realtimepublishers.com™

The Definitive Guide™ To

Windows Installer Technology

for System Administrators

Darwin Sanoy and
Jeremy Moskowitz

Introduction to Realtimepublishers

by Sean Daily, Series Editor

The book you are about to enjoy represents an entirely new modality of publishing and a major first in the industry. The founding concept behind Realtimepublishers.com is the idea of providing readers with high-quality books about today's most critical technology topics—at no cost to the reader. Although this feat may sound difficult to achieve, it is made possible through the vision and generosity of a corporate sponsor who agrees 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 publications does not in any way diminish their quality. Without reservation, I can tell you that the book that you're now reading 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. The Realtimepublishers publishing model also provides other significant benefits. For example, the electronic nature of this book makes activities such as chapter updates and additions or the release of a new edition possible in a far shorter timeframe than is the case with conventional 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 our books are by no means paid advertisements for the sponsor. Realtimepublishers is an independent publishing company and maintains, by written agreement with the sponsor, 100 percent editorial control over the content of our titles. It is my opinion that this system of content delivery not only is of immeasurable value to readers but also will hold a significant place in the future of publishing.

As the founder of Realtimepublishers, my *raison d'être* is to create "dream team" projects—that is, to locate and work only with the industry's leading authors and sponsors, and publish books that help readers 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 send an email to feedback@realtimepublishers.com, leave feedback on our Web site at http://www.realtimepublishers.com, or call us at 800-509-0532 ext. 110.

Thanks for reading, and enjoy!

Sean Daily
Founder & Series Editor
Realtimepublishers.com, Inc.

Copyright Statement

Chapter 1: Meet Windows Installer: Introduction, Features, and Benefits

by Jeremy Moskowitz

You're an administrator. You're the person who comes in early to reboot the servers on Sunday. You're the person who stays late to ensure that the backup job has really kicked off successfully. You're setting up servers and hauling around the occasional desktop, and you're the lucky one who all-too-frequently has the honor of losing weekends due to a poorly planned "move, add, change." We created this book for you.

If you're familiar with the technology and history behind Windows Installer, you know that historically, discussions about this technology have had a bit of a developer slant. However, there is another side to these discussions, and the administrator's side to the Windows Installer story is an exciting one that might get you a weekend or two back.

We're going to try hard in this book to show you why Windows Installer is useful for you—the administrator. Throughout the book, we will be getting into some of the meaty internals of Windows Installer. If you have a little bit of a developer background, that's great; but those who do not will be just as comfortable and find much useful information. With that in mind, let's start our journey.

Defining the Need for Windows Installer

Before we dive into an introduction to Windows Installer and an exploration of how it works, let's define the need for this technology. Why was it developed and what does it offer that will benefit you?

Saved Time and Effort Through Automated Installs

How are you installing your software today? If you're like many administrators, you're still tracking down the installation CD-ROM media (or you've made the installation available on a network source), running from machine to machine, and shooing the user aside for 30 to 60 minutes to install and test the application. This method works just fine for about 10 machines but quickly falls apart once the number of users and applications starts to increase. In the past, additional manpower has been brought in to shoulder the load—more people are hired to slog from computer to computer and load applications. Although hiring more people can ease the load, you and the other administrators must still answer, for each installation, the same 10 questions—such as the location of the program files, the default location of the data files, and which portions of the application should be loaded.

In addition to being a "poor administrative experience," this tack is expensive. Imagine the figure you would come up with if you had to calculate all the time that you and your staff spent ensuring that each desktop had exactly the same hand-crafted software installation settings. Think of all the other stuff you could be doing if you and your team weren't trotting to each desktop.

Enter Windows Installer technology. After we've covered all the pieces of this technology—from the Windows Installer basics to the MSI repackaging process to the actual deployment—you'll have gained the knowledge, skills, and tools necessary to save yourself and your staff tons of time that can then be devoted to more productive endeavors.

> Many IT departments try some form of automated software distribution. Oftentimes, the software distribution job is seen as "secondary work" and isn't given a dedicated person. When the new implementation starts destabilizing as a result of a lack of dedicated personnel, IT managers become frustrated and determine that it's best to just throw in the towel rather than continue throwing money at a new project that already appears doomed. The result is that the project is completely scrapped and bad feelings are felt toward those who wanted to give it a shot in the first place. We'll be talking about how to prevent this chain of events through an exploration of distribution methods and helpful tips in Chapter 6.

Application and Operating System Stability

How often have you loaded a desktop or server system, then returned months later to load yet another application only to discover the bitter taste of application incompatibility? As Figure 1.1 shows, applications can be destabilized by other applications that load on top of required components.

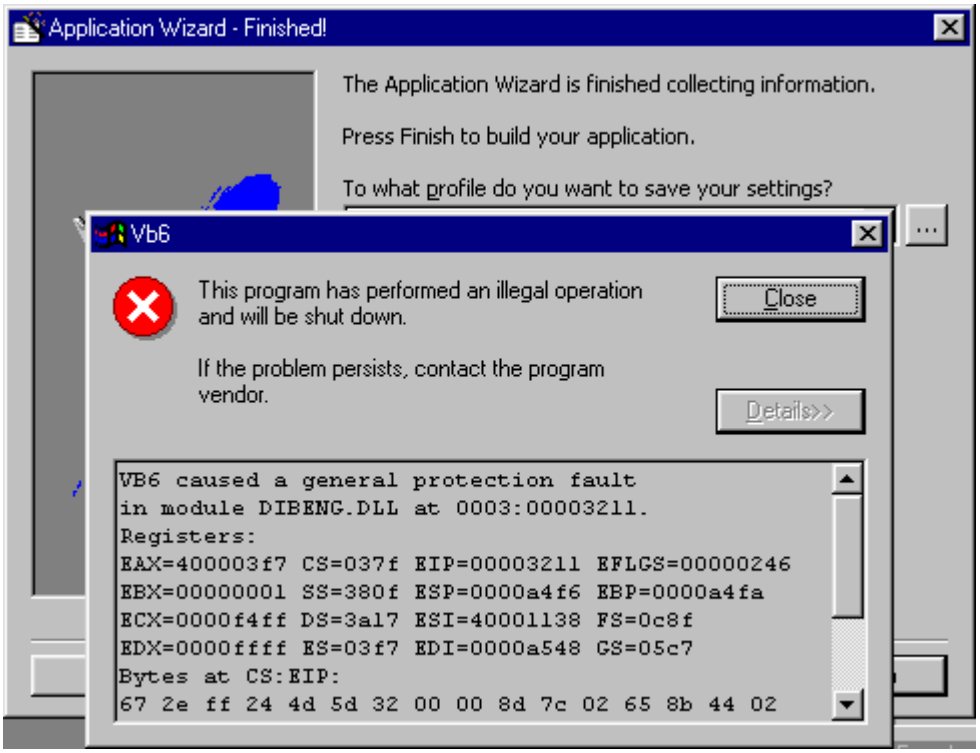

Figure 1.1: Evidence of application incompatibility.

This incompatibility can occur because the technology used to package (or re-package) the application or manipulate its components for installation was simply not aware of other components already loaded on the system. Likewise, the operating system (OS) components can find themselves in harm's way.

Figures 1.1 appears here with permission from David Joffe and his Microsoft Crash Gallery Web site at http://www.scorpioncity.com/mscrash.shtml.

In Windows ME, Windows 2000 (Win2K), and Windows XP and later, new strides have been taken to ensure that, at least, the OS has an increased layer of protection. Read about it in the following sidebar "Windows File Protection."

Windows File Protection

The Windows OS has had a reputation for crashing, but a new feature in Windows 98, Win2K, and Windows XP tries to put an end to this problem. The new feature is called Windows File Protection (WFP.) The goal of WFP is to ensure that if a critical system file, such as a .DLL or .EXE, is compromised, a secret copy can be brought back from a hidden "cache" on the disk. This copy ensures that misbehaving applications cannot take over the OS. Take a look into Windows' secret directory called *dllcache* under the %systemroot%\system32 directory, usually C:\windows or C:\winnt (see Figure 1.2).

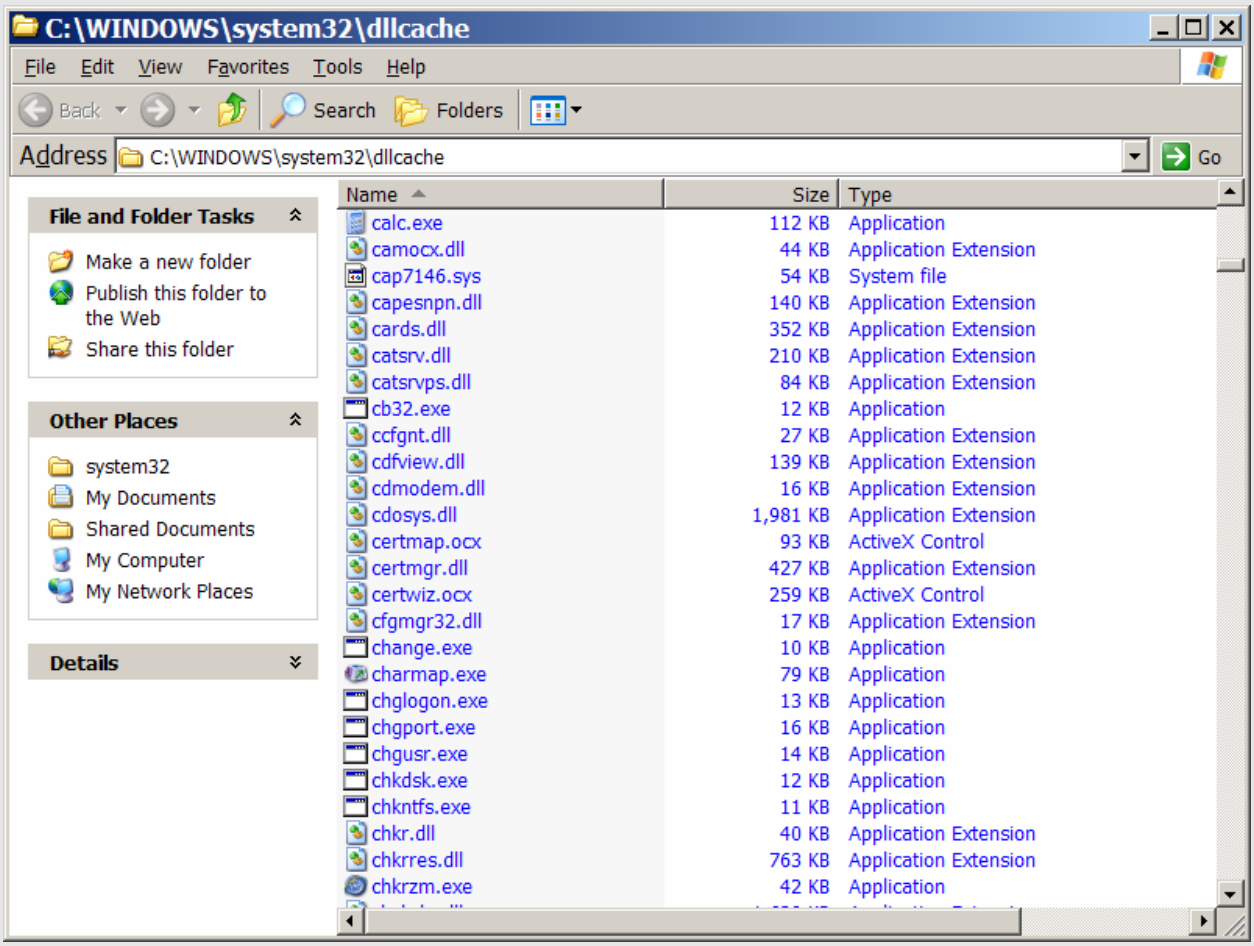

Figure 1.2: A list of some of the files in dllcache.

