET with SMS to ensure that:

- Your developers all work with a consistent installation of Visual Studio .NET.
- Service packs and fixes are applied consistently to your developer computers.
- Installation does not encounter any permissions problems if your developers are not local administrators.

For step-by-step instructions about deploying Visual Studio .NET with SMS, see article Q309657, "HOW TO: Deploy Visual Studio .NET by Using Systems Management Server," in the Microsoft Knowledge Base (*http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q309657*).

References

For more information about deploying the redistributable .NET Framework, review the articles and Web sites in Table 2.3.

Table 2.3: Further Reading

Name	Location
.NET Framework Deployment Guide	*http://msdn.microsoft.com/library/en-us/dnnetdep/html/dotnetframedepguid.asp*
Redistributing the .NET Framework	*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp*
.NET Framework Redistributable Package Technical Reference	*http://msdn.microsoft.com/library/en-us/dnnetdep/html/dotnetfxref.asp*
Microsoft .NET Framework Redistributable EULA	*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redisteula.asp*
Addendum to the End User License Agreement for Microsoft Products	*http://msdn.microsoft.com/library/en-us/dnnetdep/html/addendeula.asp*
Using Visual Studio .NET to Redistribute the .NET Framework	*http://msdn.microsoft.com/library/en-us/dnnetdep/html/vsredistdeploy.asp*

3

Visual Studio .NET Deployment Projects

In addition to providing a powerful rapid application development environment with programming languages and tools that target the Microsoft® .NET Framework, Microsoft Visual Studio® .NET development system includes new features that allow you to package and deploy your solution to the production environment.

This chapter outlines the new setup and deployment features available with Visual Studio .NET.

Note: Guidance for determining whether the features described in this chapter are suitable for your specific environment is provided in Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Application."

Use Visual Studio .NET Setup and Deployment Projects

Unlike earlier versions of Visual Studio, you can create setup programs for different types of applications in the Visual Studio .NET integrated development environment (IDE). Visual Studio .NET provides projects for creating setup applications for applications that are based on the Microsoft Windows® operating system, Web applications, components, and Internet download packages.

Manage Your Setup Routines

To take advantage of the application management features of Microsoft Windows Installer for installation, repair, and removal, you can use Visual Studio .NET to create Windows Installer files to install and manage your .NET solutions.

Microsoft Windows Installer is an installation and configuration service that ships as part of Windows 2000, Windows Millennium Edition (Me), and Windows XP. It is also available for Windows 95, Windows 98, and Windows NT 4.0. Windows Installer is based on a data-driven model that provides all installation data and instructions in a single package. With Windows Installer, each computer maintains a database of information about every application that it installs, including files, registry keys, and components. When an application is uninstalled, the database is checked to make sure that no other applications rely on a file, registry key, or component before removing it. This prevents the removal of one application from breaking another.

Windows Installer also supports self-repair. This is the ability for an application to automatically reinstall missing files that may have inadvertently or maliciously been deleted by the user. In addition, Windows Installer provides the ability to roll back an installation. For example, if an application relies on some prerequisite software and that software is not found during installation, the setup can be aborted and the computer returned to its pre-installation state.

Create Windows Installer Files for Your Application

You can create four different types of setup projects with Visual Studio .NET setup and deployment projects. They are:

- Standard setup projects.
- Web setup projects.
- Merge module projects.
- CAB projects.

Both standard and Web setup projects are compiled by Visual Studio .NET into Windows Installer files (.msi files) that are compatible with the Windows Installer service.

Merge modules are used to package components rather than complete applications. Developers who use your components in their applications can include your merge module in their Windows Installer files to ensure your component is installed correctly along with their other application files. Merge modules are discussed later in this chapter.

CAB projects are used to package Microsoft ActiveX® controls and managed controls that can be downloaded from a Web server to a Web browser. CAB projects are discussed later in this chapter.

Standard Setup Projects

Standard setup projects are typically used to install and manage client applications, business logic objects and Windows Service applications. Use standard setup projects when you want to do any or all of the following:

- Have a simple-to-use graphical user interface (GUI) installation program
- Integrate with the **Add/Remove Programs** item in Control Panel for:
  - Installing.
  - Uninstalling.
  - Adding or removing application features.
  - Repairing a broken installation.
- Have a robust setup routine that:
  - Rolls back the system to the state it was in prior to the start of installation if any part of the setup routine fails.
  - Rolls back the system to the state it was in prior to the start of installation if the user cancels the installation part-way through the setup routine.
- Require any (or all) of the following:
  - Registry manipulation
  - File associations to be created
  - Assemblies to be installed into the global assembly cache
  - Custom tasks to be run in the final stages of installation
  - Hardware and prerequisite software checks to be made prior to installation
  - Register assemblies for COM Interop

Some of these features are discussed later in this chapter.

Web Setup Projects

Web setup projects are used for packaging and deploying Web-based solutions, rather than client applications. Web setup projects differ from standard setup projects in that they install Web applications to a virtual root folder on a Web server rather the Program Files folder. The process of creating and compiling Web setup projects is very similar to that for standard setup projects. Designing and building Windows Installer files are discussed next—this information applies to both Web and standard setup projects.

Design Your Windows Installer Files

You can use the features provided by Visual Studio .NET to create powerful installers for your solution. The following topics describe each of these features.

Add Project Output Groups to your Setup Project

The recommended way to add the files needed by your application to the setup project is to add the project output groups, rather than individual files. Depending on the type of application that you are packaging, you can choose to include a number of different file types as you add your project output, such as the primary output, localized resources, content files, debug symbols, and so on. Which of these you choose depends on what you want to distribute, but the major advantage of adding project output groups is that Visual Studio can more easily detect dependencies for your application for project outputs compared to the need to manually manage which individual files get included.

Design the User Interface for Your Windows Installer Files

To define the steps that your users must take to install your application, you can specify that a number of predefined user interface dialog boxes be presented during installation to present or gather information. You specify the dialog boxes to be displayed in the User Interface Editor.

► **To open the User Interface Editor**

1. In Solution Explorer, select a deployment project.
2. On the **View** menu, point to **Editor**, and click **User Interface**.

In addition to using the dialog boxes provided with Visual Studio .NET setup and deployment projects "as is," you can customize them. To do this, you need to add a dialog box to your project and then modify that dialog box outside of Visual Studio .NET with a tool such as Orca.

For more information about each of the dialog boxes you can add to your Windows Installer, see "Deployment Dialog Boxes" on MSDN (*http://msdn.microsoft.com /library/en-us/vsintro7/html/vbconDeploymentDialogs.asp*).

For more information about Orca and the other Windows Installer Platform Software Development Kit (SDK) tools, see the "Windows Installer Platform SDK" section later in this chapter.

Manipulate Files and Folders with Your Installer

You can use the File System Editor to add project outputs and other files to a deployment project, to specify the locations where files will be installed on a target computer, and to create shortcuts on a target computer.

► **To open the File System Editor**

1. In Solution Explorer, select a deployment project.
2. On the **View** menu, point to **Editor**, and then click **File System**.

The File System Editor is divided into two parts: a navigation pane on the left and a detail pane on the right. The navigation pane contains a hierarchical list of folders that represent the file system on a target computer. The folder names correspond to standard Windows folders; for example, the Application folder corresponds to a folder beneath the Program Files folder where the application will be installed.

You can add custom folders and special folders to the installation. Special folders are folders in the File System Editor that represent predefined Windows folders. The physical location of Windows folders can vary from one computer to another: for example, the System folder may be located in C:\Windows on one computer, D:\Windows on another, and C:\Winnt on a third. Regardless of the physical location, Windows recognizes the folder as the System folder by reading special attributes. Using special folders in a deployment project allows you to choose a destination folder on a target computer without knowing the actual path to that folder.

Custom folders represent other folders on a target computer. Unlike special folders, custom folders do not necessarily depend on existing folders on the target computer, but rather allow you to create new folders at install time.

Note: For installation to succeed, the user running the Windows Installer must have the appropriate permissions for adding the files and folders that you specify in your setup project. This is not an issue for Windows 98 and Windows Me, but for Windows NT 4.0, Windows 2000, and Windows XP platforms the installation fails if the user does not have the required permissions. When a user who is not a member of the local administrators group runs a Windows Installer on these platforms, they are prompted that they may not have permissions to complete the installation and are offered the choice to run the installer as a different user (for example, as the local Administrator). To take advantage of this option, the user must provide the user name and password for the account with which they want to run the installation.

Manage the Registry Settings Required by Your Application

For your application and components to function correctly, they might rely on certain registry keys and values being present on the target computer. You can deploy these keys and values as part of your installer.

Registry keys can be added to a deployment project using the Registry Editor.

► **To open the Registry Editor**

1. In Solution Explorer, select a deployment project.
2. On the **View** menu, point to **Editor**, and then click **Registry**.

If a key doesn't exist in the registry of a target computer, it will be added during installation. Keys can be added beneath any top-level key in the Registry Editor. For instructions on how to add registry keys to your projects, see "Adding and Deleting

Registry Keys in the Registry Editor" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskAddingDeletingRegistryKeys.asp*).

The Registry Editor can be used to specify values for new or existing registry keys in the registry of a target computer. You can add string, binary, and DWORD values. The values are written to the registry during installation. The values you specify overwrite any existing values.

You can also use the Registry Editor to specify a default value for any registry key.

Registry keys and values can be added to a deployment project by importing a registry file (.reg) into the Registry Editor. This allows you to save time by copying an entire section of an existing registry in a single step. Registry files can be created using tools such as the Registry Editor (Regedit.exe) included with Microsoft Windows 2000.

Note: As with managing files and folders, the user running the installation must have the appropriate permissions to modify the registry; otherwise, installation fails when the Windows Installer attempts to create, modify, or delete registry keys and values. For example, the user must have local administrative privileges if your installer creates, deletes, or modifies keys and values in the **HKEY_LOCAL_MACHINE** hive. As mentioned previously, when a user who is not a member of the local administrators group runs a Windows Installer file, they are prompted that they may not have permissions to complete the installation and are offered the choice to run the installer as a different user (for example, as the local Administrator). To take advantage of this option, the user must provide the user name and password for the account with which they want to run the installation.

For instructions on how to specify values for registry keys, see "Adding and Deleting Registry Keys in the Registry Editor" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskAddingDeletingRegistryValues.asp*).

For more information about creating default registry values, see "Creating a Default Registry Value in the Registry Editor" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskCreatingDefaultRegistryValue.asp*).

For instructions on how to import registry files into your deployment projects, see "Importing Registry Files in the Registry Editor" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskImportingRegistryFiles.asp*).

Manage File Associations for Your Installer

If your solution includes (or creates) files of a certain type as part of its normal operation, then you might want those files to be associated with your application. For example, you might want any file that your application creates to open with your executable file if the user double-clicks the file in Microsoft Windows Explorer. The File Types Editor is used to establish file associations on the target computer, by associating file extensions with your application and specifying the actions allowed for each file type.

► **To open the File Types Editor**

1. In Solution Explorer, select a deployment project.
2. On the **View** menu, point to **Editor**, and then click **File Types**.

For more information about editing file associations, see "File Types Management in Deployment" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html /vbconTheAssociationsEditor.asp*).

Manage Dependencies for Your Installer

The setup and deployment projects in Visual Studio .NET automatically detect dependencies and add them to the deployment project whenever a project output group, assembly, or merge module is added to the project. For assemblies, all dependencies on other assemblies are detected. However, if the assembly references an unmanaged component (for example, a COM DLL), any dependencies of the unmanaged component will not be detected. Likewise, COM components added directly to a deployment project may have dependencies that are not detected.

Note: Rather than adding assemblies directly to a deployment project, it is best to add a project output group that contains the assembly. The deployment tools can more easily detect dependencies for a project output group.

You need to determine all of the possible dependencies for the COM component and include them in the deployment project. You should check the documentation for the component or contact the component's author to get a list of dependencies. Preferably, you can obtain a merge module (an .msm file) for the COM component to deploy the component and its dependencies—the developer who created the .msm file should already have defined and included the dependencies for their component and included them in the .msm file.

Deploy Shared Assemblies to the Global Assembly Cache

The global assembly cache is a computer-wide code cache provided by the .NET Framework that is used to store assemblies that need to be shared by several applications on the computer. To install an assembly to the global assembly cache, add the assembly or the project output group for the assembly to the Global Assembly Cache folder in the File System Editor. You can add your assemblies to the Global Assembly Cache folder with simple drag-and-drop operations, just as you would for any other folder in the File System Editor in your setup project.

The Global Assembly Cache folder is unlike the other folders in the File System Editor. It has no properties that are settable, and you cannot create shortcuts to the folder or to assemblies in the folder.

For more information about when assemblies should be deployed to the global assembly cache, see Chapter 4, "Deployment Issues for .NET Applications."

Include Custom Actions in Your Installer

Custom actions are Windows Installer features that allow you to run code in the final stages of the installation in order to perform actions that cannot be handled during the main setup routine. The code can be in the form of a DLL, an executable file, a script, or an assembly. For example, you might want to create a local database on the target computer during installation. You could create an executable file that creates and configures the database, and then add that executable file as a custom action in your deployment project.

The Custom Actions Editor in Visual Studio .NET is used to manage custom actions in a deployment project. Custom actions can be added and properties for the custom actions can be set. A deployment project can contain multiple custom actions.

Custom actions are often used to run installer classes to create and configure application resources. One restriction on using custom actions is that they cannot run on assemblies that will be installed into the global assembly cache. This means that all installer classes should be placed in private assemblies. One option is to include the installer classes in your executable file (if you have one). Another option, for ease of management, is to create a separate private assembly that contains only installer classes. This option groups your installer classes together and minimizes the number of assemblies that need to be added as custom actions in your setup project. Although this causes an assembly to be deployed that is technically not used after installation is complete, this does not cause any problems for your application and should be considered.

Custom actions are run after the actual installation is complete, so they do not have access to properties that are used to control installation. If you need to pass information from the installer to a custom action, you can do so by setting the **CustomActionData** property.

In addition, conditions can be placed on any custom action using the **Condition** property. This allows you to run different custom actions based on conditions that exist on a target computer during installation. For example, you might want to run different custom actions, depending on the operating system version on the target computer. Conditional deployment is discussed next.

For more information about using installer classes and running them as custom actions, see the "Use Installation Components to Create and Configure Application Resources" section in Chapter 4, "Deployment Issues for .NET Applications."

For more information about working with custom actions, see "Adding and Removing Custom Actions in the Custom Actions Editor" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskAddingRemovingCustomActions.asp*).

For more information about passing data from your installer to a custom action, see “Walkthrough: Passing Data to a Custom Action” on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxwlkWalkthroughPassingDataToCustomAction.asp*).

Manage Conditional Deployment with Your Installer

One of the most powerful features in Visual Studio .NET deployment is the ability to set conditions for installation, allowing you to customize installations on a case-by-case basis. For example, you might want to install different files based on operating system version, customize registry settings based on the value of an existing key, or even halt installation if a dependent application is not already installed on the target computer.

The deployment tools in Visual Studio .NET support conditional deployment in two ways: through launch conditions and through the **Condition** property.

Launch conditions are used to evaluate a condition on a target computer and to halt installation if the condition is not met. Launch conditions can be set to check for the operating system version, existence of files, registry values, Windows Installer components, the common language runtime, and Microsoft Internet Information Services (IIS). For example, you might set a launch condition to check for a specific operating system version such as Windows 2000—if it is not found, you could display a message informing the user of the requirement and then the installation will be halted. Launch conditions are specified in the Launch Conditions Editor.

Setting **Condition** properties is another way to manage conditional deployment. The **Condition** property is used to evaluate properties exposed by Windows Installer or set by other elements in the installer. For example, Windows Installer exposes properties for the operating system version during installation. If you want to install a file only on Windows 2000 or later, you set the **Condition** property for that file to VersionNT>=500. Multiple conditions can be specified in a single **Condition** property; for example, VersionNT>=400 AND VersionNT<500 specifies any version of Windows NT 4.0, but not Windows 2000. Conditions can be set for files, folders, registry entries, and custom actions.

For more information about working with launch conditions, see “Launch Condition Management in Deployment” on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxconLaunchConditionManagementInDeployment.asp*).

For more information about using the **Condition** property, see “Deployment Conditions” on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxconDeploymentConditions.asp*).

Create Localized Installers

The Visual Studio deployment tools include several features that allow you to distribute different versions of your application for different locales. You need to create a separate installer for each localized version of your application—it is not possible to create a single installer for multiple locales.

Note: If the core files for your application are the same for all locales, consider putting the core files in a merge module and adding the merge module plus any locale-specific files to the installer for each locale. Registry settings, custom actions, and file types can be set in the merge module project so that you do not need to recreate them for each project. For more information about creating and using merge module projects, see the "Merge Module Projects" section later in this chapter.

To create a localized installer, you set the **Localization** property of the deployment project to one of the supported languages (listed in the drop-down list in the Properties window). The **Localization** property setting determines the language for the default text displayed in the installation user interface dialog boxes during installation, such as button captions and instructions for proceeding with the installation. You cannot see the translated text in the IDE; you can see only the translated text by building and running the installer.

Text that is provided by properties is not translated. For example, the **ProductName** property that determines the name displayed in the title bar of the installation dialog boxes is not translated into the chosen locale—you need to enter the localized **ProductName** in the Properties window for each localized deployment project. Other deployment project properties that you may need to localize include the **Author**, **Description**, **Keywords**, **Manufacturer**, **ManufacturerUrl**, **Subject**, **SupportPhone**, **SupportUrl**, and **Title** properties. If the **AddRemoveProgramsIcon** property specifies an icon that contains text, you may want to specify a localized icon as well.

Additional properties that may need to be localized include the **Name** and **Description** properties for shortcuts in the File System Editor, the **Name** and **Description** properties for file types and actions in the File Types Editor, and the **Message** property for conditions in the Launch Conditions Editor.

Set Project Properties for Your Windows Installer

To have your installer contain all of the information needed to install correctly, you should manipulate properties that control how your Windows Installer files interact with Windows Installer. The following list describes the Windows Installer file properties that you can manipulate with Visual Studio .NET:

- **AddRemoveProgramsIcon**. Specifies an icon to be displayed in the **Add/Remove Programs** dialog box on the target computer. This property has no effect when installing on operating systems such as Windows 98 or Windows NT, where icons are not displayed in the **Add/Remove Programs** dialog box.

- **Author**. Specifies the name of the author of an application or component. The **Author** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in Windows Explorer. After the application is installed, the property is also displayed in the **Contact** field of the **Support Info** dialog box, which is accessible from the **Add/Remove Programs** dialog box.
- **Description**. Specifies a free-form description for an installer. The **Description** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in Windows Explorer. After the application is installed, the property is also displayed in the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **DetectNewerInstalledVersion**. Specifies whether to check for later versions of an application during installation. If this property is set to **True** and a later version is detected at installation time, installation ends. For more information about working with this property, see Chapter 6, "Upgrading .NET Applications."
- **Keywords**. Specifies keywords used to search for an installer. The **Keywords** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in the Windows Explorer.
- **Localization**. Specifies the locale for the run-time user interface.
- **Manufacturer**. Specifies the name of the manufacturer of an application or component. The **Manufacturer** property is displayed in the **Publisher** field of the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box. It is also used as a part of the default installation path (C:\Program File \Manufacturer\Product Name) displayed during installation.
- **ManufacturerUrl**. Specifies a URL for a Web site containing information about the manufacturer of an application or component. The **ManufacturerUrl** property is displayed in the **Support Info** dialog box, accessible from the **Add/ Remove Programs** dialog box.
- **ProductCode**. Specifies a unique identifier for an application. This identifier must vary for different versions and languages. Windows Installer uses the **ProductCode** property to identify an application during subsequent installations or upgrades; no two applications can have the same **ProductCode** property. To ensure a unique **ProductCode** property, you should never manually edit the GUID; instead, you should use the GUID generation facilities in the **Product Code** dialog box. For more information about working with this property, see Chapter 6, "Upgrading .NET Applications."
- **ProductName**. Specifies a name that describes an application or component. The **ProductName** property is displayed as the description of the application or component in the **Add/Remove Programs** dialog box. It is also used as a part of the default installation path (C:\Program Files\Manufacturer\Product Name) displayed during installation.

- **RemovePreviousVersions**. Specifies whether an installer removes earlier versions of an application during installation. If this property is set to **True** and an earlier version is detected at installation time, the earlier version's uninstall function is called. The installer checks **UpgradeCode**, **PackageCode**, and **ProductCode** properties to determine whether the earlier version should be removed. The **UpgradeCode** property must be the same for both versions, whereas the **PackageCode** and **ProductCode** properties must be different. For more information about working with this property, see Chapter 6, "Upgrading .NET Applications."
- **RestartWWWService**. For Web setup projects only, this specifies whether Internet Information Services stop and restart during installation. Restarting may be required when deploying an application that replaces Internet Server API (ISAPI) DLLs or ATL Server components that are loaded into memory. ASP.NET components created using Microsoft Visual Basic® .NET development system or Microsoft Visual C#™ development tool do not require a restart when replacing components that are loaded into memory. If this property is not set to **True** and a restart is required, the installation completes only after the computer is restarted.
- **SearchPath**. Specifies the path that is used to search for assemblies, files, or merge modules on the development computer.
- **Subject**. Specifies additional information describing an application or component. The **Subject** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in the Windows Explorer.
- **SupportPhone**. Specifies a phone number for support information for an application or component. The **SupportPhone** property is displayed in the **Support Information** field of the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **SupportUrl**. Specifies a URL for a Web site containing support information for an application or component. The **SupportUrl** property is displayed in the **Support Information** field of the **Support Info** dialog box, accessible from the **Add/Remove Programs** dialog box.
- **Title**. Specifies the title of an installer. The **Title** property is displayed on the **Summary** page of the **Properties** dialog box when an installer file is selected in the Windows Explorer.
- **UpgradeCode**. Specifies a shared identifier that represents multiple versions of an application. This property is used by Windows Installer to check for installed versions of the application during installation. The **UpgradeCode** property should be set for only the first version; it should never be changed for subsequent versions of the application, nor should it be changed for different language versions. Changing this property prevents the **DetectNewerInstalledVersion** and **RemovePreviousVersions** properties from working properly. For more information about working with this property, see Chapter 6, "Upgrading .NET Applications."

- **Version**. Specifies the version number of an installer. The **Version** property should be changed for each released version of your installer. For more information about working with this property, see Chapter 6, "Upgrading .NET Applications."

Build Your Windows Installer File

The Windows Installer file containing your setup logic is created when you build the setup project. The contents of the Windows Installer file, and whether any other files are created, are determined by the settings you choose for the project. You can control these settings in the **Property Pages** dialog box. (You can view this dialog box by right-clicking your setup project in the Solution Explorer and then clicking **Properties**).

Choosing How Your Files are Packaged

In most cases, you will choose to package your files in a single setup file (an .msi file). This allows you to distribute one file that contains all the files needed for installation. You can specify how files will be packaged within a Windows Installer file. The following options are available:

- **In a single setup file**. This allows you to distribute one file that contains all of the files needed for installation. The .msi file also contains the logic needed to complete the installation.
- **As loose uncompressed files**. All of the files required by your application are placed in the same directory as the .msi file (and subdirectories if your solution includes them). The .msi file itself contains the setup logic necessary to complete the installation and relies on the other files being present at install time. This option can be useful because it allows users to retrieve individual files, perhaps from the installation CD, after installation is complete if those files are damaged or have been deleted from their computers.
- **In CAB files**. To distribute more manageable file sizes, you should use the cabinet file(s) option. You can choose the maximum file size for your CAB files, which can make distribution of these files easier than for one large installer file. For example, if you plan to distribute files on a series of floppy disks, set the maximum size to 1440 KB (1.44 MB). Your files will be packaged in one or more CAB files in the same directory as the .msi file. Similar to the loose uncompressed files option, the .msi file contains the setup logic necessary to complete the installation and relies on the CAB files being present at install time.

Use the Bootstrapping Application to Ensure Windows Installer 2.0 Is Installed

If you are not certain whether the target computers have Windows Installer 2.0 available, you should include the Windows Installer bootstrapping application with your .msi file. For your .msi files to install your application successfully, they require that the target computer have the Windows Installer 2.0 service present. A bootstrapping application includes the files needed to install Microsoft Windows Installer on the target computer, if it is not already installed, before continuing with your application setup. You have two main options for including the bootstrapping application:

- **Windows Installer bootstrapping application**. A bootstrapping application is included for installation on a Windows-based computer. Your .msi file automatically checks for the Windows Installer service on the target computer and, if it is not present or is an incompatible version, launches the setup of the Windows Installer service prior to installing your application.
- **Web bootstrapping application**. A bootstrapping application is included for download over the Web. As in the case of the Windows Installer bootstrapping application, this option launches the setup of the Windows Installer service prior to installing your application, if necessary. The only distinction is that you don't distribute the bootstrapping application with your .msi file, but instead make it available for download from a Web server. A Web bootstrapping application has authentication built in, so using this approach rather than CAB files for distributing the bootstrapping application is recommended. If you choose this option, you can use the **Settings** option to specify the Web location where your application and the Windows Installer executable files are available for download.

Note: In some cases, browser content expiration settings can cause the Web bootstrapping application to fail to download. For more information about how to remedy this problem, see article Q313498, "BUG: Error 1619 When You Install a Package That Uses Web Bootstrapper," in the Microsoft Knowledge Base (*http://support.microsoft.com/default.aspx?ln=EN-US&pr=kbinfo&#mskb*).

