



realtimepublishers.comtm

*The Developer
Shortcut Guidetm To*



**Building Installations
for .NET**
*and .NET Compact Framework
Applications*

InstallShield

John Featherly and Olivier D'hose

Introduction

By Sean Daily, Series Editor

Welcome to *The Developer Shortcut Guide to Building Installations for .NET and .NET Compact Framework Applications!*

The book you are about to read represents an entirely new modality of book publishing and a major first in the publishing industry. The founding concept behind Realtimepublishers.com is the idea of providing readers with high-quality books about today's most critical IT topics—at no cost to the reader. Although this may sound like a somewhat impossible feat to achieve, it is made possible through the vision and generosity of corporate sponsors such as InstallShield, who agree to bear the book's production expenses and host the book on its Web site for the benefit of its Web site visitors.

It should be pointed out that the free nature of these books does not in any way diminish their quality. Without reservation, I can tell you that this book is the equivalent of any similar printed book you might find at your local bookstore (with the notable exception that it won't cost you \$30 to \$80). In addition to the free nature of the books, this publishing model provides other significant benefits. For example, the electronic nature of this eBook makes events such as chapter updates and additions, or the release of a new edition of the book possible to achieve in a far shorter timeframe than is possible with printed books. Because we publish our titles in “real-time”—that is, as chapters are written or revised by the author—you benefit from receiving the information immediately rather than having to wait months or years to receive a complete product.

Finally, I'd like to note that although it is true that the sponsor's Web site is the exclusive online location of the book, this book is by no means a paid advertisement. Realtimepublishers is an independent publishing company and maintains, by written agreement with the sponsor, 100% editorial control over the content of our titles. However, by hosting this information, InstallShield has set itself apart from its competitors by providing real value to its customers and transforming its site into a true technical resource library—not just a place to learn about its company and products. It is my opinion that this system of content delivery is not only of immeasurable value to readers, but represents the future of book publishing.

As series editor, it is my *raison d'être* to locate and work only with the industry's leading authors and editors, and publish books that help IT personnel, IT managers, and users to do their everyday jobs. To that end, I encourage and welcome your feedback on this or any other book in the Realtimepublishers.com series. If you would like to submit a comment, question, or suggestion, please do so by sending an email to feedback@realtimepublishers.com, leaving feedback on our Web site at www.realtimepublishers.com, or calling us at (707) 539-5280.

Thanks for reading, and enjoy!

Sean Daily

Series Editor

Introduction.....	i
Chapter 1: Understanding Microsoft .NET.....	1
An Assembly of Technological Tenets.....	1
Technology Adoption Accelerators	2
The .NET Framework and Associated Tools.....	3
Visual Studio .NET.....	3
Wizards and Toolkits	3
Products.....	3
OSs.....	4
.NET Enterprise Servers	4
Office Productivity Packages.....	4
Devices.....	4
Industry-Accepted Technology and Architecture Specifications	5
An Overview of the .NET Framework	5
Deploying the .NET Framework.....	7
Application Types that Require the .NET Framework	8
An Overview of the .NET Compact Framework	9
Anatomy of Assemblies.....	12
Basic Deployment Implications	14
Assembly Identity	15
Assembly Versioning.....	15
Shared Assemblies and the Global Assembly Cache	16
Summary	17
Chapter 2: Deploying Applications for the .NET Framework.....	18
Packaging Your Solutions.....	18
Packaging of Windows Forms Applications.....	18
Packaging Web Applications.....	20
Setup Projects.....	21
Windows Installer Files	21
Standard and Web Setup Projects	22
Merge Module and CAB Projects.....	23
Third-Party Options	24
Assembly Deployment Considerations.....	26

Deploying Private Assemblies	26
Deploying Shared Assemblies	30
Publisher Policies.....	32
COM Interop Deployment Considerations	33
Summary	34
Chapter 3: Understanding the .NET Compact Framework and Deploying .NET Applications to Smart Devices	35
The .NET Compact Framework.....	35
Deployment Options	36
Network Share	37
Flash Media.....	38
ActiveSync.....	39
Using the Compact Framework GAC.....	41
Device Setup Project.....	42
Direct Deployment via Flash Media	42
Deployment via Windows CE Object.....	46
Summary	48

Copyright Statement

© 2003 Realtimedpublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimedpublishers.com, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimedpublishers.com, Inc or its web site sponsors. In no event shall Realtimedpublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtimedpublishers.com and the Realtimedpublishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtimedpublishers.com, please contact us via e-mail at info@realtimedpublishers.com.

Chapter 1: Understanding Microsoft .NET

Deciphering the importance of new technology trends hidden amid a sea of marketing-induced confusion is a skill required of all information technology (IT) staff. However, at times, this exercise can be more arduous than others. The initiatives recently undertaken by Microsoft under the .NET banner are that type of new technology. More than the marketing flavor of the day, .NET is a vision, deeply changing Microsoft's approach to enterprise computing.

Beyond offering a new paradigm for application development, .NET redefines how we conceive and architect applications for Windows. As an integral part of the development life cycle of an application, the seamless deployment of .NET solutions is a key success factor for any development project. Microsoft .NET provides the necessary hooks for independent software vendors to provide powerful tools to manage the architecting and planning of deployment projects. Recognizing the need for concise information that is immediately useable to developers, this three-part guide provides an understanding of the deployment techniques available with .NET and shows you how to harness them in your installation-building projects.

Much has been written about .NET since the introduction of the technology by Microsoft in 2001. My intent in this chapter is not to provide yet another broad explanation of the different components that make up the .NET solution. However, I believe that it is important to establish a foundation of understanding of the different components that will impact the deployment of a .NET application; thus, in this chapter we will explore:

- An architectural view of a .NET application
- The .NET Framework and its deployment implications
- The .NET Compact Framework and its deployment implications
- The anatomy of an assembly and its deployment implications

This information will provide a foundation for the second and third chapters, in which we'll explore deployment scenarios for applications that leverage the .NET Framework and the .NET Compact Framework, respectively.

An Assembly of Technological Tenets

Before diving into the technology and in order to fully understand the breadth of Microsoft .NET, we must begin with a holistic approach to .NET. .NET is first and foremost the result of a realization that the product-centric approach to IT solutions promoted by software manufacturers is not appropriate for the enterprise. Solutions made up of an all-inclusive, monolithic-in-nature product rarely meet the business requirements at the origin of the investment. Furthermore, technological limitations too often shape business processes, constraining an enterprise's success to its mastering of technology. For example, it is a common approach to shape a procurement process according to the features of the software chosen to manage the transaction. This rather limiting outlook on solutions positions IT as a mere business-support tool with all the concerns associated with ownership costs and headaches. This approach is what is referred to as the first generation of IT solutions.

Acknowledging that enterprise requirements cannot be addressed by a single product solution implies the integration of multiple vendors' offerings. The emphasis on application migration promoted by Microsoft before .NET is now morphing into an integration strategy in which .NET applications coexist with those developed using other frameworks, such as J2EE. Migration tools—which, by definition, support limited interoperability between the source and target environments for a limited time—are slowly making way for a generation of products designed to bridge feature gaps and provide transactional as well as functional integrity throughout a heterogeneous solution. For example, Microsoft Host Integration Server 2000 is not positioned as a migration tool from a mainframe environment to the Windows Server Datacenter platform; it is a strategic component of an integration effort between non-Microsoft platforms and Microsoft-based solutions, providing messaging and protocol gateways as well as insuring transactional integrity.

In summary:

- .NET is a solution-based approach to business requirements as opposed to a product-driven answer
- .NET is driven by the need for integration between various technological implementations as opposed to the migration from or to a single vendor product or solution

The translation of the .NET vision into a marketable offering is most likely the source of the confusion blurring Microsoft's message. I rationalize the .NET offering into four tiers, which we'll briefly explore in the next section:

- Technology adoption accelerators
- Products
- Devices
- Industry-accepted technology and architecture specification

Technology Adoption Accelerators

A recurrent challenge encountered when building IT solutions remains the ability to adopt and adapt to a new technological platform in a timely fashion while minimizing the negative business impact. Accelerating the adoption rate of a new technology or architecture is a key determining factor in the selection of potential tools. Out of the plethora of tools included by Microsoft .NET, I'll discuss three major families: the .NET Framework and associated tools, Visual Studio .NET, and wizards and toolkits.

The .NET Framework and Associated Tools

The .NET Framework is the cornerstone of the integration and development technologies for the Microsoft platform. Its main role is to provide a homogeneous application runtime, called the Common Language Runtime (CLR), and a set of core services, called the Framework Class Library. Access to the resources required by applications is managed by the CLR. It exposes the different resources required in a uniform fashion, abstracting the implementation details from the application logic. The application runtime environment provides the basis for application and component versioning as well as a more granular security mechanism during code execution. The Framework is also the core of the next generation of Microsoft development-language implementations as well as the foundation for the next generation of dynamic Web applications.

Currently, the Framework is available as an add-on for the Windows 32-bit operating systems (OSs—from Windows 98 to Windows XP). Windows Server 2003 is the first Microsoft OS that provides the Framework out of the box. Microsoft and third-party initiatives are porting essential parts of the Framework to non-Microsoft platforms (such as Linux) to achieve, in turn, cross-platform deployment of .NET applications. Because the Framework is not yet installed on every Windows platform, you will need to address deployment of the Framework as part of your application deployment plan. When packaging a .NET application for deployment, it's a good idea to assume that the Framework may not be installed to your intended targets and to plan for making it available during your installation routine.

Visual Studio .NET

Visual Studio .NET is Microsoft's integrated development environment (IDE) to create and package .NET applications. It is a series of toolkits and editors that make available the different resources exposed by the .NET Framework. It is worth noting that Visual Studio is not required to build .NET applications. Compilers and language resources are a part of the .NET Framework (as opposed to Visual Studio). Visual Studio .NET is, however, a useful tool for Rapid Application Development (RAD) projects.

Wizards and Toolkits

At the disposal of .NET architects and application developers, Microsoft provides numerous wizards and toolkits that allow you to rapidly integrate technology islands within an application. These tools provide a lightweight alternative for environments in which the deployment of the .NET Framework is not possible.

Products

In addition to Visual Studio .NET, there are several Microsoft products that, from a marketing perspective, have been branded with the .NET moniker. These products can be organized into three categories: OSs, .NET Enterprise Servers, and office productivity packages. Let's take a look at each of these categories.

OSs

Microsoft .NET introduces some changes in Microsoft's OS offerings. As I previously mentioned, Windows Server 2003 is the first OS that implements significant native support for .NET-developed applications. Windows Server 2003 lets you integrate authentication operations with .NET services such as Microsoft Passport; implements the full .NET Framework, allowing for the development, deployment, and management of .NET-based applications; and enables the consumption and publishing of Web Services by integrating XML and XML-derived technologies (Simple Object Access Protocol—SOAP, Web Services Description Language—WSDL, Universal Description, Discovery and Integration—UDDI) at the core of the OS. Subsequent versions of the different Windows flavors (enterprise, consumer, and embedded) will deepen the integration of .NET components.

.NET Enterprise Servers

The .NET Enterprise Server banner commonly represents the extended BackOffice server family. .NET Enterprise Server presents a rapid solution for targeted business requirements (from electronic messaging with Microsoft Exchange Server to e-commerce applications with Microsoft Commerce Server 2002). In many regards, the .NET Enterprise Server systems can be considered technology islands. As with the OS offering, .NET Enterprise Servers are being componentized and will provide Web Services interfaces as their main integration interface.

Office Productivity Packages

Office productivity packages, such as Microsoft Office and Microsoft Visio, will provide the consumer edge of the .NET story. The next generation of Office productivity packages integrates seamlessly with Web Services, bridging the “last mile” between application service providers (ASPs) and application users (or consumers).


Devices

Microsoft .NET is the result of the acknowledgement that IT solutions need to reach beyond traditional PCs. To transcend the business reasons motivating a particular IT solution and truly offer new business horizons, applications should be pervasive and accessible seamlessly from a Compaq iPaq handheld computer, an intelligent cell phone, or a device that is adapted to a very particular environment (for example, hospitals, the mining industry, and so on). Additionally, the number of domestic or professional appliances with an embedded computer and access to the network is constantly increasing, justifying the need for a service-based solution architecture.

Microsoft .NET offers toolkits that accommodate a particular form factor or access mechanism, but beyond the interface, usability, and esthetic, .NET presents a case for a solution architect to design applications with device-independent presentation methods in mind. For these types of handheld, portable, or otherwise special-use devices, Microsoft provides the .NET Compact Framework—a lighter-weight version of the full .NET Framework that is geared towards developing applications on these resource-restricted devices.

Industry-Accepted Technology and Architecture Specifications

To promote integration, IT solutions need to be designed according to industry-accepted specifications—both on the architectural and implementation level. Microsoft has integrated the use of industry-accepted specifications when available. .NET's use of Extensible Markup Language (XML)-based interfaces and data exchange methods is an example of this commitment. In addition, Microsoft and Hewlett-Packard (HP) are actively participating in industry consortiums such as the World Wide Web Consortium (W3C) working groups and the Web Service Interoperability Organization (WS-I), helping to shape future evolutions of and requirements for the different technologies used for integration purposes.

 You can find out more about W3C and WS-I at <http://www.w3.org/> and <http://www.ws-i.org/>, respectively.

Once momentum has been gathered behind a particular innovation, the technology is handed over to a ratification body such as the W3C. After the specifications for that particular technology have been assembled and agreed upon, modular .NET solutions are easily retrofitted to accommodate the new specification.

An Overview of the .NET Framework

With an understanding of the breadth of a .NET solution, the challenge for developers and IT professionals is to package and deploy increasingly complex solutions with simplicity and nothing short of grace. To help focus this discussion, I will qualify .NET solutions as solutions that rely on the .NET Framework to function. The .NET Framework (referred to hereafter as the Framework) is the medium through which the CLR and the .NET Framework Class Library are shipped. At execution time, the CLR manages code and provides core services such as thread and memory management. *Managed code* is code that targets the runtime using these core services.

Thus, at the core of the Framework lies the CLR. Runtime environments have often accompanied modern programming languages such as Java and Visual C++. Typically, a language runtime's role is dependant on the language itself. It may actually execute the application code (as with Java and Visual Basic) or it can provide common functionality (for example, error handling) used by an application written in a native compiled language (for example, C/C++). All .NET languages are compiled to an intermediate language (the Microsoft Intermediate Language—MSIL) rather than machine code. The MSIL is just-in-time (JIT) compiled by the CLR before the code is executed. Figure 1.1 illustrates the Framework.