With WFP, no applications (except hotfixes or service packs) can overwrite these files—either here or in the files' actual locations (normally Windows or system32). Go ahead and try to delete a file listed in dllcache, such as calc.exe, from its actual location. It will come right back from the cache!

Figure 1.3 shows a famous picture taken at London Heathrow Airport by Alan Cox. As this picture shows, application incompatibilities can lead to an unstable system with disastrous results. Although this picture illustrates a humorous example of the result of an application incompatibility, your goal should be to provide as stable of a system as possible.

The picture in Figure 1.3 was reproduced with permission, courtesy of Alan Cox.

Figure 1.3: London Heathrow Airport's system has a problem.

But how does Windows Installer relate to system stability? Before Windows Installer, packaged applications haven't had the ability to sense what was going on around them, which resulted in application and OS stability problems when a new application was installed on the system. Thus, Microsoft needed to step in and release a technology that was vendor-neutral and helped increase OS and application stability. That technology is called Windows Installer. In the following sections, we'll explore how Windows Installer improves stability as well as other benefits this technology provides.

The Benefits of Windows Installer and MSI

Windows Installer works as a result of the marriage between Windows Installer and a new package type called the Microsoft Installer (MSI) file type, which I'll discuss later in this chapter, and because of the behind-the-scenes action that takes place when Windows Installer encounters the new package type. Before we get too far along and talk about the technology behind Windows Installer and MSI and the stuff you can do with that technology, let me introduce its basic benefits. Table 1.1 provides just a few of the myriad benefits that Windows Installer and the corresponding MSI technology bring to the table.

Some documentation refers to MSI packages as Medium Scale Integration files rather than Microsoft Installer files.

Windows Installer and MSI Benefit	Description
Application is installed via OS service	On Win2K and Windows XP and later, the application is installed in an administrative context. I'll explore this technology later in this chapter.
MSI provides a standard package format	A new format, the MSI package and its .MSI extension, is the new standard to interface with the Windows Installer technology.
Transactional install and rollback	Windows Installer packages can be made to either fully install the way the author intended, or if there is a failure during the install (for example, because you run out of disk space part-way through), the failed install can simply undo all the changes it has made up to that point in the installation to bring the system back to its previous state.
Self-healing (or self-repair) of corrupt or deleted critical files	As we'll explore in detail later, certain files can be *keyed* for detection of failure. If a critical file (a .DLL or .EXE file, for example) that is part of the distribution is corrupt or is deleted, the user can be prompted to repair the installation by presenting the original .MSI distribution. Additionally, if the installation media is available (for example, on a network share), the repair simply happens automatically.
Served installs	Because MSI files can be housed in a share point and delivered via a server, you can keep your installation files all in one place or move them around—closer to your users if necessary.
Install on demand	Windows Installer–deployed applications can be offered to clients at any time. Once offered, their installation can be triggered when a user clicks a corresponding registered extension. For instance, clicking a .DOC file prompts the installation of Word for Windows. Once chosen, the application is downloaded in a Just in Time (JIT—see the following entry) fashion.
JIT installation	After an application is offered to a user, it isn't *actually* installed. Instead, the application's icon appear, and when the user decides to run the application, it is installed from the media (or downloaded from the server) in a JIT fashion, and presents itself as ready to the user in a matter of moments.
Packages can utilize transform files	An application's package can be developed such that a *base* or *administrative* install is prepared for general distribution. A *transform* file can overlay the base, letting you customize specific installations. I'll discuss this benefit later in this chapter.

Windows Installer and MSI Benefit	Description
Packages can utilize patch files	After a package is on the machine, you might need to fix the source files if a bug is found or an update is needed. Windows Installer defines a clear path to rectify these problems. I'll discuss this benefit later in this chapter.
State management	In the past, it's been difficult to know whether an application is installed on a machine. You would have to query for a .DLL with a specific version number or determine whether an .EXE file with a specific name was present. Windows Installer provides an application programming interface (API) that lets programmers and administrators see whether a specific application is installed on a machine.
Administrative privileges are not required for installations	Previously, you might have found that applications needed to be loaded through the local administrator account. Windows Installer eliminates this requirement.
Scriptable API	With a little elbow grease, you could whip together a VBScript to help you with your MSI file manipulations. The API to manipulate MSI files is so powerful that it can create packages, validate packages, update packages, trigger installs and uninstalls, examine the MSI repository data on computers, and perform some custom actions. If you have to repeat the same function, scripting is the way to go.
Packages can be managed using the MSIEXEC command-line tool	The command-line tool MSIEXEC is a very powerful tool that lets you manage your MSI applications. We'll be exploring some features of the MSIEXEC command-line tool a bit later in this chapter.

Table 1.1: Benefits of Windows Installer and MSI technology.

In the rest of the chapter and in the upcoming chapters, we'll explore these benefits in detail. A roadmap of the remaining chapters can be found at the end of this chapter.

Your First Windows Installer Encounter

The Windows Installer technology has been around for a while, so it's likely that you've already had some experience with the technology, perhaps without even being aware of it. For instance, you might have casually encountered it when installing the first Windows Installer–ready application, Office 2000, as Figure 1.4 shows.

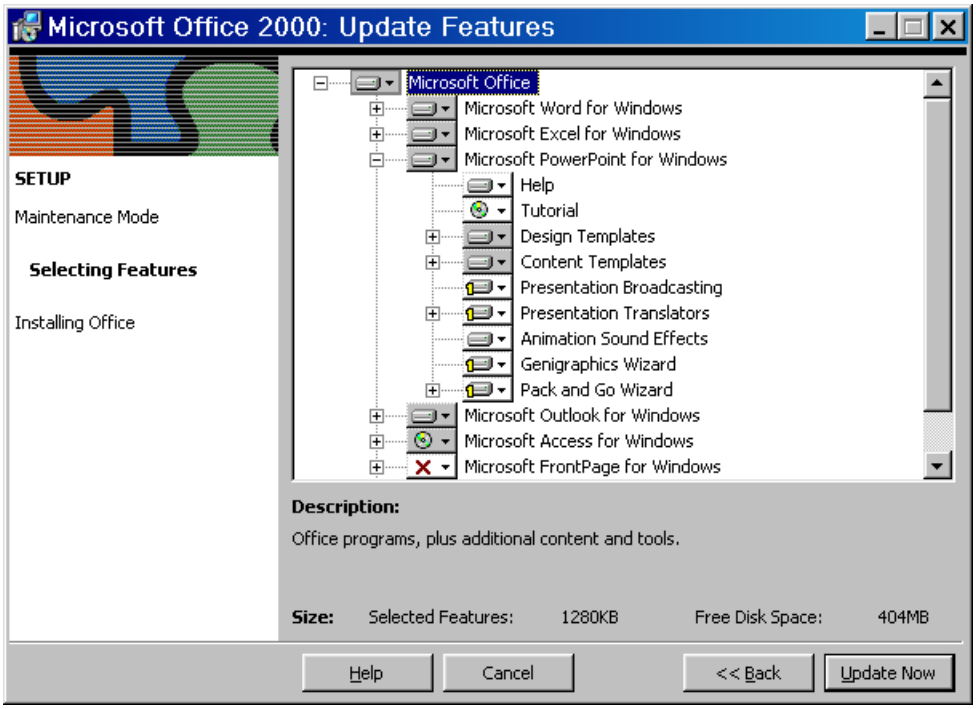

Figure 1.4: Office 2000 was the first application to ship as a Windows Installer–ready application.

Take a close look at the icons within the graphic. First, you'll notice that a product's installation is viewed in a hierarchical fashion. At the top of the hierarchy, we can see Windows Installer telling us which *product* we are installing. In this case, the product is Microsoft Office. Below the product, is a subset called *features*, as Figure 1.5 details.

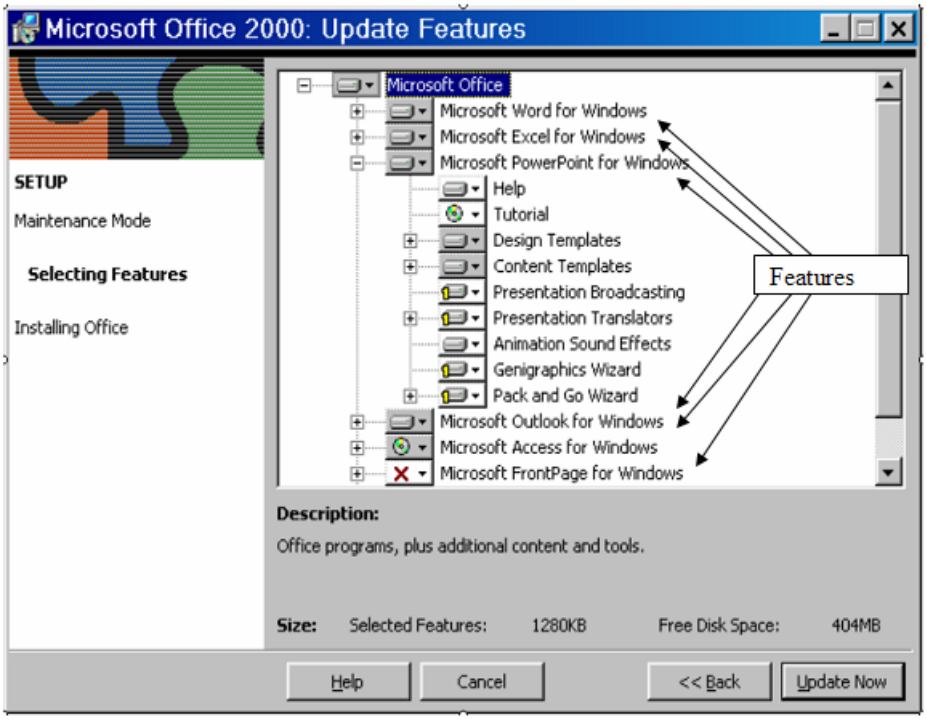

Figure 1.5: Highlighting a product's features.

The features make up the product, and there can be one or more features per product. Each feature in the hierarchy can have *sub-features*, as Figure 1.6 shows. In Office 2000, most of the components are located in the sub-features.

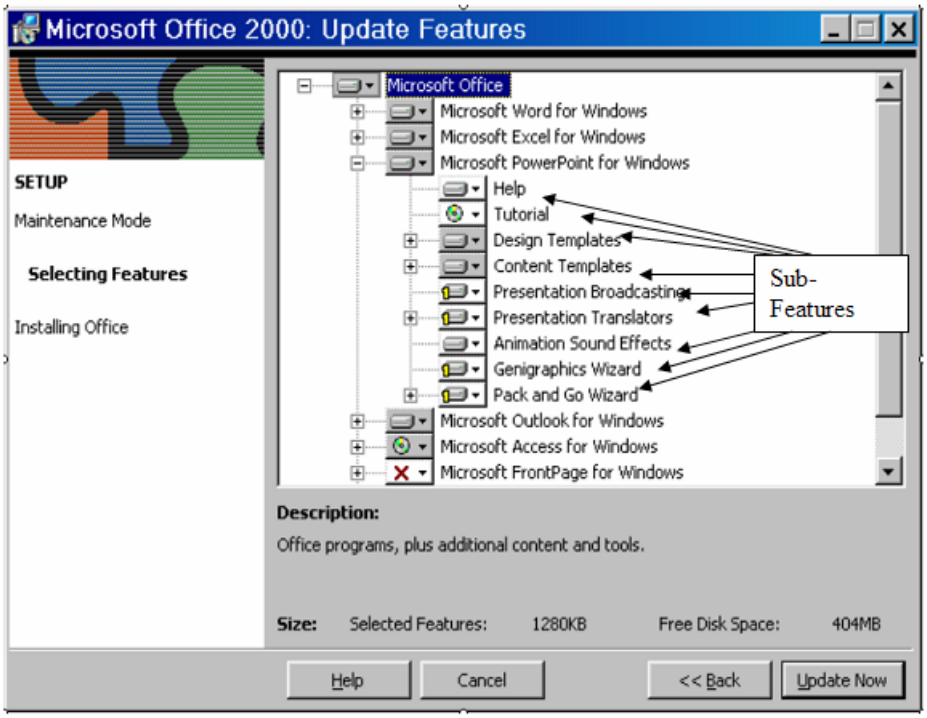

Figure 1.6: Features can have sub-features.