Compress Your Installer File(s)

To manage the size of the installer file, you can set either of two levels of compression:

- **Optimized for speed**. Files are compressed to install faster, but result in a larger file size.
- **Optimize for size**. Files are compressed to a smaller size, but may result in a slower build and installation.

These settings apply only to CAB files and to solutions packaged in a single .msi file, and not to loose uncompressed files. You should consider optimizing for size when users install your solution over a slow dial-up link. For scenarios where size is not an issue, such as when the installer will be distributed on CD or DVD, you might consider optimizing for speed.

Digitally Sign Your Installer

You can easily set your installer file to be digitally sign by selecting the **Authenticode Signature** check box. You must then provide one of the following:

- **Certificate file**. Specifies an Authenticode certificate file (.spc) that be used to sign the files.
- **Private key file**. Specifies a private key file (.pvk) that contains the digital encryption key for the signed files.
- **(Optional) Timestamp server URL**. Specifies the Web location for a timestamp server used to sign the files.

You can obtain digital certificates from a number of certificate authorities, such as Verisign (*http://www.verisign.com/*) or Thawte (*http://www.thawte.com/*).

Merge Module Projects

Microsoft merge modules (.msm files) are used to package components rather than complete applications. Developers who use your component in their applications can include your merge module in their .msi files to ensure your component is installed correctly along with their other application files.

Merge modules are, in effect, reusable setup components. Merge modules cannot be installed directly; they are merged into an installer for each application that uses the component. Much as dynamic-link libraries allow you to share application code and resources between applications, merge modules allow you to share setup code between .msi files. This ensures that the component is installed consistently for all applications, eliminating problems such as version conflicts, missing registry entries, and improperly installed files. The following list contains some recommendations for using merge modules:

- For components that are used across multiple applications you should package the component in an .msm file. This allows you to deploy that .msm file with the .msi files of the applications that use the component without needing to repackage the component every time.
- Installers can include multiple applications, allowing you to install a suite of applications in a single step. In this case, the installer should include merge modules for all components used by any of the included applications. If a merge module is used by more than one application it needs to be added to the .msi file only once.
- You should capture all of the dependencies for a particular component in your merge module to ensure that the component is installed correctly.
- Each merge module contains unique version information that is used by the Windows Installer database to determine which applications use the component,

preventing premature removal of a component. For this reason, a new merge module should be created for every incremental version of your component. A merge module should never be updated after it is included in an installer.

Use the Module Retargetable Folder to Allow Other Developers to Control where Your Files Are Installed

By default, files in your merge modules are installed into the folder locations you specify when you build the .msm file. In some cases, you may want to allow the developer who uses your merge module some flexibility in deciding where files should be installed for their application. For example, if an assembly in a merge module is used by multiple applications, the consumer of the merge module may want to install it in the global assembly cache; otherwise, they would install it in the application directory.

To allow other developers to retarget your files to a different location, you should place them in the Module Retargetable folder in your merge module project. When the resulting merge module is added to another deployment project, the author of that project can choose a location for your files by setting the **Module Retargetable Folder** property exposed by your built merge module in their .msi file.

You can provide a default location for files in the Module Retargetable folder by setting its **DefaultLocation** property. You can set this to any of the following:

- [CommonFilesFolder]
- [FontsFolder]
- [GAC]
- [ProgramFilesFolder]
- [SystemFolder]
- [WindowsFolder]
- [TargetDir]

The [TargetDir] setting corresponds to the application folder for the solution that is installed with the .msi file that includes your merge module. The other folders in the list should be self-explanatory.

Set Project Properties for Your Merge Module

In addition to the properties that merge module have in common with .msi files, they also support a **ModuleSignature** property. This property specifies a unique identifier for the merge module. The **ModuleSignature** property contains the name of the merge module followed by a GUID. Each released version of a merge module must have a unique **ModuleSignature** property in order to avoid versioning problems. You should never manually edit the GUID portion of this property—you should use the GUID generation facilities in the **Module Signature** dialog box.

Build Your Merge Module

As with project properties, merge module projects support a subset of the build options available for .msi files. These include:

- Output file name.
- Compression.
- Authenticode signature.

For more information about these options, see the "Build Your Windows Installer File" section earlier in this chapter.

CAB Projects

CAB projects allow you to create a CAB file to package ActiveX and managed controls that can be downloaded from a Web server to a Web browser. Unlike the other deployment project types, there are no editors provided for working with CAB projects. Files and project outputs can be added to a CAB project in Solution Explorer, and properties can be set in the Properties window or in the Project Property pages.

Properties of CAB projects allow you to specify a level of compression and implement Authenticode signing (as discussed for .msi and .msm files). You can also set the following properties for your CAB file project:

- **Friendly Name**. Specifies the public name for a CAB file in a CAB project.
- **Version**. Specifies the version number a CAB file. As with the other types of setup projects, the **Version** property should be changed for each released version of your CAB file.
- **Web Dependencies**. Specifies the URL, friendly name, and version of other CAB files that must be installed in order for the CAB to be successfully installed. If this property is set, all dependencies automatically download and install when the CAB file runs. You can specify multiple Web dependencies using the Web dependencies dialog box, which is accessible from the Properties window.

Windows Installer Platform SDK

In addition to the setup and deployment projects provided with Visual Studio .NET, the Windows Installer Platform SDK includes several utilities that you can use with your .msi and .msm files.

Advanced Options with Orca and MsiMsp

Merge modules (.msm files) and .msi files contain databases that define setup components and logic for your application or component. The Orca database editor is a table-editing tool available in the Windows Installer SDK that you can use to edit your .msi or .msm files. You can use Orca if you need to edit the database tables directly to control setup logic and components for an .msi file or .msm file. Although you can edit any .msi file or .msm file using Orca, you should do so for third-party installers only under instruction from the software vendor.

A more common use for Orca is to create patch creation properties (PCP) files. You can then use MsiMsp.exe to create a Windows Installer patch (MSP) file from your PCP. The MSP patch file can then be applied to your application.

Orca.exe and MsiMsp.exe are both available in the Microsoft Windows Installer SDK. To download the Microsoft Windows Installer SDK, see "Samples, Tools, and Documentation (x86) v1.2 for Windows 2000, Windows 9x, and Windows NT 4.0" on MSDN (*http://msdn.microsoft.com/downloads/sample.asp?url=/MSDN-FILES/027/001/457/msdncompositedoc.xml*).

For a complete list of other utilities included in the Windows Installer Platform SDK and for more in-depth information, see the Windows Installer Platform SDK documentation.

Use Orca to Create Nested Installations

In certain circumstances, you might need to launch one .msi file from another. This is known as a nested installation. Windows Installer technology supports nested installs, but the setup and deployment projects do not currently natively support them. However, you can use Orca or another .msi file editing tool to add a nested installation action to the custom action table of your primary .msi file in order to have that .msi file launch another one. Nested installations are performed in the same transactional context as the launching .msi file, so failure in the nested installation rolls back all work carried out by the main .msi file. However, there are some issues with nested installations that make them suitable in certain circumstances only. For example:

- Nested installations cannot share components.
- An administrative installation cannot also contain a nested installation.
- Patching and upgrading may not work with nested installations.
- The installer may not properly calculate the disk space required for the nested installation.
- Integrated progress bars cannot be used with nested installations.

For more information about nested installations, see “Nested Installation Actions” on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msi /cact_260j.asp*).

For step-by-step instructions on creating nested installation, see article Q306439, “HOWTO: Create a Nested .msi Package,” in the Microsoft Knowledge Base (*http:// support.microsoft.com/default.aspx?ln=EN-US&pr=kbinfo&#mskb*).

Third-Party Installation Tools

If you need to implement features in your setup routines that are not supported by Visual Studio .NET setup and deployment projects, you should investigate whether third-party tools can help you achieve your specific goals. Some information and links for popular installation tools are included in Chapter 5, “Choosing Deployment Tools and Distribution Mechanisms for your .NET Application.”

4

Deployment Issues for .NET Applications

This chapter discusses the new issues you face when deploying your .NET applications and provides guidance for managing these issues.

Deploy Web Applications

ASP.NET applications can take advantage of many different technologies, such as private and shared assemblies, interoperation with COM and COM+, event logs, message queues, and so on. Specific technologies are discussed later in this chapter. This section discusses the deployment of ASP.NET Web applications at a more general level—you can use this section for general guidance when you are planning the deployment of your ASP.NET solution, but you should refer to the later sections for technology-specific information and guidance.

Deploy ASP.NET Application Files

An ASP.NET application is defined as all the Web pages (.aspx and HTML files), handlers, modules, executable code, and other files (such as images and configuration files) that can be invoked from a virtual directory and its subdirectories on a Web server. An ASP.NET application includes the project DLL (if the code-behind features of .aspx files are used) and typically other assemblies that are used to provide functionality for the application. These assemblies are located in the bin folder underneath the virtual directory of the application.

The Microsoft® .NET Framework makes deployment of Web applications much easier than before. For example, configuration settings can be stored in a configuration file (Web.config) which is just a text file based on Extensible Markup Language

(XML)—these settings can be modified after the Web application is deployed without requiring the Web solution to be recompiled. In addition, ASP.NET allows changes to be made to assemblies without requiring that the Web server be stopped and restarted. Furthermore, when a newer version of a Web.config, .aspx, .asmx, or other ASP.NET file is copied to an existing Web application, ASP.NET detects that the file has been updated. It then loads a new version of the Web application to handle all new requests, while allowing the original instance of the Web application to finish responding to any current requests. After all requests are satisfied for the original application, it is automatically removed. This feature eliminates downtime while updating Web applications.

There are a number of different ways for deploying all of these elements from your development or test environment to the production Web server(s). For simple Web applications, it is often appropriate to copy the files to the target computer, using XCOPY, Windows Explorer, FTP, or the Microsoft Visual Studio® .NET development system **Copy Project** command on the **Project** menu. For more complex solutions, such as those that include shared assemblies or those that rely on specific Microsoft Internet Information Services (IIS) settings to be in place, using Windows Installer technology might be a better choice. Choosing an appropriate mechanism is discussed in Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Application."

Deploy IIS Settings

One of the issues that you need to consider as you plan for how to deploy your ASP.NET applications is how to deploy IIS settings along with your solution.

If you choose to deploy your Web application with copy operations, such as XCOPY, Windows Explorer, or FTP, IIS settings are not included with that deployment. You will need to apply any required settings separately. You can either do this manually or write IIS scripts for applying the settings to the Web folder.

If you use the Visual Studio .NET **Copy Project** command, a new Virtual directory is created for you on the target Web server. However, for the simple copy operations, the IIS settings are not copied from your development virtual directory and applied to the production copy of your solution. The new virtual directory inherits the default settings from the Web site. Again, you need to apply the appropriate settings separately, either by developing and running IIS scripts or by manually applying the setting your Web application requires.

If you package your ASP.NET application in a Windows Installer Web setup project, you can specify that certain IIS settings be applied when your solution is installed. You specify these settings by using the Properties window in the Visual Studio .NET Web setup project, when the Web Application folder (or a subfolder) is selected. The following describes properties that control the IIS virtual directory settings:

- **AllowDirectoryBrowsing**. Sets the **IIS Directory browsing** property for the selected folder. This setting corresponds to the **Directory browsing** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box, and can be set to either **True** or **False**.
- **AllowReadAccess**. Sets the **IIS Read** property for the selected folder. This setting corresponds to the **Read** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **AllowScriptSourceAccess**. Sets the **IIS Script source access** property for the selected folder. This setting corresponds to the **Script source access** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **AllowWriteAccess**. Sets the **IIS Write** property for the selected folder. This setting corresponds to the **Write** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.
- **ApplicationProtection**. Sets the **IIS Application Protection** property for the selected folder. This setting corresponds to the **Application Protection** selection on the **Directory** page of the **Internet Information Services Web Properties** dialog box. It can be set to:
  - **vsdapLow**. The application runs in the same process as IIS.
  - **vsdapMedium**. The application runs in an isolated pooled process in which other applications are also run.
  - **vsdapHigh**. The application runs in an isolated process separate from other processes.
- **AppMappings**. Sets the **IIS Application Mappings** property for the selected folder. This setting corresponds to the **Application Mappings** list on the **App Mappings** page of the **Internet Information Services Application Configuration** dialog box.
- **DefaultDocument**. Specifies the default (startup) document for the selected folder. This setting corresponds to the list of default documents on the **Documents** page of the **Internet Information Services Web Properties** dialog box. You can specify a comma separated list to specify multiple default documents—the order of precedence for the default documents in taken from their relative positions in your comma-separated list.
- **ExecutePermissions**. Sets the **IIS Execute Permissions** property for the selected folder. This setting corresponds to the **Execute Permissions** list on the **Directory** page of the **Internet Information Services Web Properties** dialog box. You can set it to:
  - **vsdepNone**. Only static files, such as HTML or image files, can be accessed.

  - **vsdepScriptsOnly**. Only scripts, such as Active Server Pages scripts, can be run.
  - **vsdepScriptsAndExecutables**. All file types can be accessed or executed.
- **LogVisits**. Sets the **IIS Log Visits** property for the selected folder. This setting corresponds to the **Log visits** check box on the **Directory** page of the **Internet Information Services Web Properties** dialog box and can be set to either **True** or **False**.

Other IIS settings cannot be set with properties of your Web setup project. These include specifying directory security settings (for anonymous access, basic authentication, or Windows authentication), and specifying custom errors. You need to determine another way to apply these settings. One useful approach you might consider is to include custom actions in your Windows Installer file for applying the required settings. Custom actions are a Windows Installer feature that allows you to run code at the end of an installation to perform actions that cannot be handled during installation. The code can be in the form of a DLL, executable file, script, or assembly.

For more information about custom actions, see "Custom Actions" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconCustomActions.asp*).

For more information about IIS Security in ASP.NET applications see "Authentication in ASP.NET: .NET Security Guidance" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/authaspdotnet.asp*).

Deploy Secure Sockets Layer Encryption

Secure Sockets Layer (SSL) is the most widely-used method of creating secure communication on the Web. SSL uses public key cryptography to securely generate and exchange a commonly-held key (called the session key) that is used for symmetric encryption. The SSL features in IIS cannot be used until a server certificate is obtained and bound to the Web site. For Internet (public) applications, you obtain a server certificate from a recognized third-party certification authority. For private (intranet) applications, you can issue a server certificate yourself. By binding, IIS associates the certificate with a particular Internet Protocol (IP) address and port number combination.

You need to determine a way to bind your server certificate to your production Web site as you deploy your ASP.NET application. You cannot do this with the deployment mechanisms previously mentioned—the most common way is to manually bind the certificate using the IIS management console. If all of the virtual directories and Web sites on your Web server can use the same certificate, you can bind a server certificate at the Master Properties level for your Web server—this binds the certificate to every Web site that has not previously been bound to a certificate. If this is your first server certificate, it means that it is effectively bound to the entire

Web server. Any new virtual directories or Web sites that you create as part of your deployment routines are bound with that server certificate.

If you are deploying your Web applications in a clustered server environment, you can use the deployment and synchronization features of Microsoft Application Center to manage SSL certificates. Application Center detects that your solution relies on a certificate and automatically configures each member of your cluster accordingly.

Certificates should be backed up onto portable media and stored in a well-known place for emergencies or for configuring new servers.

For more background information about using SSL, see “Untangling Web Security: Getting the Most from IIS Security” on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dniis/html/websec.asp*).

Deploy HTTP Handlers and HTTP Modules

ASP.NET provides the IHttpHandler and IHttpModule interfaces that allow you to use application programming interfaces (APIs) that are as powerful as the Internet Server API (ISAPI) programming interfaces available with IIS, but with a simpler programming model. Hypertext Transfer Protocol (HTTP) handler objects are functionally similar to IIS ISAPI extensions, and HTTP module objects are functionally similar to IIS ISAPI filters.

ASP.NET maps HTTP requests to HTTP handlers. Each HTTP handler enables processing of individual HTTP Uniform Resource Locators (URLs) or groups of URL extensions within an application. HTTP handlers have the same functionality as ISAPI extensions with a much simpler programming model.

An HTTP module is an assembly that handles events. ASP.NET includes a set of HTTP modules that can be used by your application. For example, the SessionStateModule is provided by ASP.NET to supply session state services to an application. You can also create custom HTTP modules to respond to either ASP.NET events or user events.

HTTP handlers and HTTP modules are easily deployed.

► **To deploy HTTP handlers and HTTP modules**

1. Compile and deploy the .NET class for the HTTP handler or HTTP module in the \bin directory under the application’s virtual root.
2. Register the HTTP handler or HTTP module in the application Web.config configuration file.

For more information about working with and registering HTTP handlers and HTTP modules, see “HTTP Runtime Support” on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhttpruntimesupport.asp*).

Use Application Center to Deploy Web Applications to Web Farms

If you are distributing your Web solution to a Web farm or clustered server environment, you should use the deployment and synchronization capabilities of Application Center to manage the deployment process. In addition to providing management and monitoring features for your server environment, Application Center simplifies many of the deployment issues associated with Web application deployment. Application Center can deploy all of the different files required for your Web application; it removes the need for you to specify detailed actions for your deployment. For example, if you script the deployment of a virtual directory, you need to include the basic copy operations and then determine how to ensure the appropriate IIS settings are in place. On the other hand, if you use Application Center, it detects that you are deploying a virtual directory and configures it appropriately on the target computer(s)—you do not need to take any further steps to ensure that it is configured correctly. As another example, if you deploy an updated COM component to a Web application with script, you need to include actions that stopped the IIS Web service, unregistered the old component version, replace the component files, register the new component, and then restart the IIS Web service. Again, if you deploy the component using Application Center, it deals with the stopping and starting of services for you, as well as ensuring the old component is unregistered and registering the new one.

For more information about using Application Center to deploy Web applications and server environments, see the "Use Application Center to Deploy and Manage Server Environments" section in Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Application."

Pre-Compile Your ASP.NET Application

There are two compilations that take place in the very first request to a Web application:

- **Batch compile**. When an ASP.NET file (such as .asmx or .aspx) is requested for the first time, ASP.NET performs a "batch compile" on that directory. It compiles all of the code in the ASP.NET files in that directory (but not subdirectories) into the Microsoft Intermediate Language (MSIL).
- **Just-in-Time (JIT) compile**. Like any assembly, the first time a method is accessed it is compiled from MSIL to machine code to execute. This occurs on a per method basis.

These two compilations cause a delay in the first request to a Web application. To avoid the delay, you can "warm" your Web application by triggering either or both of these compilations.

Batch Compile Your Web Application to Increase Performance

To avoid your users experiencing the batch compile of your ASP.NET files, you can initiate it by performing one request to a page per directory and waiting until the CPU idles again. This increases the performance your users experience and decrease the burden of batch compiling directories while handling requests.

Initiate JIT Compilation Only in Extreme Scenarios

JIT compilation occurs for all .NET assemblies that are not pre-compiled using a tool such as NGEN. For more information about NGEN, see "Native Image Generator (Ngen.exe)" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfnativeimagegeneratorngenexe.asp*). NGEN does not support pre-compiling ASP.NET applications. After a method is compiled with the JIT compiler it does not need to be compiled again. To avoid your users experiencing the JIT expense, you can trigger the JIT for common code paths by requesting the page and invoking the code path. After this is done, all subsequent requests do not need to be compiled to native code. You can access the common code paths in your application either manually or through scripts to avoid your users having experience the JIT compilation, but this is rarely a major issue.

Initiate Any Pre-Compilation Separate from Deployment

Keep in mind that pre-compilation is not just a deployment issue. These compilations occur when the application is first started. This means that whenever the computer is rebooted, IIS is restarted, or the application is unloaded for any reason, both of these compilations occur again the next time the Web application runs. In addition to the Web server being restarted, ASP.NET applications can be unloaded and restarted for a variety of reasons, including:

- ASP.NET can be configured to restart the application periodically by using the restartQueueLimit setting in the processModel section of your configuration files.
- Every time you update any part of your Web application, ASP.NET effectively creates a new application instance for reflecting your changes.

If you choose to pre-compile your application to avoid the initial performance hit for your users, you should create a process (either manually or through script) separately from your deployment steps so that you can execute that process whenever the application is unloaded and needs to be started. For example, you might create a script that is run on a scheduled basis, or you might create a Windows service that monitors the status of your Web server and runs the script in response to your application being unloaded.

Use Aspnet_regiis.exe to Manage Multiple Versions of the .NET Framework for Your Web Applications

As future versions of the .NET Framework (including ASP.NET) are released, you will need to manage which version actually runs your Web applications. For example, you may have a number of different solutions on your Web server, each of which was developed for different versions of ASP.NET. You can specify which version of ASP.NET runs your Web application by using the ASPNET_RegIIS.exe tool that ships with each version of the .NET Framework. You can run this tool from the command line and pass in the path to the virtual directory that contains the Web application you want to run with the specific version of ASP.NET. The result is that the appropriate script mappings for your virtual directory are set for that specific version of ASP.NET.

The following example installs the version of ASP.NET recursively for the MyWebApp application:

```
ASPNET_REGIIS.EXE –s W3SVC/1/ROOT/MyWebApp
```

For help with the command-line switches, type **ASPNET_REGIIS.EXE /?** at the command line.

Deploy Web Services

Many of the issues affecting the deployment of Web applications also apply to Web service deployment—IIS settings may need to be deployed with the Web service, as might HTTP handlers and modules.

One slight difference between the deployment of Web services and the deployment of Web applications is that you are deploying the Web service file (.asmx) rather than Web forms (.aspx). However, this does not raise any additional deployment issues other than those already discussed for Web applications—both .aspx and .asmx files are text files with simple deployment requirements in and of themselves.

Deploy Discovery Files

The major deployment issue that differentiates Web services from Web applications concerns the discovery process. Before developers can access and use your Web service, they usually locate and query the Web service description. Web service descriptions contain XML-based information about the Web service exposed as Web Service Description Language (WSDL). The WSDL description verifies that the Web service exists, describes its capabilities, and indicates how to properly interact with it. One way a developer can query the WSDL, is to simply browse to the .asmx file and append the ?WSDL query string (for example, *http://www.somedomainname-123.com/webservices/somewebservice.asmx?WSDL*).

However, before the developer can query the WSDL for your Web service, they must know that it exists. In reality, developers do not always know what Web services you provide. Therefore, they are not able to access the Web service description unless you provide them with a discovery mechanism. You can provide this discovery mechanism by creating and deploying a static discovery file (.disco) with your Web service. Developers can then use this discovery file when they add a Web reference to their applications—they can view the contents of the file in the **Add Web Reference** dialog box provided by Visual Studio .NET and can link to the WSDL for your service from this file.

Create Static Discovery Files

You can create static discovery files by browsing to your Web service and appending the ?DISCO query string to the URL (for example, *http://www.somedomainname-123.com/somewebservice.asmx?DISCO*). The resulting XML can be saved as the .disco file. The following example illustrates the contents of a static discovery file for the *somewebservice* Web service:

```
<?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://www.somedomainname-123.com/webservices/somewebservice.asmx?WSDL
“
      docRef=”http://www.somedomainname-123.com/webservices/somewebservice.asmx”
      xmlns=”http://schemas.xmlsoap.org/disco/scl/” />
</discovery>
```

The location of the actual discovery information is contained in the <contractRef> element—developers referencing your Web service in Visual Studio .NET can click a link in the **Add Web Reference** dialog box to view the Contract for *somewebservice* (that is, the WSDL for your Web service). If you create your DISCO files in this way as a pre-deployment step by using your development servers, you need to update the URLs in both the *ref* and the *docref* attributes of the <contractRef> section before you deploy them to your production servers, so that they point to the live Web services and not the development ones.

Allow Discovery of Multiple Web Services with a Static Discovery File

In many cases, your virtual directory contains multiple Web services. For example, as well as providing the *somewebservice*, as used in the earlier sample, you might also provide a second service known as *anotherwebservice*. You can allow developers to discover both Web services by adding the discovery information for the second

Web service to the DISCO file. The following example illustrates a discovery file that allows developers to discover both *somewebservice* and *anotherwebservice*.

```
<?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://www.somedomainname-123.com/webservices/somewebservice.asmx?WSDL
"
      docRef="http://www.somedomainname-123.com/webservices/somewebservice.asmx"
      xmlns="http://schemas.xmlsoap.org/disco/scl/" />
  <contractRef
      ref="http://www.somedomainname-123.com/webservices/
anotherwebservice.asmx?WSDL "
      docRef="http://www.somedomainname-123.com/webservices/
anotherwebservice.asmx"
      xmlns="http://schemas.xmlsoap.org/disco/scl/" />
</discovery>
```

You can see in the preceding example that you simply need to include the <contractRef> element to the DISCO file for each Web service that you want to allow developers to discover. When they browse to your DISCO file using the Visual Studio .NET **Add Web Reference** dialog box, they will see Contract links for each <contractRef> element. A simple way to add these elements to your DISCO file is to simply browse to each of your Web services with the ?DISCO query string, and then copy and paste the <contractRef> element into the discovery file.

You should consider creating a DISCO file that resides in the Web server's root directory. Without such a file, developers still need to know some arbitrary URL to locate the collection of Web services. In addition, you might consider including hyperlinks to your discovery files in the default document for the Web server (usually Default.htm, Default.asp, or Default.aspx).