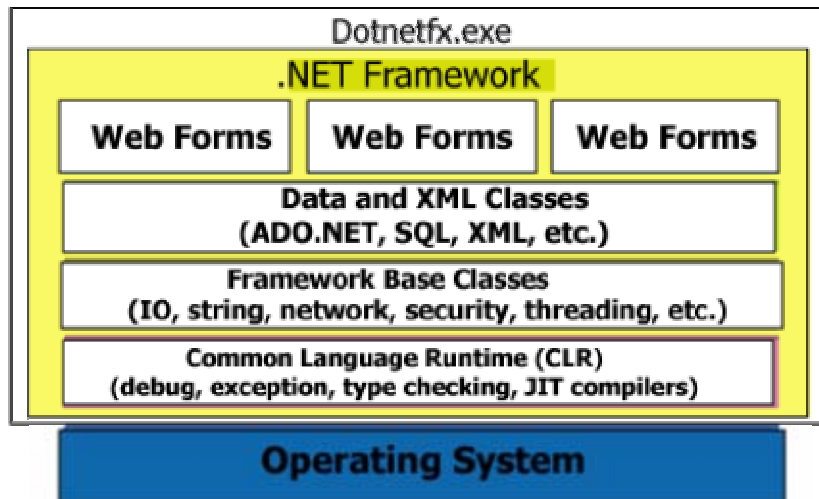



Figure 1.1: An overview of the .NET Framework.

You use the .NET Framework Class Library to develop applications. This library is an object-oriented collection of reusable classes—the abstraction layer provided by Microsoft to give you access to features such as security and graphical user interface (GUI) elements to file input and output systems and data access. Any applications that you use the library to build will require the CLR to execute.

The Framework supports the following platforms: Windows 98, Windows 98 Second Edition, Windows ME, Windows NT 4.0 (Workstation and Server) with Service Pack 6a (SP6a), Windows 2000 (Win2K—Professional, Server, and Advanced Server) with all service packs and critical updates, Windows XP (Home, Professional, and Tablet PC Edition), and Windows Server 2003. (ASP.NET applications are not supported on Windows 98, Windows ME, or Windows NT 4.0).

Microsoft's strategy and ambition is to deploy the Framework as part of the OS. Windows Server 2003, Windows XP Tablet PC Edition, and the Windows Mobile 2003 Platform (formerly known as Pocket PC) have a version of the Framework installed, and the Microsoft Windows Update Web site offers the Framework as a Recommended Update. However, major upgrades of the Framework are always tied to a new release of Visual Studio. Illustrating that fact, the current version of the Framework (version 1.1) was released at the same time as Visual Studio .NET 2003 and was included in Windows Server 2003. However, service packs for the Framework are released independently from service packs for Visual Studio. And, of course, there are still a fair number of Windows machines that had version 1.0 of the Framework deployed when Visual Studio .NET originally shipped.

 The .NET Framework SDK 1.1 is available for the following platforms:

- Windows Server 2003
- Win2K Server and Workstation SP2 and later
- Windows XP Professional (Professional is required to run ASP.NET applications)
- Windows XP Home Edition

The .NET Framework 1.1 redistributable package is available for the following platforms:

- Windows Server 2003
- Win2K Server and Workstation SP2
- NT 4.0 SP6a
- Windows 98
- Windows ME
- Windows XP (Home, Professional, and Tablet PC Edition)

Until the Framework is available on a majority of systems, you can use `Dotnetfx.exe`—a redistributable package that is free from Microsoft—to deploy the Framework as part of the application setup. However, as a result of the size of the executable, you must put forth a significant amount of planning before you deploy the Framework to client computers. Deployment administration products help address these issues by providing more customizable MSI packages.

Deploying the .NET Framework

As I have briefly mentioned, Microsoft's ambition is to use managed code as a foundation for many of its products. In the near future, desktop applications such as Office System and enterprise server solutions such as Exchange Server will provide interfaces that strictly leverage the .NET Framework. In the meantime, however, we are faced with a rather heterogeneous landscape in which it is not possible to assume the presence of the Framework.

Therefore, it is safe to assume that any packaging solution chosen to deploy an application should have a mechanism to verify that the Framework is present and optionally allow for its deployment. Although the distributable package of the Framework (`dotnetfx.exe`) cannot be included in the MSI package, a bootstrapping setup program can be included in the package to verify that the Framework is not already present and trigger the installation of the `dotnetfx.exe` package. Fortunately, setup authoring tools provide a much more granular approach to the problem by enabling you to determine which version of the Framework the setup program is checking for as well as specify the source for the Framework binaries if required. For example, you can point the setup application to download the Framework from a Web share before continuing to install your solution (see Figure 1.2).

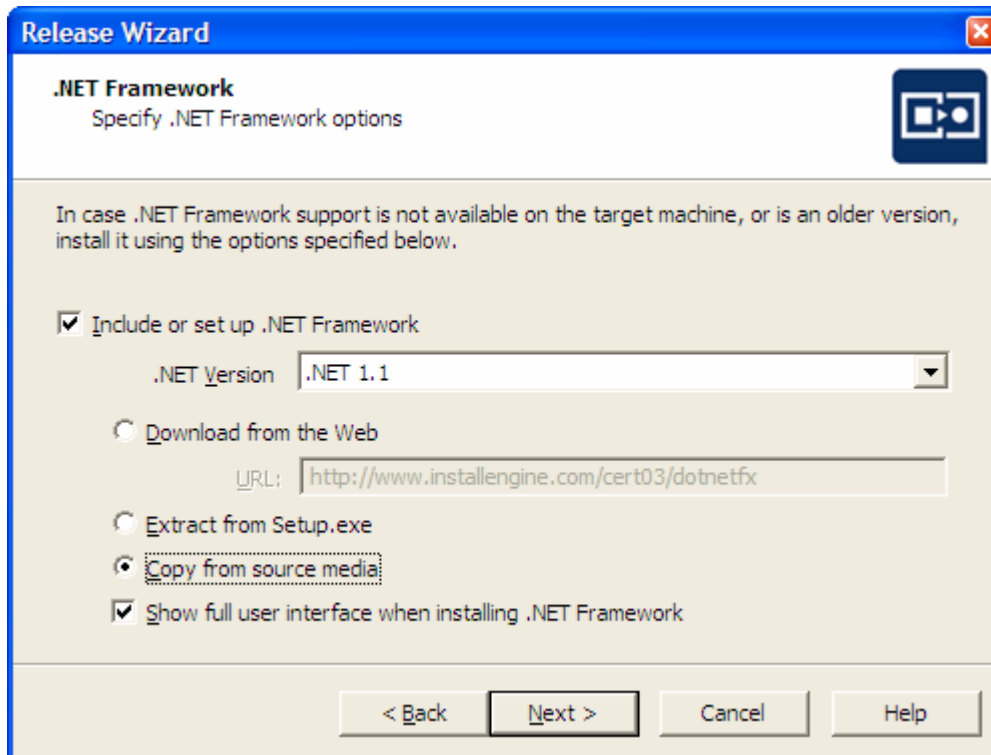




Figure 1.2: InstallShield DevStudio is one the tools available to ease Framework deployment.

Application Types that Require the .NET Framework

To help in the planning of your .NET application deployment, it is useful to be aware of the application types that require the .NET Framework. The following list identifies these application types:


- Windows Forms-based (WinForm) applications built using the Framework require that the client computer have the Framework installed before they can operate. WinForm controls and menus are part of the System.Windows.Forms library.
- Client Web applications do not require the Framework to be deployed on the client computer. The GUI of a Web application is rendered by the browser. An important exception must be made for Web applications using managed controls. Managed controls are components referenced from Web pages that are downloaded to the user's computer and executed upon demand. As .NET components, these assemblies require the Framework to be installed on the client computer in order to be executed. It is the same principle for Web-based setup applications.
- .NET Web applications—applications leveraging ASP.NET on the application server or hosting .NET-based Web services (.ASMX files)—require the Framework to be deployed on the application server. It is worth noting that because ASP.NET controls are part of the System.Web library, the Framework is required on presentation servers as well as middle-tier (or business logic) servers.
- Business logic servers—servers hosting the business-logic tier (or tiers) of a distributed application—require the Framework to be deployed if you have developed components using managed code or resources from the .NET Framework Class Library.

 The user context initializing the .NET Framework setup must have administrator privileges for the target computer. If users are expected to execute dotnetfx.exe, you must ensure that they have elevated privileges on that particular machine. Alternatively, using software publishing policies from Active Directory (AD) or using an automated software management and deployment system such as Microsoft Systems Management Server (SMS) allows systems administrators to deploy the Framework with the correct permissions.

 We will go into more details about .NET deployment scenarios in Chapter 2.

An Overview of the .NET Compact Framework

In the pre-.NET Compact Framework IT world, developers had to put forth time and effort for the training required to acquire the specific skills necessary for developing software for devices. Through the software development environment provided by the .NET Compact Framework and Smart Device Extensions for Visual Studio .NET, Microsoft has enabled the desktop and mobile or situational developer to create software for a variety of form factors. In the .NET Compact Framework, Microsoft includes the same development tools, programming languages, and a subset of class libraries that the .NET Framework offers. Although this solution allows development by less-trained developers, situational application development still requires those developers to design for specific considerations—and learning how to do so requires irreplaceable experience. (Figure 1.3 shows a new project within Visual Studio 2003's Smart Device Application Wizard.)

 We will extensively explore the challenges presented by deploying .NET Compact Framework applications in Chapter 3.

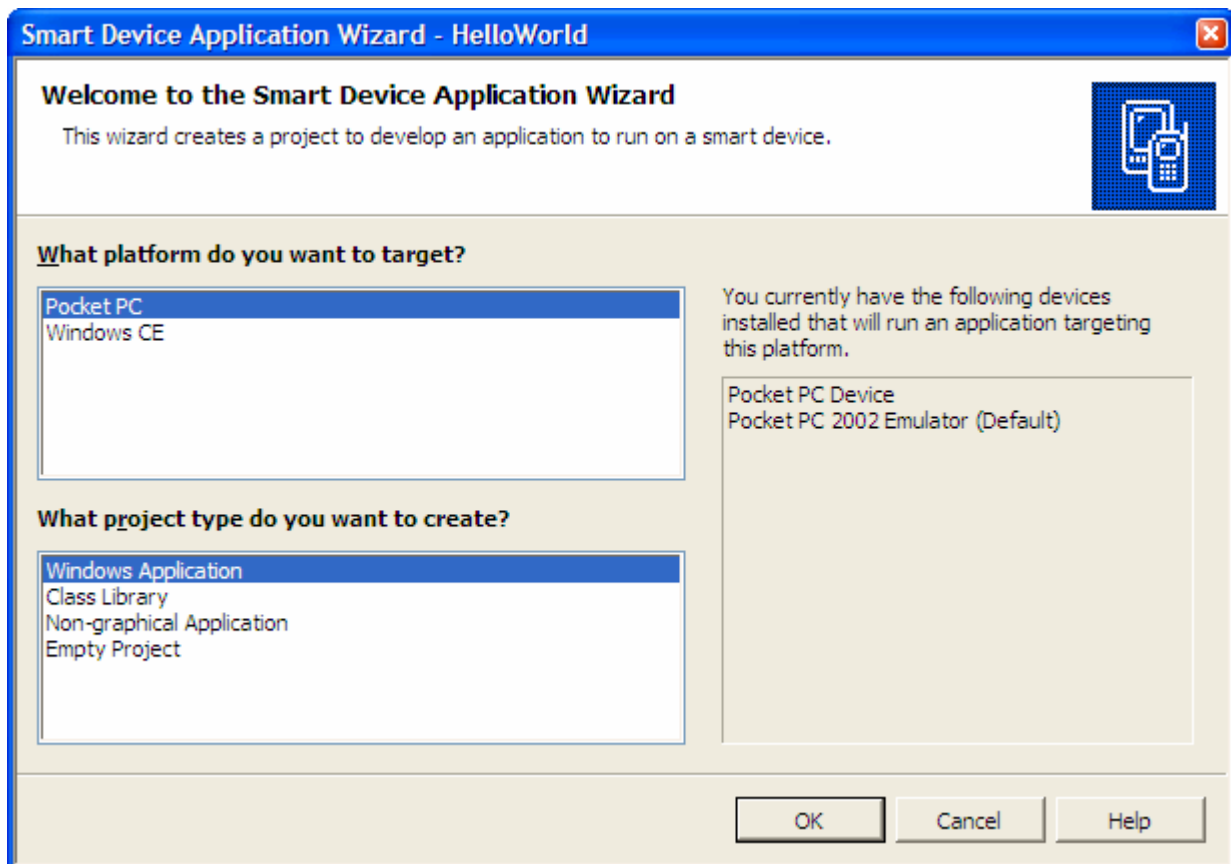



Figure 1.3: A new project within the Smart Devices Application Wizard.


The .NET Compact Framework is an abstraction layer, which enables you to develop applications separately from the underlying OS. In addition, the .NET Compact Framework implements the CLR and, as I previously mentioned, includes a subset of the .NET Framework Class Library. This subset provides classes for the specific needs of mobile devices.


 The following Microsoft platforms support the .NET Compact Framework: Pocket PC 2000, Pocket PC 2002, Windows Mobile 2003, Pocket PC 2002 and 2003 Phone Edition, and Windows CE .NET 4.1 or later.

As you can see in Figure 1.4, you can deploy the .NET Compact Framework on a Pocket PC 2002 emulator.



Figure 1.4: Deploying the .NET Compact Framework on a Pocket PC 2002 emulator.

 With the exception of Pocket PC 2000 and 2002, the .NET Compact Framework is not supported by Windows CE 3.0 and earlier, the handheld PC 2000 and earlier, and the Microsoft Smartphone 2002. Currently, Windows CE .NET 4.1 includes the Compact Framework natively, and the 2003 release of the device OS will include the .NET Compact Framework.

 You can use eMbedded Visual Tools 3.0 to build applications for devices not supported by the .NET Compact Framework. This solution lets you use eMbedded Visual Basic and eMbedded Visual C++ to program in a standalone development environment. You can also use third-party tools to manage the deployment of embedded and mobile solutions. For more information about the .NET Compact Framework, check out Andy Wigley, Stephen Wheelwright, and Mark Sutton's *Microsoft .NET Compact Framework (Core Reference)* (Microsoft Press, 2003).

Anatomy of Assemblies

To deploy a solution efficiently, you must have an understanding of the anatomy of the basic deployment unit composing the solution—in the case of .NET, the basic deployment unit is an assembly. Assemblies are installation units that refer to or contain several file types and physical files necessary at runtime for execution to be successful. Deployment units for types and resources, assemblies have been called logical Dynamic Link Libraries (DLLs). They can contain files such as bitmaps, .NET portable execution (PE) files, and so on. Allowing assemblies to be logical structures rather than monolithic components enables the system to load only the elements of code required at any given moment (as opposed to loading the whole component in memory at once). As we will explore in the next chapter, this feature widens the deployment options for a particular application.


 In Chapter 2, we'll explore assemblies in greater detail.

Figure 1.5 illustrates an overview of a .NET PE file or assembly.