> Occasionally sub-features are also called features, so these terms can be misleading. Common use defines features as the first level below product, and sub-features as the second level below product (as well as all additional levels below the feature level).

As you can see in the previous graphics, four possible installation states exist for any specific subsection of Office 2000:

- Solid gray means that the feature or sub-feature will be installed and available for use as soon as the install is complete.
- White with a yellow "1" icon means that the feature or sub-feature will be installed at first use. When the feature is installed, this sub-feature isn't really installed. When the user tries to use this sub-feature for the first time, it is pulled from the installation media and installed in a JIT fashion. A benefit of this installation state is that it saves space—this feature will only be installed (and consume disk space) if users are going to use it.
- The CD-ROM icon means that the feature or sub-feature will not be installed directly on the hard drive. The feature or sub-feature will be accessible and able to run when the source CD-ROM is present in the drive or through the network if a network connection is available. This installation state is handy, for example, for features that consume a lot of space, such as multimedia files and reference files, that will be used occasionally, if ever. In the previous figures, for example, Microsoft Access has been set to run solely from the installation media—either a CD-ROM drive, network share, or other source.

- The red X icon means that the feature or sub-feature will not be installed and won't be accessible during normal use of the product. If this feature is desired, the installation setup program needs to be re-run and this feature's state needs to be changed to one of the other three states.

Windows Installer Version Numbers

Like most software products, Windows Installer has versions associated with it. There are major revisions (such as 1.0) and minor revisions (such as 1.1). Windows Installer is unique in that it is versioned for each platform. That is, Windows NT, Windows 9x, and so on each has its own Windows Installer version number. However, automatic updates can occur without a user's knowledge, so users in your environment could possibly have different Windows Installer versions.

What Is Your Windows Installer Version Number?

To find out which version of Windows Installer is on your machine, you can simply use Windows Explorer to navigate to the %systemroot% directory (usually \Windows or \Winnt), enter the system32 directory, right-click MSI.DLL, and select Properties. Doing so reveals the window that Figure 1.7 shows.

> ☞ You can also run the command-line tool MSIEXEC from the Start menu Run text box to discover the version number.

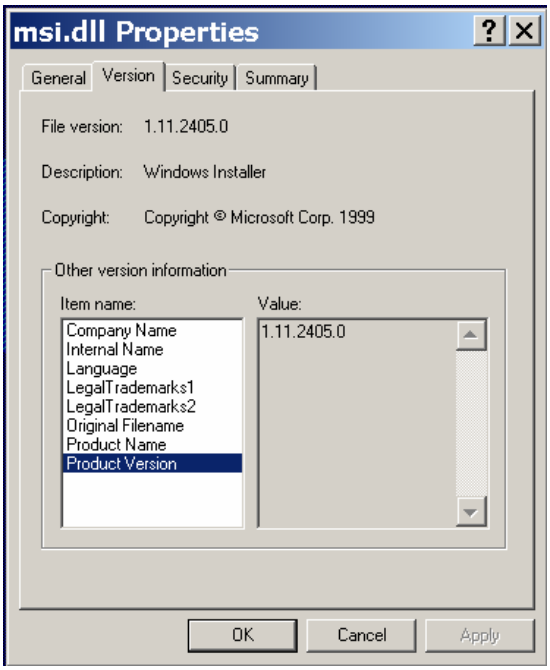

Figure 1.7: Inspecting your Windows Installer version number.

The file version provided on the Version tab of the properties window should match the product version (in Figure 1.7, they both have a value of 1.11.2405.0). The 2405.0 entry is simply the specific build number. The Windows Installer filer version is 1.11. So why would your machine have a different file version?

The Internals of Version Numbers

As one might expect, Windows Installer started with version 1.0. As I previously mentioned, Office 2000 was the first Windows Installer–ready application. Being the first presented a problem: How would the system perform the Office 2000 installation if Windows didn't yet have the Windows Installer *bits* (the executables necessary for Windows Installer to install the program)?

The Office 2000 development team came up with a brilliant little plan: Before actually loading Office, load a little piece of code that loads the bits required to perform a Windows Installer–type install, then perform the rest of the Office 2000 install as a Windows Installer install. (Current applications will attempt to install the latest version of the Windows Installer using a similar method.) So, the bits for Windows Installer 1.0 are included on the Office 2000 CD-ROM and are automatically installed when the setup routine is run.

Let's take a look at the Windows Installer versions from then until now:

- Version 1.10 was the first version included in Win2K directly (build 1.10.1029.0). For other platforms, there was a version made available for download from the Microsoft Web site (build 1.10.1029.1).
- Version 1.11 appeared in Service Pack 1 (SP1) for Win2K (build 1.11.1314.0). Another, later build of Version 1.11 appeared in SP2 (build 1.11.2405.0).
- Version 1.20 appeared in Windows ME (build 1.20.1410.0) and was available for download from the Microsoft Web site (build 1.20.1827.1).
- Version 2.0 and later will be discussed in the next section.

Note the pattern of version numbers. Specifically, versions that end in .0 ship with and are built into the OS. Those that end in .1 are downloads.

Windows Installer Version 2.0

Windows Installer 2.0 is the latest major release for Windows Installer. You might casually encounter the upgrade in a manner similar to the original Office 2000 installation. That is, you might simply double-click to install your latest application acquisition, and you'll be presented with the pronouncement that Figure 1.8 shows.

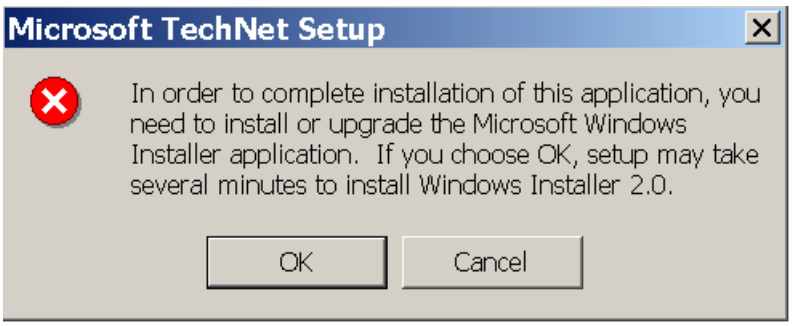

Figure 1.8: The installation of Windows Installer 2.0.

> You must have Windows NT SP6 installed before you are able to install Windows Installer 2.0.

Windows Installer 2.0 provides some new features, which the following list describes. These features make Windows Installer a much more efficient engine.

- Hash-based calculations—Windows Installer 2.0 is a lot smarter than previous versions about recognizing when files need to be replaced—for either a repair, patch, or upgrade to an existing package. Should an application need a repair, upgrade, or patch, Windows Installer 2.0 performs a much faster hash-based calculation (rather than comparing each installed file to the original source installation, which is a really slow process) to determine whether a file needs to be replaced. The added bonus is that you might not need the original source media available unless there is a problem with a file that specifically needs to be replaced. Thus, you can more quickly perform a repair, patch, or an upgrade.
- Delayed reboot—When a new version of the Windows Installer bits becomes available, you're prompted to install the new version (as Figure 1.8 shows). Previous versions of Windows Installer required that you install the new bits and reboot the system before you could progress with the installation of your application. Windows Installer 2.0 lets you delay the reboot (which completes the Windows Installer upgrade) until all your MSI packages are fully installed.

> An upgrade from Windows Installer 1.x to 2.0 requires a reboot, but you can delay the reboot until you're ready. In addition, you must be logged on as a local Administrator to perform the upgrade.

- Improved logging—Windows Installer 2.0 provides better logging in the event log and when files are installed. Each error receives an ID (the error codes for previous versions of Windows Installer all fell within two or three non-unique event IDs), which greatly improves how you can search for and filter Windows Installer events.
- Increased security and multiple user isolation—Previously, an MSI application was installed for one user; however, another user might be able to use that application. Windows Installer 2.0 makes a great effort to ensure that each install is a personal and customized experience so that when Joe sits at Sally's machine, Joe cannot run an application that Sally installed just for Sally.
- Digital signature support—Files can now be digitally signed within an MSI file (as well as .MSP and .MST files) to ensure that the file came from a trusted source.

- 64-bit service with 64-bit Windows—Windows .NET server and Windows XP will each have 64-bit versions that will run on the Itanium II processor. These OSs will be able to take advantage of the new 64-bit Windows Installer services.

> ☞ To download Windows Installer 2.0, go to www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm.

Windows' Relationship to Windows Installer

Let's take a closer look at how Windows Installer and Windows are related. Recall that the Windows Installer bits are built into Win2K and Windows XP; moreover, they run as a service. To see the service, simply right-click My Computer, select Manage, and click the Services entry under Services and Applications. The Windows Installer service is highlighted in Figure 1.9.

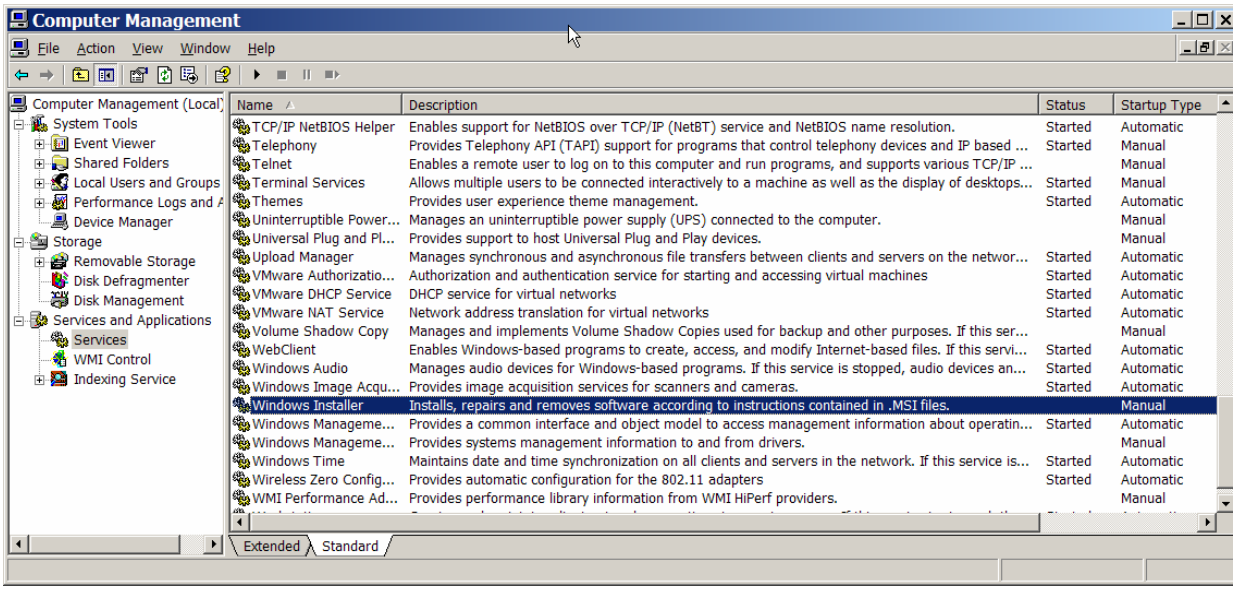

Figure 1.9: Locating the Windows Installer service.