Deploy Static Discovery Files

After you create your DISCO file, you need to deploy it to the production environment. Because the discovery file is a simple XML-based text file, it does not have any special deployment requirements—it can deployed to the appropriate location on the production Web server with simple copy operations (such as XCOPY or FTP), or it can be included in a Web setup Windows Installer file.

Do Not Deploy Dynamic Discovery Files

Discovery can also be achieved with a process known as dynamic discovery. If you create your Web services with Visual Studio .NET, a dynamic discovery file (.vsdisco) is created for you. ASP.NET can use this file to perform a search through all subfolders in your Web service virtual directory to locate and describe Web services.

You should not deploy this file to the production environment, because you will lose control over which Web services can be discovered by other developers—you should use this file only on development computers.

Deploy XML Schema Definition (XSD) Files

With ADO.NET, you can serialize datasets into XML. There are two general variations on serialized datasets:

- **Datasets**. Datasets store data in a disconnected manner. The structure of a dataset is similar to that of a relational database; it exposes a hierarchical object model of tables, rows, and columns.
- **Typed datasets**. A class that inherits from DataSet and provides strongly typed methods, events, and properties.

Typed datasets are defined by XML schema definition (XSD) files—XSD files describe the tables, columns, data types, and constraints that are in the dataset. If your Web service returns typed datasets, the WSDL (and hence the DISCO file) for your Web service will contain references to the XSD files that define the schemas for those datasets. The following example illustrates a discovery file that references the schema definition for a typed dataset.

```
<?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://www.somedomainname-123.com/webservices/somewebservice.asmx?WSDL “
    docRef=”http://www.somedomainname-123.com/webservices/somewebservice.asmx”
    xmlns=”http://schemas.xmlsoap.org/disco/scl/” />
  <schemaRef
    ref=”Schema.xsd” xmlns=”http://schemas.xmlsoap.org/disco/schema/”/>
</discovery>
```

When developers browse to your DISCO file from the **Add Web Reference** dialog box in Visual Studio .NET, they can access the dataset schema definition by clicking on the *View Schema* link. The schema definition is retrieved from the file referenced in the <schemaRef> element (as illustrated earlier).

You need to ensure that you deploy these XSD files with your Web service if it returns typed datasets. Because XSD files are simply XML-based text files, they do not have any special deployment requirements.

Deploy Windows Forms Applications

One of the major advances in application development provided by .NET is the ease with which powerful, fully-featured Windows-based applications can be developed. The new application development model for Windows-based applications is provided by the Windows Forms platform.

Windows Forms applications can take advantage of many different technologies, such as private and shared assemblies, interoperation with COM and COM+, event logs, message queues, and so on. Specific technologies are discussed later in this chapter. This section treats the deployment of Windows Forms applications at a more general level—you can use this section for general guidance when you are planning the deployment of your Windows Forms solution, but you should refer to the later sections for technology-specific information and guidance.

Deploy with Windows Installer Files

Traditionally, Windows-based applications have been installed with executable setup programs and, more recently, with Windows Installer files. You can use Visual Studio .NET to create Windows Installer files for your Windows Forms application. Creating Windows Installer files with Visual Studio .NET allows you to fulfill the typical requirements of modern .NET Windows Forms applications, including:

- Provision of desktop and **Start** menu shortcuts for running the application.
- Management of file and folder location.
- Integration with the **Add/Remove Programs** item in Control Panel.
- Registry manipulation.
- Installation of shared components.
- Installation of COM components.
- Management of installation rollback if errors occur during setup.
- Management of installation rollback if users cancel the setup routine.
- Management of file associations.
- Uninstall routines.
- Self-repair.

For more information about creating Windows Installer files, see Chapter 3, “Visual Studio .NET Deployment Projects.” For guidance about when using Windows Installer files is appropriate, see Chapter 5, “Choosing Deployment Tools and Distribution Mechanisms for your .NET Application.”

Ensure Users Have Required Privileges

One issue that affects deployment of Windows Forms applications to users with Windows Installer files is whether the user running the setup will have the required privileges to complete the installation. The privileges required depend on the actions that the Windows Installer file performs and the platform that users are installing your solution onto. For example, no special privileges are required to install applications on Windows 95, Windows 98, or Windows Me, whereas even creating an application folder beneath the Program Files system folder on Windows 2000 or Windows XP by default requires that the user to be a member of a local group with elevated privileges, such as Power Users or Administrators.

When a user who is not a member of the local Administrators group attempts to run a Windows Installer file on Windows 2000, he or she is first prompted that the user may not hold sufficient permissions to complete the installation. They are also offered the choice of running the installer file as a different user who does have all required permissions (Administrator by default). To be able to run the installer file as a different user, the person running the installation needs to know the password for that account.

Perform Administrator-Initiated Deployment

One way to ensure that your Windows Installer file will not fail due to insufficient privileges is to distribute it with an electronic software deployment tool, such as Microsoft Systems Management Server (SMS) or Microsoft Active Directory® directory service group policies for software distribution. Both of these tools allow you to run your Windows Installer file with administrator privileges, regardless of the privileges held by the user logged on to the computer.

For more information about using SMS and Active Directory to deploy applications, see Chapter 5, “Choosing Deployment Tools and Distribution Mechanisms for your .NET Application.”

Deploy with Simple Copy Operations

The new architecture of the .NET Framework allows you to use simple copy operations for installing Windows Forms applications, rather than creating Windows Installer files. These copy operations can include any of the following:

- Distribution with the command prompt’s XCOPY command
- Distribution with the copy and paste operations in Windows Explorer
- Distribution with FTP commands
- Distribution to users via e-mail

For more information about these copy operations, including comparisons to Windows Installer files, see Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Application." However, for now, be aware that many of the features typically required by Windows-based applications (listed in the previous topic) are not provided by simple copy operations.

Use Internet Deployment to Achieve Zero Install and Zero Administration for Client Computers

Another way to package Windows Forms applications is to use a new feature of the .NET Framework. This approach is called Internet Deployment of .NET applications. The way this works is:

1. You store your application files (such as executables and DLLs) on a Web server.
2. Users connect to your Windows Forms application using HTTP from their Web browsers (either by clicking a link that you provide or typing the URL directly in the address bar), the **Run** dialog box, or by way of an application stub. For more information about these different approaches, see Table 4.1.
3. The initial files and assemblies that are immediately needed when the application is first run are downloaded to the .NET Framework assembly cache download folder (<windir>\assembly\download\) and the Temporary Internet Files folder.
4. As each additional resource is used by your application, it is automatically downloaded to the client computer, and stored in the .NET Framework assembly cache download folder and the Temporary Internet Files folder.

The advantage of this approach is that you can combine all of the richness of a traditional Windows graphical user interface with the manageability and maintainability of Web applications that have become so popular in recent years.

Because resources are downloaded on an "as needed" basis, download time for the initial run of the application is minimized. Then, when other assemblies are needed later, they are downloaded at that time. All of this happens automatically—when your executable file requests a class from another assembly, .NET Framework locates that assembly in the same location on the Web server and downloads it.

Although the application is effectively running from the .NET cache, you can still update your application files on the Web server, and those changes will take place immediately. Before the user's computer loads the assembly, it verifies that the downloaded application files have not changed since it last requested them—if they have, it downloads the later versions as needed.

There are two main approaches to implementing Internet deployment of .NET applications, described in Table 4.1.

Table 4.1: Internet Deployment of .NET Applications

Approach	Description
URL-Launched Executable	With this approach, the user can run your Windows Forms application either by directly browsing to the executable file, or by clicking a control or hyperlink in a Web browser that launches your application. Your application and any assemblies that are required are downloaded and then run in a security sandbox — they are downloaded to the .NET Framework assembly cache download folder and executed from there with either Internet or intranet zone privileges (which are minimal). The advantages of this approach include: No install is necessary on the client computer — all code is downloaded as needed. Automatic updating of your application — this approach uses the same approach as Microsoft Internet Explorer for determining whether Web content has changed on the Web server since it last downloaded them. If files have changed, the newer versions are downloaded as needed. However, you should be aware that if the user sets Internet Explorer to work offline, then your application is also offline — no checks will be made for updated application files on the Web server until Internet Explorer is set back to online. This can lead to problems if an assembly is required while the user is offline, but that assembly has not yet been downloaded because it was not accessed in a previous session.
Code Download to an Executable Stub	With this approach, you distribute an application stub to your users. The stub simply contains code that loads assemblies from a Web server using Assembly.LoadFrom(). For more information about this type of code, refer to the articles in Table 4.2. The advantages of this approach include: You can have a fully trusted executing stub installed on the client computer. You can manage security for the downloaded assemblies to grant them more privileges than those usually allowable for URL-launched applications. Immediate updating of application functionality with zero administration effort — you simply need to replace your application files on the Web server. However, any changes to the application stub will need to be redeployed to the user's computer. A reliable connection to the Web server is typically required with this approach — by default, the application stub will not load assemblies from the cache without first checking the versions on the Web server. Instead, it fails with an exception. You can work around this issue to provide an offline mode for your application, but this involves additional coding. Additionally, you still have the deployment issues for the application stub. However, it often contains a minimal amount of functionality other than connecting to the Web server for assemblies, so a simple copy operation should suffice — you could simply send the stub to your users in an e-mail message, or distribute it in any number of different ways, such as with FTP or by making it available on a network share for your users.

Note: Although it is more of a design issue than a deployment one, you should note that assemblies downloaded to the client computer in either of the preceding scenarios can only call back to the server from where it was downloaded. So, for example, a downloaded assembly cannot call a Web service on another computer. This is a security precaution that, for example, prevents people from distributing code that performs denial of service attacks on some other server.

For more information about Internet deployment of .NET applications, see the articles in the Table 4.2.

Table 4.2: Internet Deployment Articles

Article	Location
Death of the Browser?	*http://msdn.microsoft.com/library/en-us/dnadvnet/html/vbnet10142001.asp*
Security for Downloaded Code	*http://msdn.microsoft.com/library/en-us/dnadvnet/html/vbnet12112001.asp*

Deploy Assemblies

As well as defining the building blocks of your .NET application, assemblies also define units of deployment. They can contain up to four different elements:

- The assembly manifest that contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes
- Type metadata that defines the types that the assembly contains, much as type libraries have done in the past (for COM objects)
- Microsoft intermediate language (MSIL) code that implements the types to provide the assembly's functionality
- A set of resources, such as graphics or localized strings

You can group all elements in a single portable executable (PE) file, such as a .dll or an .exe file. Alternatively, you can have different elements of your assembly reside in separate files. For example, you might want large graphics or seldom used types to reside in separate files—that way they will only be loaded when required. For this scenario, your PE file (.dll or .exe) will contain the assembly manifest and core MSIL code and type metadata, and the assembly manifest will link to the external resource file(s) for graphics and to compiled .NET module(s) for seldom-used types and their MSIL implementation.

There are a number of different issues affecting deployment of assemblies. These issues are discussed next.

Deploy Private Assemblies

If an assembly is designed for the sole use of your application, you should deploy it as a private assembly. Private assemblies are usually deployed directly to the application base directory. The application base directory is either:

- The directory that contains your executable, if you have built a Windows-based application.
- The bin folder located in the virtual directory, if you have built an ASP.NET Web application.

Alternatively, you can specify the location of your assembly in your application configuration file by using the <codebase> element. For most cases, deploying private assemblies to the application base directory is recommended.

Private assemblies are used only by the application for which they were developed, allowing for your application to be isolated from other applications. This resolves many issues with shared components, including those problems collectively known as “DLL Hell.” Private assemblies do not need to be created with a strong name. You can still strong name your private assemblies, but this can lead to other issues. The next topic discusses strong-naming your private assemblies.

If your private assemblies are not strong-named, you can use a simple copy operation, such as XCOPY, to install them into the production environment. (Strong-named private assemblies require a further step, which is discussed in the next section). Private assemblies are self-describing and, unlike COM components, do not require information to be added to the registry. If you update your assembly, you can simply copy the new version to the appropriate folder and your application will automatically use that new one. Again, unlike COM components, the earlier version does not need to be unregistered, and the new version does not need to be registered.

This copy-style deployment is one of the major advantages of the .NET Framework, as far as deployment of private assemblies is concerned.

For more information about specifying assembly locations, see “Specifying an Assembly’s Location” on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconspecifyingassemblyslocation.asp*).

For more information about how the common language runtime (CLR) locates assemblies, see “How the Runtime Locates Assemblies” on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconhowruntimelocatesassemblies.asp*).

Strong Name Your Private Assemblies

As mentioned previously, you can create your private assemblies with strong names.

Strong names provide the following features:

- They guarantee name uniqueness by relying on unique key pairs.
- They protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they load comes from the same publisher that created the version the application was built with.
- They provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built.

A strong name consists of the assembly's identity (its simple text name, version number, and localization information) plus a public key. You can create public/private key pairs with the Sn.exe tool and then strong name your assemblies with Visual Studio .NET by setting assembly attributes. Alternatively, you can use the assembly linker utility (Al.exe) to strong name your assembly with the keys generated by Sn.exe after they are built. This latter approach is most useful if you need to delay the signing of your assemblies for security reasons.

Note: When strong-named assemblies are loaded from a location other than the global assembly cache, the CLR must perform a verification of the manifest to ensure that the files contents have not been tampered with. This performance penalty occurs every time the assembly is loaded. When assemblies are installed in the global assembly cache, the verification occurs at installation time and the performance penalty is not incurred at load time.

For more information about strong-naming your assemblies with attributes, see "Signing an Assembly with a Strong Name" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconassigningassemblystrongname.asp*).

Strong Name Components Built for Other Developers

One other reason to create strong-naming assemblies is if you are developing components that will be used by other developers. You will not know when you develop your assembly whether the developers who use it will need to install it as a private assembly or as a shared one. If they need to install it as a shared assembly in the global assembly cache, it must have a strong name. The developer should not apply a strong name to your assembly after you create it, because then it cannot be verified that you (or your company) is the author of the component. If you create your assembly with a strong name, you can leave the decision of whether the assembly will be installed into the global assembly cache or as a private assembly to the developer who is using your component. The global assembly cache is discussed later in this chapter.

Deploy Your Strong-Named Private Assemblies

Although you can use simple copy operations for the initial deployment of your strong-named assemblies, updating them introduces a further deployment issue. You cannot simply copy a new version of your strong-named assembly and have your application (or other assemblies) use it automatically. The strong name of the assembly is stored in the manifest of the assembly that references it, and different versions of a strong-named assembly are considered to be completely separate assemblies by the CLR. Because of this, the types exposed by different versions of an assembly are not considered equal, so by default the CLR does not load a different version of a strong-named assembly than the one your application was originally built against. You can instruct the CLR to load an updated version by providing binding redirects in your application's configuration file. You must then deploy not only the updated strong-named private assembly, but also the updated application configuration file that contains the binding redirect information. The following code snippet shows an example of redirecting your application to use an updated assembly:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns=”urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name=”myAssembly” publicKeyToken=”32ab4ba45e0a69a1"
                                       culture=”en-us” />
        <bindingRedirect oldVersion=”1.0.0.0" newVersion=”2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Note: The CLR attempts to locate assemblies in the global assembly cache before searching in the application base folder. Consequently, if you deploy a strong-named private assembly with your application but it is already present in the global assembly cache, the CLR uses the shared version, rather than the private one, at run time. This should not lead to any problems, because assemblies with the same strong name should always be identical.

For more information about redirecting assembly binding, see “Redirecting Assembly Versions” on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library /en-us/cpguide/html/cpconassemblyversionredirection.asp*).

Deploy Shared Assemblies to the Global Assembly Cache

The global assembly cache is a computer-wide storage area for shared assemblies. You can install assemblies that need to be shared among multiple applications in the global assembly cache. Installing your assemblies into the global assembly cache provides the following advantages:

- The global assembly cache provides a centralized location for managing assemblies that need to be shared among multiple applications.
- Assemblies installed into the global assembly cache support side-by-side execution. Side-by-side execution occurs when you have multiple versions of the same assembly present in the global assembly cache, and different applications need to use the different assembly versions. Because all assemblies installed into the global assembly cache must be created with strong names, the CLR can differentiate between the versions of a shared assembly, thereby allowing the appropriate version to be used by the referencing application.
- The global assembly cache provides a version-aware and publisher-aware assembly store.

However, installing an assembly into the global assembly cache introduces the following issues that are not encountered with private assemblies:

- Installing an assembly into the global assembly cache requires administrative privileges by default, because the physical location of the global assembly cache is a subfolder of the Windows directory.
- The shared assembly must be created with a strong name. You cannot install assemblies without strong names into the global assembly cache because the CLR performs integrity checks on all files that make up shared assemblies based on the strong name. The cache performs these integrity checks to ensure that an assembly has not been tampered with after it has been created (for example, to prevent a file from being changed without the manifest reflecting that change).
- You cannot simply copy an assembly into the global assembly cache and have your applications use it. The assembly must be installed in the global assembly cache. The preferred (and most common) approach for installing an assembly in the global assembly cache is to use Windows Installer technology.

Note: In a production environment, you should always install assemblies into the global assembly cache with some mechanism that can maintain a count of the number of references to that assembly. This prevents premature removal of the assembly from the global assembly cache by the uninstall routine of another application, which would break your application that relies on that assembly. Windows Installer has very robust reference counting features and is the recommended way to install assemblies into the global assembly cache. You can actually install an assembly into the global assembly cache without using Windows Installer technology, by using the Gacutil.exe utility or by using a drag-and-drop operation to move the assembly into the Global Assembly Cache folder in Windows Explorer. However, using drag-and-drop operations to move assemblies into the global assembly cache does not implement any reference counting; therefore, it should be avoided. If you use the Gacutil.exe tool, you should use the **/ir** switch, which installs assemblies into the global assembly cache with a traced reference. These references can be removed when the assembly is uninstalled by using the **/ur** switch.

After an assembly is installed into the global assembly cache, you cannot simply copy a new version and have your application use that updated assembly. As with all strong-named assemblies, applications contain the strong name (complete with version number) of the assembly that they reference in their own manifests. Instead of simply copying a new version of the strong-named assembly, you can either recompile against the newer version or provide binding redirection for the referencing application as well. For shared assemblies that reside in the global assembly cache, you can achieve this in a number of different ways:

- Redirect assembly versions using publisher policy. You can state that applications should use a newer version of an assembly by including a publisher policy file with the upgraded assembly. The publisher policy file, which is located in the global assembly cache, contains assembly redirection settings. If a publisher policy file exists, the runtime checks this file after checking the assembly's manifest and application configuration file. You should use publisher policies only when the new assembly is backward compatible with the assembly being redirected. New versions of assemblies that claim to be backward compatible can still break an application. When this happens, you can use the following setting in the application configuration file to make the runtime bypass the publisher policy: <publisherPolicy apply="no">.
- Redirect assembly versions using your application configuration file. As with strong-named private assemblies, you can specify that your application use the newer version of a shared assembly by putting assembly binding information in your application's configuration file.
- Redirect assembly versions with the machine configuration file. This should **not** be considered as a first choice for most redirection scenarios, because the machine configuration file overrides all of the individual application configuration files and publisher policies, and applies to all applications. However, there might be rare cases when you want all of the applications on a computer to use a specific version of an assembly. For example, you might want every application

to use a particular assembly version because it fixes a security hole. If an assembly is redirected in the machine configuration file, all of the applications using the earlier version will use the later version.

As far as deployment is concerned, updating shared assemblies is more complex than updating private assemblies—not only do you need to ensure that the upgraded assembly is installed in the global assembly cache, but you also need to ensure that configuration or publisher policy files are also deployed.

Upgrading shared assemblies with publisher policies is discussed in Chapter 6, "Upgrading .NET Applications."

Use Installation Components to Create and Configure Application Resources

Modern distributed .NET applications consist not only of the traditional program files but also of associated resources, such as message queues, event logs, performance counters, and databases. These elements should be created as part of your deployment process. The .NET Framework provides installation components that allow these resources to be created and configured when your application is installed, and removed when your solution in uninstalled. These installation components integrate with widely available deployment tools, such as Installutil.exe, Windows Installer files, and the Windows Installer service. There are two types of installation components: installer classes and predefined installation components.

Use Installer Classes to Perform Custom Installation Operations

You can add installer classes to your .NET application to perform specific actions during installation. For example, you can use installer classes to create databases required for your solution, or you can use them to pre-compile a certain assembly to native code after the main installation is complete.

Your installer classes are consumed and executed by the Windows Installer service. To allow Windows Installer to consume and execute your installer classes, you can either deploy your solution using Windows Installer files and add your installer classes as custom actions that are run at the end of the installation process, or you can use the Installutil.exe tool to run your installer classes after the remainder of your solution has been installed.

For an example of how to create and use installer classes, see "Walkthrough: Using a Custom Action to Create a Database During Installation" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxwlkwalkthroughusingcustomactiontocreatedatabaseduringinstallation.asp*).

For more information about using the Installutil.exe utility, see "Installer Tool (Installutil.exe)" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconinstallerutilityinstallutilexe.asp*).

Use Predefined Installation Components to Create and Configure Application Resources

Predefined installation components are conceptually similar to the installer classes discussed in the previous topic. However, they provide you with a head start for installing certain resources that otherwise might need you to write complex installer class code. Visual Studio .NET provides five predefined installation components:

- EventLogInstaller
- MessageQueueInstaller
- PerformanceCounterInstaller
- ServiceInstaller
- ServiceProcessInstaller

Using these components is discussed later in this section.

Much like installer classes that you develop yourself, predefined installation components in your application project are run at the end of the installation process. If you create a Windows Installer file for your project, you need to add these installer classes to the Windows Installer project as custom actions. If you do not create an installer, you need to use the Installutil.exe utility on the production servers to run these installer classes. While there is no functional disadvantage to using Installutil.exe, it is often easier to have the Windows Installer file perform the necessary actions as part of the deployment process.

Use Standard Development Practices for Deploying Application Resources

To simplify the process of deploying application resources such as message queues, event logs, and performance counters, you should adhere to the following general guidelines:

1. Create your resources in your development (or test) environment. For example, you might create a message queue named RecOrders.
2. Set properties for your resources in the development (or test) environment. For example, you might the maximum queue size for the RecOrders message queue, and you might activate journal recording.
3. Use Server Explorer in Visual Studio .NET to create your components and link them to your resources. For example, you can use Server Explorer to drag and drop the RecOrders message queue to your project. The message queue will be added as a MessageQueue component, which you might name mqRecOrders.

4. Write the code for your mqRecOrders component and then compile and test your project using the resources you have created in your development (or test) environment.
5. Add installation components to your project. For example, you can add an installation component for your mqRecOrders component by right-clicking the component and then clicking **Add Installer**.
6. Deploy your project to the production environment. Your installation components can be run either as custom actions in your Windows Installer files, or you can run them manually using Installutil.exe. The installation components that you add to your project automatically create and configure your resources in the production environment. For example, the installation component reads the properties of your mqRecOrders MessageQueue component and determines that it uses a message queue names RecOrders. It then accesses the message queue itself in order to retrieve the properties of that resource, such as maximum queue size and journal recording settings. When you deploy your project to the production environment, it recreates the message queue for your production server and ensures that those settings are applied. All of these settings are stored in the code for the installation component—this happens automatically, so you do not need to manage these settings manually.

You do not need to follow this design process, but it provides an efficient way to work with your test and production resources. If you do not create and configure a test resource from which property values can be copied to the installation component, you can access the installation component in the ProjectInstaller class (or any class with the RunInstallerAttribute value set to **true**) and manually set the necessary values to create and install the resource in the state you desire.

Deploy Event Logs with the EventLogInstaller Component

The easiest way to incorporate event logging into your .NET application is to use the EventLog component available in Visual Studio .NET. You can then use the EventLogInstaller component to install and configure the event log to which your application writes. The EventLogInstaller is used during deployment to write registry values that are associated with your event log. You can run the installation component either as a custom action in your Windows Installer file or with the Installutil.exe utility.

The EventLogInstaller component deploys the settings you have associated with EventLog components. These settings include:

- **Log**. The name of the log to read from or write to.
- **Source**. The name to use when writing to the log.

When you deploy your solution to the production environment, the installer creates the log you specify in the **Log** property; it also creates the event source that you specify in the **Source** property. An event source is simply a registry entry that indicates what logs the events. It is often the name of the application or the name of a subcomponent. Because event sources are necessary only when writing to an event log, it is not necessary to use an event log installer when you will be only reading, but not writing, to a log.

When deploying event logs with your application, you should be aware that:

- The EventLogInstaller class can install event logs only on the local computer.
- If you set the **Source** property for an existing log, but the specified source already exists for that log, the EventLogInstaller deletes the previous source and recreates it. It then assigns the source to the log you are using.
- The name you use as a source for your log cannot be used as a source for other logs on the computer. An attempt to create a duplicated **Source** value throws an exception. However, a single event log can have many different sources writing to it.
- Event logs are supported only on the following platforms:
  - Windows NT Server 4.0
  - Windows NT Workstation 4.0
  - Windows 2000
  - Windows XP Home Edition
  - Windows XP Professional
  - Windows .NET Server family
- The ASPNET security account cannot create event sources. If you use the SourceExits() method on the EventLog class and the **Source** exists it will return **True**, but if the **Source** does not exist, it throws an exception. This should not cause problems if you install event logs and event sources using the EventLogInstaller class with Windows Installer files or Installutil.exe, because the installation is probably running with the Administrator security account.