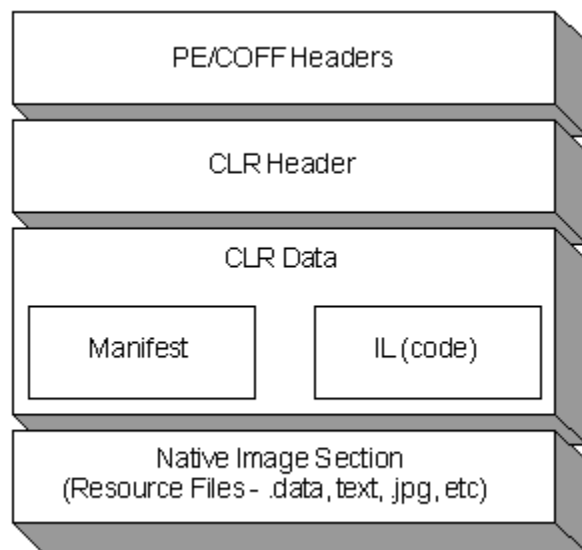


Figure 1.5: Anatomy of a .NET PE file.

An assembly must contain a manifest that describes its content. The manifest is meta data (represented by an XML v1.0 annotation) that describes everything about the assembly from identity and versioning information to files belonging to the assembly and references to external assemblies. In short, assembly manifests describe the component, making it discoverable programmatically. A manifest can contain the following information:

- A simple name
- A four-part version number (we'll explore this version number in more detail later in this chapter)
- A culture (such as nl for Dutch or fr for French)
- The publisher's public key
- The list of files that make up the assembly
- A list of dependent assemblies and their identity parameters
- Permission requests
- Exported types
- Resource files such as graphic files

To view an assembly's manifest, you must *disassemble* the package (either a DLL or an executable file). To do so, you can use the IL Disassembler utility (ILDASM.exe) provided with the .NET Framework. Figure 1.6 shows an ILDASM.exe screen shot of a very basic assembly.

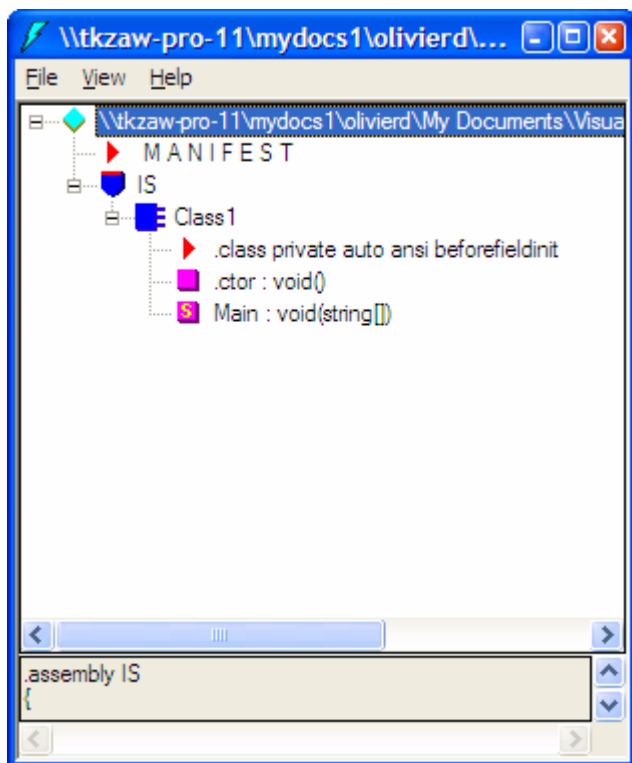



Figure 1.6: A very basic assembly manifest disassembled through ILDASM.exe.

We can classify assemblies into four distinct categories:

- **Static assemblies**—These assemblies are the .NET PE files created at compile time. These assemblies are created when modules (written in a managed code language) are compiled into MSIL.
- **Dynamic assemblies**—These assemblies are PE-formatted, in-memory assemblies dynamically created at runtime using the classes grouped under the `System.Reflection.Emit` namespace. These assemblies are the equivalent to code dynamically created at runtime.
- **Private assemblies**—These assemblies are used only by the application for which they were developed and allow the application to be isolated from other applications. Private assemblies are usually deployed directly to the application base directory. Unlike COM components, private assemblies are self-describing and do not require entries to be made in the system registry.

 You can read more about private assemblies in Chapter 2.

- **Shared assemblies**—These assemblies are shared system-wide and are available for consumption by any .NET application.

This categorization is somewhat artificial. For example, static assemblies can be either private or shared.


As you begin to understand how .NET application structure is based on assembly types and structure, and how that differs from traditional Windows applications' structure, you will quickly realize that .NET application deployments will require additional considerations. Although assemblies are self-describing, .NET application deployments will need to handle private assemblies in different ways than shared assemblies. Application deployment management tools that are .NET-aware can be a big help in handling these new deployment intricacies.

Basic Deployment Implications

One assembly can be dependent upon another as part of an application installation. The meta data associated with an assembly describes this dependency and is very specific about the name, public key signature (or strong name), and version of the assembly upon which it's dependent. This prevents problems such as those that were encountered in the COM world where the wrong version of a shared or dependent component could very easily be installed and accidentally called by an application, with often unpredictable results. Although the dependencies are much better defined and controlled in .NET, the complete collection of components still need to be available to execute the application. In order to deploy a complete and correct collection of components, it is helpful to use a third-party tool that can access assembly meta data in order to manage dependency and version requirements of the overall application.

Assembly Identity


Type uniqueness is important in remote procedure call (RPC), COM, and .NET. In COM, uniqueness of a component is provided by the means of a Globally Unique Identifier. GUIDs are less than elegant (not to mention impractical) to use in development and deployment. In .NET, a type is referred to by its readable name and its namespace. Because a readable name and a namespace are not enough to be globally unique, an assembly's identity consists of four distinct parts: a simple text name, a version number, an optional culture, and an optional public key, required if the assembly is built to be a shared assembly. These identity tokens are stored in the assembly's manifest.

 I'll discuss identity tokens in more detail in Chapter 2.

When the publisher includes its cryptographic key in the assembly's name, the assembly is described as a strong-named assembly. Shared assemblies must be strong-named assemblies. It is important to note that strong names and digital signing in this context are not the same as giving an Authenticode digital signature with the File Signing Tool (SIGNCODE.EXE in the .NET Framework Software Development Kit—SDK). Strong names by themselves have no inherent trust associated with them. Authenticode signatures work with certificates that have associated trust. Because these are independent concepts, assemblies can and should be given both strong names and Authenticode signatures.

Assembly Versioning

Each assembly has a version number, which is a critical piece of its identity. This version number consists of four parts. These four parts represent major revision, minor revision, build, and revision numbers. Developers can update this version number, as the CLR doesn't associate any semantics to the version number's parts, thus it doesn't infer anything based on the assignment of an assembly's version number.

 Although the semantic associated to the versioning information is not dictated by the environment, it is recommended that a standard for versioning annotation is set at an organizational level.

Visual Studio automatically creates the source file `AssemblyInfo.cs` in which assembly attributes are grouped in a single view, making it easy to update them using the source code editor. Versioning information is created by the global attribute `[assembly: AssemblyVersion("1.0.*")]`. (The asterisk in this attribute sets the default revision and build numbers, which causes Visual Studio to increment the value each time the solution is compiled.)

To compile applications using versions of Visual Studio earlier than 2003, you can execute against the .NET Framework version 1.0 or 1.1; however, for applications to compile by default using Visual Studio 2003, version 1.1 is required.

Shared Assemblies and the Global Assembly Cache

In the introduction of this chapter, I explained the benefits of .NET's design guideline, which includes the idea that assemblies are used by only one application. However, it is sometimes a necessity for applications on a machine to share an assembly. (The .NET Framework Class Libraries are a perfect example of shared assemblies.) When an application consumes a shared assembly, the dependencies list of the manifest includes a reference to the identity of the external assembly. Providing the ability to reference the required components allows for side-by-side execution.

As I have mentioned earlier in this chapter, a shared assembly must be a strong-named assembly. Shared assemblies are installed and registered in a machine-wide repository called the Global Assembly Cache (GAC). The cache is found in the %windir%\Assembly directory, as Figure 1.7 illustrates.

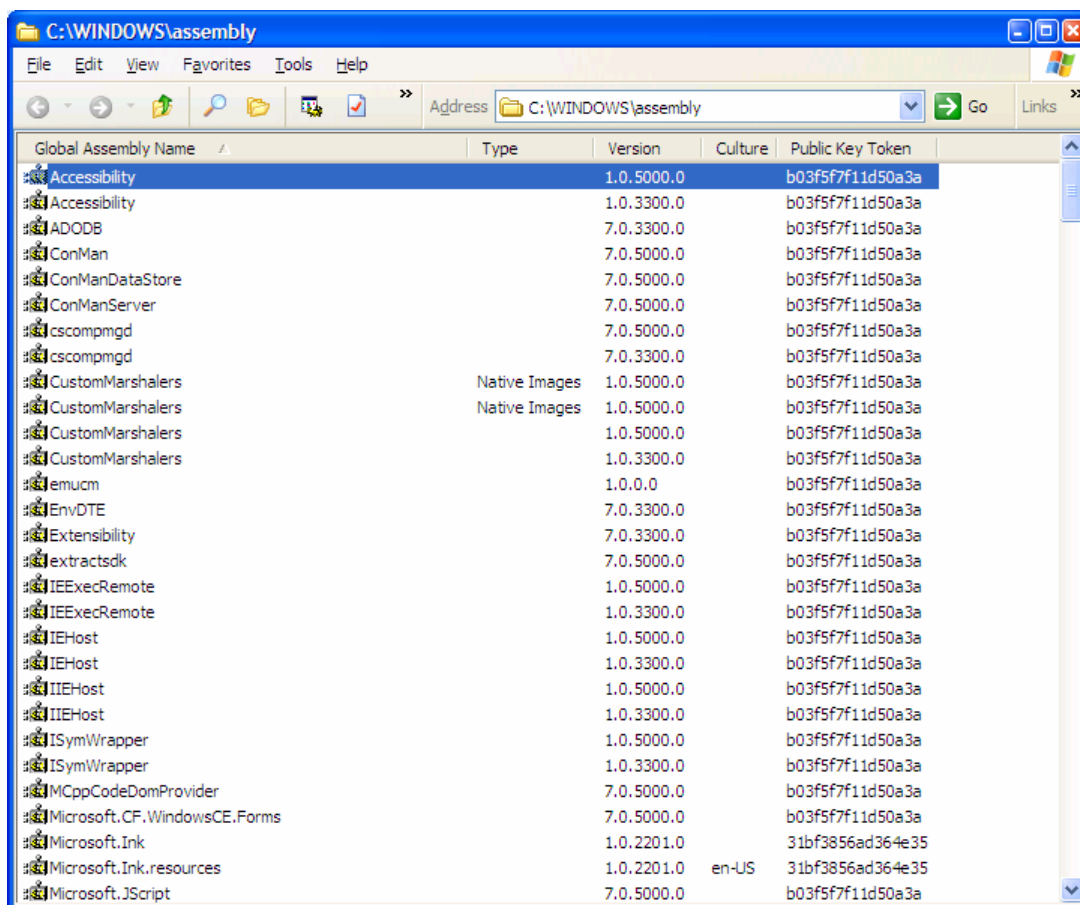





Figure 1.7: The GAC as viewed through Windows Explorer.

Although the GAC is conceptually a flat directory with a list of components, it differs from the System32 directory in that it allows multiple versions of the same assembly to live and be executed side-by-side. Because the version number of an assembly directly affects its identity, each version of the assembly is considered unique in the GAC. With Windows Installer 2.0, Microsoft added explicit functionality for installation of the unique elements that make up a .NET application, such as support for installation of assemblies into the GAC.

As you might imagine, managing installation of assemblies into the GAC and the associated task of creating strong names is a complex undertaking when building installations for applications. Although it is possible to script the use of basic Framework tools, such as gacutil.exe and sn.exe, it is easier and less error-prone to use a powerful third-party deployment tool set.

 We will explore gacutil.exe and sn.exe as well as third-party options in more detail in Chapter 2.

 Unlike the full .NET Framework, shared assemblies in the Compact Framework are verified at runtime, not during the deployment. The verification consists of using the digital signature that is part of the assembly's strong name to ensure that the file has not been tampered with.

 Using shared assemblies does present some deployment considerations that we will explore in greater detail in the next chapter.

In the .NET Compact Framework, files installed in the GAC go in the device\Windows directory. At runtime, the CLR uses the gacutil.exe utility to update the GAC with the information of the shared assembly.

Deploying .NET applications to devices that use the Compact Framework is potentially the most complex deployment scenario. Managing deployment of the Compact Framework as well as the application can quickly become an intricate deployment project. Sophisticated third-party tools can go a long way in helping to manage this complexity.

Summary

Beneath the marketing sheen, which dresses very different initiatives in a common banner, the substance of the changes introduced by Microsoft .NET is of enormous proportion. .NET implies changes not solely to products but in how we architect IT solutions. As developers and IT professionals, we need to deal with a different paradigm for conceiving, developing, and deploying our solutions. The unit of deployment of a .NET solution is an assembly. Although being intimate with the anatomy of an assembly might not ultimately be required to build a deployment project, understanding the identity mechanisms as well as the versioning principles and the different execution paradigms allows us to make smarter, more creative decisions when it comes time to deploy the solution.

In Chapter 2, we'll dive deeper into .NET application deployment. We'll explore Web deployment (one-touch deployment), hybrid (managed plus unmanaged) application deployment, and more.

Chapter 2: Deploying Applications for the .NET Framework

Planning the successful deployment of a software solution is a critical factor in any development project. To be truly prepared, you must first be aware of the many deployment and packaging options for desktop solutions available. In this chapter, we'll build on the development knowledge of Chapter 1 by exploring the deployment and packaging options offered by Microsoft .NET and .NET-based tools, including:

- .NET solution packaging options
- Setup projects
- Assembly deployment considerations


Packaging Your Solutions

Numerous options are available for packaging a .NET application for deployment. Let's look at your options in detail, review their characteristics, and talk about the environment to which each packaging option is best suited.

Packaging of Windows Forms Applications

Windows Forms applications, often referred to as Smart Client applications, are typically desktop productivity or configuration management applications that have distinct installation considerations. Applications that require rich integration features with the target operating system (OS) environment as well as applications developed in a hybrid mode (that is, applications that exploit features outside the realm of control of the CLR) are best served by a Windows Installer (MSI package) packaging and deployment solution. Consider an MSI package solution if an application meets the following criteria:

- Manipulates the registry
- Creates a file association
- Installs assemblies in the GAC

 For more information about the GAC, see Chapter 1.


- Depends on legacy COM components
- Runs custom tasks after the installation is complete (such as a registration program)

In addition, for such applications, consider the following inclusions during your installation program design:

- Familiar GUI-based installation routines for end-user installation.
- Complete integration with the Control Panel's Add/Remove Programs applet.
- A transactional setup routine that enables you to roll back the system to its pre-installation state should the deployment fail or be cancelled by the end user.
- The ability to deploy the application in silent mode (without user interaction) following an automation script.
- Integration with electronic software distribution tools, such as Microsoft Systems Management Server (SMS).