Windows Installer is installed as a service, so it's capable of intercepting shortcuts and file extensions that link to applications and prompt their install from the source media. For instance, you might have Office 2000 loaded, but set to load PowerPoint upon first use (that's the icon with the little '1' we saw in Figure 1.4).

The version that runs on Win2K and Windows XP and later is also capable of receiving and executing instructions when running in an Active Directory (AD) environment. Specifically, Win2K and Windows XP and later can take instructions to load MSI applications via Group Policy. This functionality allows for applications deployed via AD to run in an administrative context—allowing applications to be loaded when administrators want systems protected, and preventing regular users from installing applications they shouldn't. We'll be discussing this method of installation in Chapter 6.

Windows Installer on Downlevel Clients

Services are incapable of running on Windows 9x machines, so you won't be able to perform the aforementioned procedure. You could see the Windows Installer service on NT clients; however, downlevel clients can't receive Group Policy, so they are incapable of receiving instructions from AD telling them to install or upgrade an MSI file.

Although downlevel clients can't receive Group Policy, and hence, are incapable of receiving MSI installation instructions, the clients are still capable of using nearly all the remaining Windows Installer features. For example, recall that files can be keyed for proper operation, and if a required file should get damaged, the application can simply prompt for the original installation source. This capability is present with downlevel clients and is a major win for system stability on older clients.

> ☞ Microsoft has recently provided a very good FAQ about Windows Installer at http://www.microsoft.com/windows2000/community/centers/management/msi_faq.asp.

MSI File Foundations

In this section, we'll explore how an administrator might typically encounter MSI files. Not every MSI application will be interfaced in the same way, but this section will provide you with general exposure to the process.

Setup or MSI?

As we've discussed, you might already be familiar with MSI files because a vendor has delivered some of your applications in the MSI format. Interestingly, though, some vendors, including Microsoft, still include a legacy setup.exe program. This setup.exe program is often simply checking the system for the presence of MSI bits, then calling the corresponding .MSI file to launch. Oftentimes setup.exe is provided as a backward-compatibility measure; that is, those who don't know to click the .MSI file will simply use the setup.exe program, which then calls the .MSI file.

For example, as Figure 1.10 illustrates, the Win2K Support Tools installation comes with both a setup.exe program and a file named 2000RKST.MSI. Clicking either SETUP or 2000RKST will ultimately launch the MSI file, and start the installation.

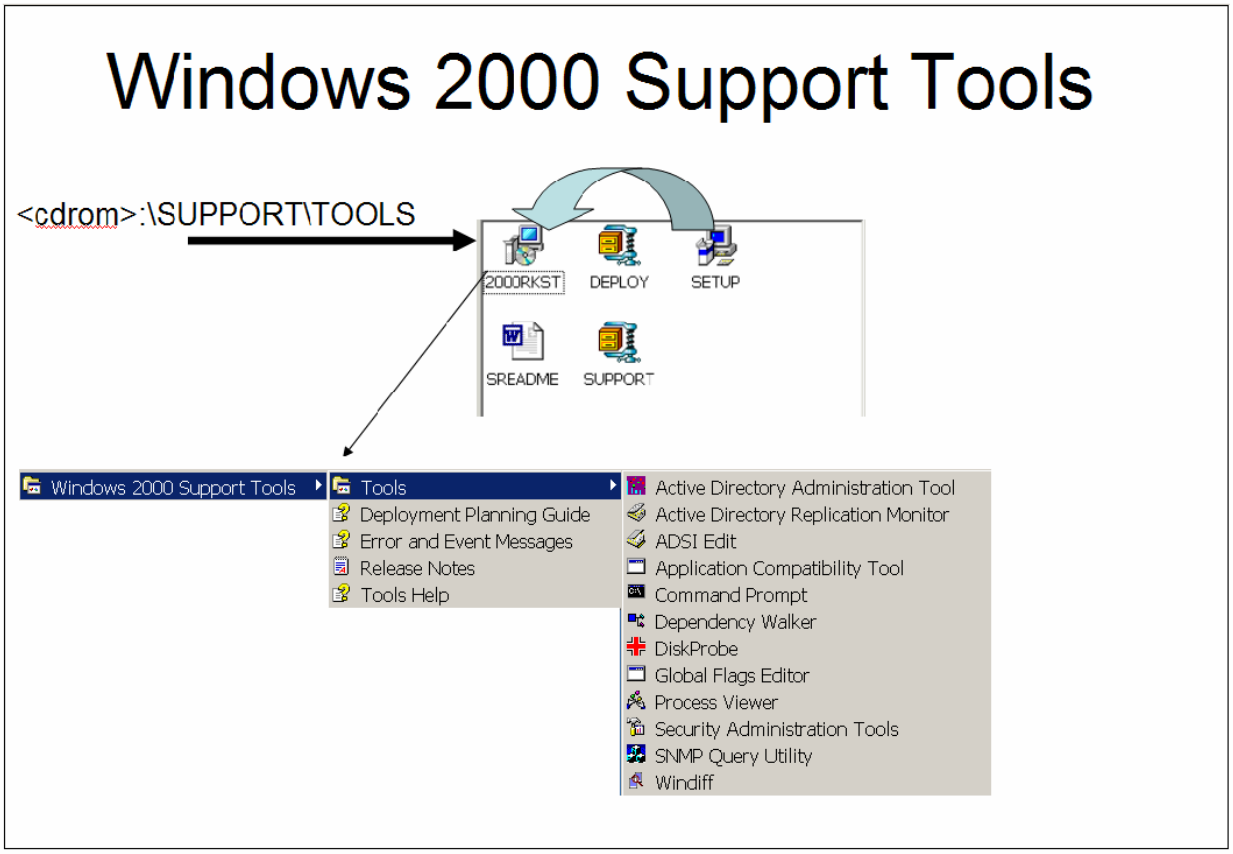

Figure 1.10: The Win2K Support Tools can be launched in either fashion.

Base Installations, Transforms, and Patches

We can build on this foundation of knowledge about MSI and Windows Installer so that you understand how to manage applications using all the new MSI functions. When you receive an application package from a vendor, what is the actual process you're supposed to follow—after you get the software, what do you do?

Base Installations

First, you'll need to understand that the application will come from a vendor and ship as a set of *base* installs. The base installs are the bits that are downloaded from the Web or the bits on the CD-ROM that constitute the original distribution of the software. (In just a moment we'll compare base installations to transforms and patches.)

To make use of the base installation, you might need to *prepare* the installation, creating an *administrative installation*. In an administrative installation, the base installation's files are basically yanked from the packed source (in this case, the MSI file) and placed in the format that the application needs in another, alternative directory structure that is suitable for file sharing. (This technique is similar to the way that Windows 95 and Office 97 rollouts were prepared.) Not all .MSI packages must be prepared as an administrative installation; check your vendor's documentation to be sure.

When an administrative installation is required to manipulate a vendor-supplied package, you'll usually use the built-in Win2K MSIEXEC command to create the administrative installation point. Oftentimes, an administrative installation is performed by running the MSIEXEC utility with the /a switch, as follows:

```
msiexec /a {packagename}.MSI
```

When you do, the familiar Windows Installer window will pop up, showing you that the command is working, as Figure 1.11 shows.

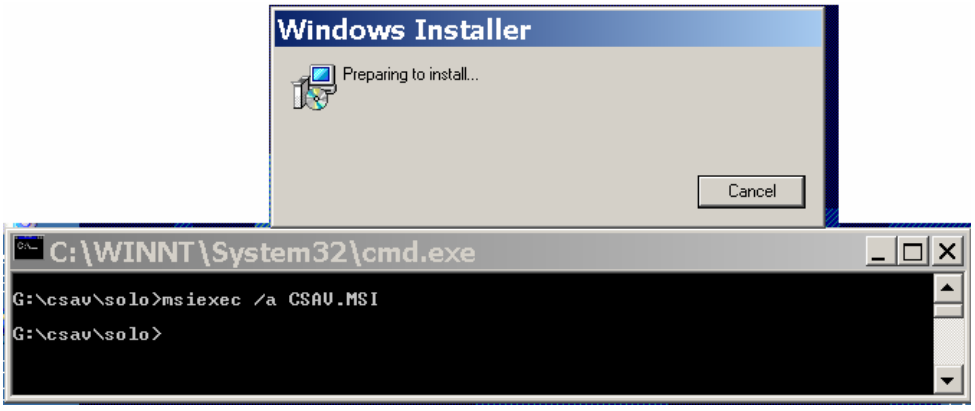

Figure 1.11: Run MSIEXEC to prepare the installation.

After this command has executed, you'll typically get a wizard that asks where you want to place the administrative installation. Simply place the files into a shared folder, and you're most of the way done. Your users could then connect to the administrative point by mapping a drive or via logon script or one of the many alternative methods, and run the installation.

The problem with running the installation in this fashion is that it's simply not customized or tailored for the many users who might want to install and use the software. Indeed, all users who connect back to either the base bits or the administrative installation are usually presented with the default settings as the MSI package is coded. If an end user isn't savvy, he or she could be faced with many potential installation choices, as we saw in Figures 1.5, 1.6, and 1.7. That is where transforms come in.

Transforms

So, although both the base and administrative installations are useful, the installations that they create can simply be too broad. If a client were to double-click the setup's .MSI in the base installation, the client would be prompted to install every default option described in the package, which might not be the ideal installation for the client. You might want to specify that certain users get some options in a package and ensure that others do not, as well as specify the default installation directory, the default *Save as* location, and so on.

To do so, the MSI format allows you to create *transforms*. A transform filters and shapes what your MSI file will look like for a specific user base. A transform file has an .MST extension.

Transforms can be handy in a variety of situations. For instance, you might choose to have a base MSI installation package that loads the DogFoodMaker 5.0 application. However, you might want a customized installation for the sales group that places the default installation point on the D drive, a customized installation for the marketing group that places the default installation point on the E drive, and a customized installation for the nurses that has only one feature.

Vendor-Supplied Transform-Generation Tools

Transforms can be created for a specific MSI file in multiple ways. One way is that a vendor supplies a standalone tool that examines their product's setup .MSI file, allows for user input, then spits out a customized .MST file. Figure 1.12 shows an example of a custom–installation–creation wizard.