For instructions on using the EventLogInstaller class, see "Walkthrough: Installing an Event Log Component" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbwlkWalkthroughCreatingEventLogInstallers.asp*).

Deploy and Configure Message Queues with the MessageQueueInstaller Component

You can incorporate message queuing functionality into your .NET applications by using the MessageQueue component in Visual Studio .NET. You can then use the MessageQueueInstaller component to create and configure message queues for your application as you deploy to the production environment. The MessageQueueInstaller component deploys all the settings you have associated with your MessageQueue component, such as path, journal settings, label, and so on, and creates and configures the message queue itself. You can run the installation component either as a custom action in your Windows Installer file or with the Installutil.exe utility.

For an example of installing message queues with the MessageQueueInstaller component, see the "Use Standard Development Practices for Deploying Application Resources" section earlier in this chapter.

Deploy Performance Counter Categories and Counters with the PerformanceCounterInstaller Component

The easiest way to incorporate performance counters in your .NET application is to use PerformanceCounter components. You can use these components to read values from existing predefined counters, such as memory or processor counters, and you can also read from and write to custom counters that you create. For example, you might use the Server Explorer to create a new performance counter category named MyApp, and then create counters for monitoring your application performance, such as NumFilesOpen for the number of files that are in use, and UserCons for how many users are currently connected, and so on.

If you create custom counters, you need to ensure that they are deployed with your application. Registry entries need to be added to your production computers for custom categories and counters. The easiest way to deploy these counters and categories is to use PerformanceCounterInstaller components for your PerformanceCounter components. You can run the installation component either as a custom action in your Windows Installer file or with the Installutil.exe utility.

When deploying performance counters with your application, you should be aware that:

- Performance counters are supported only on the following platforms:
  - Windows NT Server 4.0
  - Windows NT Workstation 4.0
  - Windows 2000
  - Windows XP Home Edition
  - Windows XP Professional
  - Windows .NET Server family
- Performance counters need to be created by security accounts with administrative privileges—the easiest way to ensure this is to have your Windows Installer file run by an administrator, or to have an administrator run the Installutil.exe utility to install your performance counter.

Deploy Windows Services with the ServiceInstaller and ServiceProcessInstaller Components

Microsoft Windows services, formerly known as Windows NT services, are long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots, can be paused and restarted, and do not show any user interface. This makes Windows service applications ideal for use on a server or whenever you need long-running functionality that does not interfere with other users who are working on the same computer.

Windows service applications have specific development and installation requirements:

- You cannot simply start the service from the Visual Studio .NET environment to test and debug it—you must install the application as a service first, using Installutil.exe or a Windows Installer file.
- Registry entries need to be made for each service in your Windows service application, such as the start type (manual, automatic, or disabled).
- Registry entries need to be made for the service executable, such as the security account used to run the service (the default is LocalSystem, but you can change this to suit your specific needs).
- The Service Control Manager must be informed that your service is installed.

The easiest way to install your Windows service application to meet these requirements is to add installation components to your project. You will use two types of installation components to install Windows service applications:

- ServiceInstaller. You will use one ServiceInstaller component per service contained in your project. You will use this component to specify settings such as start type.
- ServiceProcessInstaller. You will use only one ServiceProcessInstaller for your Windows service application, regardless of how many services it contains. This component controls installation settings for the executable file itself, such as the security account used to run the service application.

You can run the installation components either as custom actions in your Windows Installer file or with the Installutil.exe utility.

For more information about creating and installing Windows service applications, see "Adding Installers to Your Service Application" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbtskAddingInstallersToYourServiceApplication.asp*).

Deploy Enterprise Services Components

To take advantage of COM+ services, such as transaction management, object pooling, activity semantics, and so on, your .NET application uses managed classes that are built and then installed in COM+ applications. In the .NET Framework, these managed classes are known as serviced components.

Because your serviced components are hosted by a COM+ application, they must be accessible to that application. This introduces the following registration and configuration requirements:

- The assembly must be strong-named.
- The assembly must be registered in the Windows registry.
- Type library definitions for the assembly must be registered and installed into a specific COM+ application.

Use Dynamic Registration Only if Your Process Has Administrative Privileges

Serviced components can actually register automatically the first time they run. The first time a client attempts to create an instance of a serviced component, the CLR registers the assembly, the type library, and configures the COM+ catalog. Registration occurs only once for a particular version of an assembly. This is the easiest method for registering your serviced components, but it works only if the process running them has administrative privileges. Also, assemblies that are marked as a COM+ server application require explicit registration, and dynamic registration does not work for unmanaged clients calling managed serviced components.

Custom attributes are used to specify the services that are required, such as the Transaction custom attribute. These attributes store the configuration options for a service in the assembly metadata. The following example illustrates the use of the Transaction attribute for an e-commerce component that processes orders:

```
Imports System.EnterpriseServices
Imports System.Runtime.CompilerServices
Imports System.Reflection
‘ Supply the COM+ application name.
<Assembly: ApplicationName(“CommerceTransaction”)>
‘ Supply a strong-named assembly.
<Assembly: AssemblyKeyFileAttribute(“CommerceTransaction.snk”)>
Namespace CommerceTransaction
<Transaction(TransactionOption.Required)> _
Public Class Orders
  Inherits ServicedComponent
  <AutoComplete()> _
  Public Sub AddOrderLine _
    (ByVal OrderID As Integer, ByVal ProductID As Integer, _
     ByVal UnitPrice As Double, ByVal Quantity as Integer)
    ‘ Insert Data Access Code Here
  End Sub
End Class
End Namespace
```

The preceding example specifies that transaction services are required for this component. Also, the AutoComplete Attribute is applied to the AddOrderLine method. This attribute instructs the runtime to automatically call the SetAbort function on the transaction if an unhandled exception is generated during the execution of the method, or to call the SetComplete function if no unhandled exceptions occur.

The custom attributes are used by having code load the assembly and using reflection to extract the service configuration stored in the attribute. The information is then written to the COM+ catalog. The code that performs these and other steps is contained in EnterpriseServices.RegistrationHelper. However, be aware that the process that attempts to load the assembly and write the configuration to the COM+ catalog must have administrative privileges, otherwise it does not work.

For more details about the RegistrationHelper class, see "Understanding Enterprise Services in .NET" on the .NET Framework Community Web site, GotDotNet (*http://www.gotdotnet.com/team/xmlentsvcs/espaper.aspx*).

Use an Installer Class to Register Your Serviced Component

In most cases, the process that uses your assembly does not have the required privileges for dynamic registration to occur successfully. For example, when the assembly is used in a Web application, ASP.NET is not able to register your serviced component. ASP.NET does not run with administrative privileges by default (unless you set it to run as SYSTEM, which is not recommended for security reasons) so when your serviced component attempts to register, it fails with an access denied exception.

To deploy a serviced component where the process using it will not have the required privileges for dynamic registration to occur, you can register the serviced component in one of the following ways:

- Use the Regsvcs.exe utility from the .NET Framework Software Development Kit (SDK) to manually register your assembly containing serviced components.
- Programmatically register your serviced component using an installer class.

If you programmatically register your component with an installer class, you still need to ensure that the process running your installation code has the required privileges. A typical way to ensure this is to create a Windows Installer file for your solution, such as a Web setup project. You add the installer classes as custom actions to your setup project, in a similar manner to the predefined installation components described previously in this chapter. The Windows Installer files for your Web applications are typically run by an administrator, so the required privileges are met.

However, unlike the predefined installation components, you need to write the installation code for your components yourself—for example, you need to override the Install and Uninstall methods for your installer class. In these methods, you use the RegistrationHelper class to register (and unregister) your component. For an example of how to do this, see "Deploying N-Tier Applications with Visual Studio .NET Setup Projects" on GotDotNet (*http://www.gotdotnet.com/team/xmlentsvcs/deployntier.aspx*).

Install Serviced Components into the Global Assembly Cache

In most cases, you should deploy your serviced components into the global assembly cache. Only if the component is going to be used by a single application and is a library component, should you consider not installing it in the global assembly cache.

Consider Traditional COM+ Deployment Issues for Your Serviced Components

Because serviced components take advantage of COM+ services, they still require some of the deployment considerations associated with COM+ components to be taken into account. These include:

- If you are using distributed COM (DCOM) instead of .NET remoting, you need to deploy application proxies to client computers just as you would for a traditional COM+ application.
- In addition to any COM+ settings that can be set via attributes on the assemblies themselves, you must make sure other configuration settings (such as adding users to roles and setting the process security identity) are created and assigned correctly. This is usually done via script at install time and can also be done programmatically in a custom action in your setup project. This process could be added to the same custom action that registered the component.

For more information about proxies, see "Deploying Application Proxies" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgdeployingapplications_65gz.asp*).

Understand How Side-By-Side Issues Affect Serviced Components

As you update your serviced components, or if you have a scenario where multiple versions of the same component are on the same computer, you can encounter several issues caused by having these components installed side by side. To help you maintain your serviced components, you should adhere to the following advice:

- Do not use globally unique identifiers (GUIDs) as the GuidAttribute class or ApplicationIDAttribute. Instead use the ApplicationNameAttribute. If you do use a GUID, you need to hard code the GUID into the assembly, which requires you to manually change this GUID if you update your component and have it run alongside the original version; otherwise, the COM+ installation overwrites the settings for the earlier version.
- Remember that assembly versioning applies only to the assemblies themselves, not the COM+ application. There is no automatic way to version the application itself. If the newer version of a serviced component requires changes at the COM+ application level that will break previous versions of the component, then you need to use the Application Name to indicate versioning by installing the newer version of the serviced component into its own COM+ application.

For more information about Enterprise Services Components, see "Understanding Enterprise Services (COM+) in .NET" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/entserv.asp*).

Deploy Instrumented Assemblies

If your solution uses Windows Management Instrumentation (WMI) features, you need to ensure that the core WMI is installed on your target computers. WMI is included as a standard feature in Windows XP, Windows 2000, and Windows Me. Additionally, it is automatically installed by Visual Studio .NET on Windows 95, Windows 98, and Windows NT 4.0. However, if your target computers do not have the core WMI installed, a setup package can be downloaded and installed from MSDN Code Center (*http://msdn.microsoft.com/downloads/?url=/downloads/sample.asp?url=/msdn-files/027/001/576/msdncompositedoc.xml*).

Note: Although it is not really a deployment issue, it is worth mentioning that you can download and install the Visual Studio .NET Server Explorer Management Extensions if you are developing WMI-enabled applications. The Server Explorer Management Extensions are a set of development tools that provide browsing and RAD capabilities for building WMI-enabled applications. For more information and to download these management extensions, see "Managing Windows Management Instrumentation (WMI) Date and Events with Service Explorer" on MSDN (*http://msdn.microsoft.com/library/en-us/WMISE/wmioriManagingWindowsManagementInstrumentationWMIDataEventsWithServerExplorer.asp*).

In addition, an instrumented application needs to undergo a registration stage, in which its schema can be discovered and registered in the WMI repository. Schema publishing is required on a per assembly basis. Any assembly that declares instrumentation types (events or instances) must have its schema published to WMI. You can use installer classes for deploying and registering instrumented classes.

If your project does not include installer classes for other application resources (such as message queues, event logs, services, or performance counters), you can use the DefaultManagementProjectInstaller class to install your instrumented application. To use this default project installer, simply derive a class from DefaultManagementProjectInstaller inside the assembly. No methods need to be overridden. However, if your project already uses a master project installer for other application resources, you should instead use a ManagementInstaller class for your instrumented assembly. To install successfully, your installer classes must be run by a process with administrative privileges. The easiest way to ensure this happens is to have your installer classes run as custom actions as part of a Windows Installer file—typically, Windows Installer files are run by users who are members of the local administrators group, so required privileges are in place. Alternatively, you can have an administrator manually run the installer classes with the Installutil.exe utility, similar to the predefined installation components and the installers for serviced components discussed earlier.

For more information about registering WMI schemas for your instrumented applications with the ManagementInstaller class, see "Registering the Schema for an Instrumented Application" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconregisteringtheschemaforaninstrumentedapplication.asp*).

Manage the Deployment of Configuration Files

Configuration files can be used to store information about everything from process models, through application settings, to security and authentication models. There are three main types of configuration files:

- **Application configuration files**. Application configuration files contain settings specific to an application. This file contains configuration settings that the CLR reads (such as assembly binding policy, remoting objects, and so on), and settings that the application can read. The name and location of the application configuration file depend on the application's host, which can be one of the following:
  - For an executable file, the configuration file resides in the same folder as the executable file, and will be called <AppName>.exe.config, where <AppName> is the name of your executable (for example, MyApp.exe.config).
  - For Web applications, the configuration file is named Web.config and resides in the virtual directory for your application. Subdirectories of your Web application can also have a Web.config file, and the pages in the subdirectory are governed by both the settings contained in that specific Web.config as well as the settings contained in Web.config for the parent application's virtual

  directory. The settings contained in the configuration file in the virtual directory in which the Web files exist take precedence over the configuration file in the parent application's virtual directory if the settings conflict.
  - For Internet Explorer-hosted applications, a configuration file can be referenced by the <link> tag with the following syntax:

    <link rel="ConfigurationFileName" href="<location>">. For more information about Internet Explorer-hosted applications, see "Configuring Internet Explorer Applications" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconconfiguringieapplications.asp*).

- **Machine configuration files**. The machine configuration file, Machine.config, contains settings that apply to an entire computer. This file is located in the <runtime install path>\Config directory. Machine.config contains configuration settings for computer-wide assembly binding, built-in remoting channels, and ASP.NET.
- **Security configuration files**. Security configuration files contain information about the code group hierarchy and permission sets associated with a policy level. For more information about security configuration files, see the "Deploy Security Policy" section later in this chapter.

Manage the Deployment of Environment-Specific Configuration Settings

As far as physical deployment is concerned, configuration files do not raise any additional special requirements than those for the rest of your application—they are, after all, simple text files. For example, if you are using Windows Installer files you can simply include your configuration files for deployment; if you are using copy mechanisms, then again you simply need to ensure you copy the configuration files with your application; if you are using Application Center to deploy to server environments, then you simply need to ensure that the configuration files are also deployed and synchronized across your server clusters. Rather than introducing physical deployment issues, distributing configuration files raises issues that have more of a management nature.

The most pervasive issue concerning the deployment of configuration files is how you manage settings for your application as you deploy to different environments. For example, if you store Web service URLs for your Web references in your application configuration file, these settings are often different in the development, test, staging and production environments. The challenge you face is how to deploy the appropriate settings to each environment.

Create a Configuration File per Environment

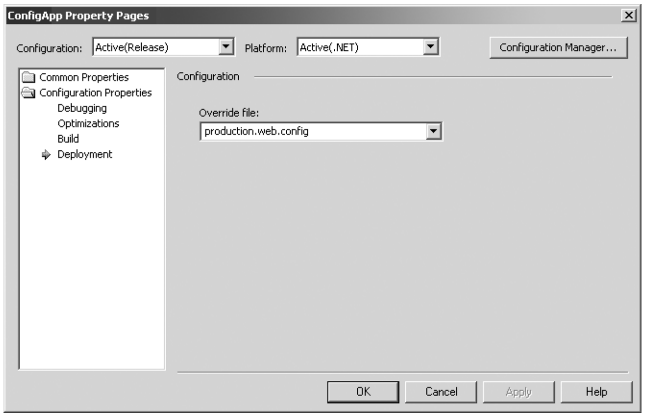
To manage different configuration files for different environments, you should:

1. Add the configuration file that is to be used in your development environment to your Visual Studio .NET project. This file is named either Web.config or App.config, depending on your application type. (Note that for ASP.NET applications, Visual Studio .NET automatically adds the Web.config file to your project for you, when you create the application).
2. Create separate configuration files for the test, staging, and production environments, in addition to the application configuration file your developers use in the development environment. Name these files so that it is immediately obvious which environment they are to be used in. For example, if you are developing a Web project, you might name your files Test.web.config, Stage.web.config, and Production.web.config.
3. Add each file to your project. They are then maintained in your source control system.
4. When you deploy your application to a specific environment, include only the file you need for that environment and rename it to the required name as part of the deployment. The required name is either Web.config for ASP.NET applications, or <AppName>.exe.config for Windows applications. (Note that you could deploy all of the other configuration files along with your application and it does not affect your application in any way. However, because these files are not actually needed, you should avoid deploying them along with the application for efficiency.)

If you are using Visual Studio .NET setup and deployment projects to package and distribute your solution, you should manage the renaming of your environment-specific configuration files with the configuration override file option for your project. This option is discussed in the next topic. If you are using some other mechanism for deploying your solution, such as XCOPY deployment, or Application Center deployment, then you need to manage the renaming of the environment-specific configuration file some other way. Typical approaches include renaming the configuration file with setup script, or even having the person deploying your solution manually rename the file.

Use the Configuration Override File in Visual Studio .NET Projects

Visual Studio .NET allows you to specify a configuration override file for your application. This setting is available on the Build page of the **Configuration Properties** section in the **Property Pages** dialog box for applications developed with the Microsoft Visual Basic® development system, as illustrated in Figure 4.1.



Override file: production.web.config

Figure 4.1
Configuration override file setting for Visual Basic projects

The next topic discusses how to use the configuration override file option for Microsoft Visual C#™ development tool projects.

The override file allows you to specify which configuration file should be built into a Windows Installer file when you build the setup project for your application. To use the configuration override file option, you should:

1. Maintain different configuration files for each environment in your project (and source control system), as recommended earlier. (If the environment-specific configuration files are not included in your project, then you will not be able to use them as configuration override files).
2. Add a Visual Studio .NET setup and deployment project to your solution.
3. Add the project output for your application to the setup project.
4. Specify the appropriate configuration file in the configuration override option in the property pages of your *application's* project, before you build the setup project. For example, if you are about to build a Windows Installer file for deploying your application to the production environment, you will specify Production.app.config or Production.web.config as the override file for your application's project (if you follow the naming convention described in earlier).
5. Build your setup project for the environment in question. Visual Studio .NET packages the configuration override file with your other build outputs and renames it to the required name for you in the Windows Installer file. For example, if you are packaging an ASP.NET application, Visual Studio names the configuration override file Web.config. When you run the installer, the configuration file is deployed along with your application.

Use the Configuration Override File Option with Your C# Projects

The override file setting is not available in the Property pages of C# applications, but you can still use configuration override files for your C# solutions. You need to edit the .csproj file to achieve this. The following example shows how to specify a configuration override file in the .csproj file for a C# Web application.

```
…
…
<Config
  Name = “Release”
  AllowUnsafeBlocks = “false”
  BaseAddress = “285212672”
  CheckForOverflowUnderflow = “false”
  ConfigurationOverrideFile = “production.web.config”
  DefineConstants = “TRACE”
  DocumentationFile = “”
  DebugSymbols = “false”
  FileAlignment = “4096”
  IncrementalBuild = “false”
  Optimize = “true”
  OutputPath = “bin\”
  RegisterForComInterop = “false”
  RemoveIntegerChecks = “false”
  TreatWarningsAsErrors = “false”
  WarningLevel = “4”
/>
…
…
```

In the preceding example, note that the ConfigurationOverrideFile entry is set for the Release section—it is possible to specify different override files for different build configurations, either by editing the .proj file as shown, or by using the project property pages for Visual Basic applications.

Deploy the Production Configuration File to the Test Environment and Replace it after Install

The deployment process used to install your application to the test environment should be as similar as possible to the process you use to deploy your solution to the production environment. If you are deploying different configuration files to these environments, this raises the issue that the deployment of the production configuration file will not be fully tested before your roll out your solution. One approach to solving this issue is to actually deploy the *production* version of your configuration file to the test environment, rather than the test version. That way, you can ensure that it is distributed correctly with your Windows Installer file, copy operations, Application Center, or whichever mechanism you choose. Your testers can then perform the supplementary task of replacing the production version of the configuration file with the test version *before* running the solution in the test environment. This should be a separate step, in order to avoid introducing unanticipated issues into the deployment process.

Store Application Settings in Your Application Configuration File

Another issue is which configuration files you should use to store certain settings. Many settings, such as assembly redirection, can be specified at either the application level or at the computer level. In general, you should store application settings in the application configuration files (such as Web.config or the configuration file for your executable file) rather than in the Machine.config file. The Machine.config file stores settings for the entire computer—this can lead to an environment where settings become difficult to manage. Although this is more of a design issue than a deployment one, it is nevertheless important to bear this in mind as you plan for the deployment of your .NET applications.

Deploy Security Policy

Security policy is managed by administrators at three different levels: enterprise, computer, and user. In addition, developers can specify security policy in code for their application domains.

Use the .NET Framework Configuration Tool to Package Security Policy in Windows Installer Files

As far as deployment is concerned, security policy can be easily distributed and applied in Windows Installer files. The .NET Framework configuration tool (Mscorcfg.msc) provides a wizard for creating Windows Installer files for your security policy. The wizard creates a Windows Installer file that corresponds to one of the three configurable policy levels (enterprise, computer, and user), but not all of them concurrently. If you are deploying security policy for all three configurable levels, you must create three different Windows Installer files and deploy them individually.

The wizard creates the Windows Installer file using the current policy settings of the computer where the wizard executes. For example, to create a user policy for deployment to a group of users, you configure the user policy on your current computer, create the Windows Installer file with the wizard, and then return the user policy of the current computer to its original state.

You need to ensure that the user account under which the policy is installed has adequate privileges to access the configuration files you are modifying. For example, if you are currently logged on using an account that does not have permission to modify the enterprise configuration file, and the Windows Installer file needs to modify that file, the installation does not succeed. Note that the Windows Installer package does not produce an error if the current account does not have sufficient permission to modify the configuration file.

Deploy Your Security Policy Windows Installer File

Like any other Windows Installer file, you can deploy your security policy Windows Installer file with a number of different mechanisms. These include having your users run the installer locally or from a network share, deploying the installer with SMS, and deploying the installer with Active Directory Group Policy for software distribution. For a discussion on using these (and other) deployment mechanisms, see Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Application."

If you need to deploy security policy with your application, you can create a nested setup whereby the security policy installer is launched from the application installer. For more information about creating nested setups, see Chapter 3, "Visual Studio.NET Deployment Projects."

Use Scripts and Caspol.exe to Apply Security Policy

Rather than deploying security policy with Windows Installer files, you can use scripts and batch files along with the Code Access Security Policy tool (Caspol.exe) to apply security policy settings. If you use this approach, you need to disable the prompt for your users that security policy has been changed. You can achieve this by including the following command as the first entry in your batch file:

```
Caspol –pp
```

For more information about using this tool, see "Code Access Security Policy Tool (Caspol.exe)" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfcodeaccesssecuritypolicyutilitycaspolexe.asp*).

For more information about administering security policy, see "Security Policy Administration Overview" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsecuritypolicyadministrationoverview.asp*).

For more information about specifying security policy in code for application domains, see "Setting Application Domain-Level Security Policy" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconsettingapplicationdomainlevelsecuritypolicy.asp*).

For more information about using the Mscorcfg.msc wizard, see ".NET Framework Configuration Tool (Mscorcfg.msc)" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconnetframeworkadministrationtoolmscorcfgmsc.asp*).

Verify Permissions Granted to Your Assemblies

After you set security policy, you can verify the permissions that are granted to your assemblies by that security policy. The .NET Framework configuration tool contains a wizard that you can use to view the permissions given to an assembly by current security policy.

► **To access and run the .NET Framework Configuration wizard**

1. In Control Panel, open the Administrative Tools folder.
2. Double-click the **Microsoft .NET Framework Configuration** icon.
3. In the Microsoft .NET Framework Configuration tool, right-click **Runtime Security Policy**.
4. Click **Evaluate Assembly**.
5. Browse to or enter the location of the assembly you want to test.
6. Under **Choose the type of evaluation**, indicate whether you want to see the permissions the assembly receives from policy or which code groups in security policy apply to it.
7. Keep the All Levels policy level setting, unless you want to analyze a specific policy level's contribution to the overall set of permissions the assembly receives.
8. Click **Next** to view the list of permissions granted.

For answers to frequently asked question regarding the deployment of security policy, see ".NET Framework Enterprise Security Policy Administration and Deployment" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/entsecpoladmin.asp*).

Deploy Session State Settings with Your Solution

ASP.NET offers three models for storing session state for you Web applications:

- **In process (InProc)**. State information is stored in memory by ASP.NET. This is conceptually similar to previous versions of ASP, and is the default setting.
- **StateServer**. State information is stored and managed by the StateServer service, separately from ASP.NET. This provides for more robust state management. For example, if the IIS process fails, it does not affect your state information—when IIS is restarted, saved state information is still available. Furthermore, the StateServer can reside on a different computer, meaning that your state information can actually survive reboots of the Web server. One StateServer can be used to manage state information for multiple Web servers, such as in a Web farm configuration.
- **SQLServer**. State information is stored in a Microsoft SQL Server™ database. This provides for very robust and scalable state management. One SQL Server

can be used to manage state for many Web servers concurrently. Furthermore, because the state information is stored in a database, rather than in memory, it can survive reboots of both the Web server *and* the computer running SQL Server.

Important: There is a deployment issue with using InProc session state of which you should be aware: When a file is updated in a Web application, ASP.NET loads a new application that incorporates your changes and uses that application to deal with new requests. At the same time, it maintains the previous application to deal with existing requests. After those existing requests are satisfied, ASP.NET removes the original application from memory. This results in a loss of state information that was stored in the Application object, the Session object, and other cached values. To avoid this potential problem, you should use either a StateServer (which runs independently from the ASP.NET Web application process), or use the SQLServer option for complete and robust persisted state management.