Alternatively, you can use Microsoft .NET to package Windows Forms applications through .NET's Internet deployment or trickle-feed deployment functionality. The main advantage of this approach is to simplify maintenance and deployment challenges by providing a central repository for the application binaries; users simply access the application using their Web browsers—as if the application were a Web solution. A bootstrap assembly is downloaded to the end-user computer and executed from the user's download cache. The bootstrap (or loader) program then requests the necessary assemblies—those required to run the features of the solution to be downloaded—on an as-needed basis and executes the assemblies locally. If earlier versions of the assemblies are already present in the user's cache, the bootstrap program verifies that a new version of the assembly is available and should be requested before execution. Although this mode of execution is attractive, it has the following requirements:

- Clients must have the .NET Framework installed. This packaging option will not allow you to automate the deployment of the Framework if it is missing from the target environment.

 For more information about how the deployment of the .NET Framework is simplified via the use of third-party tools, see Chapter 1.

- Code Access Security policies have been set so that your application can access resources outside of Internet security (for example, File I/O).
- The target environment is completely managed so that application dependencies such as legacy COM components are present and properly configured before the application is executed.
- The network environment is robust enough to support potentially large assemblies and resource files to be transferred without a significant impact on the user experience.

XCOPY Deployments

A much-touted feature of .NET is the concept of zero-impact deployment. This deployment option allows .NET applications that use only managed code and private assemblies to be copied to their desired destinations. Although the feature is attractive in a development or test scenario and opens possibilities for application management and distribution automation, its use is rather limited. Software deployment is, unfortunately, more complex than simply transferring files between computers. The following common installation tasks are difficult, at best, and often impossible to perform by using the XCOPY command:

- Creating shortcuts on the desktop or Start menu
- Allowing user interactions for setting application options
- Installing files in a relative path on the target machine
- Adding assemblies to the GAC
- Creating or configuring databases during the installation
- Adding custom event logs and performance counters to the target machine
- Performing checks for pre-requisite variables
- Presenting a user-friendly and branded user interface
- Allowing license key management and user registration

Packaging Web Applications

Web applications can vary considerably in complexity. Unlike Windows Forms applications, Web applications are usually installed by a systems administrator and therefore afford a less interactive installation model. Depending on the complexity of the application, several packaging options are available.

The most basic packaging option for Web applications is a simple collection of build outputs, referring to the application files that compose the solution, such as ASPX files, executables, DLLs, configuration files, graphics, and other resources. Such files can simply be copied to the target environment by using XCOPY or the Visual Studio .NET Copy project command. As with Windows Forms applications, this packaging option can only be applied to very simple Web applications.

The build outputs method presents the following challenges:

- You can deploy strong-named private assemblies using the build outputs method; however, updating these assemblies will require you to manually redirect the application to use the newer version. You can do so by modifying the application configuration file, but you will then need to redeploy the configuration file as well.
- You must use the Installutil.exe utility to package and manually install predefined installation components for application resources, such as performance counters, with the application.

For more complex solutions, I recommend using MSI files to deploy the application. Visual Studio .NET provides Web setup projects that you can use to deploy Web applications. These setup projects differ from the standard setup projects by installing Web applications to an IIS virtual root folder rather than a Program File folder. Web setup projects allow for the following features:

- Installation of shared assemblies in the GAC
- Installation and registration of legacy COM components
- Creation and assignment of IIS settings
- Installation of application resources through custom actions

Alternatively, you can use MSI authoring tools, which provide more granular options to manage your Web setup project.

Setup Projects


There are many tools at your disposal for managing and automating deployment actions—and overcoming the challenges presented by a .NET application deployment. We'll focus on the features offered by Windows Installer, the IDE provided by Visual Studio .NET, and the features provided by a third-party solution.

Windows Installer Files

Out of the box, Visual Studio .NET setup and deployment projects offer four deployment options:

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

Standard and Web setup projects compile the files that compose your applications into Windows Installer files (.MSI files). Merge modules are used to package components rather than complete applications.

 We will discuss merge modules later in this chapter.

You typically use CAB packages to package applications delivered over the Web (see Figure 2.1).

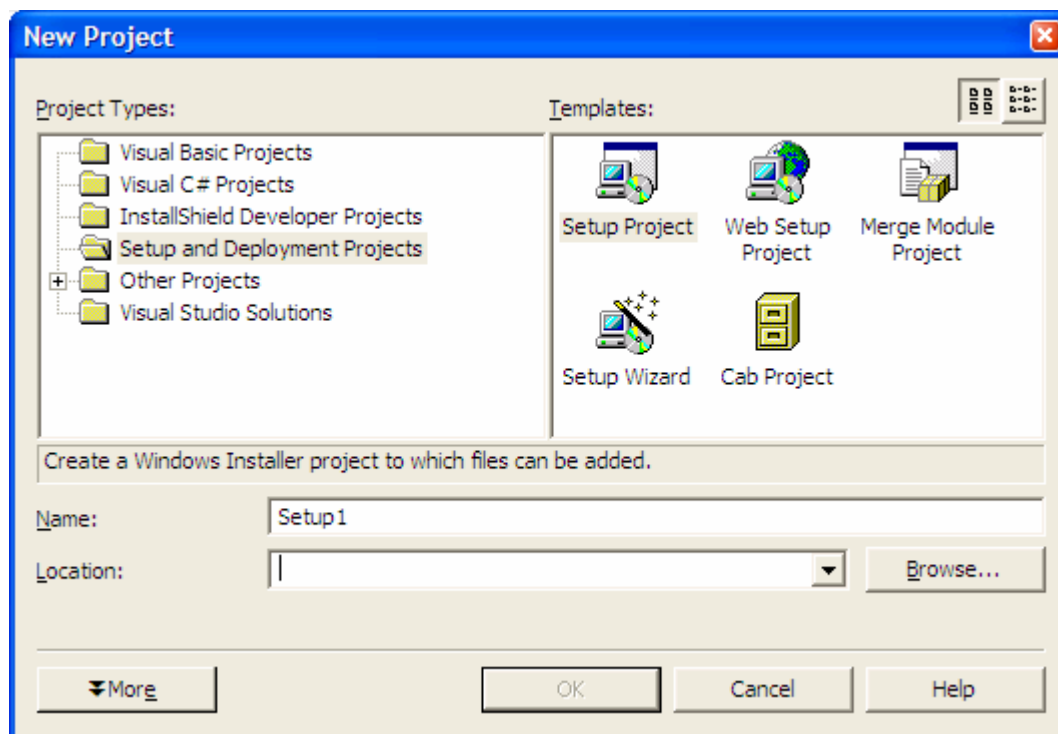


Figure 2.1: Standard Visual Studio .NET setup and deployment projects.

Standard and Web Setup Projects

Typically, you use standard setup projects to install and manage client (desktop) applications. Using a standard project lets you create a familiar GUI environment for an installation's interactive phases. The standard setup project provides the following features:

- Integration with the Add/Remove Programs Control Panel applet
- Transactional behavior for the installation routine (rollback capabilities upon failure or cancellation)
- Registry manipulation
- File association creation
- Assembly registration in the GAC
- Assembly registration for COM interop
- Environmental (hardware and software) prerequisites checks
- Custom tasks (such as registration mechanisms)

Figure 2.2 shows a standard Visual Studio .NET setup project.

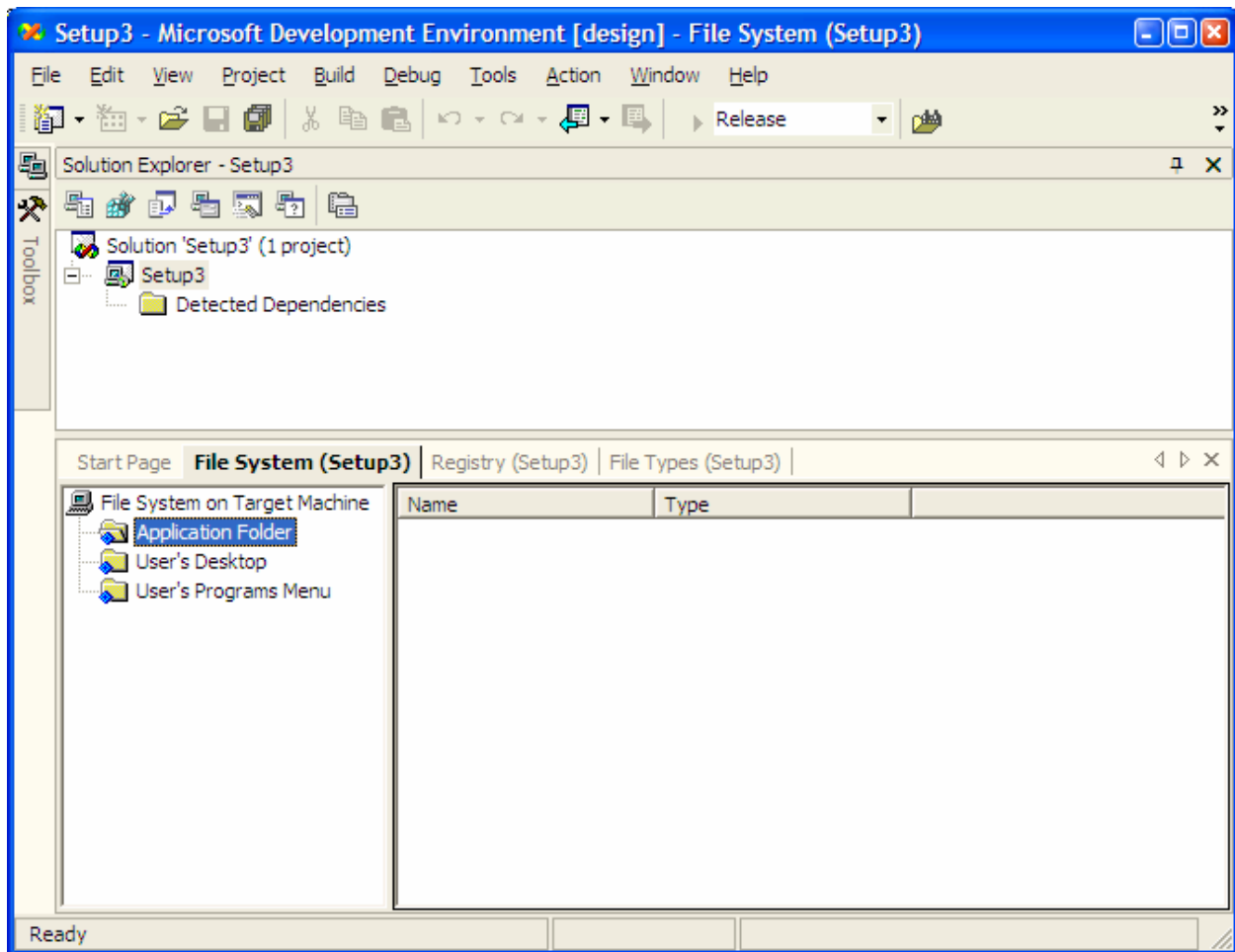



Figure 2.2: A Visual Studio .NET standard setup project.


You use Web setup projects for packaging and deploying Web applications. This feature allows for the deployment of an application to an IIS virtual root folder, enabling the configuration of IIS settings in the process.

 Windows Installer 2.0 technology assumes the presence of Windows Installer 2.0 on the target computer. If Windows Installer 2.0 is not present on the target system, you can bootstrap Windows Installer in your MSI application.

Merge Module and CAB Projects

You use merge modules (.MSM files) to package components rather than to complete applications. Merge modules can best be described as reusable setup components—they cannot be installed directly. The modules are merged into an installer for each application that uses the components. This functionality lets developers package their components and all their dependencies (such as registry settings or resource files) to be shared consistently between different setup routines.

CAB modules let you package managed controls for Web applications. Like Active X components, managed controls are assemblies embedded in Web applications that are downloaded to the target computer and executed by the Framework.

 Windows Installer packages (MSI, MSM, and CAB) can and should be signed using an Authenticode certificate. The user interface of the different projects lets you automatically sign the installation project. As such, you must provide one of the following:

- A certificate file (.SPC), which specifies the location of the Authenticode certificate used to sign the files
- A private key file (.PVK), which specifies the location of a digital encryption key for the signed files
- The timestamp server URL, which you can use as an option to specify the location of an Internet server used to sign the files

Third-Party Options

Visual Studio .NET provides projects to create Windows Installer packages; even with this tool, the management of large and customized deployment projects can be a challenge. For a more granular level of control over your deployment project, you can look to the Windows Installer SDK, which provides low-level utilities, or take advantage of one of the third-party MSI authoring tools on the market, which provide a combination of wizards, high-level views, and low-level utilities. Some third-party tools even provide an extensive MSI authoring tool integrated with the .NET development environment (see Figure 2.3).

These third-party options offer several powerful features including:

- File dependency scanning—An extensive scan for code dependencies at build-time, which helps alleviate the headache of ensuring that all the solution's assemblies and their dependencies have been included in the deployment package. You can scan for both dependencies and properties at build time, checking every assembly for its dependencies before compiling the MSI package.
- Dialog box editing—Easily modify existing dialog boxes or create new ones. In addition, dialog boxes can be exported and imported for sharing across projects, making project collaboration more efficient.
- Automation interface—Edit an installation project programmatically. You can also automate your build process by building and compiling your project from the command line.
- Patch creation—Create patches or upgrades that can modify all aspects of a previously deployed package.
- Debugging tools—Step through the entire installation to confirm behavior or identify and eliminate trouble spots in your installation. You can also test run an installation without copying files.

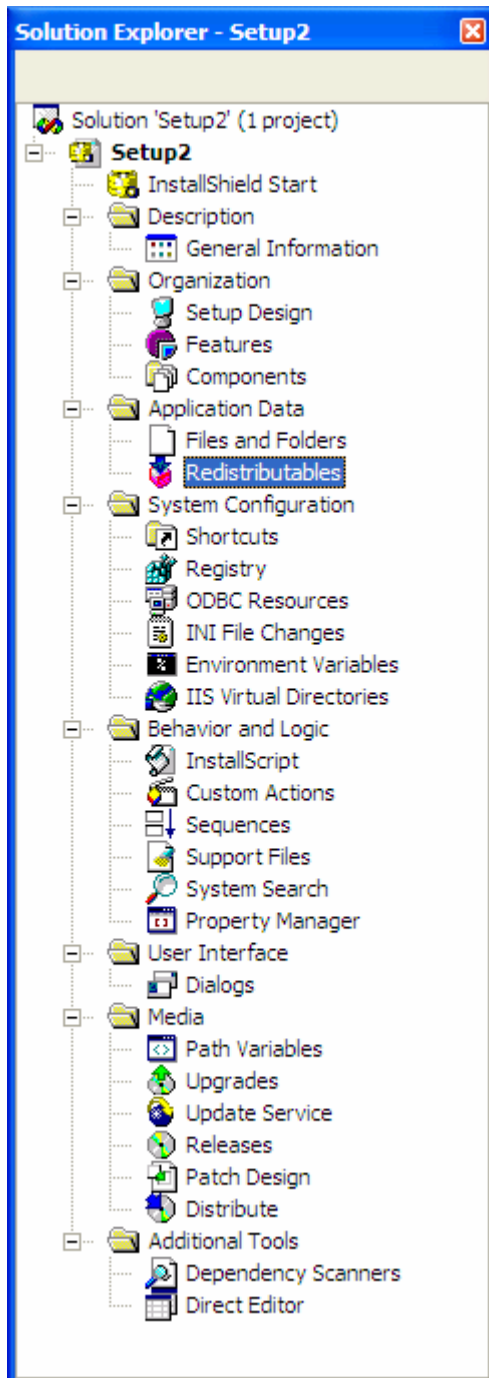


Figure 2.3: InstallShield DevStudio is one of the third-party products that lets you set up projects in Visual Studio .NET.