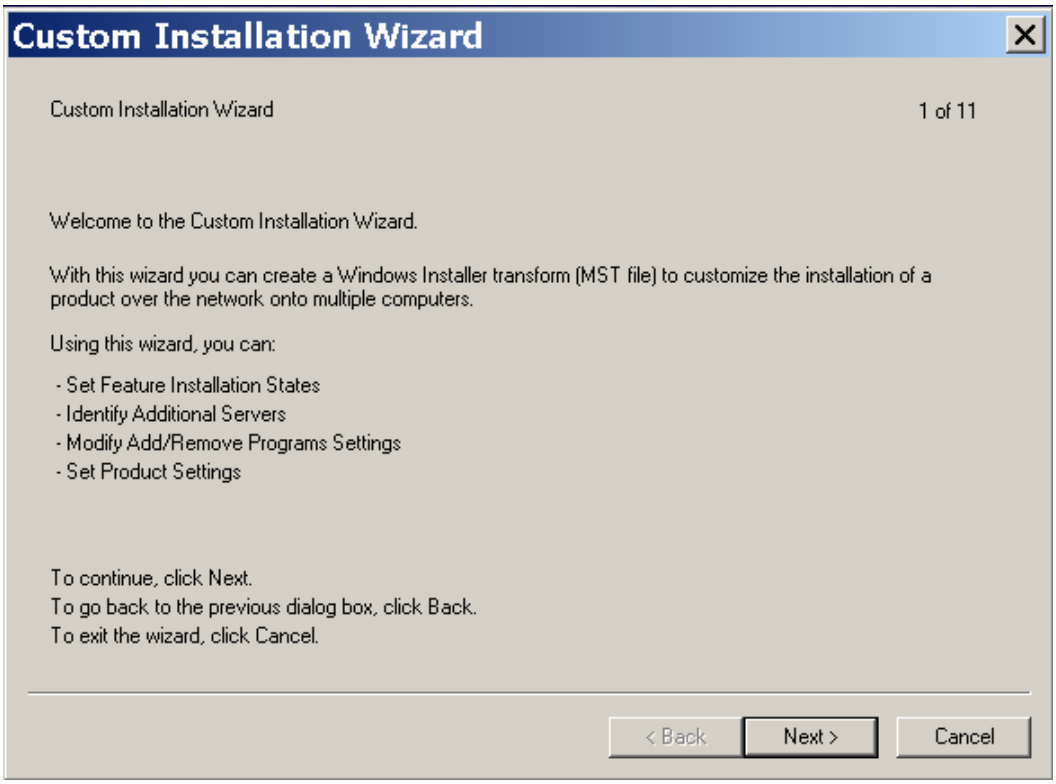

Figure 1.12: A vendor's custom–installation–creation wizard for creating transforms.

Such a tool uses the vendor's MSI file as an initial starting point, then walks you through which settings you can change to customize an installation. After you've chosen your customizations, the tool spits out both an MST file and instructions for its use (as Figure 1.13 shows).

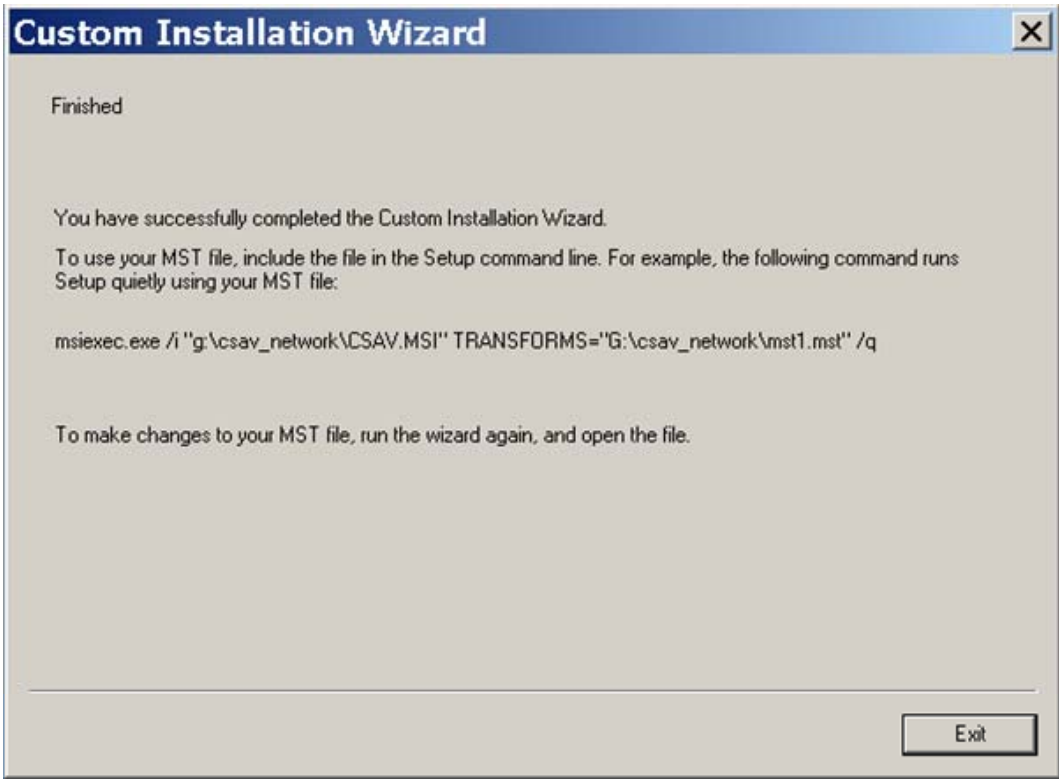

Figure 1.13: The final window of a custom–installation–creation wizard.

Don't run the package just yet. In the following sections, I'll explore third-party transform-generation tools, then discuss what to do with the customized installation file.

> Figure 1.13 shows output that uses a specific drive letter—the G drive. For you to use the command that Figure 1.13 shows, G would actually need to be mapped to the administrative installation. If you don't want to worry about drive mappings, consider substituting using universal naming convention (UNC) pathnames.

Third-Party Transform-Generation Tools

Although many vendors are starting to produce their software as MSI files, a lot of them don't yet offer a standalone tool to generate MST files. If you have applications that fall into this category, you'll need to find a third-party tool that can crack open an existing MSI file and help you generate an associated MST file, such as the InstallTailor tool included in Wise Solutions' Wise Package Studio. As Figure 1.14 shows, you simply point InstallTailor to an existing MSI file (either one that you create or a vendor-supplied MSI file).

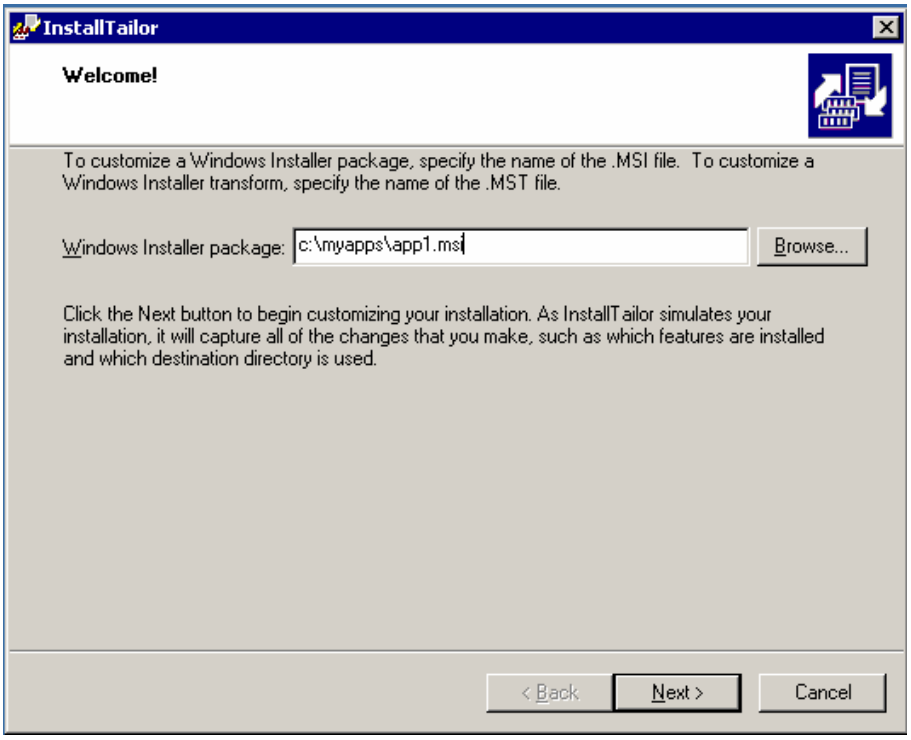

Figure 1.14: Opening an existing MSI file to create an MST file.

You'll then be asked to "simulate" the MSI file's installation. Make your installation choices as if you were actually installing the MSI package. When you're finished, the tool will create the MST file, as Figure 1.15 shows.

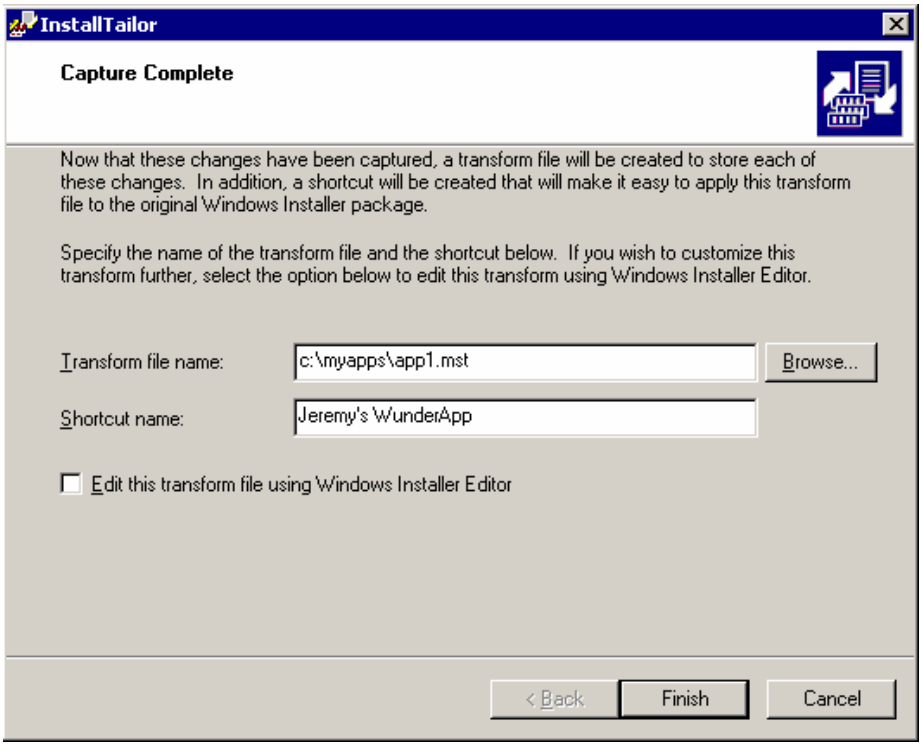

Figure 1.15: Creating an MST file for your applications.

Executing MSIs with Transforms

After you've created a transform by using either a vendor-supplied standalone application or third-party transform-generation source, you're ready to run. If you want a client to get the MSI package with your MST customizations, simply execute MSIEXEC in the following fashion:

```
msiexec /I {packagename}.MSI TRANSFORMS={transform.mst}
```

This command line runs the MSI package and applies the changes that you included in the MST file.

Patches

Getting the base bits online via the administrative installation is important. It's also important to make use of transform files, as they let you specify which bits that you want to make their way onto the client desktop. But what if the original bits need to be fixed in some way? That's where the MSI file definition has room to be adjusted.

If an update or a fix is available for a current MSI installation, you can *patch* the original installation with the latest bits to ensure that the most modern bits are being used. These files come in the form of .MSP files, signifying that they are patch files.

After you download the MSP file, you'll need to run the MSIEXEC command to update the MSI with the latest patch file. To do so, use the following command syntax, as Figure 1.16 shows:

```
msiexec /a {packagename}.MSI /p {patchfile}.MSP
```

Figure 1.16: Execute the patch against the MSI file.

Depending on the application, you might be prompted for the path of the administrative installation. After the file installs, all users who run the MSI will have the latest updates.

The good news about using MSP files for patching is that you're actually modifying the source MSI file. Thus, all new installations that spring forth from this MSI will be up to date with the latest patches. The bad news about using MSP files for patching is that you're stranding the installations that used the previous (un-patched) version of the MSI. If a client that used this un-patched MSI has a problem and must pull down a file or two from the source, it won't be able to because the MSI file is now different.