The state management used by your Web application is specified in either your Machine.config file or in the Web.config file for your Web application. The default setting is InProc. To use a state server, specify StateServer and provide a stateConnectionString in your configuration files. To use a SQL Server, specify SQLServer and provide a sqlConnectionString in your configuration files. If you use SQL Server to manage state for your Web applications, you must also run the InstallSqlState.sql script (located in <Windows Dir>\ Microsoft.NET\Framework\ <VersionNumber>) on your SQL Server to create the session schema. This script creates a database called ASPState that includes several stored procedures, and adds the ASPStateTempApplications and ASPStateTempSessions tables to the TempDB database. There is also a UninstallSqlState.sql script in the same directory that can be used to uninstall the session database.

As far as deployment is concerned, you simply need to ensure that the correct configuration settings are in place for each environment. For example, if you are using state servers or SQL Servers for your state management, it is likely that you will use different computers for your development, test, staging, and production environments. The key to a successful deployment between these environments is to manage your configuration files so that the appropriate settings and connection strings are deployed for these different environments.

For more information, see the "Manage the Deployment of Configuration Files" topic earlier in this chapter.

COM Interop Deployment Considerations

The interoperability features of .NET allow you to work with existing unmanaged code (that is, code running outside the CLR) in COM components as well as Microsoft Win32 DLLs. It also allows you to use managed components from your unmanaged, COM-based code.

When a .NET component is called from COM, the runtime generates a wrapper object to bridge the gap between the managed and unmanaged environments. In this case, the runtime reads the type information for the component from its assembly metadata and generates a compatible COM callable wrapper (CCW). The CCW acts as a proxy for the unmanaged object. When a COM object is called from .NET, the runtime generates a runtime callable wrapper (RCW). Similar to the CCW, the RCW acts as a proxy between the managed .NET code and the unmanaged COM code. RCW and CCW are created at run time; therefore, they are not a deployment issue.

The RCW is created at run time from type information that is stored in an interop assembly. An interop assembly, unlike other .NET assemblies, contains no implementation code. Interop assemblies contain only the type definitions of types that are already implemented in COM components. It is from these type definitions that the CLR generates the RCW to allow managed code to bind to the types at compile time and provides information to the CLR about how the types should be marshaled at run time. In order for .NET to call COM components, the appropriate interop assemblies must be deployed with the solution.

Deploy All Required COM Components with Your Application

As with the deployment of unmanaged applications (before the advent of .NET), COM objects with which your .NET applications interoperate must be registered on the target computer. Similarly, COM+ components with which your .NET applications interact must be properly installed into the COM+ catalog.

Any COM components required by your .NET application should either already exist on the target computers or be deployed along with your application. If you are using Visual Studio .NET setup and deployment projects, the COM dependencies of your .NET assemblies are detected and included in the Detected Dependencies list. However, any other dependencies of the COM object cannot be automatically detected and must be manually included into the setup project for deployment.

For more information about deploying all required COM components, see the following articles on MSDN:

- "Deploying COM+ Applications" (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/pgdeployingapplications_5xmb.asp)
- "Application Deployment Using Microsoft Management Technologies" (*http://www.microsoft.com/windows2000/techinfo/howitworks/management/apdplymgt.asp*)

.NET Calling into COM

For .NET to talk with COM, you need an interop assembly, preferably a primary interop assembly.

For .NET to correctly call to COM the interop assembly, each COM object referenced must be deployed along with the application. There are two types of interop assemblies:

- **Primary interop assemblies**. Primary interop assemblies are interop assemblies that are provided by the same publisher as the type library they describe, and they provide the official definitions of the types defined with that type library. They are always signed by their publisher to ensure uniqueness. A primary interop assembly is created from a type library by running TlbImp.exe with the "/primary" switch.
- **Alternative interop assemblies**. Alternative interop assemblies are any non primary interop assembly. They are not signed by the COM object publisher.

If you do not have a primary interop assembly, alternative interop assemblies can be created by Visual Studio .NET (by setting a reference to a COM object) or by Tlbimp.exe. If you use Visual Studio .NET to create a reference to a COM object, Visual Studio .NET looks in the registry to see if a primary interop assembly is registered on that computer for the COM object being referenced. If one is registered, it uses the primary interop assembly instead of generating one. If there is no primary interop assembly, Visual Studio .NET creates an alternative interop assembly. If you create a reference to a COM object in Visual Studio .NET, the interop assembly generated cannot be strong named. If you want a strong named alternative interop assembly, you must manually create it, strong name it, and then set a reference to it. Alternatively (and preferably), you can just use a primary interop assembly, because all primary interop assemblies are strong named.

Note: The Type Library Importer (Tlbimp.exe) converts most COM method signatures into managed signatures. However, several types require additional information that you can specify by editing the interop assembly. For more information about Tlbimp.exe, see "Type Library Importer (Tlbimp.exe)" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrftypelibraryimportertlbimpexe.asp*).

If you use Visual Studio .NET to create a reference to a COM object, it looks in the registry to see if a primary interop assembly is registered on that computer for the COM object being referenced. If it has, Visual Studio .NET uses the primary interop assembly instead of generating an alternative interop assembly. If there is no primary interop assembly, Visual Studio .NET creates an alternative interop assembly. If Visual Studio .NET creates an alternative interop assembly, it cannot be strong named. If you want a strong named interop assembly you must manually create it, strong name it, and then set a reference to it. Alternatively (and preferably), you can use a primary interop assembly if one exists, because all primary interop assemblies are strong named.

Always Use a Primary Interop Assembly When Available

When using the types defined in a type library, you should always refer to the primary interop assembly (if available) for that type library, rather than creating an alternative interop assembly. Doing so helps to prevent type incompatibilities in code written by developers using the same COM object. You should also provide primary interop assemblies for any components you develop that might be used by others, especially if third-party developers or customers will use these components in their applications.

Note: Primary interop assemblies are important because they uniquely identify a type. Types defined within an interop assembly that was not provided by the component publisher are not compatible with types defined in the primary interop assemblies. For example, consider two developers in the same company who are writing managed code that will interoperate with an existing third-party supplied COM component. One developer acquires the primary interop assembly from the component publisher. The other developer generates his or her own interop assembly by running Tlbimp.exe against the COM object's type library. Each developer's code works properly until one of the developers (or worse yet, a third developer or customer) tries to pass the object to the other developer's code. This results in a type mismatch exception; although they both represent the same COM object, the type checking functionality of the CLR recognizes the two assemblies as containing different types.

Use Regasm.exe and the Global Assembly Cache for Primary Interop Assemblies on Developer Computers

When deploying a primary interop assembly to developers' computers, you should always register the primary interop assembly with Regasm.exe on the developer computer. This places entries into the registry that associate the primary interop assembly with the COM object. In addition to using Regasm.exe, you must then place the primary interop assembly in the global assembly cache on the developer's computer. When a developer then adds a reference to the COM component in Visual Studio .NET, the fact that there is a primary interop assembly associated with this COM component is detected and Visual Studio .NET does not generate an alternative interop assembly. At run time, the primary interop assembly is found in the global assembly cache and loads properly on the developer computer.

Deploy Primary Interop Assemblies in the Global Assembly Cache

On non-developer computers, there is no need to use Regasm.exe against the primary interop assembly. The registry entries created by this are simply used by Visual Studio .NET to designate that a primary interop assembly exists for the COM component. On a non-developer computer, such as your production environment, you can either place the primary interop assembly in the global assembly cache or deploy it as a private assembly. Because COM components have a computer-wide visibility, you should install primary interop assemblies in the global assembly cache so that the primary interop assembly for the component is visible to all .NET applications on that computer.

COM Calling to .NET

When deploying .NET assemblies that will be consumed by COM, you must make the .NET assemblies visible to COM.

► **To make an assembly visible to COM**

1. Strong name your NET assemblies (optional, but **highly** recommended).
2. You must register the .NET assembly by creating an entry in the registry that points to Mscoree.dll, which loads and executes the .NET assembly. This can be easily done in one of two ways: You can use a Visual Studio .NET setup project (or third-party installation tool) that will create the appropriate registry entries upon install, or you can use the Regasm.exe tool to register the .NET assembly. There is a Codebase switch on the Regasm.exe tool creates a Codebase entry in the registry. The Codebase entry specifies the file path for an assembly that is not installed in the global assembly cache. This switch is mainly used only in development and you should not specify this option if you subsequently install the assembly that you are installing into the global assembly cache
3. After the .NET assembly is registered, it must be located where it can be found by the .NET runtime. This means it must be placed in one of the following locations:
   - The global assembly cache, which is the usual and recommended place
   - The application base directory, which is usually the .exe directory or the bin directory in a Web application

Do Not Deploy COM Visible .NET Assemblies to the Application Base Directory

The recommended location for .NET assemblies that are being called by COM components is in the global assembly cache for most scenarios. If you do not deploy these to the global assembly cache, you may encounter problems. For example, for traditional Active Server Pages (ASP) applications, the root application is either Inetinfo.exe or Dllhost.exe (depending on the isolation setting in IIS), both of which are located in the System32 directory. This means that you would need to deploy the .NET assembly to this directory—this is **not** recommended. In addition, when developing in Microsoft Visual Basic 6.0 you must put the .NET assembly in the VB6.exe directory because that is where it runs during debug. To avoid the need to deploy .NET assemblies to these locations, and to have a cleaner deployment of your application, you should deploy a .NET assembly that is going to be called by COM in the global assembly cache. If your application is completely isolated and all of its components are deployed in the same place, there is no need to deploy an assembly to the global assembly cache; it can safely be deployed with other components.

Manage Debug Symbol Files for Deployment

You can manage the configuration of your build outputs when you build your solution. Visual Studio .NET provides two default configurations for your project: release and debug. One of the differences between these two configuration modes is the way that debugging symbols are managed. In the debug configuration, a debugging symbols file program database is created, whereas for the release configuration, debugging information is not generated by default.

The program database file contains the information needed to map the compiled Microsoft Intermediate Language (MSIL) back to the source code; this enables debugging tools to fully report information such as variable names. In addition, the just-in-time (JIT) compiler can generate tracking information at run time to map the native code back to the MSIL. Both tracking information and symbol files are needed to effectively debug managed code. In Visual Studio .NET, generation of the program database file and tracking information is controlled by the **Generate debugging information** option on the build page of the configuration section in the project property pages of your projects. If you compile your solution with the command line, you can specify the **/debug** switch with various options to control the generation of debugging symbols and tracking information.

Although the release configuration does not, by default, generate debugging symbols, you should consider changing this default behavior. If you have deployed your solution to the production environment and unexpected behavior occurs, you will not be able to debug that version if you have not generated the debugging symbols for that build. Because you can use the debugging symbols for only a particular build, you will not be able to use the debugging symbols generated for the debug build with the release version of your solution. If you have not generated debugging symbols for the release version of your application, but you find that you need to debug it in the production environment, you need to rebuild the solution, including debugging information, and redeploy the solution to be able to debug it in the production environment. Not only does this mean that you have to undertake a significant amount of additional work in order to debug the production version of your solution, it can also introduce factors that may prevent you from reproducing the errors that were previously encountered, because you will be working with a different application build.

You can specify that the release configuration should generate the debugging symbols files by modifying the project property pages when the **Release** configuration is selected.

You do not want to distribute the symbol files with your application to the production environment initially, because this allows customers or other users to more easily reverse engineer your solution (which is obviously a security concern). Instead, you should generate the debug symbols file with your release configuration, but keep them secured separately from the deployed solution. That way, you can install them into the production environment if (and only if) you need to debug the released application.

To help ease the management of symbols across your enterprise, you can use Microsoft Symbol Server to store your symbols in a centralized location and have the debuggers access them when they need them. An in-depth discussion of symbol management is beyond the scope of this guide. For more information about symbol management, see the following:

- "Bugslayer: Symbols and Crash Dumps" (*http://msdn.microsoft.com/msdnmag/issues/02/06/Bugslayer/Bugslayer0206.asp*)
- "How to Get Symbols" (*http://www.microsoft.com/ddk/debugging/symbols.asp*)
- "INFO: Use the Microsoft Symbol Server to Acquire Debug Symbol Files" (*http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q311503*)
- "INFO: PDB and DBG Files - What They Are and How They Work" (*http://support.microsoft.com/default.aspx?scid=kb;en-us;Q121366*)

Deploy Remoting Information with Your Application

You can use .NET remoting to enable different applications to communicate with one another, whether those applications reside on the same computer, on different computers in the same local area network, or thousands of miles away over a wide area network (WAN) link or the Internet. The applications can even be running on different operating systems. The .NET Framework provides a number of services such as activation and lifetime control, as well as communication channels responsible for transporting messages to and from remote applications.

You must provide the following information to the remoting system to make your types remotable:

- The type of activation required for your type.
- The complete metadata describing your type.
- The channel registered to handle requests for your type.
- The URL that uniquely identifies the object of that type. In the case of server activation, this means a URI that is unique to that type. In the case of client activation, a URL that is unique to that instance will be assigned.

There are two ways to provide this information to configure your remotable types: You can either:

- Call configuration methods directly in your server and client code. For more information about how to achieve this, see "Programmatic Configuration" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconprogrammaticconfiguration.asp*).

- Create a remoting configuration section and include it in your application's configuration file, Web.config file, or Machine.config file. For more information about the remoting elements you can use in your configuration file, see "Remote Object Configuration" on MSDN (*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconremoteobjectschannels.asp*).

As far as deployment is concerned, ensuring that remoting information is distributed with your application is a simple matter. If you call configuration methods directly in your application code, you do not need to take any remoting-specific deployment steps. However, you are not able to change key properties at run time, such as the end point URL. If you configure remoting in configuration files, you need to ensure that the appropriate configuration file is deployed along with your application. You can then make any changes as required, and redeploy just the configurations file to have those changes applied.

For more information about deploying configuration files, see the "Manage the Deployment of Configuration Files" topic earlier in this chapter.

Deploy Localized Solutions

The CLR provides support for retrieving culture-specific resources that are packaged and deployed in satellite assemblies. Satellite assemblies contain only resource files, or loose resources such as .gif files. They do not contain any executable code.

In the satellite assembly deployment model, you create an application with a default assembly (the main assembly) and several satellite assemblies. You should package the resources for the default language neutral assembly with the main assembly and create a separate satellite assembly for each language that your application supports. Because the satellite assemblies are not part of the main assembly, you can easily replace or update resources corresponding to a specific culture without replacing the application's main assembly.

If you have many different languages for which you need to localize your application, you can streamline the packaging and deployment process by:

- Packaging your core, language neutral files into a merge module (MSM) using a Visual Studio .NET setup and deployment project.
- Creating a project that contains the localized resources for *all* languages that you want to support.
- Creating a separate Windows Installer file for each language you are localizing to, using Visual Studio .NET setup and deployment projects. This allows you to localize the installer as well as the application.
- In each Windows Installer project, add the output for the project that contains the localized resources. That essentially includes *all* of the localized resources. You can then use the ExcludeFilter to filter out all but the one localized resource that you need for your specific language.

As an alternative, instead of filtering out the localized resources that are not needed, you can simply distribute them all. That way, your clients can use the different localized resources on the same computer; for example, they can change their locale settings and have the appropriate resources loaded by your application. Of course, this requires more disk space, but probably not a significant amount.

Another approach is to create a core installer and a separate set of localized installers (typically known as "language packs"). Your global customers can then install your core application and then one (or more) of your language packs.

For more information about localizing Windows Installer files, see Chapter 3, "Visual Studio.NET Deployment Projects."

Deploy Registry Settings with Your Solution

Modern .NET applications should be less reliant on registry entries than traditional solutions. For example, assemblies do not require registry entries, unlike COM components in the past. However, your solution might still rely on registry settings in some cases.

If you do need to include registry settings with your solution, you can either distribute and run .reg files as you deploy your settings, or, more typically, include registry settings with your Windows Installer files.

For more information about deploying registry settings with Windows Installer files, see Chapter 3, "Visual Studio .NET Deployment Projects."

5

Choosing Deployment Tools and Distribution Mechanisms for Your .NET Application

As you plan for how to deploy your Microsoft® .NET application, you need to answer two important questions:

- How will the application be packaged for deployment?
- How will the application be distributed to the production environment?

The answers to these questions are to some extent interrelated—your goal is to choose a strategy both for the packaging and the distribution of your solution that meets your specific needs.

The following sections discuss each of the factors that influence your choices. A number of different scenarios for each factor are discussed, and guidance is provided for choosing appropriate packaging and distribution mechanisms for each scenario. Different variations for each scenario are also discussed—this helps you choose an exact strategy that meets your specific needs.

Choose How to Package Your Solution

There are a number of different options available to you for packaging your .NET application for deployment, and there are a number of different factors that affect which option you choose. This section discusses the implications that each factor has on a number of different scenarios.

Determine How to Package Windows Forms Applications

Windows Forms applications that consist of an executable file (.exe) and perhaps a collection of DLLs, such as a desktop productivity application or a game, usually have distinct installation requirements. You can use a number of different approaches for packaging Windows Forms applications, depending on your specific requirements.

Package Windows Forms Applications in Windows Installer Files to Meet Traditional Installation Requirements

When you require traditional installation features of applications that run on the Microsoft Windows® operating system, such as those in the following list, you should use Microsoft Windows Installer files to package and deploy your solution:

- A simple-to-use graphical user interface (GUI) installation program
- Integration with Control Panel's Add/Remove Programs utility for:
  - Installing.
  - Uninstalling.
  - Adding or removing application features.
  - Repairing a broken installation.
- A robust setup routine that:
  - Rolls back the system to the state it was in prior to the start of installation if any part of the setup routine fails.
  - Rolls back the system to the state it was in prior to the start of installation if the user cancels the installation part-way through the setup routine.
- Requirements for any (or all) of the following:
  - Registry manipulation
  - File associations to be created
  - Assemblies to be installed into the global assembly cache
  - Custom tasks to be run after installation is complete
  - Hardware and prerequisite software checks to be made prior to installation

The Windows Installer service and the Windows Installer database manage the installation process of applications packaged in Windows Installer files, as well as managing those applications when installation is complete.

From the perspective of the end-user, applications packaged into the Windows Installer file format (.msi) provide the following features:

- Familiar GUI-based install routines for end user installation

- Integration with the Add/Remove Programs utility for install, uninstall, adding/removing features, and repair operations
- Robust installation routines with complete rollback to prior system state in case a problem arises during installation or if the user cancels the setup routine

From the perspective of the developer, packaging applications into the Windows Installer file format provides the following features:

- The ability to manage the location of the application files that are installed as part of the setup routine
- The ability to install shared assemblies into the global assembly cache
- The ability to manage registry settings that may be needed by the application
- The ability to design the user-interface for the setup program, so that information can be collected from the user for use in the setup, and so that information can be supplied to the user, such as license agreements, installation progress, and so on
- The ability to manage file associations for the application
- The ability to cancel an installation if the computer does not meet minimum hardware requirements
- The ability to cancel an installation if the computer does not meet minimum software requirements
- The ability to run supplementary tasks after the installation is complete
- The ability to utilize the application version management architecture of the Windows Installer and .msi files to ensure that patches and upgrades for an application install in the correct order

From the perspective of system administrators, applications packaged into the .msi file format provide the following features:

- Integration with electronic software distribution tools, such as Microsoft Systems Management Server (SMS) or Microsoft Active Directory® directory service Group Policies for software distribution, which allows administrators to specify that certain users or certain computers should have a copy of the application installed
- The ability to install applications in silent mode, with no user interaction

Use Visual Studio Setup Projects to Create Windows Installer Files for Your Windows Forms Applications

You can package your applications in .msi files using Microsoft Visual Studio® .NET development system setup projects or using third-party utilities such as Wise or InstallShield. Setup projects allow you to build .msi files from your project outputs (or even just for a collection of files).

For information about how to create Windows Installer files with Visual Studio .NET, see to Chapter 3, "Visual Studio .NET Deployment Projects."

Use Internet Deployment to Achieve Zero Install and Zero Administration for Client Computers

Another way to package Windows Forms applications is to use a new feature of the .NET architecture. This approach is called Internet deployment of .NET applications.

The main advantage of this approach is that although the application runs on the user's download cache, you can still update your application files on the Web server; those changes take place immediately. Before the user's computer loads the assembly, it verifies that the downloaded application files have not changed since it last requested them—if they have, it downloads the later versions as needed.

For information about how this works, see the "Use Internet Deployment for Installing Windows Forms Applications" section in Chapter 4, "Deployment Issues for .NET Applications."

Take Advantage of Zero Deployment Features for Intranets

If you use the Internet deployment of Windows Forms applications in an intranet environment, you can take advantage of the following:

- You can ensure that clients have the .NET Framework installed, which is required for this deployment approach.
- You can deploy security policy with your assemblies over your intranet, which is required if the application is to perform useful tasks. (You are not usually able to deploy security policy for Internet applications.)
- You can more easily ensure that other dependencies of your application, such as COM components, are installed correctly on your users' computers in your intranet environment.

Also note that downloading large application files and assemblies is likely to be less of an issue for intranets than for the Internet.

Be Aware of Caching Issues If You Run Your Application from a Network Share

A variation on the Internet deployment approach, described in Chapter 4, "Deployment Issues for .NET Applications," is to use a network share rather than a Web server to host your .NET applications. Your users can connect to the share and start the executable file from there. This approach is similar to the URL-launched executable file, except that the download cache is not used to cache the downloaded application and assemblies—all assemblies are loaded directly into memory as needed. Consequently, they are downloaded from the network share each time the user runs the application. This variation is a less compelling solution than using a Web server to host your application because it lacks the efficiency of caching downloaded assemblies.

Determine How to Package Web Applications

Different Web applications vary considerably in complexity. They may include any (or all) of the following:

- ASPX or ASMX files (and the associated compiled project DLL)
- A database (or databases)
- Private assemblies
- Shared assemblies that reside in the global assembly cache
- Legacy COM components
- Serviced components (COM+)
- Specific Internet Information Services (IIS) settings
- MessageQueue components
- EventLog components
- PerformanceCounter components
- Other resources that need to be installed on the target computer

Unlike Windows Forms applications, Web applications are usually installed by an administrator or other skilled information technology (IT) professionals. In many cases, they are **not** installed, uninstalled, or repaired using Add/Remove Programs in Control Panel. Additionally, rolling back the installation is not usually required if a problem occurs within the setup routine—it is often easier for the administrator to fix minor problems, such as modifying IIS settings or managing the failed copy of a certain file, than it is to roll back the entire process and start again.

Depending on the complexity of your Web application, some packaging options are more appropriate than others.

Consider Deploying Web Applications as a Collection of Build Outputs

For many Web applications, you can consider deploying your solution as a simple collection of build outputs, rather than by applying any further packaging. The term “build outputs” refers to the application files that comprise your application, such as ASPX files, executables, DLLs, configuration files, graphics, and other resources.

The advantages of packaging your application files as a collection of build outputs include:

- **Ease of deployment**. The build outputs and other files can simply be copied to the Web server.
- **Ease of update**. Updated files can simply be copied once again to the Web server.

Put simply, to distribute your Web solution as a collection of build outputs and other required files, you need only to choose a mechanism for copying those files to the required destination. Typical ways to achieve this include:

- Copying the files directly out to the target using Windows Explorer copy-and-paste or drag-and-drop operations, File Transfer Protocol (FTP) operations, or the command prompt's XCOPY command.
- Using Microsoft Application Center 2000 (for Web farm deployment).
- Using the Visual Studio .NET **Copy Project** command.

For more information about these distribution mechanisms, see the "Choose How to Distribute Your Solution" section later in this chapter.

Include Installer Classes and Predefined Installation Components with Your Web Application Build Outputs

If you need to deploy application resources, such as message queues, event logs, and performance counters, along with your solution files, you can still deploy your application as a collection of build outputs. You need to include predefined installation components for these types of resources with your Web application, and you also need to use the Installutil.exe utility with them in the production environment. You can use a similar approach if you have used installer classes to create application resources for which there are no predefined installation components (such as serviced components).

For more information about using predefined installation components and installer classes, see the "Use Installation Components to Create and Configure Application Resources" section in Chapter 4, "Deployment Issues for .NET Applications."

Manage the Process of Updating Strong-Named Private Assemblies with Your Build Outputs

Although you can initially deploy strong-named private assemblies as simple build outputs with your application, updating those assemblies is slightly more complex than for assemblies that are not strong named. If you need to upgrade a strong-named private assembly, you cannot simply copy a new version and have your .NET applications use it. When you build a .NET application against a specific version of a strong-named assembly, the application uses that version of the assembly at run time. To use an updated version of a strong-named assembly, you need to redirect your application to use the newer version. You can do this for private assemblies by modifying your application configuration file to redirect your application to the newer assembly. Consequently, you also need to re-deploy that configuration file.

For more information about strong-named assemblies and the global assembly cache, see the "Deploy Assemblies" section in Chapter 4, "Deployment Issues for .NET Applications."