Assembly Deployment Considerations

As we briefly discussed in the previous chapter, assemblies are units of deployment for your solution. As they are such an integral piece of a deployment, you will benefit from understanding assemblies and how they affect your deployments.

Deploying Private Assemblies

Application isolation is a key design feature of building .NET solutions. Sharing as little context as possible between components of different applications allows for solutions to be more reliable and easier to manage. When an assembly is designed for the sole use of an application, you should deploy it as a private assembly. Private assemblies are best deployed directly to the application base directory (as opposed to %windir%\system32\). The application base directory is either the directory containing the executable files, in the case of a smart client, or the bin folder located in the IIS virtual directory of an ASP .NET application.



Although using the application base directory is strongly recommended, it is possible to specify an alternative location for some of your binaries. Using the <codebase> element of the application configuration file, you can specify an alternative location for assemblies.

It is unlikely that enterprise-class applications would ever deploy private assemblies that are not strong name assemblies. Strong names assemblies provide the following benefits:

- Guarantee uniqueness by relying on unique key pairs to sign the code base.
- Insure a consistent version lineage for the assembly. Assembly versions are signed by the publisher, ensuring that subsequent versions are published by the same publisher (and preventing Trojan horse attacks).
- Meet requirements by the .NET Framework code access security for evidence-base security policies.

A strong name consists of the assembly's identity (simple text name, version number, and culture information) combined with a public key. A key pair can be generated by the sn.exe tool provided with Visual Studio, as Figure 2.4 shows.



```

C:\>sn

Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

Usage: SN [-q<quiet>] <option> [<parameters>]
Options:
  -c [<cspp>]
    Set/reset the name of the CSP to use for MSCORSM operations.
  -d <container>
    Delete key container named <container>.
  -D <assembly1> <assembly2>
    Verify <assembly1> and <assembly2> differ only by signature.
  -e <assembly> <outfile>
    Extract public key from <assembly> into <outfile>.
  -i <infile> <container>
    Install key pair from <infile> into a key container named <container>.
  -k <outfile>
    Generate a new key pair and write it into <outfile>.
  -m [y|n]
    Enable (y), disable (n) or check (no parameter) whether key containers
    are machine specific (rather than user specific).
  -o <infile> [<outfile>]
    Convert public key in <infile> to text file <outfile> with comma separated
    list of decimal byte values.
    If <outfile> is omitted, text is copied to clipboard instead.
  -p <infile> <outfile>
    Extract public key from key pair in <infile> and export to <outfile>.
  -pc <container> <outfile>
    Extract public key from key pair in <container> and export to <outfile>.
  -q
    Quiet mode. This option must be first on the command line and will suppress
    any output other than error messages.
  -R <assembly> <infile>
    Re-sign signed or partially signed assembly with the key pair in <infile>.
  -Rc <assembly> <container>
    Re-sign signed or partially signed assembly with the key pair in the key
    container named <container>.
  -t [p] <infile>
    Display token for public key in <infile> (together with the public key
    itself if -tp is used).
  -T [p] <assembly>
    Display token for public key of <assembly> (together with the public key
    itself if -Tp is used).
  -v [f] <assembly>
    Verify <assembly> for strong name signature self consistency. If -vf is
    specified, force verification even if disabled in the registry.
  -U
    List current settings for strong name verification on this machine.
  -Ur <assembly> [<userlist>]
    Register <assembly> for verification skipping (with an optional, comma
    separated list of usernames for which this will take effect). <assembly>
    can be specified as * to indicate all assemblies or *,<public key token> to
    indicate that all assemblies with the given public key token. Public key
    tokens should be specified as a string of hex digits.
  -Uu <assembly>
    Unregister <assembly> for verification skipping. The same rules for
    <assembly> naming are followed as for -Ur.
  -Ux
    Remove all verification skipping entries.
  -?
  -h
    Displays this help text.

C:\>_

```

Figure 2.4: Using the sn.exe tool.

For example, to generate a key pair for signing an assembly, use the following option:

```
Sn -k c:\keys\mykeyfile
```

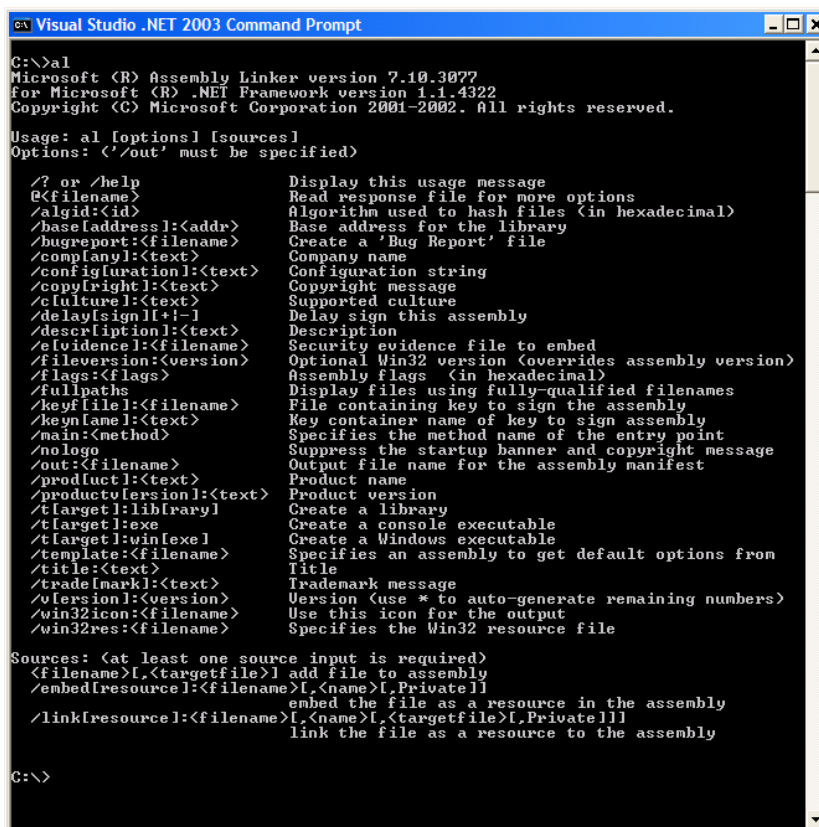
To strong name your assembly, use the key generated by the previous code to set the assembly attributes within your code:

```
[assembly: AssemblyKeyFile("<insert key file here>")]
```

Alternatively (and probably more relevant in an enterprise-class context), you can use delay signing to strong name assemblies after the assemblies are built. Doing so allows for quality and security checks to be performed on the code before release. You can use delay signing to strong name assemblies after they are built by using the Assembly Linker utility (Al.exe), which has the following syntax:

```
Al /out:myassembly.dll mycodemodule.module
  /keyfile:c:\keys\mykeys
```

where the output myassembly.dll is the signed output assembly, mycodemodule.module is the source code module for the assembly, and mykeys is the key file generated by sn.exe (see Figure 2.5).



```

C:\>al
Microsoft (R) Assembly Linker version 7.10.3077
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

Usage: al [options] [sources]
Options: ('/out' must be specified)


/? or /help           Display this usage message
@<filename>          Read response file for more options
/algid:<id>           Algorithm used to hash files (in hexadecimal)
/baseaddress:<addr>   Base address for the library
/bugreport:<filename> Create a 'Bug Report' file
/company:<text>       Company name
/configuration:<text> Configuration string
/copyright:<text>     Copyright message
/culture:<text>       Supported culture
/delay/sign[:+|-]    Delay sign this assembly
/description:<text>   Description
/evidence:<filename> Security evidence file to embed
/fileversion:<version> Optional Win32 version (overrides assembly version)
/flags:<flags>        Assembly flags (in hexadecimal)
/fullpaths           Display files using fully-qualified filenames
/keyfile:<filename>   File containing key to sign the assembly
/keyname:<text>       Key container name of key to sign assembly
/main:<method>        Specifies the method name of the entry point
/nologo             Suppress the startup banner and copyright message
/out:<filename>       Output file name for the assembly manifest
/product:<text>       Product name
/productversion:<text> Product version
/target:library      Create a library
/target:exe           Create a console executable
/target:winexe       Create a Windows executable
/template:<filename> Specifies an assembly to get default options from
/title:<text>         Title
/trademark:<text>     Trademark message
/version:<version>    Version (use * to auto-generate remaining numbers)
/win32icon:<filename> Use this icon for the output
/win32res:<filename> Specifies the Win32 resource file

Sources: (at least one source input is required)
<filename>[,<targetfile>] add file to assembly
/embed[:resource]:<filename>[,<name>][,Private]]
embed the file as a resource in the assembly
/link[:resource]:<filename>[,<name>][,<targetfile>][,Private]]
link the file as a resource to the assembly

C:\>

```

Figure 2.5: Using the Assembly Linker tool.

 An assembly signed with a strong name does not necessarily assert the identity of the publisher. To provide a token of the publisher's identity, you need to use an Authenticode signature. With the Authenticode signature, the publisher's identity is asserted by a third-party Certificate Authority (CA). To sign the code, you can use the signcode.exe tool, which attaches an Authenticode signature to the assembly.

When signing an assembly using both sn.exe and signcode.exe, you MUST sign the assembly for strong name purposes using sn.exe before applying the Authenticode signature.

The strong name of an assembly is stored in the assembly's manifest. Because the version of an assembly is a key tenet of its identity, each version is considered a separate assembly by the CLR. A strong-named assembly has to be unique, so the CLR does not load the updated version of a strong-named assembly as part of your solution; the CLR needs to be instructed to load the updated assembly. You can do so by providing a binding redirect statement in the application configuration file. When deploying an updated strong-named assembly, deploy the binaries as well as an updated configuration file.

Application Configuration Files

In addition to dynamic properties, the Framework supports runtime configuration through the use of machine and application configuration files. These configuration files are XML files based on a schema set by the .NET Framework. These settings are more widely used to configure ASP .NET applications but have some use for Windows Form-based applications as well.

At a system level (that is, the computer running the application), the `machine.config` file (located under the `config` directory for each version of the Framework installed on the system) defines system-wide .NET settings such as the location of the different Framework binaries (see Figure 2.6).

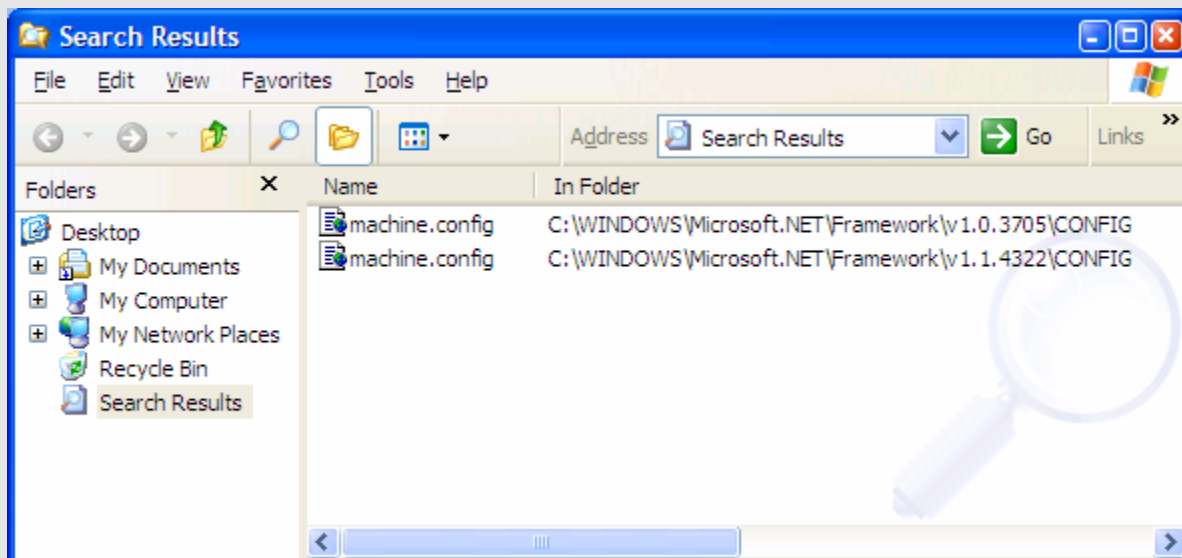


Figure 2.6: The `machine.config` file in Windows Explorer.


Configuration settings set in the `machine.config` file can be overruled by application-specific configuration files. Application configuration files (`app.config` or `web.config` in the case of an ASP .NET application) reside in the application base directory. These files can either be modified manually with an XML editor (Notepad, for example) or modified using the .NET Framework Configuration tool.

Deploying Shared Assemblies

Although best practices recommend isolating applications as much as possible, the need for sharing complex feature sets across applications does not warrant the administrative and logistical cost of duplicating assemblies.

To allow multiple applications to find and consume shared components, you need to implement a registration and discovery mechanism. In the COM days, components were registered in the system registry. In .NET, shared assemblies are registered (or, more accurately, installed) in the GAC (see Figure 2.7). The GAC, however, provides the version-aware and publisher-aware assembly store required for a side-by-side execution model.

Because sharing components requires that you can uniquely identify these components, assemblies stored in the GAC must be strong named. The CLR performs integrity checks on all files that compose shared assemblies based on the strong name. The cache performs integrity checks to ensure that an assembly has not been tampered with after it has been created.


 When strong-named assemblies are loaded from a location other than the GAC, the CLR must perform integrity checks at load-time. The performance penalty associated with these verifications occurs every time the assembly is loaded. In contrast, assemblies installed in the GAC are verified at install-time and do not require integrity checks from the CLR at load-time. The potential performance hit, although not enormous, should be taken into consideration when architecting and deploying your solutions.