Therefore, you have to instruct all systems that used the previous version of the MSI that a *new* MSI file is available and to reinstall from that source. A typical command line to do so might look like the following example:

```
msiexec /fvomus package.msi REINSTALL=ALL
```

This command instructs the application to perform a full reinstallation. After running this command, if a re-installation is required or even just one file is damaged, the target system matches the administrative installation.

> ☞ The /f option in MSIEXEC simply specifies that you want to do a repair. The v, o, m, u, and s options essentially overwrite all previous files and registry entries if they are encountered.

Roadmap for the Rest of the Text

At this point, you should have a good handle on what the Windows Installer technology does and why the MSI file type is necessary. With this knowledge, we can start committing to the idea of consistently using the MSI file type so that every application we deploy is delivered as an MSI package.

In future chapters, we'll show you how to start creating your own packages and make use of all that the MSI technology has to offer. We'll do so via both free and third-party tools that help in MSI file generation, as we'll show you in Chapter 2.

In this chapter, we touched on the relationship between Windows Installer and MSI files. But to really make the best use of the technology, we'll need to dig deeper. In Chapter 3, we'll tackle the package structure in greater detail, talk about how to further manipulate packages using transforms, and learn how to secure our packages. Then, we'll move on in Chapter 4 to the secret world of the MSI software development kit (SDK).

Many companies are moving toward Win2K and AD, but many are staying put with either Novell or NT for the foreseeable future. If you're not planning on going to AD anytime soon or you simply don't want to make use of the built-in Win2K software deployment features, Chapter 5 will be a must-read for you.

Finally, we'll wrap things up in Chapter 6 by exploring various ways to distribute the packages you've learned to make while repackaging and authoring. Have a small user base? Have a giant user base? Have Win2K and AD? Chapter 6 will highlight various methods for you to get the right package to the right people.

Chapter 2: MSI Tools Roundup

by Jeremy Moskowitz

In the last chapter, you were introduced to the current crop of installation headaches and how the Windows Installer technology in conjunction with MSI files helps get you closer to the way you should be installing your software. Remember, Windows Installer allows you several tangible benefits that you can't get via the widespread setup.exe methods. A quick review of the major highlights follows:

- Transactional install and rollback—If the package fails to install midway, the package automatically removes itself as if the installation was never attempted.
- Self-healing (or self-repair) of corrupt or deleted critical files—If the installation is damaged, a file (or many files) can be grabbed from the installation source to fix the application.
- Just-in-time (JIT) installation—A package need not be *fully* installed right away. If a feature or component of the package is needed later, it can be grabbed from the source and loaded JIT—all while the application is already started.

These benefits all sound terrific, and they are. But if your application isn't packaged as an MSI file already, what are you going to do? Quite simply, you'll need to get your applications to the MSI "promised land." And you'll do that with one or more of the myriad tools available. You'll use these MSI creation tools to manage the applications you buy or develop in-house and repackage them into new MSI packages. This process is usually as simple as rounding up the existing setup.exe-type applications and repackaging them into the Windows Installer and MSI format.

Some tools are free and some cost a bit, and they all have different angles and philosophies for repackaging. That's what this chapter is about—to expose you to the different options you have for authoring or repackage your application. You'll get glimpses into how to use each tool; however, this chapter isn't meant to discuss every nook, cranny, and feature that any specific application provides. In addition, this chapter isn't meant to guide you to a specific tool, and it doesn't include specific recommendations for a tool that you should use.

Each of these tools will lead you to the end goal, which is to repackage your application into the MSI format. Indeed, each tool will do the job. I will be minimizing the step-by-step instructions, and, instead, try to emphasize each application's overall methodology differences. It is also my goal to briefly expose you to a little bit of each application's interface so that you can be somewhat comfortable and familiar with each one should you encounter it on your own. By being exposed to many tools, you can make an informed purchasing decision as well as be a little more comfortable with using the tool upon first use. You will use this information about the tool or tools you choose in the next several chapters, especially Chapter 4 about the best practices for building packages.

Basics of the Repackaging Approach

Many of tools that we'll discuss share a common approach. Specifically, these tools perform a *snapshot*; they take a before and after picture and generate a package based on the *delta* (the differences between the two pictures). Although these steps represent the basic approach of the majority of the packages we'll discuss, each tool will bring something extra to the table, to enhance the experience. A snapshot occurs as follows:

- Start the tool to perform the snapshot.
- Have the tool examine the contents of the hard drive to see the current state.
- Run the Setup.exe program for the application that you want to repackage.
- Set the desired installation options.
- Customize the installation as desired.
- Have the tool re-examine the contents of the hard drive to see the changes to the original state.
- The tool creates the repackaged application.

Regardless of which tool you're working with, you should keep one main principal in mind: ensure that the machine from which you're creating the snapshot is as *clean* as possible. A clean machine is one with the OS installed and almost nothing else. That way, when the tool performs the snapshot, it has only the most minimal interference. Even seemingly innocuous applications, such as screensavers, could impact what is seen in the before and after scans of the hard drive.

However, if you cannot use a totally clean machine (for instance, the application you want to repackage has dependencies on another preloaded application), you can use the following tips to improve your chances at a successful repackage:

- Close all applications
- Stop all unnecessary services
- Clear out the Recycle Bin
- Disable screensavers
- Disable antivirus programs
- Disable anything else that runs in the background

Microsoft's Offerings

Because Microsoft is the biggest proponent of the Windows Installer technology, it would stand to reason that the company would have several tools to assist in MSI file creation. After all, MSI does stand for Microsoft Installer. With that in mind, we're going to explore three tools that Microsoft provides for administrators to repackage an application into the MSI format.

WinInstall LE

WinInstall LE is a free tool provided by Microsoft to help with creating MSI files. WinInstall LE wasn't designed by Microsoft, rather, it has a long and strange history. WinInstall LE was developed by OnDemand Software. Seagate bought WinInstall LE, then Veritas bought Seagate. WinInstall LE didn't get much attention from Veritas, as the company's focus is mainly backup software. OnDemand Software reformed and took the product back into development.

> It's easy to get confused by the naming convention that OnDemand Software has chosen. WinInstall LE is the tool we're discussing that you can use to repackage setup.exe programs into MSI files. The primary job of WinInstall (without the LE) is to deploy and install applications throughout your environment. However, the full package of WinInstall does contain a standalone repackaging application that is basically a revised cousin of WinInstall LE.

WinInstall LE Operation

To get started on your WinInstall LE journey, you'll need to grab hold of either a Win2K Professional or Win2K Server (or Advanced Server) CD-ROM. Once you have it, locate SWIADMLE.MSI, which can be found in the {cdrom}:\VALUEADD\3RDPARTY\MGMT\WINSTLE directory, as Figure 2.1 shows.

Figure 2.1: Traverse to the appropriate directory to reach the SWIADMLE.MSI file.

The snapshot utility of WinInstall LE is called Discover. After running Discover, you'll title your application and give the tool a location at which to store the application, as Figure 2.2 shows.

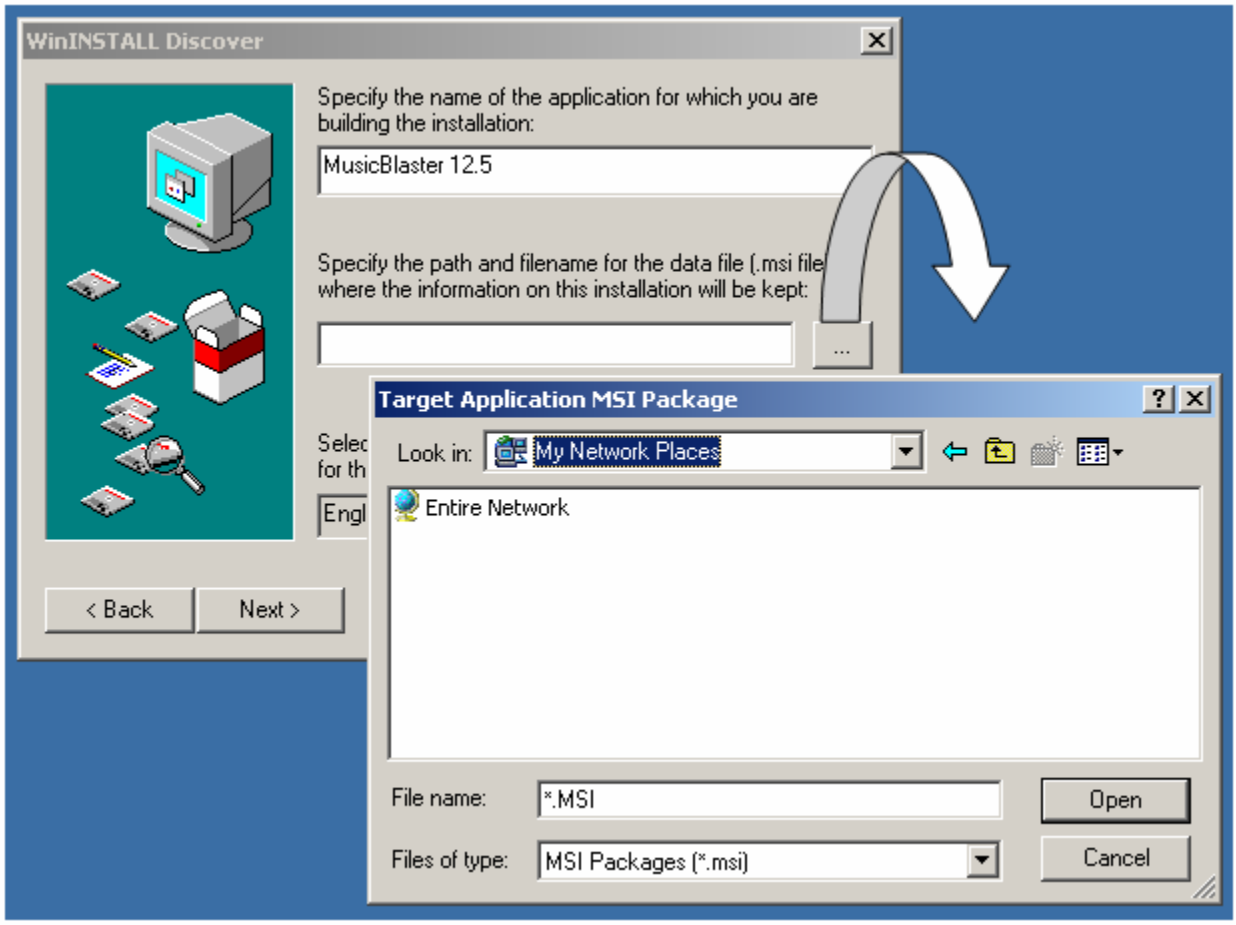

Figure 2.2: Put the MSI file you are creating on a network point.

Then, you'll simply run the setup.exe program from your application and complete the snapshot; out will pop your first MSI file.

After you have your MSI file, you could, if you were so inclined, venture into the murky world of WinInstall LE's package-editing tool. At this point, WinInstall LE's usefulness starts to break down. The WinInstall LE editing tool is called the Veritas Software Console, and can be launched via Start menu, Program Files, Veritas Software, Veritas Software Console. With this program, you can load an MSI package into the Veritas Software Console to manipulate your package, as Figure 2.3 shows.