Ensure You Include All Files in Your Copy Operation

If you use a copy operation to deploy the build outputs for your Web solution, you must make sure that all required files are included. Your Web application probably consists of more than just the build outputs—it contains the .aspx files, graphics and other resources, and so on. You can ensure that all required files are deployed by writing a script or deployment utility that interrogates the Visual Studio .NET project file (*.csproj or *.vbproj) to see which other files are required for your project. The project file is actually an XML-based text file, so retrieving the required information from it is a simple matter. It has a <Files> element that lists the other files used by the solution and specifies each file's build action, which you use to determine whether it should be deployed. The following is an example of the <Files> element in a project file:

```
<Files>
 <Include>
  <File RelPath = “AssemblyInfo.cs” SubType = “Code” BuildAction = “Compile” />
  <File RelPath = “Global.asax” SubType = “Component” BuildAction = “Content” />
  <File RelPath = “Global.asax.cs” DependentUpon = “Global.asax” SubType = “Code”
        BuildAction = “Compile” />
  <File RelPath = “Web.config” BuildAction = “Content” />
  <File RelPath = “WebForm1.aspx” SubType = “Form” BuildAction = “Content” />
  <File RelPath = “WebForm1.aspx.cs” DependentUpon = “WebForm1.aspx”
        SubType = “ASPXCodeBehind” BuildAction = “Compile” />
 </Include>
</Files>
```

The BuildAction attribute can be one of the following:

- **None**. The file is not included in the project output group and is not compiled in the build process. An example is a text file that contains documentation, such as a Readme file.
- **Compile**. The file is compiled into the build output. This setting is used for code files.
- **Content**. The file is not compiled, but it is included in the Content output group. For example, this setting is the default value for an .aspx, .htm, or other kind of Web file or graphic.
- **Embedded Resource**. This file is embedded in the main project build output as a DLL or executable file. It is typically used for resource files.

Your deployment script or utility should iterate through the <File> elements contained in the <Include> section of the <Files> node, and should retrieve the RelPath values for each file that has a BuildAction set to **Content**. Any file that has the BuildAction set to **Compile** or **EmbeddedResource** is included in the build outputs themselves. (Of course, you should also remember to include the build output in your copy operation.) Those files set to **None** are not typically required by your solution.

Consider Packaging Your Web Applications in Windows Installer Files

Although you can distribute your Web application as a collection of build outputs, installer classes, and database creation scripts, it is often easier to deploy complex solutions with Windows Installer files. Visual Studio .NET provides Web setup projects that can be used to deploy Web applications. (For more information about Web setup projects, see Chapter 3, "Visual Studio .NET Deployment Projects.") These Web setup projects differ from standard setup projects in that they install Web applications to a virtual root folder on a Web server rather the Program Files folder.

Table 5.1 describes the installation issues associated with different features of complex Web applications and explains the advantages of using Windows Installer files for deploying solutions that include these features.

Table 5.1: Installation Issues for More Complex Web Applications

Feature	Installation Issues
Shared Components	Windows Installer files provide an easy and robust mechanism for installing assemblies into the global assembly cache.
Legacy COM components	If your Web application includes legacy COM components, they need to be installed and registered properly before they can be used. Windows Installer files provide an easy and robust mechanism for installing and registering COM components. Although you can use RegSvr32 to manually register your COM libraries, Windows Installer files provide an easier approach.
IIS Settings	Simply copying build outputs for your Web applications does not copy IIS settings from your development computer to the production Web server. You need to manually change any settings that need modifying or develop a script for ensuring the correct settings are in place. An easier approach is to use a Web setup project to package your solution in Windows Installer files — you can specify IIS settings for your Web setup project to have them applied when you run the .msi file. For more information about managing the IIS settings for Web setup projects, see Chapter 3, "Visual Studio .NET Deployment Projects."
Application Resources	Windows Installer files provide an easy way to deploy application resources, such as message queues, event logs, and performance counters, along with your project outputs. You can use predefined installation components provided by Visual Studio .NET in your project to deploy this type of application resource. After you add the required predefined installation components to your application's project, you can easily add them to the Windows Installer project as custom actions. Similarly, if you have created installer classes to handle the deployment of other application resources for which there are no predefined installation components (such as serviced components), you can also add them to your setup project as custom actions. For more information about using predefined installation components and installer classes, see the "Use Installation Components to Create and Configure Application Resources" section in Chapter 4, "Deployment Issues for .NET Applications."

For more information about deploying IIS settings, shared assemblies, COM components, and application resources, see Chapter 4, "Deployment Issues for .NET Applications." For more information about packaging Web applications in Windows Installer files, see Chapter 3, "Visual Studio .NET Deployment Projects."

Determine How to Package Web Applications and Serviced Components

You will need to give special consideration to packaging your Web applications if they include serviced components. Serviced components enable context sharing between COM+ and .NET Framework classes. A serviced component is a class that is authored in a Common Language Specification (CLS) compliant language and that derives directly or indirectly from the System.EnterpriseServices.ServicedComponent class. Classes configured in this way can be hosted by a COM+ application and can use COM+ services.

Serviced components are hosted by a COM+ application and must be accessible to that application. They have the following registration and configuration requirements:

- The assembly must be strong-named.
- The assembly must be registered in the Windows registry.
- Type library definitions must be registered and installed into a specific COM+ application.
- Services added programmatically must be configured in the COM+ catalog.

You can either have your serviced components registered dynamically, or you can use installer classes to ensure that they are installed correctly. For a comparison of these two approaches, see Chapter 4, "Deployment Issues for .NET Applications." If you use installer classes, you should consider packaging your solution in a Windows Installer file—that way, you can have your installer classes run as custom actions as part of the installation process. If you do not use Windows Installer files to run your installer classes, you need to run them with the Installutil.exe tool.

Consider Host Header Issues for Web Applications

Although for many scenarios you can consider packaging your Web applications in a Web setup Windows Installer file, there is one situation in which this is not an appropriate choice. This scenario is where you need to deploy your Web application to a Web server that distinguishes between sites based on host headers.

Host header names can be used to host multiple domain names from one IP address. To distinguish one Web site from another on the same computer, IIS uses the following three elements:

- TCP/IP address
- TCP port
- Host header name

As long as at least one of these three items is unique for each Web site, IIS can manage multiple sites. When IIS receives a request for a Web page, it looks at the information sent in by the browser. If the browser is HTTP 1.1 compliant (for example, Internet Explorer 3.x and later, or Netscape Navigator 3.x and later), the HTTP header contains the actual domain name requested. IIS uses this to determine which site should answer the request. Although Web setup projects allow you to specify which port to install the application on, they do not currently support deployment of a Web application to a site differentiated by host headers. Therefore, you need to consider alternatives to using Web setup projects to package your Web applications if you need to deploy in this scenario.

One approach that you might consider is using the Visual Studio .NET **Copy Project** command or the command prompt XCOPY operation to copy the build outputs of your Web application to the appropriate location on the target Web server, and then build a separate **standard** Windows Installer file for deploying the components that have more complex setup requirements. The person running the installer needs to ensure that the components are installed into the correct folders.

Consider Packaging Multi-tiered Applications in Multiple Installers

For a multi-tiered application, you should build separate installers for each different physical tier. Building separate installers is appropriate for most scenarios because it is extremely complex (and in some cases impossible) to run an installer on one specific physical tier and have other physical tiers deployed to other computers as part of the same installation process. For example, if you are running the installer on your Web server, it is difficult to deploy your business-logic components to a separate computer. Additionally, different physical tiers may require different packaging strategies. For example, the Web tier might best be deployed as a collection of build outputs, while the business logic tier and database server might best be packaged in one Windows Installer file, with perhaps a Windows Forms application packaged in another installer.

Package Managed Controls in CAB Files

The most appropriate way to package managed controls is to use a Visual Studio .NET CAB setup project. You can then simply copy the CAB file to your Web server and use the uniform resource locator (URL) for the CAB as the code base for the managed control in your Web pages.

Managed controls are assemblies referenced from Web pages that are downloaded to the user’s computer and executed upon demand. They add functionality to Web pages much like Microsoft ActiveX® controls hosted in a Web browser have done in the past.

For information about creating managed controls, see "Writing Secure Managed Controls" on MSDN (*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingsecuremanagedcontrols.asp*).

Creating CAB files provide the following advantages:

- They can be downloaded and the managed controls that they contain can be extracted and executed upon demand.
- They support a number of compression levels, allowing for a quick download time.
- You can use Microsoft Authenticode® technology to sign your CAB files so that users will trust your code.
- They have zero client footprint (with the exception of the download cache).
- The controls that the CAB files contain can easily be updated simply by repackaging the new version into a CAB and replacing the existing copy on the Web server. The CAB projects support versioning, so you can ensure that end users are always using the most recent copy.

You can create CAB files for your managed controls by using CAB setup projects in Visual Studio .NET. Files and project outputs can be added to a CAB project in Solution Explorer, and properties can be set in the Properties window or in the Project Property pages.

Properties of CAB projects allow you to specify a level of compression and implement Authenticode signing, which allows administrators to grant higher trust to these controls without reducing their security with regard to other intranet/Internet code. You can also set the following properties for your CAB file project:

- **Friendly Name**. Specifies the public name for the CAB file.
- **Version**. Specifies the version number of the CAB file. As with the other types of setup projects, the **Version** property should be changed for each released version of your CAB file.
- **Web Dependencies**. Specifies the URL, friendly name, and version of other CAB files that must be installed in order for the selected CAB project to be successfully installed. If this property is set, all dependencies are automatically downloaded and installed when the CAB file is run. You can specify multiple Web dependencies using the **Web Dependencies** dialog box, which is accessible from the Properties window.

Manage Security for Managed Controls

From a code access security perspective, there are two types of managed controls: those that are run under the default security policy and those that require higher trust.

To write managed controls that are intended to run under the default security policy, you only need to know what operations are allowed by the default security policy for the intranet or the Internet zones, depending on where the CAB file is deployed. As long as a managed control does not require more permission to execute than it receives as a result of its zone of origin, then it runs. To execute managed controls that demand higher trust, the administrator or user must adjust the security policy of any computer that runs the code.

Controls that require higher trust should always be strong named or signed with a publisher certificate. This allows policy administrators to grant higher trust to controls signed with a particular certificate without reducing their security with regard to other intranet/Internet code. After the assembly is signed, the user must create a new code group associated with sufficient permissions, and specify that only code signed by the user's company or organization be allowed membership to the code group. After the security policy is modified in this manner, the highly trusted control receives sufficient permission to execute.

Because security policy must be modified to allow high trust downloaded controls to function, deploying this type of control is much easier on a corporate intranet than on the Internet—there is usually an enterprise administrator who can deploy the described policy changes to multiple client computers on the intranet. In order for high-trust controls to be used over the Internet by general users, the user must be comfortable following the publisher's instructions to modify the policy and to allow the high-trust control to execute. Otherwise, the control is not allowed to run.

For more information about security policies, see Chapter 4, "Deployment Issues for .NET Applications."

For more information about managing the security requirements for downloaded CAB files, see "Writing Secure Managed Controls" on MSDN (*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwritingsecuremanagedcontrols.asp*).

Determine How to Package Windows Service Applications

Windows service applications enable you to create long-running executable applications that run in their own Windows sessions. Services are characterized by the following:

- They can be automatically started when the computer boots.
- They can be paused and restarted.
- They do not show any user interface.
- They can run in the security context of a specific user account that is different from the logged-on user or the default computer account.

All of these characteristics make Windows service applications ideal for use on a server when you need long-running functionality that does not interfere with other users who are working on the same computer. However, as far as installation is concerned, Windows service applications have the following requirements:

- They must be installed and registered as a service.
- An entry must be created for the service with the Windows Services Control Manager.

To meet these requirements, you should add predefined installation components to your Windows service applications. For more information about the predefined installation components you should use for Windows services, see the "Deploy Windows Services with the ServiceInstaller and ServiceProcessInstaller Components" section in Chapter 4, "Deployment Issues for .NET Applications." As discussed for message queues, event logs, and performance counters, if you create a Windows Installer file for your Windows service project, the installer classes you create can be added to the Windows Installer project as custom actions. However, if you do not create a Windows Installer file, you need to use the Installutil.exe utility on the production servers to run these installer classes. There is no functional disadvantage to using Installutil.exe, but it is often easier to have the Windows Installer file perform the necessary actions as part of the deployment process.

For more information about creating setup projects for your Windows service applications, see "Adding Installers to Your Service Application" on MSDN (*http://msdn.microsoft.com/library/en-us/vbcon/html/vbtskAddingInstallersToYourServiceApplication.asp*).

Package Components in Merge Modules for Other Developers

As a developer, you are familiar with the idea that the code you write can have one of two different audiences, end users or other developers.

End users use your applications "as is," while other developers incorporate your controls and DLLs into their own applications. Given the distinction between code that is used by end users and that is used by other developers, a similar comparison can be made for the actual packaging of your code—you can package it as a complete setup routine that can be used "as is" for end users, or you can package it so that other developers can include it in their own install routines.

In addition to allowing you to create Windows Installer files for complete applications, Visual Studio setup projects allow you to create merge modules (.msm files) for components that will be used by other developers. Whereas Windows Installer files are used to install applications and files on another computer, merge modules are used by developers who need to include the setup routines required by your components with their application's installer. When developers create Windows Installer files for their applications, they include these merge modules in their Windows Installer, alongside their project outputs, executable files, and other files, for distribution to the production environment.

Merge modules could be described as reusable setup components. Much like DLLs allow you to share application code and resources between applications, merge modules allow you to share setup code between .msi files. The following list contains recommendations for using merge modules:

- As a general rule, any component that is shared by more than one application (or that has the potential for being shared) should be put into its own merge module. While it is possible to put multiple components into a single merge module, it is best to create a separate merge module for each component to avoid distributing unnecessary files.
- To avoid versioning problems, you should always use merge modules for any component or file that will be shared by multiple applications.
- Installers can include multiple applications, allowing you to install a suite of applications in a single step. In this case, the installer should include merge modules for all components used by any of the included applications. If a merge module is used by more than one application it needs to be added to the .msi file only once.
- Merge modules allow you to capture all of the dependencies for a particular component, ensuring that the correct versions are installed. After you distribute a merge module, it should never be modified. Instead, you should create a new merge module for each successive version of your component.
- Merge modules cannot be installed directly, but are merged into an installer for each application that uses the component. This ensures that the component is installed consistently for all applications, eliminating problems such as version conflicts, missing registry entries, and improperly installed files.
- Each merge module contains unique version information that is used by the Windows Installer database to determine which applications use the component, preventing premature removal of a component. For this reason, a new merge module should be created for every incremental version of your component. A merge module should never be updated after it is included in an installer.

The main advantage of using merge modules is that you need to define the setup logic for the assemblies in the .msm file only once, rather than for each instance of the application. The .msm file needs only to be added to the Windows Installer file once, even though multiple applications in the installer file depend on their components. If you need to upgrade the assemblies in the .msm file, you need to only create a new .msm file and include that in your Windows Installer file for redeployment.

For more information about updating and redeploying components packaged in .msm files, see Chapter 6, "Upgrading .NET Applications."

For more information about merge modules, see "Introduction to Merge Modules" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconWhatYouNeedToKnowAboutMergeModules.asp*).

For step-by-step instructions on how to create and use merge modules, see "Creating or Adding a Merge Module Project" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbtskcreatingoraddingmergemodule.asp*).

Manage Additional Packaging Issues

In addition to choosing packaging mechanisms based on application type and whether you are packaging components or complete applications, there are a number of other issues that apply to all of the previously described scenarios.

Use Authenticode Signing to Guarantee Package Integrity

The software industry must provide users with the means to trust code published on the Internet. The Internet itself cannot provide any guarantee about the identity of the software creator. Nor can it guarantee that any software downloaded was not altered after its creation. Browsers can exhibit a warning message explaining the possible dangers of downloading data of any kind, but browsers cannot verify that code is what it claims to be. A more active approach must be taken to make the Internet a reliable medium for distributing software.

One approach to providing guarantees of the authenticity and integrity of files is attaching digital signatures to those files. A digital signature attached to a file positively identifies the distributor of that file and ensures that the contents of the file were not changed after the signature was created.

Visual Studio .NET deployment tools make it possible for you to sign an installer, a merge module, or a CAB file using Microsoft Authenticode technology. You must first obtain a digital certificate in order to use these features of Visual Studio .NET deployment tools. The Signcode.exe utility available in the Microsoft Platform SDK can be used to obtain a digital certificate.

More information about digital certificates is available in the Platform SDK CryptoAPI Tools Reference (*http://msdn.microsoft.com/library/en-us/security/security/cryptoapi_reference.asp*). For more information about using Authenticode signing for your Visual Studio .NET setup projects, see "Deployment and Authenticode Signing" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vxconDeploymentAuthenticodeSigning.asp*).

Version Your Setups for Multiple Application Releases

Another requirement for many applications is that they can be upgraded in a manageable way. If you need to manage upgrades and patches in a manner consistent with the Windows Installer, you should consider packaging your files in .msi (or .msm) files. Otherwise, you need to manage the versioning of your solutions manually, and you are not able to take advantage of the robust versioning control for applications provided by the Windows Installer.

Include Windows Installer Bootstrapping with Your .msi Files

If you decide that .msi files are an appropriate choice for packaging your solutions, you need to determine if the target computers in the production environment are actually capable of running your Windows Installer files. For your .msi file to install properly, you need to ensure that:

- Target computers have the Windows Installer service present.
- Target computers have the correct version of the Windows Installer service.

If you are deploying to your own corporate network, you may have prior knowledge that the correct version of the Windows Installer service is present. However, in many other cases you might not be certain that the correct version is installed. Therefore, you need to devise a strategy for installing or upgrading the Windows Installer service prior to running the installer for your application.

Every installation that attempts to use the Microsoft Windows Installer begins by checking whether the installer is present on the user's computer, and if it is not present, whether the user and computer are ready to install Windows Installer. Two setup applications, InstMsiA.exe and InstMsiW.exe, are available with the Windows Installer SDK that contains all of the logic and functionality to install Windows Installer.

Note: InstMsiA.exe is the ANSI version of the installer redistributable package, whereas InstMsiW.exe is the Unicode version. The version used by the bootstrapping application depends on the platform to which the Windows Installer service is being installed. InstMsiA.exe is used for Windows 95, Windows 98, and Windows Me, and InstMsiW.exe is used for Windows NT 4.0 and later.

You can include these files with your Windows Installer file to ensure that the correct version of the Windows Installer service is present before your .msi file runs.

However, a bootstrapping application must manage this process. If Windows Installer is not currently installed, the bootstrapping application must query the operating system to determine which version of the InstMsi is required. After the installation of Windows Installer has initiated, the bootstrapping application must handle return codes from the InstMsi application and handle any reboots that are incurred during the Windows Installer installation.

Visual Studio .NET setup projects make this bootstrapping process very simple. All you need to do is set some project properties for your .msi file, and the build process creates the appropriate bootstrapping application. On the property pages for your deployment project, you can specify one of the following bootstrapping application options:

- **None**. No bootstrapping application will be included.
- **Windows Installer Bootstrapper**. A bootstrapping application will be included for installation on a Windows-based computer.

- **Web Bootstrapper**. A bootstrapping application will be included for download over the Web. A bootstrapping application has authentication built in, so using a bootstrapping application rather than .cab files is the preferred method for downloading.

When you build your setup project, the bootstrapping application (Setup.exe and associated .ini file) and the InstMsi file are included in the build output—users need to run Setup.exe rather than the .msi file to install your application.

For more information about the process for installing bootstrapping applications, see "Bootstrapping" in the Platform SDK (*http://msdn.microsoft.com/library/en-us/msi/boot_4puv.asp*).

Include .NET Framework Bootstrapping with Your .msi Files

As discussed in Chapter 2, "Deploying the .NET Framework with your .NET Applications," your .NET applications need the .NET Framework to be installed on your target computers. If you are packaging your application so that it can be installed onto unknown target computers, you need to ensure that the .NET Framework is present before your own setup routine starts. To do this, you can:

- Package the redistributable .NET Framework (Dotnetfx.exe) with your application.
- Run a check that determines whether the .NET Framework is installed on the target computer.
- Install the .NET Framework if necessary.
- Install your application.

As mentioned earlier in this chapter, you cannot include the .NET Framework in your Windows Installer files. Instead, you must develop a separate bootstrapping application that performs the tasks listed above. A sample application (Setup.exe) that performs these tasks can be downloaded from "Microsoft .NET Framework Setup.exe Bootstrapper Sample" on MSDN (*http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/001/830/msdncompositedoc.xml*).

The Setup.exe bootstrapping application sample demonstrates how to create a setup program that verifies whether the .NET Framework is installed. The code checks for a specified version number of the .NET Framework in the HKLM\SOFTWARE\Microsoft\.NETFramework\policy\v1 registry key. You can determine which version number should be checked for by examining the properties of the redistributable .NET Framework, Dotnetfx.exe. (For more information about Dotnetfx.exe, see Chapter 2, "Deploying the .NET Framework with your .NET Applications.") The code performs a comparison between the build number in the registry key and the build number of the .NET Framework being hosted by the application. If there is not a matching build number in the registry key in this location, Setup.exe performs a silent install of the .NET Framework included with your application, and then runs your Windows Installer file.

You can use the sample for your own applications by downloading the precompiled application and simply modifying the Setup.ini file that ships with the sample. Alternatively, you can download the source code for the sample and modify it to meet your specific needs.

For more information about how to work with this sample, see "Redistributing the .NET Framework" on MSDN (*http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp*).

Consider Automating the Creation of Installers

If you are an independent software vendor (ISV), or if you need to deploy a large number of .NET applications for your enterprise, you might want to automate the creation of your setup routines. For example, you might want to build your own application for creating setup programs to standardize deployment practices, or you might want to write scripts that create installers for multiple applications. If the other factors described in this chapter bring you to the conclusion that an .msi (or .msm) file is appropriate for your solutions, you can automate the creation of your setups by using the automation interface provided by the Windows Installer, rather than creating your installers using Visual Studio .NET.

For in-depth information about the Windows Installer automation interface, see the "Automation Interface" section of the Platform SDK on MSDN (*http://msdn.microsoft.com/library/en-us/msi/auto_8uqt.asp*).

Consider Using Third-Party Installation Tools

In addition to the deployment tools included in Visual Studio, installation tools that support Windows Installer are available from third-party vendors. These tools may support additional Windows Installer authoring features that are not available in Visual Studio deployment projects.

The following list describes some third-party tools:

- InstallShield Developer is a Windows Installer setup-authoring solution that provides complete control of the Windows Installer service and full support for .NET application installations. The Developer edition provides the option to create .NET installations directly from within the Microsoft Visual Studio .NET integrated development environment (IDE) or from the traditional InstallShield Developer IDE. It features Visual Studio .NET project wizards, dynamic links with Microsoft Visual C#™ .NET development tool and Microsoft Visual Basic .NET development system projects, .NET COM interop support, .NET assembly installation configuration, and .NET Framework distribution in addition to a Windows Installer direct table editor. For more information, see the InstallShield Web site (*http://www.installshield.com/isd/*).

- Wise for Visual Studio .NET operates directly within Microsoft Visual Studio .NET. It merges the installation and application development lifecycles so that applications and installations are designed, coded, and tested together. For more information, see the Wise Web site (*http://www.wise.com/visualstudio.asp*).

Choose How to Distribute Your Solution

As introduced at the beginning of this chapter, you not only need to determine how to package your application for deployment, but you also need to decide how you will distribute your solution to the production environment. There are a number of different options available to you for distributing your .NET application, and there are a number of different factors that affect which mechanism you choose. This section discusses the different distribution mechanisms.

Choose a Distribution Mechanism that is Compatible with Your Packaging Strategy

How you distribute your .NET application to the production environment depends, to some extent, on how you package the solution.

Determine How to Distribute Your Windows Installer Files

You can distribute your Windows Installer files in any of the following ways:

- Make your .msi file available to end users on network shares, Web servers, via e-mail, or on portable media (such as CD, DVD, or floppy disk)—your users can then run the .msi file to install your solution.
- Use administrative software deployment mechanisms, such as SMS or Active Directory group policies for software deployment—administrators can then have your application installed and configured with little or no user interaction.

The mechanisms you choose depend on a number of different factors, which are discussed later in this chapter.

Determine How to Distribute Your Build Outputs

Build outputs are best deployed to the production environment with simple copy operations. You can use any of the following mechanisms for copying your build outputs:

- XCOPY deployment from the command prompt
- Windows Explorer copy and paste operations
- File Transfer Protocol (FTP)
- The **Copy Project** command in Visual Studio .NET for Web applications

The end result of using any of these copy mechanisms is that your build outputs and other solution files are simply copied to the production environment, so functionally, they are all very similar. However, the deployment of your application should be consistent. To avoid human error, you should either script the copy deployment (using XCOPY or FTP), use the **Copy Project** command in Visual Studio .NET, or for the most robust solution use Application Center. Copy operations for build outputs are discussed in more detail later in this chapter.

Deploy Your Web Farms with Application Center

You should use Application Center to deploy Web applications and Web content to multiple servers in your Web farm—the ability to manage and synchronize content and Web applications across multiple Web servers is one of the major features of Application Center. Application Center is best suited for deploying build outputs and content files for Web solutions and COM+ applications—it is not suitable for deploying applications that are packaged in Windows Installer files or other executable setup programs.

Details of Web farm deployment with Application Center later in this chapter.

Distribute Windows Installer Files for User-Installed Applications

There are a number of appropriate mechanisms for distributing Windows Installer files to your users, each with their own advantages. You can use the Table 5.2 to determine a suitable approach for your own specific scenario.