Global Assembly Name	Type	Version	Culture	Public Key Token
Accessibility		1.0.5000.0		b03f5f7f11d50a3a
Accessibility		1.0.3300.0		b03f5f7f11d50a3a
ADODB		7.0.3300.0		b03f5f7f11d50a3a
ConMan		7.0.5000.0		b03f5f7f11d50a3a
ConManDataStore		7.0.5000.0		b03f5f7f11d50a3a
ConManServer		7.0.5000.0		b03f5f7f11d50a3a
cscompmgd		7.0.5000.0		b03f5f7f11d50a3a
cscompmgd		7.0.3300.0		b03f5f7f11d50a3a
CustomMarshalers	Native Images	1.0.5000.0		b03f5f7f11d50a3a
CustomMarshalers	Native Images	1.0.5000.0		b03f5f7f11d50a3a
CustomMarshalers		1.0.5000.0		b03f5f7f11d50a3a
CustomMarshalers		1.0.3300.0		b03f5f7f11d50a3a
emucm		1.0.0.0		b03f5f7f11d50a3a
EnvDTE		7.0.3300.0		b03f5f7f11d50a3a
Extensibility		7.0.3300.0		b03f5f7f11d50a3a
extractsdk		7.0.5000.0		b03f5f7f11d50a3a
IEExecRemote		1.0.5000.0		b03f5f7f11d50a3a
IEExecRemote		1.0.3300.0		b03f5f7f11d50a3a
IEHost		1.0.5000.0		b03f5f7f11d50a3a
IEHost		1.0.3300.0		b03f5f7f11d50a3a
IEHost		1.0.5000.0		b03f5f7f11d50a3a
IEHost		1.0.3300.0		b03f5f7f11d50a3a
ISymWrapper		1.0.5000.0		b03f5f7f11d50a3a
ISymWrapper		1.0.3300.0		b03f5f7f11d50a3a
IMCCodeDomProvider		7.0.5000.0		b03f5f7f11d50a3a
Microsoft.CF.WindowsCE.Forms		7.0.5000.0		b03f5f7f11d50a3a
Microsoft.Ink		1.0.2201.0		31bf3856ad364e35
Microsoft.Ink.resources		1.0.2201.0	en-US	31bf3856ad364e35
Microsoft.JScript		7.0.5000.0		b03f5f7f11d50a3a

Figure 2.7: The GAC on a Windows XP system.

The CLR attempts to locate assemblies in the GAC before searching the application base folder. Therefore, if a strong-named assembly is found in the GAC that matches the identity requirements, the CLR will load the shared assembly first. This behavior should not be a problem because a strong-named assembly is unique and therefore would be the same in the GAC as in the application base folder.

To prevent the premature removal of a shared assembly from the GAC, you need to maintain a count of the number of references to that assembly—you need to maintain the number of solutions consuming the shared component. Windows Installer provides robust reference-counting features. These features are also the base for more elaborate third-party tools, which allow for a granular control of the registration process into the GAC. Alternatively, you can install shared assemblies into the GAC by using the `gacutil.exe`.

 When using `gacutil.exe` to manage the GAC from the command line, always use the `/ir` switch to create a traced reference for the assembly. You can remove these references by using the `/ur` switch when the assembly is uninstalled.

Publisher Policies

To deploy an updated version of a shared assembly, you cannot deploy the new version and expect the application to consume the updated version. Shared assemblies are strong-named assemblies that use the version number as part of their identity. The deployed solution needs to be redirected in order to use the updated assembly. To achieve this goal, .NET provides publisher policy files. A publisher policy file is an XML document similar to an application configuration file and compiled into a publisher policy assembly. The publisher policy assembly is then installed in the GAC to allow for redirection of a call to a shared assembly to its updated version. Listing 2.1 shows a configuration policy file.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="HelloWorld"
          publicKeyToken="32ac4ba45e0a85a1"
          culture="en-us" />
        <!-- Redirecting to version 2.0.0.0 of the assembly. -->
        <bindingRedirect oldVersion="1.0.0.0"
          newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 2.1: A configuration policy file.

In the configuration policy file example that Listing 2.1 shows, the assembly with the simple name HelloWorld has been superseded with version 2.0. By uploading this policy to the GAC, solutions consuming the HelloWorld assembly will be redirected to the 2.0 version.

Using the Assembly Linker tool (Al.exe), you can create a publisher policy assembly from the policy file using the following syntax:

```
Al.exe /link:publisherPolicyFile /ou:publisherPolicyAssemblyFile
/keyfile:keyPairFile
```

The deployment strategy of a solution that runs side-by-side shared assemblies requires significant planning during the design phase. Application dependencies beyond assemblies are shared. For example, if the application relies on a particular registry key value, the uninstallation routine of the defunct assembly should not remove the registry key; alternatively, the installation routine of the updated assembly will have to recreate the key with the right value.

Careful thought has to be dedicated to resource sharing as well. If two versions of the same application are executed side-by-side—as allowed by the .NET Framework—you will need to manage potential conflict when accessing resources concurrently. This management becomes increasingly important when working on the instrumentation of your application. When you report events or expose health monitors for your solution, you clearly need to identify which version of the application is running.

COM Interop Deployment Considerations


Applications and components written before .NET have not been replaced by managed code overnight. Fortunately, the .NET Framework provides extensive mechanisms to consume COM components as well as Win32 DLLs. It also allows managed components to be consumed from unmanaged, COM-based code.

When a COM object is called from .NET, the CLR generates a runtime callable wrapper (RCW). The RCW acts as a proxy between the managed .NET code and the unmanaged COM code. When a .NET assembly is called from COM, the runtime generates a COM callable wrapper (CCW) from the assembly meta data. Because both wrappers are created at runtime, they do not constitute a deployment issue.

The RCW is generated according to type information stored in an interop assembly. An interop assembly contains no implementation code, only type definitions implemented in the COM components. There are two types of interop assemblies:

- Primary interop assemblies—These assemblies are interop assemblies provided by the same publisher as the type library they describe. They are signed by the publisher to ensure uniqueness.
- Alternative interop assemblies—These assemblies are interop assemblies that are not signed by the COM object publisher.


When using Visual Studio to set a reference to a COM object into your solution, Visual Studio checks the system registry for a primary interop. Primary interop assemblies are always strong-named. If no primary interop assembly exists, Visual Studio will generate an alternative interop assembly. This assembly cannot be strong-named from Visual Studio.

 If you need to create a strong-named alternative interop assembly, you will need to create the interop manually using the `tlbimp.exe` utility. For example, if I have a COM typelib called `comtypes.tlb`, I can create an interop assembly using the following command:


```
Tlbimp comtypes.tlb myinteropassembly.dll
```

Once the interop is created, you can strong name it, then add it as a reference to your project.

You can deploy and register your COM component using the same methods you used before .NET. Authoring tools allow for the creation of hybrid deployment projects, scripting the deployment and registration of COM components and Win32 DLLs in the same deployment project. The primary interop assembly can be deployed like any other assembly, either in the application base directory or in the GAC.

 COM components have system-wide visibility, so a good practice is to install primary interop assemblies directly into the GAC.

To make a .NET assembly consumable from a COM environment, assemblies must first be strong-named because they need to be uniquely registered in the COM application server. An entry in the registry is created to point to the mscoreee.dll, which loads and executes the .NET assembly. You can do so easily either from a Visual Studio .NET project or using installation authoring tools. Once the assembly is registered, it needs to be located where the runtime can access it.

 Although, technically, COM-visible assemblies can either be deployed in the application base directory or the GAC, it is strongly recommended to use the GAC. Doing so simplifies the process by not spreading application resources to the host service directory, for example.

Summary

In this chapter, we explored the different options available when designing a .NET deployment solution, from XCOPY to Windows Installer (MSI). A number of tools are available to create the installations, from built-in Visual Studio .NET utilities to robust third-party tools. Deploying .NET applications requires you to understand the different dependencies presented by the .NET deployment unit, the assembly. In Chapter 3, we'll explore the .NET Compact Framework and how to deploy .NET applications to smart devices.

Chapter 3: Understanding the .NET Compact Framework and Deploying .NET Applications to Smart Devices

One of the exciting things about .NET application development is the ability to develop mobile applications. Prior to .NET, mobile application developers' only options were eMbedded Visual C++ (eVC++) and eMbedded Visual Basic (eVB)—both of which are very limited. Now developers can use powerful new languages, such as C# and VB.NET, and the familiar Visual Studio development environment. In addition, developers can utilize Windows Forms and controls from the .NET class libraries.

The enabling technology for these developers is the .NET Compact Framework, which is the mobile device version of the .NET Framework. However, deploying .NET mobile applications requires you to overcome challenges similar to those we explored with .NET application deployment in the previous chapter. In this chapter, we'll explore these challenges as well as those presented by traditional mobile device deployment—connectivity, access to deployment media, file system differences, and more. Let's begin by exploring the .NET Compact Framework.

The .NET Compact Framework

Developers encounter the basic mechanisms of deploying a mobile application when they first write and debug an application using Visual Studio .NET 2003. Using a Pocket PC with an ActiveSync connection to the development host running Visual Studio, the application is deployed simply by building and starting a debug session, as Figure 3.1 shows.

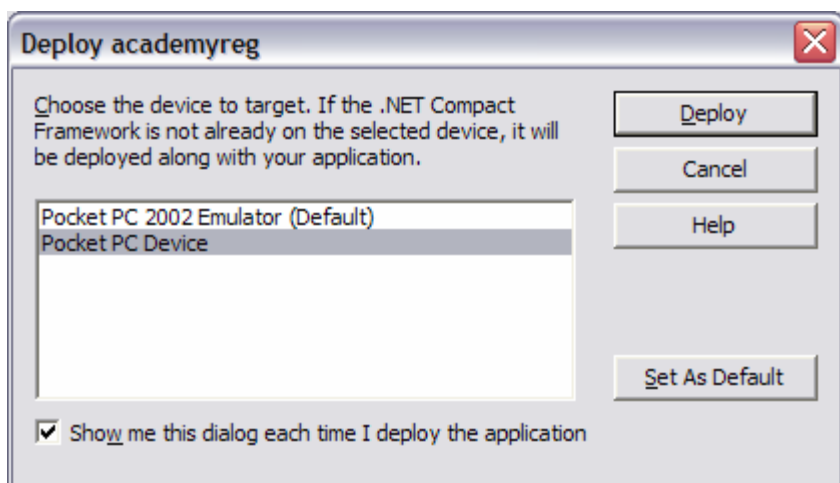


Figure 3.1: Deploying an application to a Pocket PC device.

In the dialog box that Figure 3.1 shows, you see Visual Studio set to deploy the application *academyreg* to either an actual device or an emulator. This application is a .NET application, so it needs a framework (that is, the CLR execution engine and base class libraries) in order to run. For smart devices such as the Pocket PC, that framework is called the .NET Compact Framework. Visual Studio handles deployment of the Compact Framework to a device during development, as you can see in Figure 3.1. It is important to remember that the Compact Framework is a different package of bits for each different type of device hardware. For example an iPAQ Pocket PC device needs ARM/XScale bits, while a Pocket PC emulator device needs x86 bits. Once the Compact Framework is deployed to a device, the Compact Framework will remain there until a hard reset clears the device storage.

Deployment Options

Although deployment using Visual Studio is convenient for a developer, it is certainly not convenient for a general user. It is nonsensical and absurd to require users to bring their devices to a developer station or to install Visual Studio on their desktops.

The fundamental mechanism for deploying a mobile application is to build a CAB file that is transported to the device and then “executed” or activated on the device, thereby installing the application. An icon to run the application is placed in the device Programs folder and an entry is placed in Remove Programs for application removal. You can create the CAB file install package by selecting Build Cab File from the Build menu in Visual Studio, as Figure 3.2 shows.

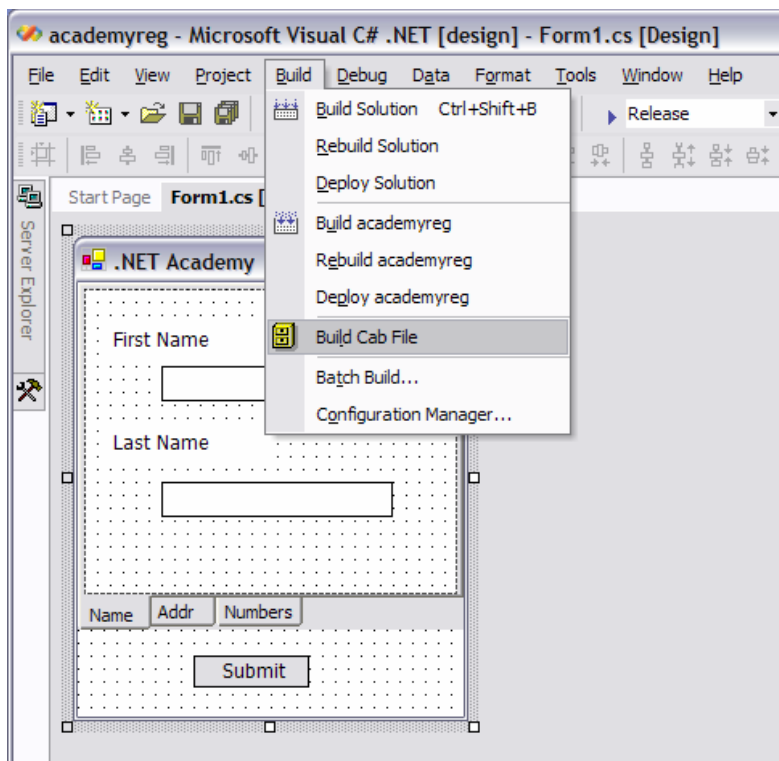


Figure 3.2: Building a CAB file from within Visual Studio.

Actually, you will find that this selection results in more than one CAB file (the number depends on the platform and hardware support you have loaded). Visual Studio builds a package for each supported platform (Pocket PC, Windows CE) and hardware type (ARM, x86). You can find these CAB files in the <VS Project>/cab/Release (or /Debug) folder in the Visual Studio project directory.

Once you have the CAB package, you can use one of the following methods to deploy the package to the device and have it installed:

- Network share
- Flash media
- ActiveSync

It is also possible to deploy the application to a desktop/device partnership. In doing so, the application becomes a Control Panel Add/Remove Programs item on the desktop partner that manages transport and registration to the device partner by using an ActiveSync connection. The complications of building, tracking, and placing the CAB file(s); tracking availability of the Compact Framework on the device; and other deployment issues are much easier to manage using deployment third-party deployment tools, as we will explore in detail later in this chapter.

Network Share

A network share can be used to provide a central location to deploy mobile applications. Using File Explorer on the mobile device, users connect to the network share and activate the mobile application CAB setup file. This method is best suited for situations in which the mobile devices are network connected and users are comfortable using mobile File Explorer.

Flash Media

Flash media is a convenient mechanism to move files—in this case, CAB files—to a mobile device. Most Pocket PC devices are capable of reading one or more flash media types, typically SD/MMC, CF I/II, or both. You can load the application CAB file onto media cards (see Figure 3.3) at a development or duplicating station.