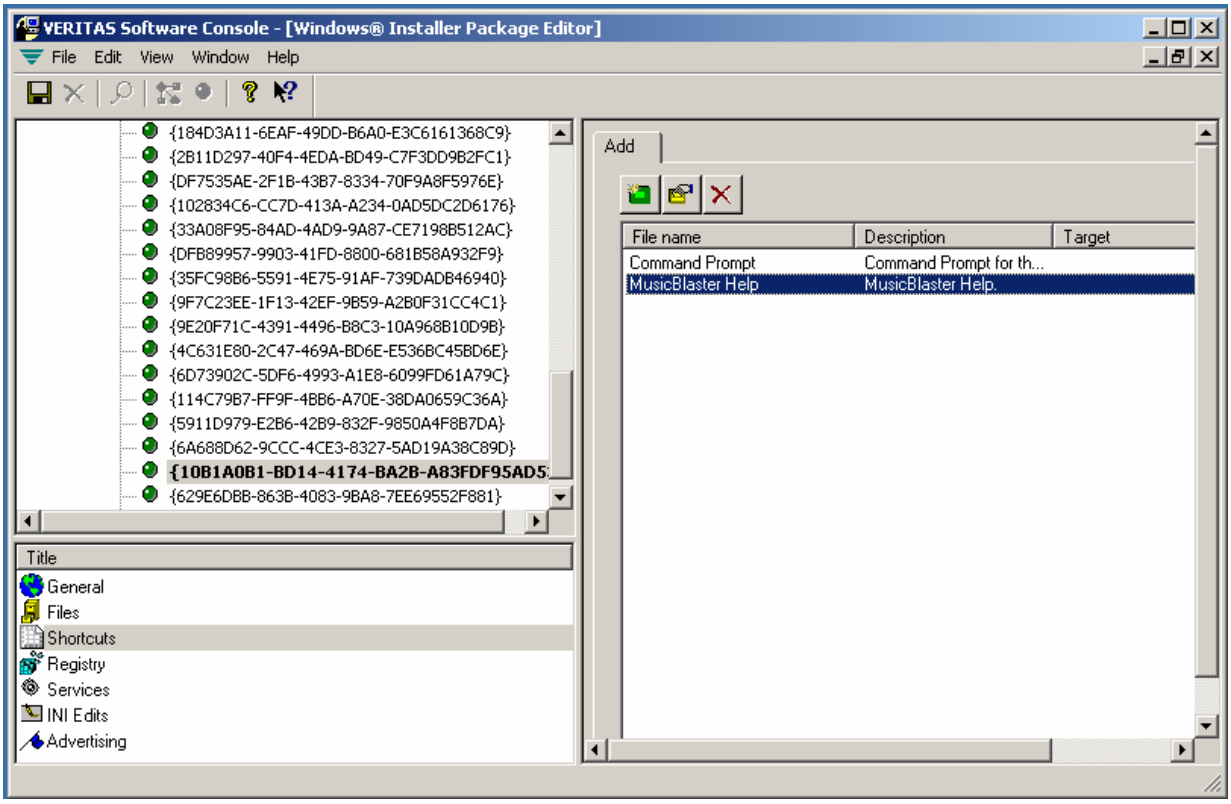

Figure 2.3: The post-editing tool is complex to navigate and negotiate.

Trying to customize a package using this tool is difficult at best. It's a fairly cumbersome, abstract, and difficult-to-use interface and requires an intimate knowledge of the relationships between features, components, and the like.

SMS Installer

The SMS Installer utility is another tool available from Microsoft for those who own and operate SMS environments. The SMS Installer utility was originally a Wise product. Wise licensed Microsoft a chunk of source code that became the SMS Installer, and Wise continued on developing the Wise Installer, and later, Wise Package Studio (which we'll explore later).

The SMS Installer is a poorly named product, but has some powerful features. It's poorly named because the name suggests that this program offers only a specific installation method using SMS, which doesn't incorporate all that the program does. The SMS Installer provides three ways to capture events and eventually create MSI files:

- Straight snapshot (similar to WinInstall LE)
- Via the SMS Installer Watch utility
- Through a custom/manual script editor

We'll be briefly exploring each of these three methods.

> You'll only be able to download and unpack the setup routine for the SMS Installer if you've actually got an SMS 2.0 site server up and running. This requirement is simply a protection mechanism to ensure that only people who have SMS licensed and up and running are able to use the SMS Installer tool. You can find the SMS Installer tool at http://www.microsoft.com/smserver/downloads/20/tools/installer.asp.

MSI creation is a fairly new feature in the SMS Installer feature set. Historically, the SMS Installer didn't create MSI files; rather, its only output was in the form of self-installing executables (.EXE files.) The latest revisions of the SMS Installer can create MSI files—via both the SMS Installer tool and an external program called the Installer Step-Up utility (ISU).

The SMS Installer Repackage Installation Wizard Tool

After you've installed the SMS Installer and launched it for the first time, you'll see plenty of options to get started, as Figure 2.4 shows.

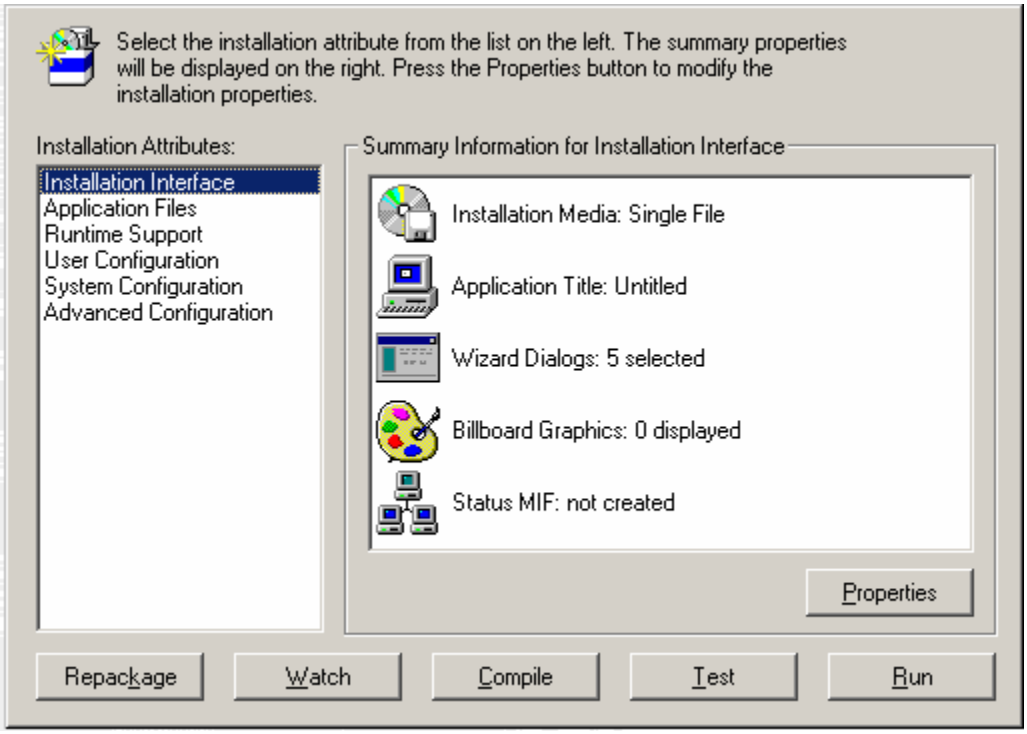

Figure 2.4: The SMS Installer has a lot of nooks and crannies.

I suggest that you start by exploring the Repackage option, which, as you might expect, repackages applications. After you click Repackage, you'll be prompted for the location of the setup.exe file as well as which portions of the hard drive to scan. After you provide this information, the first scan is performed, as Figure 2.5 shows.

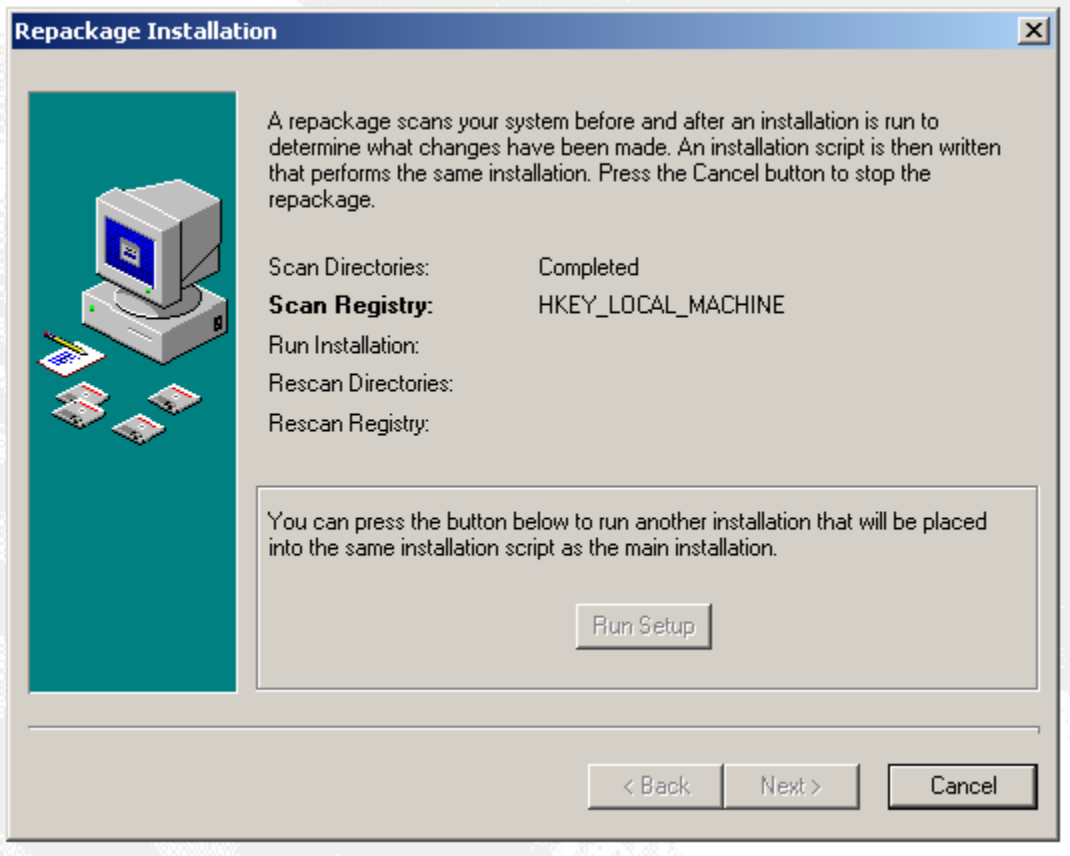

Figure 2.5: The repackage option starts off in a manner similar to how WinInstall LE starts off.

> ☞ You can have the tool capture multiple installations at the same time by clicking Run Setup when it becomes available (after the scan completes). However, it's best to repackage one application at a time as doing so lets you create individual MSI applications, as opposed to one MSI file with lots of applications inside.

After the scan completes, the setup.exe program that you selected will be launched. Complete the program's installation, and finish the Repackage Installation Wizard. At this point, you'll have captured the setup.exe program's actions but you're not quite to an MSI file. In a moment, you'll see two ways to make that capture into an MSI file.

The SMS Installer Watch Tool

In those rare circumstances in which your application is already installed (and you cannot find the setup.exe program) or the application doesn't come with a setup.exe file, you can use the SMS Installer Watch tool, as Figure 2.6 shows.