Table 5.2: Distributing User Installed Windows Installer Files

Mechanism	Details
Make your installer available on a network share	Users can connect to the network share to download your installer when required. They can then run your installer locally. Alternatively, users can also run the installation over the network, but this might affect network resource usage, and typically results in a slower install. If the user runs the installation over the network, they can uninstall your application without the need to reconnect to the server — their local Windows Installer database takes care of uninstall actions. However, they may need to have a connection to the server for adding program features.

Mechanism	Details
Make your installer available on a Web server	Downloading and running installers from a Web server is similar to doing so from a network share. You can make your installer available on a Web server for downloading on your intranet, and you can also allow your customers to download your software over the Internet. The only real difference is that security considerations must be taken into account — proxy servers may prevent the download or running of your installation and security policies may prohibit installation of your application if your installer is not digitally signed. For more details about signing your installers, see the "Use Authenticode Signing to Guarantee Package Integrity" section earlier in this chapter. A typical approach for allowing customers and the general public to download your software over the Internet is to further package your installation files into a Zip or self-extracting executable file to speed up download time. This approach can also ensure that users actually download the installation files rather than running the installation over the Internet.
Send your installer over e-mail to your users	In certain situations you might decide to distribute your installation files to users via e-mail as an attachment. Conceptually, this is similar to having the users download from a network share or Web server — the only distinction is that your e-mail server stores the installation files rather than a network or Web server. This mechanism is usually only suitable for small installers or when only a small number of users require your application; otherwise, you place considerable stress on your e-mail system. The one advantage of this approach is that users do not have to search for your installation files on a network or Web server. Of course, you might consider sending a link via e-mail to the installation files on a Web or network server much to the same effect, without placing undue stress on your e-mail system.
Distribute your installer on CD or DVD	If you package your solution for the general public or for a number of customers, you might decide to distribute your installation files on CD or DVD. You can simply create the CD with your installation files and the user can simply run your .msi file or bootstrapping application from the CD or copy the contents to their local computer or network share.

Use Software Distribution Tools to Deploy Administrator-Installed Applications

If your corporate strategy specifies that your application **must** be installed for specific users or computers, you may not want to rely on those users manually installing your solution. Instead, you can use Active Directory group policies for software distribution, or Systems Management Server, to ensure your applications are installed. Both can use "elevated privileges" during the install routine.

Using these electronic software distribution tools, you can:

- Implement and enforce standard corporate desktops.
- Target specific users who must have your software installed.
- Target specific computers to which your application must be deployed.

Use Microsoft Systems Management Server (SMS) for Robust Deployments

Systems Management Server (SMS) is Microsoft's preferred platform for software distribution. Corporations whose businesses change rapidly often receive the greatest value by using SMS for enterprise software management. SMS provides all the key elements for software management:

- **Inventory management**. Provides a repository of the existing software deployed throughout a corporation and a hardware inventory that is critical for planning and deploying enterprise software.
- **Software distribution**. Allows enterprises of all sizes to ensure that the business logic their users need is available with the proper service level. This includes advanced features such as:
  - Software distribution to all Windows-based platforms
  - WAN-aware automated distribution of software—the ability to define software distribution points that are managed in a WAN-aware way.
  - Targeting to computers based on user names, group names, computer names, domain names, network addresses, or inventory collection values.
  - Scheduled application deployment.
  - Status of application deployments.
  - Reporting of successes/failures.
- **Application usage tracking (metering)**. Allows administrators to make smart purchasing and deployment decisions because they understand what applications their customers are using.
- **Status and reporting**. Allows administrators to monitor the activity across each of the above components to deliver the correct software to the correct locations when needed.

For more instructions on using SMS to deploy applications, see the product documentation (*http://www.microsoft.com/smserver/techinfo/productdoc/default.asp*).

Use Active Directory in Certain Scenarios

You can use Active Directory to distribute your solution by assigning or publishing your application using group policies. Because Active Directory and Windows 2000 are required and because the software management components of Active Directory are not as sophisticated as those offered by Systems Management Server, you typically use group policy for software distribution under the following conditions:

- Your solution is packaged as Windows Installer files.
- You have deployed Active Directory.
- Your target computers are running Windows 2000 or later.
- Your company is a small- to medium-sized business.
- You are managing a single geographic location.
- You have consistent hardware and software configurations on desktops and servers.
- You do not require centralized reporting.
- Your package is small and will not be affected by slow WAN links.
- You only need 32-bit application support.
- You do not require background installation (group policy distribution requires a reboot).

Table 5.3 describes how assigning and publishing applications with Active Directory makes your solutions available to end users.

Table 5.3: Assigning and Publishing Applications with Active Directory

Distribution Strategy	Description
Assigning Software	You can assign a program distribution to users or computers. If you assign the program to a user, it is installed when the user logs on to the computer. When the user first runs the program, the installation is finalized. If you assign the program to a computer, it is installed when the computer starts, and is available to all users that log on to the computer. When a user first runs the program, the installation is finalized.
Publishing Software	You can publish a program distribution to users. When the user logs on to the computer, the published program is displayed in the **Add/ Remove Programs** dialog box, and it can be installed from there.

For more instructions on using Active Directory group policies to deploy applications, see the Active Directory documentation (*http://www.microsoft.com/windows2000/ technologies/directory/ad/default.asp*).

Comparing SMS and Active Directory Distribution Strategies

Table 5.4 describes the capabilities of Active Directory and SMS for key software distribution and management issues.

Table 5.4: Comparing SMS and Active Directory Group Policies for Software Distribution

Issue	Active Directory Group Policies	SMS
Reporting	Windows Installer events and messages are stored on the local computer, rather than in a central location. If an administrator requires reporting on which users or computers were updated, they need to connect to each computer and view the event logs.	SMS provides for centralized reporting and management.
Distribution	Active Directory requires an administrator to manually create and manage software distribution points. If multiple installation points are required, administrators need to synchronize them. Windows 2000 Distributed File System (DFS) can be used to streamline this process. Each distribution point requires a separate group policy.	SMS uses a distribution point hierarchy which is automatically synchronized and managed. SMS also compresses the packages prior to delivery to remote install points
Targeting	Targeting with group policy is based on organizational unit membership and policy application.	SMS targets collections based on inventory values. Collections are dynamic and query-based.

Note: Active Directory group policies are capable only of distributing applications that are packaged in .msi or .zap file formats. You cannot use it to deploy your application build outputs. Similarly, while SMS can distribute other file types, including executable files, it is not particularly suited to distributing build outputs. You should consider a different strategy for distributing your solution if it is packaged as a collection of build outputs.

Choose a Strategy for Distributing Your Build Outputs Directly

If you decide to distribute and update your solution as a simple collection of build outputs, then you are not able to use some electronic software distribution tools such as Systems Management Server or Active Directory group policies. Instead, you should consider using simple copy operations. For example, you can use Windows Explorer copy-and-paste or drag-and-drop operations, File Transfer Protocol (FTP) operations, the command prompt's XCOPY command or the Visual Studio .NET **Copy Project** command (for Web projects).

Use Visual Studio .NET Copy Project Command for Quick and Easy Deployment of Web Applications

The Visual Studio .NET **Copy Project** command creates a new Web application on the target server. You can choose to include:

- Only the files required to run to the application, including build outputs, DLLs and references form the bin folder, and any files with the **BuildAction** property set to **Content**.
- All project files, including build outputs, DLLs, and references form the bin folder, and all other files (including the project file and source files). Because source files are included with this option, it is not recommended for deploying to the production environment.
- All files that are in the project folder and subfolders. Again, because source files are included with this option, it is not recommended for deploying to the production environment.

For a production environment, you will typically choose the first option.

You can also choose a Web access method for copying your project. You can specify either Microsoft FrontPage® Web site creation and management tool or file share.

Note that FrontPage Server Extensions must be installed on the target server to use the **Copy Project** command with the FrontPage Web access method. Also, be aware that some companies do not allow FrontPage Server Extensions on their Web servers because of potential security concerns, so you may not be able to use this Web access method in all cases.

For more information about using the **Copy Project** command, see "Deployment Alternatives" on MSDN (*http://msdn.microsoft.com/library/en-us/vsintro7/html/vbconDeploymentAlternatives.asp*).

Copy Your Web Application

Due to the self-describing nature of .NET assemblies and the ability of ASP.NET to automatically detect changes in your solution files and use them automatically, deployment of simple .NET applications using copy mechanisms is now feasible. This is because, in many cases, there is no longer a need to modify the registry or perform other tasks, such as stopping Internet Information Services (IIS) when changes have occurred.

Some applications that lend themselves very well to copy-type deployment are XML Web services and ASP.NET Web applications. Previously, these types of applications were built using IIS and traditional COM components. Installing the components for these applications usually entailed registering the component during the deployment process by using the Regsvr32.exe utility. Updating existing components was far more troublesome, because IIS placed an exclusive file lock on the components. The only way to release the lock was to shut down IIS. Replacement of existing components typically included stopping the IIS service, using the Regsvr32.exe utility to remove the registration of the old component, copying and then registering the new component (again using Regsvr32.exe), and then starting the IIS service again. These steps are no longer needed for XML Web services and ASP.NET Web applications. IIS does not place exclusive file locks on .NET assemblies and can detect changes to your solution files and automatically use the new versions "on the fly." Additionally, because assemblies are self-describing, they don't have to be registered. This makes ASP.NET Web-based applications ideal for simple copy deployment and management.

However, using a simple copy mechanism to deploy your applications suffers from some drawbacks, including:

- An additional step is required to install the shared assemblies into the global assembly cache. You can install an assembly into the global assembly cache by using the Gacutil.exe utility or by using a drag-and-drop operation to move the assembly into the Global Assembly Cache folder in Windows Explorer. However, using drag-and-drop operations to move assemblies into the global assembly cache does not implement any reference counting; therefore, it should be avoided. Without reference counting, the uninstall routine of another application can result in the premature removal of a shared assembly from the global assembly cache, which would break your application that relies on that assembly. If you use the Gacutil.exe tool, you should use the **/ir** switch, which installs assemblies into the global assembly cache with a traced reference. These references can be removed when the assembly is uninstalled by using the **/ur** switch.
- You cannot automatically configure IIS directory settings for your Web application.
- You cannot automatically manage the rollback or uninstall of your application.
- You cannot include launch conditions, such as those available in Windows Installer files.
- You cannot automatically manage conditional deployment scenarios.
- You cannot automatically manipulate registry settings and file associations—you will manually have to edit the registry on your target computer or import .reg files to ensure appropriate settings are in place.
- You cannot take advantage of Windows Installer features, such as automatic repair (although this is usually not a major issue for Web applications).

If your deployment scenarios require any of the preceding features, copy-oriented operations may be an inappropriate choice. You should consider creating Windows Installer files and using one of the other deployment mechanisms previously mentioned.

Use Application Center to Deploy and Manage Server Environments

You can use Application Center to manage, deploy, and update content and components in your server farms. In addition, you can use Application Center to manage the deployment of applications between different environments, such as from development servers to test servers, from test servers to staging servers, and then to the production environment. This staged deployment is discussed later in this section.

Large Web application farms of Web servers and application servers typically cluster together multiple servers to provide scalability and increased availability in order to satisfy client requests. With these advantages come some challenges, including deploying and keeping content and functionality updated across all members of the cluster. With the .NET architecture, deploying and updating content (such as Web pages) and functionality (such as business components) is easier than before. However, the deployment and updating processes still need some managing.

Application Center makes it easy to implement clusters to provide scalability and availability for client requests, and also provides management, deployment, and synchronization of the content and components for your server environments.

An Application Center cluster is a set of servers that serve the same content to clients (for example, Web pages and COM+ components). Application Center clusters provide:

- **Increased application availability**. With multiple members serving sites and applications, clients can experience uninterrupted service even through failures or problems on individual members.
- **Increased scalability**. You can add and remove members of your cluster to satisfy availability and performance requirements for your clients
- **Increased performance**. Client workload is distributed throughout the cluster, so that each individual member receives reduced load, which enhances performance.

In terms of managing your Web farm clusters, Application Center provides the following capabilities:

- **Cluster synchronization**. You can synchronize all members by setting an interval for periodic synchronization or manually initiating synchronization. This includes synchronizing server configuration, application content, and configuration and network settings for network load balancing (NLB).
- **Component load balancing (CLB) capabilities**. You can load balance components for COM+ applications and process COM+ applications on designated application servers.

- **Simplified application deployment**. You can use the Deployment Wizard to deploy applications from a stager cluster to your production cluster (the next section provides more details on this).

In terms of deployment, you can use Application Center to synchronize and deploy any of the following:

- Web sites, virtual directories (and their associated ISAPI filters), and ASP.NET applications (including XML Web Services)
- Configuration files (such as Web.config, application configuration files, and machine configuration files)
- Private assemblies
- Discovery files
- Symbols
- HTTP modules and HTTP handlers
- Localized content
- Secure Sockets Layer (SSL) Certificates
- Server Configuration information (such as IIS metabase settings, CLB configuration, and NLB configuration)

Deployment of many of these elements requires administrative privileges, which Application Center manages. Application Center deploys and synchronizes the preceding content and configuration types across clusters.

Deploy and Synchronize in Your Server Environments

Application Center supports two concepts that are useful for setting up your clustered server environments and keeping each member up to date: deployment and synchronization.

Deployment is the transfer of content and configuration settings from one cluster (usually a stager) to another cluster (usually the production cluster). Synchronization, on the other hand, is the transfer of content and configuration settings from the cluster controller to one or more cluster members. Other than this distinction, the processes and resources that are used are identical, and the difference is merely a semantic one. You can synchronize and deploy content by using the Application Center snap-in, the New Deployment Wizard, or **AC DEPLOY** from the command line.

Use the Intelligent Deployment Features of Application Center

Application Center automatically configures your server applications (and their constituent parts) when you deploy or synchronize to cluster members. This removes the need for you to specify detailed actions for your deployment. For example, if you script the deployment of a virtual directory, you need to include the basic copy operations, and then determine how to ensure the appropriate IIS settings are in place. On the other hand, if you use Application Center, it detects that you are deploying a virtual directory and configures it appropriately on the target computer(s)—you do not need to take any further steps to ensure that it is configured correctly. As another example, if you deploy an updated COM component to a Web application with script, you need to include actions that stop the IIS Web service, remove the registration of the old component version, replace the component files, register the new component, and then restart the IIS Web service. Again, if you deploy the component using Application Center, it deals with the stopping and starting of services for you, as well as removing the registration of the old component and registering the new one.

Use Application Center to Deploy between Development, Testing, Staging, and Production Environments

One of the major advantages of using Application Center to deploy Web solutions and server applications in your clustered server environment is that you can use it to manage the different stages of your product lifecycle. Put simply, you can deploy your server-based solutions from your development environment to your test servers, from your test servers to your staging environment, and finally from your staging server(s) to your production cluster with the same mechanism. Using Application Center to deploy your solutions between these different environments provides you with consistency advantages. For example, having successfully deployed your application from development to the test environment, and then from the test servers to your staging servers, you can be certain that, as you deploy your application to the final production environment, no unexpected changes or problems are introduced that arise from the deployment mechanism itself. This makes it easier to track down any issues in the live environment, should they arise.

Stage Content and Components for Deployment

A stager is a server on which you can experiment with and fully test the quality and functionality of content and components from development and test environments before deploying to production environments. It can be useful to operate Application Center on a single server, without the context of a multi-member cluster, to provide a staging server. Application Center treats a single-node cluster, or stand-alone server, as a cluster of one member. You can use this single-node cluster as a stager. You can also use the stager in your strategies for updating your server environments. Updating server environments is discussed in Chapter 6, “Updating .NET Applications.”

Use Application Center to Manage Your Server Environments

In addition to deploying content, components, and configuration settings to servers, Application Center contains robust support for managing those servers through the Health Monitor tool. It provides facilities for managing a cluster as if it was a single server. It does this through providing a single view of the event logs of all servers in a cluster. In addition, Application Center also allows you to aggregate performance monitors to view the health of the entire cluster, in addition to that of a single server.

You can customize and extend common operations through the use of custom actions associated with Health Monitor monitors.

For more information, see the white paper, "Running Custom Actions with Health Monitor" (*http://www.microsoft.com/applicationcenter/techinfo/administration/2000/ACScpLch.doc*).

Use Application Center for Server Farms

Application Center must be installed on all servers that participate in Web farms or application server farms, so it is very suitable for deploying server applications and Web solutions to this sort of environment.

Application Center is designed for managing and deploying content and components to a server environment—it is not suitable for deploying desktop applications, Windows service applications, or other client-based applications. If you are deploying these types of applications, you must consider using one of the other distribution mechanisms already discussed. Also, Application Center is not designed to deploy Microsoft SQL Server™ databases and manage their schemas, or message queues. To deploy these technologies, you must rely on another mechanism.

At the time of this writing, Application Center cannot install assemblies into the .NET global assembly cache for members of your Web farm or application farm clusters. However, this capability is scheduled for inclusion in Service Pack 2 for Application Center. Expected release of the global assembly cache deployment ability is Summer 2002.

Use a Mix of Distribution Mechanisms

As you will have guessed by now, you will often want to use a mixture of deployment mechanisms, especially if you are deploying a multi-tiered application. For example, you might want to use:

- SMS or Active Directory to deploy and manage Windows Forms applications to client computers and objects to business servers.
- Copy commands (such as XCOPY, FTP, and Copy Project) to deploy your simple Web applications.
- Application Center to deploy and manage your Web farms.

6

Upgrading .NET Applications

Occasionally, you need to upgrade Microsoft® .NET applications that have already been deployed to the production environment. Typically, applications are upgraded in order to:

- Apply bug fixes to remedy problems with your application.
- Provide enhancements for existing features.
- Provide new features or capabilities.

One of the major advantages of the .NET architecture is that it provides comprehensive support for modifying or upgrading applications after they have been deployed. This chapter provides guidance on developing and implementing upgrade strategies for your .NET applications.

Manage the Trade-off for Your Upgrade Strategy

Whatever your upgrade strategy, and whatever tools you use to implement that strategy, there is always a certain trade-off between maintaining compatibility with the original application and implementing new features (or fixing existing ones).

The trade-off stems from the fact that to guarantee 100% compatibility with your existing solution and its component parts, you cannot change anything at all. This means that to guarantee 100% compatibility, you cannot even apply bug fixes, let alone add new features. On the flip side, if you add new features and fix problems by completely re-architecting your application, then your new solution bears little resemblance to the original one. In effect, you design a completely new solution that is not at all compatible with the original one. Of course, your goal, when upgrading your solutions, is to seamlessly provide new versions that resemble the previous ones, but to include new features and/or bug-fixes. In short, the way you manage the trade-off is determined by how willing you are to break compatibility in order to provide new features or fix existing ones. There is no right or wrong approach to this issue—the way you manage the trade-off is determined by your specific needs. However, you should be aware that the more changes you implement in your upgrades, the less you can guarantee compatibility with the previous versions.

Plan Your Upgrade Strategy

It is likely, yet by no means required, that you will upgrade your solutions using similar mechanisms to those with which you originally deployed your application. For example, if you originally packaged your application in a Microsoft Windows Installer file, then you will typically upgrade application files with an updated Windows Installer file or a Microsoft patch (MSP) file, managed by Windows Installer. If you distribute your solution with Microsoft Systems Management Server (SMS), you typically re-deploy your solution using the same approach. However, there will undoubtedly be some situations where a packaging or distribution mechanism that is different to the original deployment strategy might be more suitable. For example, it is possible that a change to a configuration file or Web file might best be implemented simply by copying the new version, even though the application might originally have been packaged in a Windows Installer file and distributed with Microsoft Active Directory® directory service group policies for software deployment. Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Application" provides information that helps you determine the advantages and disadvantages of each packaging and distribution mechanism if you decide that upgrading your application requires a different approach to that used for the original deployment.

Manage Application Versions

Versioning your applications requires careful management. Upgrading solutions introduces some management questions. Table 6.1 highlights these questions and provides some typical solutions.

Table 6.1: Upgrade Management Questions

Question	Typical Solutions
Does the new version contain features that require it to be differentiated from previous versions?	Create an upgrade for your application and increment the application version number. This allows users, administrators, and the operating system to distinguish between different releases of your solution. If you originally packaged and installed your application with a Windows Installer file, you can use Microsoft Visual Studio® .NET development system or the Windows Installer Platform Software Development Kit (SDK) to create upgrades. Creating upgrades with these tools is discussed later in this chapter. If you used some other mechanism, you need to develop your own custom solution.

Question	Typical Solutions
Is there a distinct order in which multiple upgrades to the same application must be applied?	If you continually modify and improve your application, you will want your users to have access to different upgrades for your solution as you develop them. This typically results in a number of different upgrades being available for distribution. There may be a distinct order in which upgrades should be applied to the production environment. For example, your original product may be version 1.0 and you may have two different upgraded versions identified as 1.1, and 1.2. You need to determine whether version 1.0 can be directly upgraded to 1.2, or whether the 1.1 upgrade needs to be applied first. The ordering of updates is purely a design issue. The ordering of upgrades is discussed later in this chapter.
Should a specific upgrade be prevented from installing over newer versions?	Typically, where multiple versions of an application exist, you want to prevent earlier versions from installing over later versions. If you originally packaged and installed your application with a Windows Installer file, you can version your setups and set properties that prevent earlier versions of an application from installing over later versions. Otherwise, you need to build your own upgrade logic that checks for later versions. The Windows Installer file properties for controlling this behavior are discussed later in the chapter.
Should upgrades remove previous versions?	In many circumstances, you want upgrades to completely replace earlier versions of your application. In other cases, it may be appropriate to allow different versions to coexist on the same computer. If you originally packaged and installed your application with a Windows Installer file, you can use Visual Studio .NET or the Windows Installer Platform SDK to control whether or not earlier versions are removed as you install your upgrades. Otherwise, you need to implement custom checks and setup logic for deciding whether previous versions should be removed. The Windows Installer file properties for controlling this behavior are discussed later in the chapter.

Should I Patch or Reinstall My Application?

If you are planning to upgrade your application using Windows Installer technology, you have two choices for implementing the upgrade:

- You can build a new version of a Windows Installer file and deploy your application in a similar manner to the original version.
- You can build a patch package and apply it to the currently installed application.

Visual Studio .NET does not currently support patching an installed application using Windows Installer technology. You need to use the Windows Installer Platform SDK tools (such as Orca.exe and MsiMSP.exe), or third-party utilities, to create patch files that can be applied to an application that was originally installed with a Visual Studio .NET setup project.

For more information about creating and applying patch packages, see the "Patching and Upgrades" topic in Windows Installer Platform SDK Help.

Should I Upgrade an Existing Windows Installer File or Create a New One?

When you plan for upgrading your application with a Windows Installer file, you need to decide on one of the following methods:

- Create a completely new setup project and add your project outputs and other files, just as you did for the first release of the application.
- Upgrade the existing setup project by ensuring changes to your application are reflected in the setup, and then rebuild that version.

Although both approaches can be used to install your upgraded application, you will usually upgrade your existing setup project and rebuild the Windows Installer file, for the following reasons:

- Windows Installer uses various versioning and upgrade properties and globally unique identifiers (GUIDs) to associate Windows Installer files with applications that are already installed. (See Table 6.2 for details of these version properties and GUIDs). If you create a new setup project, then Visual Studio .NET assigns new GUIDs, which prevents upgrade management features (such as removing earlier versions and preventing earlier versions from installing over later ones) from working. You can, of course, apply the original GUIDs to your new setup project, but this is an error-prone process, because of the format of the GUIDs.
- It is often easier to make a few modifications to an existing setup project to reflect changes in your application, than it is to start from the beginning again. For example, you will already have designed the user interface, registry settings, file associations, custom actions, launch conditions, and so on, for your original application. If you are simply shipping new versions of your executable file or DLLs, you will not want to step through these processes again. Instead, you can simply update your project output or individual files, and those other features that do not need changing will remain in place.

Reinstall Your Application with an Upgraded Windows Installer File

It is usually appropriate to re-deploy your enterprise solutions that were originally installed with Windows Installer packages with upgraded versions of your Windows Installer file.

The robust versioning features of Windows Installer files provide solutions to many of the issues associated with managing and upgrading multiple versions of the same application. The versioning properties of Windows Installer files built with Visual Studio .NET Setup and Deployment projects are described in Table 6.2.

Table 6.2: Versioning Properties

Property	Description
Version	Represents the version of your Windows Installer file. Because the version property is used by Windows Installer to manage upgrades, you should change this property for each released version of your installer (unless you are creating a "small update" —small updates are discussed later in this chapter). When you change the **Version** property, Visual Studio prompts you that you should also update the **ProductCode** and **PackageCode** properties and can automatically update these two other properties for you. However, you will not always want to do so. For guidance on when you should update these two properties, see the "What Type of Upgrade Should I Supply?" topic later in this chapter:
ProductCode	Specifies a unique identifier for an application, represented by a string GUID. You should update this property when you release a major upgrade version of your application; otherwise, it prevents the **DetectNewerInstalledVersion** and **RemovePreviousVersions** properties from working properly. Windows Installer uses the **ProductCode** to identify an application during subsequent installations or upgrades so no two major versions of an application should have the same **ProductCode**. To ensure a unique **ProductCode**, you should never manually edit the GUID — instead, you should use the GUID generation facilities in the **ProductCode** dialog box available from the Properties window. Note: You will not update the **ProductCode** property for small updates or minor upgrades. For guidance on when this property should be changed see the "What Type of Upgrade Should I Supply?" topic later in this chapter.
PackageCode	The package code is a GUID identifying a particular Windows Installer package. The package code associates an .msi file with an application. The **PackageCode** property is not accessible in the Visual Studio .NET Properties window — instead it is created and managed when you build your setup project. This property manifests itself as the revision number of the file — you can view this revision number on the **Summary** tab of the properties sheet for your Windows Installer file in Windows Explorer. When you rebuild your project, Visual Studio .NET also changes the **PackageCode** property for you.