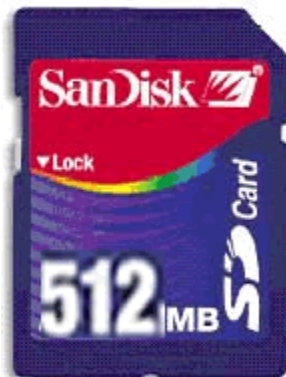


Figure 3.3: A flash media card.

The cards are then distributed to end users and inserted into the target device. Once inserted, the storage card appears as a directory folder in the device file system (see Figure 3.4). The user opens the folder and selects the application CAB file to activate the installation.



Figure 3.4: The card appears to the end user as a directory folder in the device's file system.

You can eliminate the hassles of dealing with the complexity of instructing users about how to find the Storage Card folder and the application CAB file by using the *autorun* feature for storage cards. By creating an *autorun.exe* file and placing it along with the application CAB file on the flash media, Pocket PC will automatically find and load *autorun.exe* when the media card is inserted into the device. You can use the Pocket PC SDK to help create *autorun.exe* manually. However, it is much easier to use a sophisticated third-party deployment tool to create an *autorun* setup for storage cards.

ActiveSync

ActiveSync is the desktop-to-device partner tool used to establish a tethered connection between a mobile device and a desktop machine. Typically implemented using a USB wired connection and a device cradle, ActiveSync connections can also use wireless Ethernet connections from the device.

You can use the ActiveSync desktop application, which Figure 3.5 shows, as a quick and dirty method to deploy a mobile application CAB file to a device. Simply use the Explore toolbar icon to open a window to the device file system and drop the CAB file in the device.

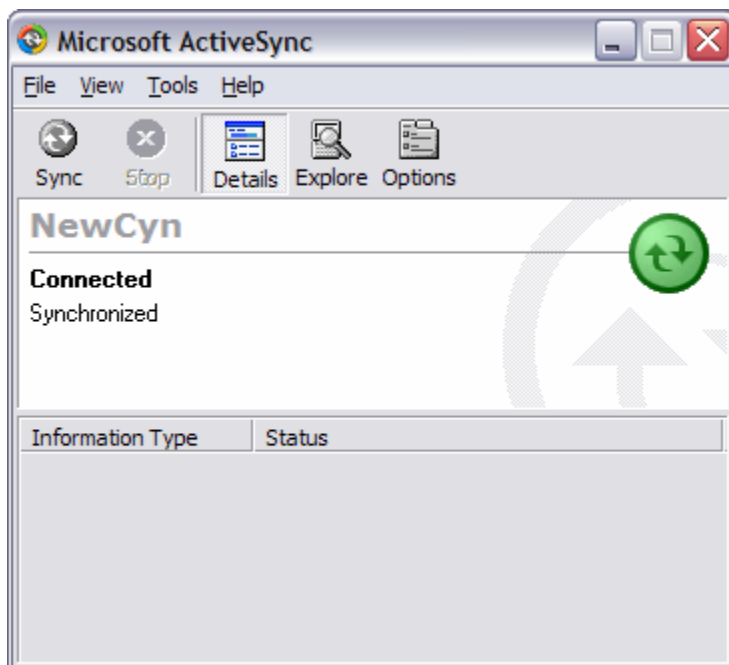


Figure 3.5: Using ActiveSync as a deployment method.

ActiveSync also offers the more sophisticated Windows CE Application Manager tool. It is essentially an Add/Remove remote Control panel for the mobile device that you can launch from the ActiveSync Tools menu, as Figure 3.6 shows.

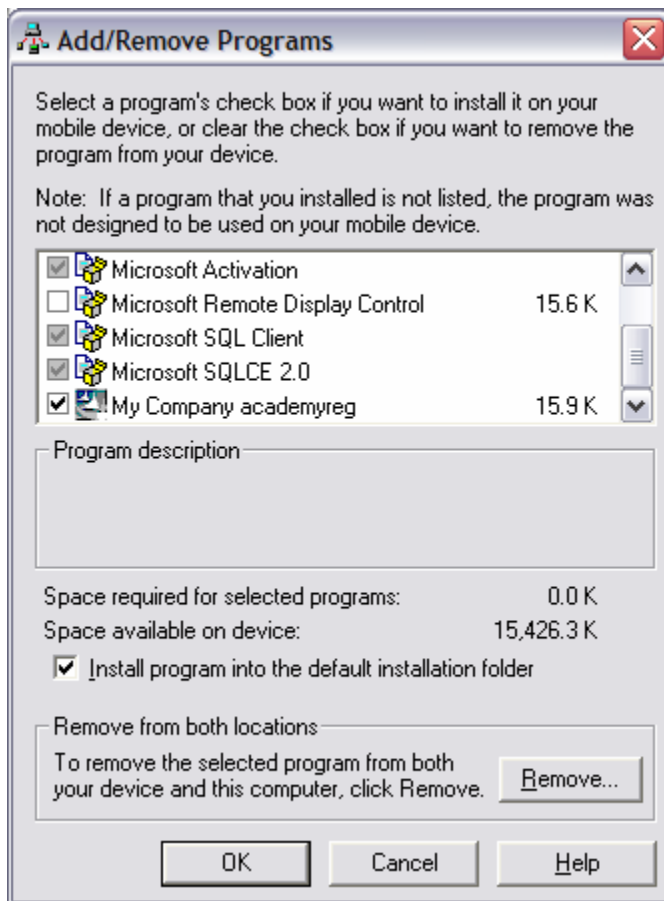


Figure 3.6: The ActiveSync Windows CE Application Manager tool.

Application Manager uses an .ini file to identify a mobile application and one or more CAB files that need to be copied to the device for installation. Typically, with the Application Manager .ini file, a desktop setup package is created that contains the files for the mobile application. When the package is installed on the desktop machine, Application Manager runs and processes the .ini file. If a connection is in place with the mobile device at the time the Application Manager runs, the tool copies files to the device and remotely activates setup on the device. If a connection is not in place, Application Manager registers the setup, and the next time ActiveSync establishes a connection, the deployment takes place. You can use third-party deployment tools to streamline and automate this process.

Using the Compact Framework GAC

As we explored in Chapter 2, assemblies that are shared by multiple applications must be placed in the GAC. The Compact Framework has a GACUTIL (cgacutil.exe) but it is not used in exactly the same way that its bigger cousin is used. To place an assembly in the Compact Framework GAC, first strong name it, then place it in a folder on the device. Next, place a text file in the \Windows directory that has a .gac extension and contains the path and filename of the .dll file containing the assembly. When the application using the assembly first runs, the assembly is moved to the GAC. The text file must remain in the \Windows directory; if it is removed, the assembly will be removed from the GAC the next time the application runs.

A log file is kept in cgacutil, GACLOG.TXT, in the root folder of the device. The Compact Framework GAC is not a folder named “assembly” like the regular .NET Framework GAC; rather, the assemblies are renamed with a GAC_ prefix and placed in the \Windows directory. Figure 3.7 shows a sample of the assemblies that make up the Compact Framework.

Name	Size
GAC_Microsoft.VisualBasic_v7_0_5000_0_neutral_1.dll	136KB
GAC_Microsoft.WindowsCE.Forms_v1_0_5000_0_neutral_1.dll	10.5KB
GAC_mscorlib_v1_0_5000_0_neutral_1.dll	382KB
GAC_System.Data_v1_0_5000_0_neutral_1.dll	393KB
GAC_System.Drawing_v1_0_5000_0_neutral_1.dll	37.5KB
GAC_System.Net.IrDA_v1_0_5000_0_neutral_1.dll	11.0KB
GAC_System.SR_v1_0_5000_0_neutral_1.dll	91.0KB
GAC_System.Web.Services_v1_0_5000_0_neutral_1.dll	94.0KB
GAC_System.Windows.Forms.DataGrid_v1_0_5000_0_neutral_1.dll	38.0KB
GAC_System.Windows.Forms_v1_0_5000_0_neutral_1.dll	136KB
GAC_System.Xml_v1_0_5000_0_neutral_1.dll	196KB
GAC_System_v1_0_5000_0_neutral_1.dll	249KB

Figure 3.7: A sampling of the assemblies that comprise the Compact Framework.

The corresponding Microsoft .NET Compact Framework 1.0.GAC file contains:

```

\Program Files\.NET Compact Framework\mscorlib.dll
\Program Files\.NET Compact Framework\System.dll
\Program Files\.NET Compact Framework\System.XML.dll
\Program Files\.NET Compact Framework\System.Drawing.dll
\Program Files\.NET Compact Framework\System.Windows.Forms.dll
\Program Files\.NET Compact
Framework\System.Windows.Forms.DataGrid.dll
\Program Files\.NET Compact Framework\System.Net.IrDA.dll

```

If you look, you will find that the directory \Program Files\.NET Compact Framework\ is empty. The reason is that cgacutil has renamed the files and moved them to the \Windows directory.

Device Setup Project

As we've explored, there are several complex parts to creating a mobile application deployment using the various deployment methods. Creating autorun.exe for flash media deployments, packaging the correct versions of the .NET Compact Framework, creating the desktop package and Application Manager .ini file for ActiveSync deployments, and so on are all much easier to deal with if you use a powerful third-party deployment tool that has been designed to handle mobile application deployment.

Third-party tools usually provide a method for creating mobile application installations for direct deployment to a device via flash media and for ActiveSync desktop-device partnership deployments. Deployments using the network share method are based on the application CAB file that you can build using either Visual Studio or a third-party tool. Let's walk through a device setup project using a third-party tool.

Direct Deployment via Flash Media

The first step is usually to enter identification and description information for the mobile application, as Figure 3.8 shows. All .NET Compact Framework mobile applications will use the CE project type, and the company name information is usually used to identify the application in the Pocket PC Remove Programs removal tool.

Figure 3.8: Providing information about the mobile application using InstallShield's DevStudio tool.

You also need to specify the destination folder default, which is the application name folder inside a company name folder (see Figure 3.9).

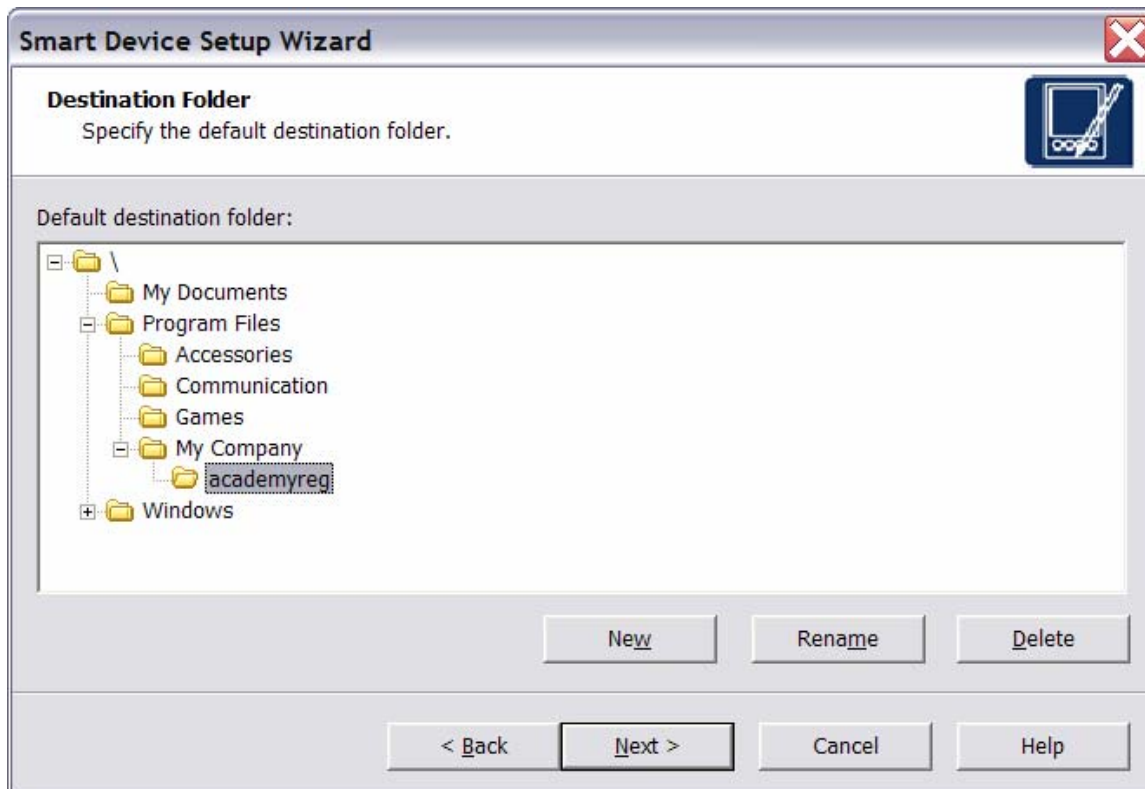


Figure 3.9: Identifying the mobile application's default destination folder.

As Figure 3.10 shows, the next step is to specify which platform and hardware support to include in the setup package. As the figure shows, platforms that the .NET Compact Framework doesn't support are listed. The reason is that some third-party tools are also used for non-managed code deployment. If you are deploying a .NET shared assembly to the GAC, you can select that option. Remember that a .NET assembly contains MSIL managed code that is platform/processor independent.

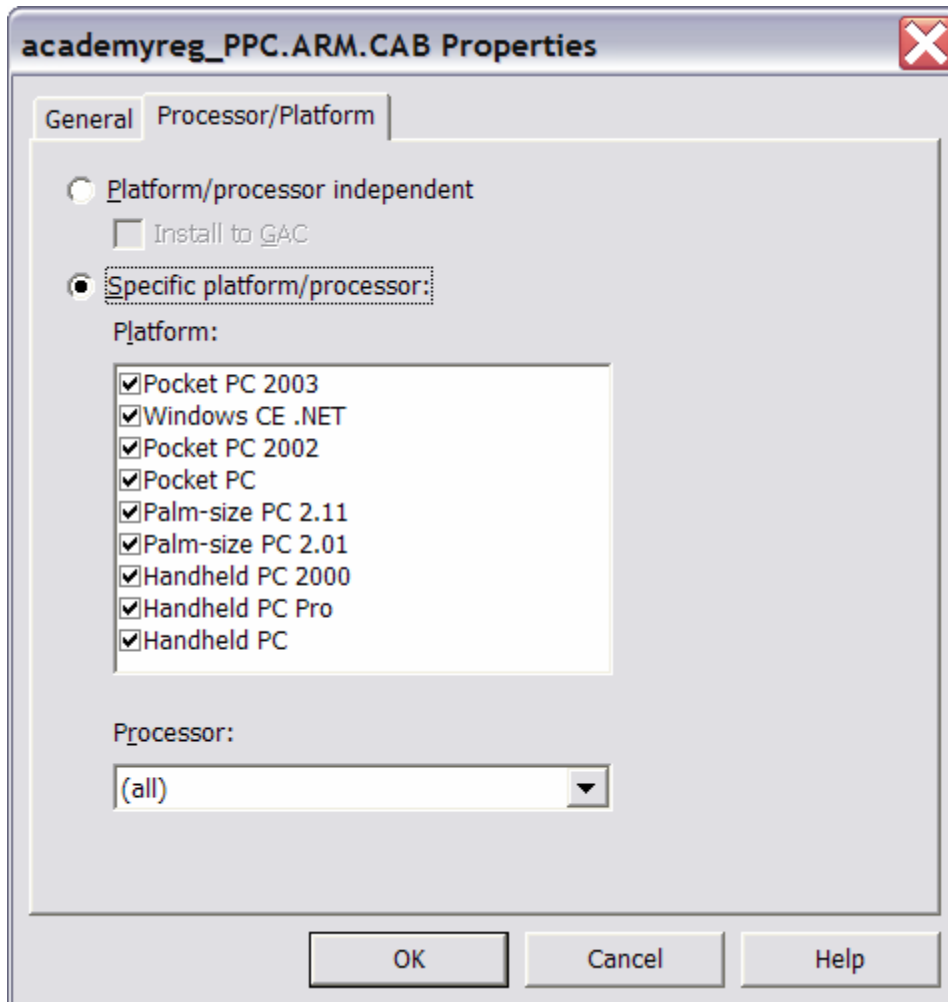


Figure 3.10: Specifying which platform and hardware support to include in the setup package.

Finally, you are presented with the option to include the .NET Compact Framework and SQL Server CE support and to generate autorun.exe for automatic installation using flash media (see Figure 3.11).

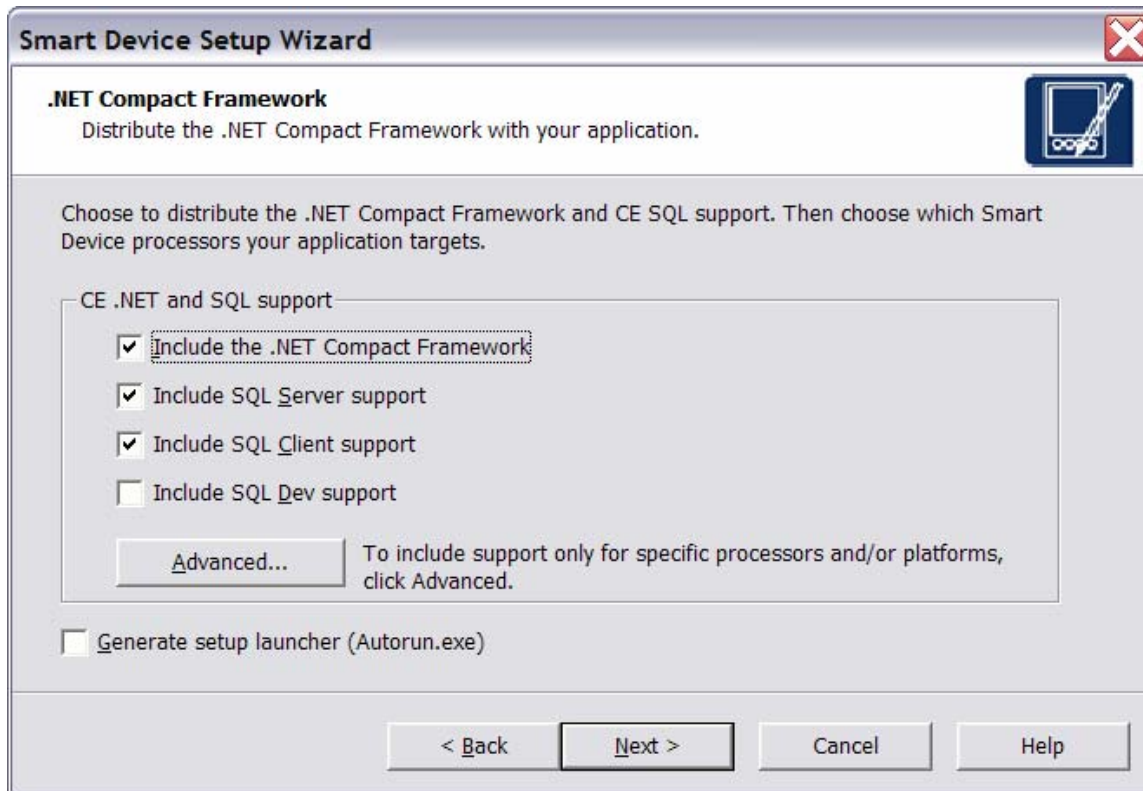


Figure 3.11: Selecting whether to include the .NET Compact Framework and SQL Server CE support and whether to generate autorun.exe for automatic installation using flash media.

Once the third-party tool has completed the creation of the setup project, you can use the IDE to build the project. You simply copy the build to a flash media storage card and insert the card in the mobile device. The autorun feature will execute automatically when the card is inserted and will install the application along with the Compact Framework and SQL Server CE support if they were included in the package. If the same version of the Compact Framework is already installed on the device, the user is given the choice to reinstall the Compact Framework or skip that part of the setup. Version 1.0 is the only version of the Compact Framework available currently. As new versions are released—version 1.1, for example—side-by-side installation and versioning—as we saw with the full Framework in Chapter 2—will come into play.

Deployment via Windows CE Object

You can also use a third-party tool when you want to create a desktop-device partner installation that will use ActiveSync Application Manager to load and set up the application on the device, as Figure 3.12 shows.

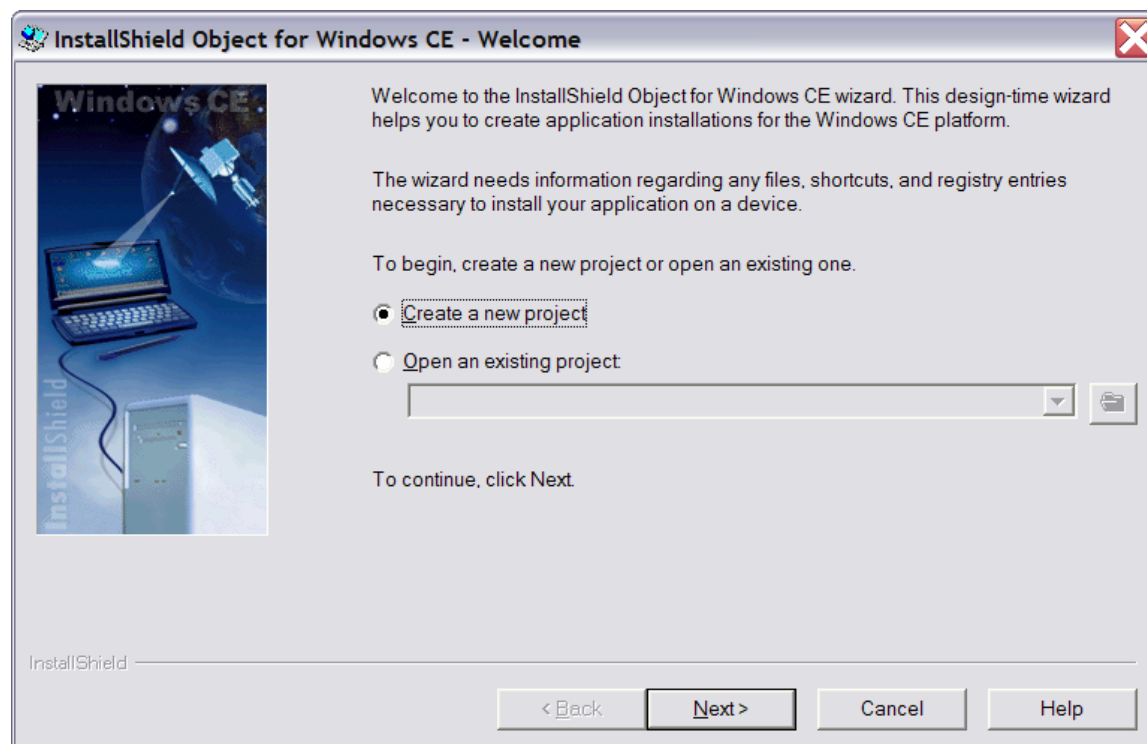


Figure 3.12: Creating a desktop-device partner installation that will use ActiveSync Application Manager.

As with the direct deployment creation process, this process requires you to enter the identity information for the application, as Figure 3.13 shows.

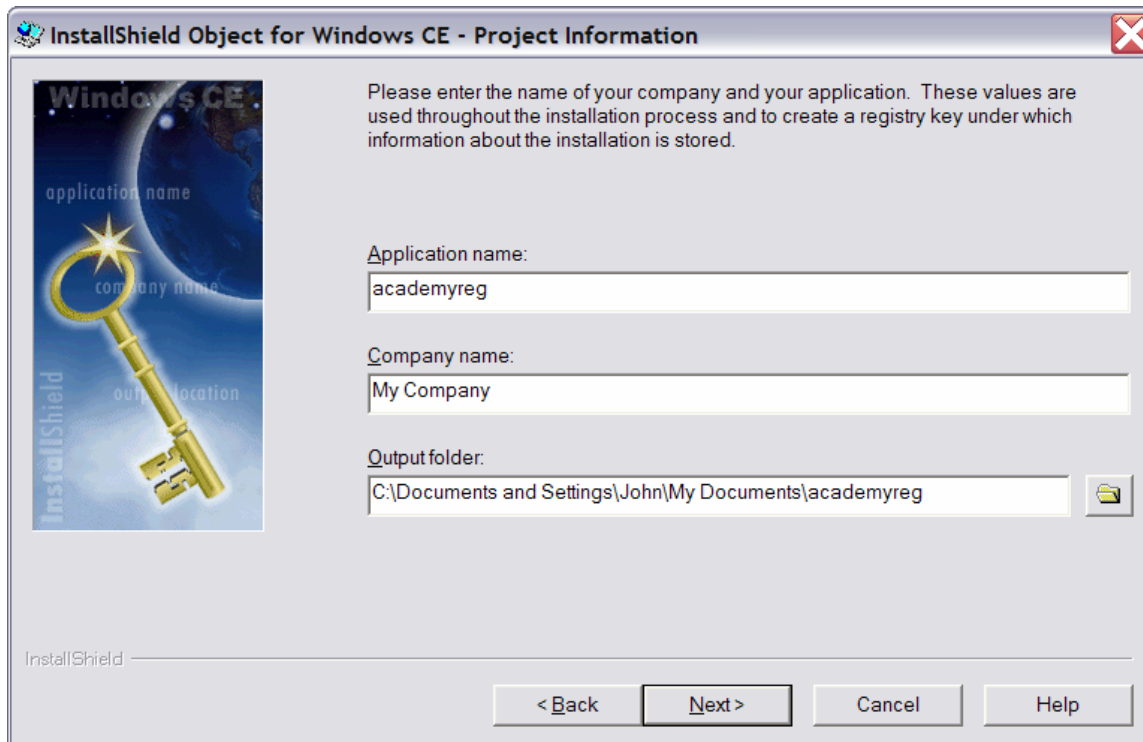


Figure 3.13: Proving identity information about the application.

Next, you will need to locate the application source file and destination folder on the device. You also need to select the platform and processor hardware that you want to support on the device (see Figure 3.14).

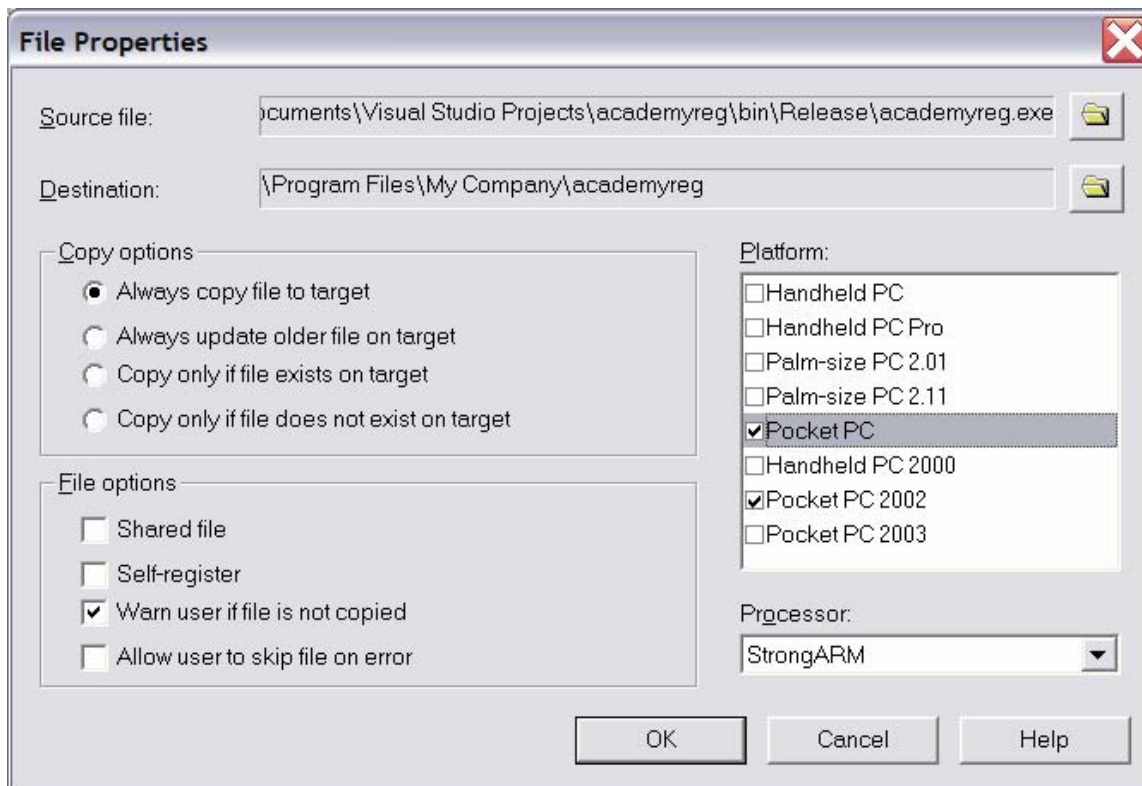


Figure 3.14: Specifying the source and destination files.

Once the third-party tool builds the project, the setup is distributed and run on a desktop machine that has a device partnership; the desktop install will use Application Manager to install the mobile application to the device. As you can see, using a third-party tool is much quicker and easier than manually writing an .ini file for Application Manager.

Summary

Mobile applications are much easier to develop using Microsoft .NET and the .NET Compact Framework. However, deploying the applications once they are developed continues to be a complex and challenging undertaking—multiple deployment scenarios, standalone devices versus desktop-device partnerships, .NET requirements on distributing the Compact Framework, and setting up shared assemblies all contribute to the challenge. You can ease this task by using powerful third-party deployment tools, which make the deployment process quick easy to complete.