(continued)

Property	Description
UpgradeCode	Specifies a unique identifier that is shared by multiple versions of the same application. The **UpgradeCode** should be set for the first version and should never be changed for subsequent versions of the application. Changing this property prevents the **DetectNewerInstalledVersion** and **RemovePreviousVersions** properties from working properly.
DetectNewerInstalledVersion	Specifies whether to check for later versions of an application during installation on a target computer. The **UpgradeCode** and **ProductCode** are used to determine if a later version is present on the target computer. If this property is set to **True** and a later version number is detected at installation time, this version is not installed over the later version.
RemovePreviousVersions	Specifies whether an installer will remove earlier versions of an application during installation. If this property is set to **True** and an earlier version is detected at installation time, the earlier version's uninstall function is called. The installer checks **UpgradeCode** and **ProductCode** properties to determine whether the earlier version should be removed. The **UpgradeCode** must be the same for both versions, whereas the **ProductCode** must be different for the earlier version to be uninstalled. (Note: There is an issue with removing earlier versions that were installed for "Everyone." For more information on this issue, see the "You Need to Uninstall Previous Versions with Add/Remove Programs in some Situations" topic later in this chapter.)

After you update your solution files and create a new version of your .msi file by rebuilding your solution, you can then distribute the new Windows Installer file to the production environment. The properties described in Table 6.2 will control how versions are managed by the Windows Installer service. For more information about which properties to change in different situations, see the "What Type of Upgrade Should I Supply?" topic later in this chapter.

The advantages of building a new version of an .msi file for upgrading your application include:

- It's an easy and familiar process for building an updated .msi file. You can simply use the same tool (such as Visual Studio .NET) to create the new version, just as you did for the original application. The properties that control how versions are treated by the Windows Installer service are easily manipulated via the Properties window in Visual Studio .NET.
- It's a familiar process for distributing and running the Windows Installer file. You will already have solved the issues for how to deploy the original application to the production environment and will probably use a similar approach for distributing the upgraded .msi file.

Consider Using the Same Version Numbers for Your Application and Windows Installer Package

To avoid confusion as to which application version is contained within which .msi file, you might consider using the version number of each released application to identify its associated installers. For example, if your application has versions 1.0, 1.5, 1.51, and 2.0, you can use those same numbers for the version property of the Windows Installer file in which they are packaged. This makes it easier for you to determine which installer should be used for which application version.

What Type of Upgrade Should I Supply?

There are three general types of upgrade that you can create for your applications. They are known as:

- Major upgrades.
- Minor upgrades.
- Small updates.

The only real difference between these types of upgrade is how you manipulate the various properties that Windows Installer uses for version management. As described in Table 6.2, the version management logic implemented by Windows Installer is based on the **Version**, **ProductCode**, **PackageCode**, and **UpgradeCode** properties. The implications of the different upgrade types are discussed following topics.

Do Not Modify the UpgradeCode Property for any Type of Upgrade

Put simply, if you modify the **UpgradeCode** from the value used to install the original version of your application, you cannot use the upgraded .msi file to upgrade that installation. Windows Installer service identifies and manages upgrades for installed applications by the **UpgradeCode** property. If you do change the **UpgradeCode** for an .msi file, the Windows Installer service does not upgrade your existing application when you run it—instead, it installs a new application that is separate from the one already present. Consequently, setting properties such as **DetectNewerInstalledVersion** and **RemovePreviousVersions** has no effect.

Create a Major Upgrade for Applications with New Features or Significant Changes

If you make significant changes or add significant new features to your application, you typically create a major upgrade. Major upgrades are distinguished by a new version number, along with both a new **ProductCode** property and a new **PackageCode**. Major upgrades are the default type provided by Visual Studio .NET and many of your upgrades will be major upgrades. (Other types of upgrades, such as minor upgrades and small updates, which are discussed next, must adhere to some strict requirements for what you can change in your installer—major upgrades do not need to comply with these requirements, and so most of your upgrades are likely to be of this type.)

To create a major upgrade, you update the **Version** property for your Windows Installer file. Visual Studio .NET then prompts you that the **PackageCode** and **ProductCode** properties should also be updated and offers to do this for you. You should accept this offer if you want to create a major upgrade. However, as for all upgrade types, you should never update the **UpgradeProperty** of your Windows Installer file.

The reason for changing the **Version** property is to allow Windows Installer to differentiate between different versions of your application. After Windows Installer can do that, it can manage the order in which updates can be applied. For example, if you create an upgrade to update version 1.0.0 of your solution to 2.0.0, and then you create another upgrade to update version 2.0.0 to version 3.0.0, Windows Installer can enforce the correct order by checking the version before applying the upgrades. This can also prevent the upgrade for updating version 2.0.0 to 3.0.0 from being applied to version 1.0.0.

Note: Windows Installer uses only the first three fields of the product version (for example, 1.2.5). If you include a fourth field in your version property (for example, 1.2.5.1), the installer ignores the fourth field. Also, it is tempting to think that minor upgrades modify lower-order elements in the **Version** property, whereas major upgrades modify the higher elements. For example, it might seem sensible that changing the version from 1.0 to 1.1 constitutes a minor upgrade whereas changing from 1.0 to 2.0 represents a major upgrade. However, which elements of the **Version** property you change is irrelevant to the type of upgrade you are supplying—the difference between a minor and major upgrade is actually whether the **ProductCode** property changes as well the **Version**. Minor upgrades are discussed in the next topic.

The reason for changing the **ProductCode** is to allow Windows Installer to implement features such as automatically removing earlier versions of your application as you install the new version.

You can apply your major upgrades by simply running the new installer.

Create a Minor Upgrade to Provide Differentiation between Minor Product Changes

Minor upgrades provide product differentiation without actually defining a different product. You typically create minor upgrades when you add or modify a number of features and want to be able to distinguish between versions of an application. For example, you might want your original installation to be known as version 1.0, whereas your updated version may be version 1.1 or 1.5 (depending on your preferences).

A minor upgrade requires that you change the **PackageCode** property and the **Version** property of your package, but not the **ProductCode** or the **UpgradeCode**. When you change the **Version** property, Visual Studio .NET prompts you that you should also change the **PackageCode** *and* **ProductCode** and offers to do so for you. You should decline this offer. The **PackageCode** will still be updated when you rebuild the solution, but the **ProductCode** (and **UpgradeCode**) properties will remain unchanged.

Because the **ProductCode** property is not changed, you must adhere to some strict requirements if you are to provide a minor upgrade. For a list of these requirements, see "Changing the Product Code" on MSDN (*http://msdn.microsoft.com/library/en-us/msi/updat_3rad.asp*).

Also, because the **ProductCode** will not have changed for a minor upgrade, you cannot simply run the upgraded Windows Installer file to update your application. If you attempt this, the Windows Installer service detects that the product has already been installed and displays the message "Another version of this product is already installed. Installation of this version cannot continue. To configure or remove the existing version of this product, use Add/Remove Programs on the Control Panel." However, if you run your Windows Installer file from the command line, and provide certain switches, you do not need to use the **Add/Remove Programs** utility to uninstall the application before running your upgrade. The following is an example of a typical command-line that a user (or deployment tool such as SMS) needs to specify to run a minor upgrade without first uninstalling the application:

```
Msiexec /i d:\MyApp.msi REINSTALL=ALL REINSTALLMODE=vomus
```

Effectively, this command line reinstalls the existing product, rather than treating it as a major upgrade (because the **ProductCode** is the same). Crucially, the reinstall is performed with your new Windows Installer package, rather than the original one that will have been cached by the Windows Installer service. The "v" in the "vomus" switch specifies that the installation should be run from your new source, rather than from the cached Windows Installer file that was originally used to install the application, and it also specifies that your new Windows Installer file should replace the original installer in the cache.

For more information about the command-line switches, see the following topics in the Windows Installer Platform SDK:

- REINSTALL property
- REINSTALLMODE Property

Create a Small Update for Bug-Fixes or Minor Modifications

A small update is typically used when a small number of changes to your solution need to be made. These changes might include bug fixes for your application, or perhaps minor modification of a few files. Small updates are typically used when no product differentiation is required between the previous version of your application and the newer one.

To create a small update, you change the **PackageCode** property of the installer, but not the **ProductCode**, the **Version**, or the **UpgradeCode** properties. You can create a small update by rebuilding a modified Windows Installer file using Visual Studio .NET. Rebuilding an .msi file with Visual Studio .NET automatically creates a new **PackageCode** for the installer, regardless of whether you have changed the **Version** or **ProductCode** properties.

Because small updates do not change the **Version** or the **ProductCode** properties of an installation, you can use them only if your modifications comply with some strict requirements. For a list of these requirements, see "Changing the Product Code" on MSDN (*http://msdn.microsoft.com/library/en-us/msi/updat_3rad.asp*).

As with minor upgrades, you cannot simply run a small update and have it replace your existing application. This is, again, because the **ProductCode** has not changed and Windows Installer detects that the product has already been installed. As for minor upgrades, Windows Installer displays the message "Another version of this product is already installed. Installation of this version cannot continue. To configure or remove the existing version of this product, use Add/Remove Programs on the Control Panel." Again, as for minor upgrades, you can reinstall the application from the command line, rather than uninstalling the application with the Add/Remove Programs utility. For an example of such a command line and more information, see the previous topic.

There will undoubtedly be many situations where small updates will *not* suit your needs. For example, you may want to develop a series of upgrades that can be applied in sequence (which cannot be achieved with small updates because the **Version** property will not have been changed). In this case, you should consider creating either a major or minor upgrade for your application. Furthermore, if you have made modifications that do not comply with the requirements for retaining the **ProductCode** property, you cannot create a small update—you should instead create a major upgrade for your solution.

You Need to Uninstall Previous Versions with Add/Remove Programs in some Situations

When an administrator installs an application with an .msi file, they can choose whether it is installed "For Everyone" or for "Just Me." If the administrator chooses "For Everyone" and then you subsequently provide an upgraded version of the installer that specifies earlier versions should be removed (by setting the

RemovePreviousVersion property), this setting does not work correctly and instead the installation overwrites files without prompting. This can obviously cause serious problems for your application. As of this writing, there is not a good workaround for this, so users must use the Add/Remove Programs utility to remove the earlier version before installing a later one.

Re-Deploy Your Application with Active Directory

If you originally deployed your Windows Installer file by publishing or assigning software to users or computers using Active Directory group policy, you can redistribute your upgraded application using a similar approach. You can upgrade your applications by:

- Creating a new image for your application, either by patching the software package that is currently assigned or published to users and computers, or by creating a new software package.
- Using the Software Installation node in Active Directory to specify that your new package is an upgrade to the existing package.
- Choosing whether the existing package should be uninstalled, or whether the upgrade package can upgrade over the existing package. Typically, the uninstall option is for replacing an application with a completely different one. The upgrade option is for installing a later version of the same product while retaining the user's application preferences, document type associations, and so on.
- Choosing whether or not the upgrade is to be mandatory.

For more information about deploying and updating applications with group policies, see the Active Directory documentation and Microsoft Windows 2000 Server Documentation (*http://www.microsoft.com/windows2000/en/server/help/upgradeApps.htm*).

Re-Deploy Your Application with Systems Management Server

If you originally deployed your Windows Installer file by using SMS, you can redistribute your upgraded application using the same mechanism.

To upgrade a Windows Installer application, you need to complete two processes:

- Create a new image for your application, either by patching the original source files, or by creating a new software package. Then update the package's distribution points to propagate the new version of the source files. Any clients installing the application after you upgrade the source files and update the package's distribution points will install the upgraded application.
- Upgrade existing clients by creating a new program in the existing package. Then, send an advertisement for this new program to a collection of clients that have installed the original version of the application.

For more information about deploying and updating applications with SMS, refer to the SMS product documentation and the white paper, "Deploying Windows Installer Setup Packages with Systems Management Server 2.0" (*http://www.microsoft.com/smserver/docs/deploymsi.doc*).

Upgrade Your Merge Modules

Unlike Windows Installer files, Microsoft merge modules (.msm files) are not installed on their own. Instead, they are included in Windows Installer files. The components and setup logic you add to your merge module are incorporated with that of the Windows Installer file when the installer is built. Therefore, your merge modules do not interact directly with the Windows Installer service.

If you modify the components you have included in a merge module, you need to:

- Update and rebuild the .msm file.
- Make the new .msm file available to developers for inclusion in their .msi files.

The developers who use your merge module then need to replace your previous version with your new .msm file. Because their application has essentially been modified by the inclusion of your new .msm file, they will need to rebuild and redistribute their installer to the production environment in order for changes to take effect. Their considerations are identical to those outlined in the previous sections. Whether they want to upgrade their application with a patch or by reinstalling their solution with an upgraded installer is their decision. Also, whether they want the upgrade to be a small update, a minor upgrade, or a major upgrade is their concern. They need to make these decisions in the same way as described in the previous sections.

Because merge modules do not interact directly with the Windows Installer service, they have different versioning properties to those for .msi files. The following describes the properties:

- **ModuleSignature**. Each released version of a merge module must have a unique **ModuleSignature** in order to avoid versioning problems when they are added to .msi files. You should never manually edit the GUID portion of this property. Instead you should use the GUID generation facilities in the **Module Signature** dialog box, accessible from the Properties window in Visual Studio .NET.
- **Version**. Each time you update and rebuild your merge module, you should update the **Version** property.

The **Version** property is useful for allowing the developer to distinguish between different releases of your merge module. The **ModuleSignature** property is used by Visual Studio and other installer creation tools to uniquely identify the merge modules when they are incorporated in .msi files.

Upgrade Your CAB Files

CAB files are used to package .NET managed controls (or Microsoft ActiveX® controls). CAB files are typically referenced from a Web page. They are automatically downloaded, and their controls extracted and installed, by Web browsers. As such, they are not installed as applications with the Windows Installer service. When you update the controls that have been distributed in this way, you will need to rebuild your CAB file. Like .msi and .msm files, CAB files support the **Version** property. To avoid version conflict, you should update this property for every new release of your CAB.

Upgrade Your Build Outputs

If you distributed your application to the production environment as a collection of build outputs and other files, you can simply upgrade your solutions by copying the new build outputs or files to the appropriate locations. For example, if you are updating Windows Forms applications or private assemblies that are not strong named, or if you are distributing updated configuration files, you can copy the new files to their destination folder using any of the following mechanisms:

- Using Windows Explorer copy and paste or drag and drop operations
- Using the command prompt XCOPY command
- Using FTP commands
- Using Microsoft Application Center (if you are updating Web farms with new configuration files—updating Web farms is discussed in more detail later in this chapter)

If you are updating a Web application, you can use the Visual Studio .NET **Copy Project** command as an alternative to the methods listed previously. For more information about using the **Copy Project** command, see Chapter 5, "Choosing Deployment Tools and Distribution Mechanisms for your .NET Applications."

Update Strong Named Private Assemblies

If you need to upgrade a strong-named private assembly, you cannot simply copy a new version and have your .NET applications use it. When you build a .NET application against a specific version of a strong-named assembly, the application uses that version of the assembly at run time—to use an updated version of a strong-named assembly, you need to redirect your application to use the later version. You can do this for private assemblies by updating your application configuration file to redirect your application to the newer assembly. Consequently, you also need to re-deploy the updated configuration file along with your new strong named private assembly. Using simple copy operations or updated Windows Installer files are both suitable methods for redistributing the assemblies and configuration files. The method you choose probably depends upon the mechanisms with which you originally distributed the application—if you used a Windows Installer file, you might want to take advantage of the upgrade versioning capabilities provided by Windows Installer, whereas if you originally used a copy operation, then simply re-copying the files might be more appropriate.

For an example of redirecting assembly versions using configuration files, see the "Deploy your Strong-Named Private Assemblies" section in Chapter 4, "Deployment Issues for .NET Applications."

Provide a Publisher Policy for Updated Assemblies in the Global Assembly Cache

If you have updated an assembly that resides in the global assembly cache, you cannot just copy a new version and have your applications use that one. Assemblies that are installed in the global assembly cache are strong named, and your application will be built to use a specific version. You need to redirect your application to use the upgraded assembly.

You can create a publisher policy file to specify assembly redirection for upgraded versions of shared assemblies. A publisher policy file is an XML document that is similar to an application configuration file. You need to compile it into a publisher policy assembly and place it in the global assembly cache for it to take effect.

► **To create and deploy a publisher policy**

1. Create a publisher policy file.
2. Create a publisher policy assembly from your publisher policy file, using the Assembly Linker utility (AL.exe).
3. Add your updated shared assembly to the global assembly cache.
4. Add the publisher policy assembly to the global assembly cache.

Because you need to add both the new assembly version and the publisher policy assembly to the global assembly cache, you need to determine how this will occur. You can use the Gacutil.exe tool to achieve this, but, as previously stated, you should favor installing shared assemblies into the global assembly cache with Windows Installer to take advantage of its robust installation capabilities.

For more information about publisher policy files, see the following articles on MSDN:

- "Redirecting Assembly Versions" (*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconassemblyversionredirection.asp*)
- "Creating a Publisher Policy File" (*http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcreatingpublisherpolicyfile.asp*)

Design Your Application Carefully if You will Implement Side-by-Side Shared Assemblies

There are design issues that come into play if you are going to have different shared assembly versions running side by side. (For background information about side-by-side assemblies, see the "Deploy Shared Assemblies to the Global Assembly Cache" topic in Chapter 4, "Deployment Issues for .NET Applications.")

You need to ensure that your installs (and uninstalls) work as expected. For example, if you create a registry key during the install of an assembly, and then design a second version of that assembly that also requires the registry key, you need to determine how install and uninstall routines should behave with respect to the registry key. For this example, you want to ensure that the registry key is not removed if either assembly version is uninstalled, because this will break the remaining version.

You will also want to ensure that side-by-side execution works properly. For example, if multiple versions of an assembly rely on a registry key or any shared resource (such as a file, directory, and so on) and both versions are executing side-by-side, you will need to determine how the assemblies affect each other. Put simply, you will need to ensure that changes to shared resources by one version of the assembly will not adversely affect the other version(s).

Furthermore, if more than one version of an assembly is used by a process, you will need to determine if there are type issues. For example, let's say that you have an assembly which has had both version 1 and 2 installed into the global assembly cache. Now imagine that two controls are used in one application, and that one control references version 1 and the other control references version 2 or your assembly. You will need to determine whether the assemblies can be used together in the same application in this way. The types in the different assemblies are considered to be different by the common language runtime, and so cannot be exchanged.

These (and other issues) strictly fall within the remit of application design, rather than deployment, but you should bear them in mind if you are to use side-by-side versions of shared assemblies in the global assembly cache.

Use Application Center to Update Your Server Environments

Application Center provides broad support for updating your server environments. You will update content, configuration settings, and COM+ applications on the cluster controller, and then have Application Center synchronize the other members of your cluster with the controller. For example, if you modify your Web.config file on the cluster controller, you can use the update features of Application Center to ensure that your changes are replicated to other cluster members. Similarly, if you add Web files, graphics, private assemblies, and so on, to your solution on the cluster controller, Application Center allows you to ensure that copies of these files are received by the other members of your cluster.

Use Automatic Update Features

For ease of administration, you can have Application Center automatically synchronize both content and configuration settings on the members of your cluster. You can use either of the two following automatic synchronization methods:

- **Change-based synchronization**. Application Center detects when content or configuration settings have changed on the cluster controller, and automatically replicates those changes to the other members of the cluster immediately. COM+ applications are not automatically updated. (Updating COM+ applications with Application Center is discussed later in this chapter.)
- **Interval-based synchronization**. You can set a synchronization interval that governs when Application Center replicates changes that have occurred on the cluster controller to the other members of the cluster. The default interval is 60 minutes.

Use Manual Synchronization

If you need a finer level of control over the synchronization of your clustered server environment, you can use the manual synchronization features of Application Center. There are three different types of manual synchronization:

- **Cluster synchronization**. All cluster members in the synchronization loop are synchronized with the controller and all applications are synchronized (except for COM+ applications).
- **Member synchronization**. Only the specified cluster member is synchronized with the controller. All applications (except for COM+ applications) are synchronized.
- **Application synchronization**. A specified application on all cluster members in the synchronization loop is synchronized with the cluster controller.

For more information about synchronization and synchronization loops, see the Application Center documentation.

Update COM+ Applications with Application Center

When you update COM+ applications or COM components, you need to stop any process that is currently using those components. This is because processes lock components they are using. After you update your COM component or COM+ application, you typically re-start the process that you previously stopped, so that it can it resume its operations and take advantage of your updated functionality. In the Web environment, this typically means stopping and restarting the IIS Web service when you update COM/COM+ applications. These steps need to be taken for all COM/COM+ applications, regardless of how you have updated them.

You can use Application Center to ease the process of stopping and restarting services when you update COM+ applications. Application Center deploys COM+ applications to member servers when they are initially added to the cluster. However, rather than automatically synchronizing COM+ applications across your cluster when subsequent changes are made, Application Center allows you to defer this synchronization to off-peak periods. By deferring the synchronization of COM+ applications, you can minimize the effect of stopping and restarting the services on your cluster members.

To upgrade COM+ applications with Application Center, you can deploy them using the New Deployment Wizard or with Application Center command line operations. You should schedule this deployment for off-peak periods, because the Web service will be stopped and started on the cluster members, effectively leaving only the cluster controller available to respond to requests while synchronization takes place.

For more information about deploying and synchronizing COM+ applications, see the Application Center documentation.

Update Clustered Server Applications with Side-By-Side Application Instances

When deploying upgraded server applications to your cluster, you should consider using Application Center to implement side-by-side application instances. Side-by-side application instances allow you to retain your existing version of your application while you deploy your new version to the production environment and verify that it functions as expected. The following scenario explains how this works.

Imagine the application you need to upgrade has been installed into a folder called CommerceApp2001, and that this application has been deployed to all members of your cluster to implement a Web farm. Rather than directly upgrade this application and have it synchronized across your cluster, you can install your upgraded application into a different folder, for example, CommerceApp2002, and create a new virtual directory that maps to this folder. You can have this new virtual directory synchronized across your cluster and the application located in CommerceApp2001 will not have been modified and will continue to handle requests from clients.
At this stage, you will have two versions of the application both able to function in the production environment. You can then verify that the application in CommerceApp2002 functions as expected in the real production environment, and you might consider advertising to selected customers the fact that your application has been upgraded by informing them of the URL for the new application. They can then provide you with valuable feedback on your upgraded application before you release it to a wider audience. When you are satisfied that the solution functions as

expected and you are confident that you can release it for use by all clients, you can perform the following simple operations:

- Modify the virtual directory settings on your cluster controller so that your original application now references the CommerceApp2002 folder, rather than CommerceApp2001.
- Synchronize your application across the cluster—the mapping between virtual directory and local folder is stored in the IIS metabase and this setting will be replicated from the cluster controller to the other cluster members.

If it becomes necessary to revert to the original version, rolling back this change is simply a matter of setting the virtual directory to CommerceApp2001 and then Application Center will synchronize the change across the cluster.

For more information about implementing your specific deployment scenarios with Application Center, see "Best Practices for Phased Deployment Using Application Center 2000" (*http://www.microsoft.com/applicationcenter/techinfo/deployment/2000 /wp_phaseddeploy.asp*).

Automatically Update .NET Applications

One alternative to various approaches for patching, repackaging, and updating applications (described in the previous sections of this chapter) is to move the responsibility of updating the application from administrators or users to the application itself. Instead of the user or administrator obtaining and installing a software update, the client application itself can be designed to automatically download and install updates from a server. To achieve this you can include code with your application so that it:

- Automatically checks for updates.
- Downloads updates if available.
- Upgrades itself by applying those updates.

In order to manage updates in this way, the application needs to know:

- When to check for updates.
- Where to check for updates.
- How to check for updates.
- How to apply updates.

To address when to check for updates, you can use a number of approaches, such as creating a thread that polls the server periodically. For example, you could specify that the thread sleeps for most of the time but checks for updates every hour while the application is running.

To address where the application should check for updates, you can code a path or URL in the application, or call a Web service that returns the location of the update.

This latter method will allow you to change the location simply by modifying the Web service.

You can use a number of different methods to define how the application checks for updates. These could range from simple date/time checks for files on the server, to using a Web service for indicating when updated components are available.

The most difficult issue is how to apply the updates. The problem is that the application needs to update its own files while it is running—but those files will be locked *because* the application is running. The only way to unlock the files is to stop the application, but if you stop the application, it cannot perform the update. There are a number of different solutions to this problem—for more information, see the white paper ".NET Applications: .NET Application Updater Component" on GotDotNet (*http://www.gotdotnet.com/team/windowsforms/appupdater.aspx*). This white paper provides sample code for a component that you can incorporate into your own applications. You will need to set a few properties, such as the location to check for updates, but apart from those minor modifications, you can actually use the sample component as is. This component is not a Microsoft product. It is intended as a sample to get you started and as such the source is also included with this white paper. However, it is worth mentioning that this component has had real world use already—it has been used internally in Microsoft to enable auto-updatability for certain applications.