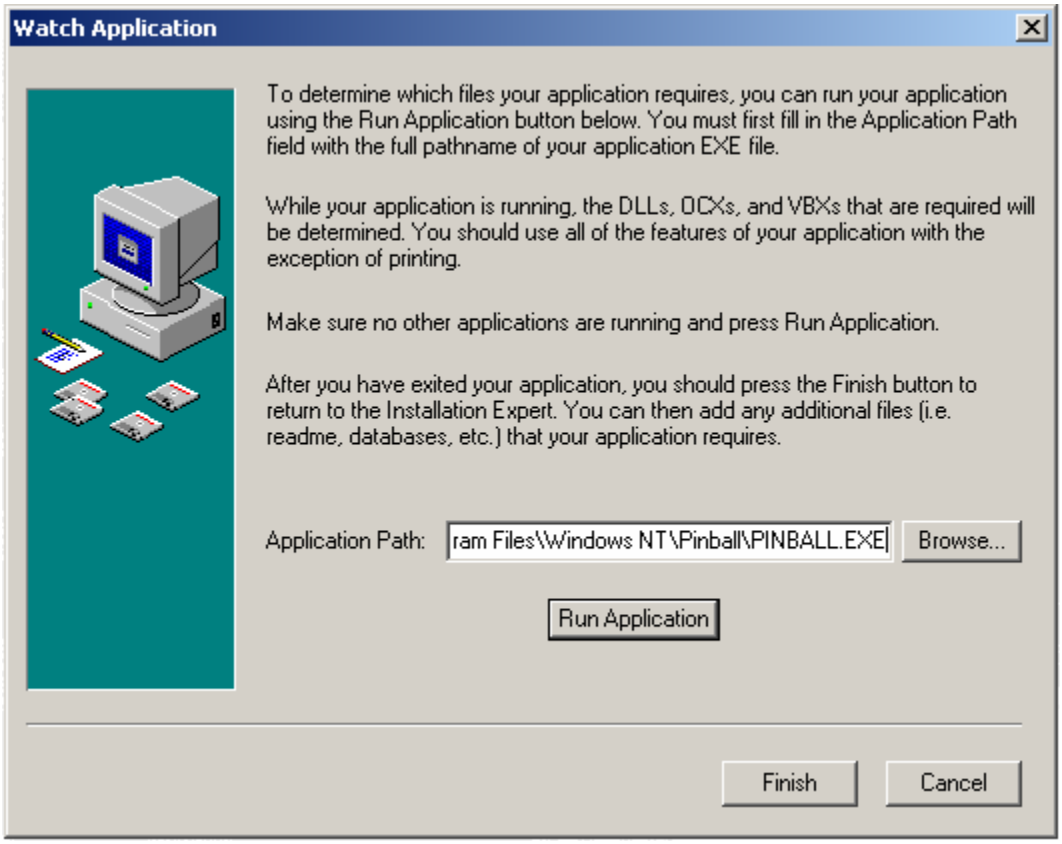

Figure 2.6: Use the Watch application when you don't have a setup.exe program for your application.

After running the application, SMS Installer looks for every call the OS makes to see which files are being accessed. When this process completes, you'll be able to perform a caption using the method I previously discussed.

This option is worthwhile in a pinch, but is ultimately not going to be as accurate as a true snapshot or capture. The reason is that when you use the Watch method, you'll have to be sure that you're selecting *every* option that the application provides. If the application is a word processor, you'll have to make sure you run the spell checker, mail merge, and Help features—everything that could possibly be selected—to ensure that every DLL, OCX, and VBX file are all "seen" by SMS Installer.

The SMS Installer Script Editor

The SMS Installer Script Editor is the most powerful feature of the SMS Installer. After you've either performed a recapture or used the Watch application, you have the option to make modifications to *how* the package is going to be installed and which options will be seen during the actual installation process. You perform these changes inside the Script Editor. What if you wanted to make sure your resulting repackage only ran on Win2K? What about making specific changes to certain INI files? You can do these things and more with the Script Editor, which Figure 2.7 shows.

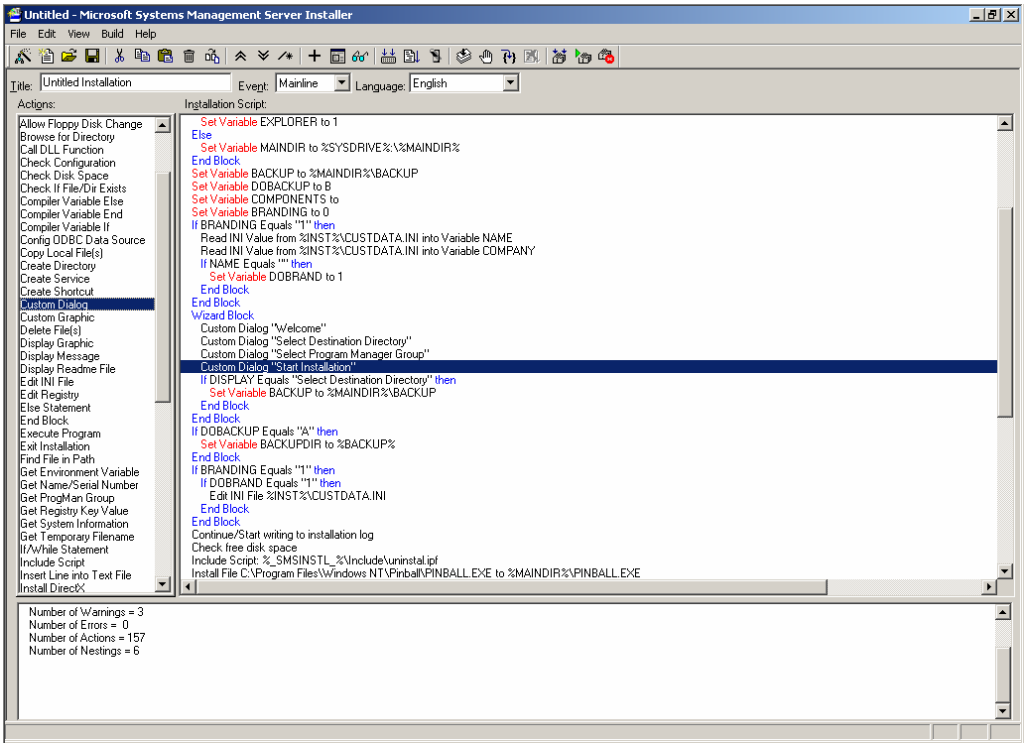

Figure 2.7: The SMS Installer Script Editor has ton of options for configuring installation options.

You simply drag Actions from the left to a point in the script on the right. After the action is where you want it in the script, you can modify the action. For instance, if you wanted to modify the standard setup screens by adding some custom information, you can edit the dialog box within the Script Editor, as Figure 2.8 illustrates.

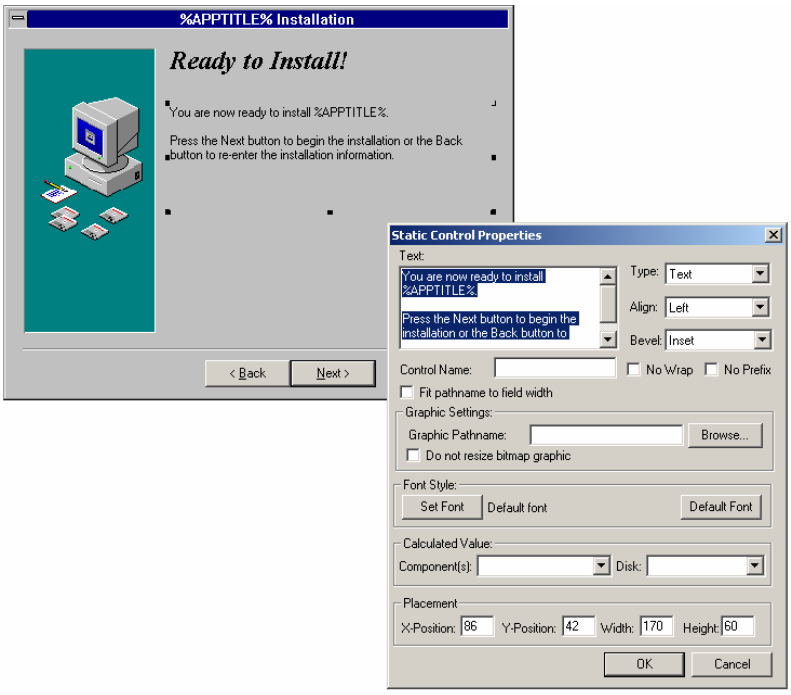

Figure 2.8: You can customize each installation screen using the Script Editor.

The SMS Installer Script Editor is an immensely powerful tool and has simply too many options to discuss in detail here. However, there are many resources that provide step-by-step directions for its use; I recommend Rod Trent's *Microsoft SMS Installer* (McGraw-Hill Osborne Media). Rod also has an excellent Web site that is full of both SMS and SMS Installer information http://myitforum.com.

If you make customized script additions (as discussed previously), then choose to make MSI files from them (as we're about to explore), you'll need to thoroughly test your resulting MSI files. Sometimes customized scripting made inside the SMS Installer will not be perfectly reflected in a resulting MSI file (but will be perfectly reflected in a compiled .EXE file).

Creating MSI Files with the SMS Installer

Your last step is to actually create an MSI file. You have two options to do so: by using the built-in MSI Compilation Method or the external ISU utility.

After the capture or Watch-application process is completed (as well as the optional script editing), you can simply choose the Build menu option to *Compile As Windows Installer Package*, as Figure 2.9 shows.

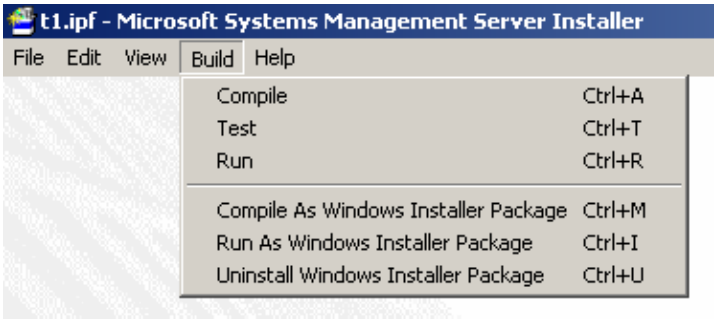

Figure 2.9: You can create MSI packages directly from SMS Installer.

When you do, you'll see the MSI file being created, as Figure 2.10 shows.

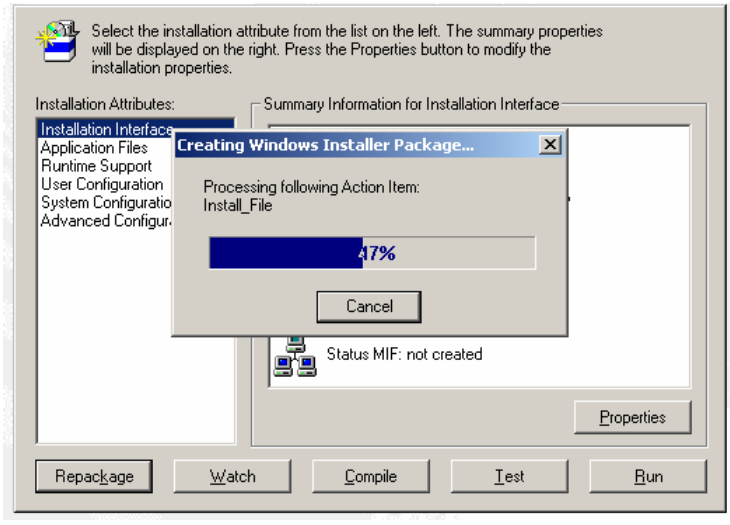

Figure 2.10: The MSI file is created from within the SMS Installer.

Alternatively, you could use the command-line ISU utility, which converts SMS Installer self-executables to MSI files. Figure 2.11 shows the ISU utility's options.

Figure 2.11: The ISU utility options as well as a compile.

The ISU utility is useful when you have existing SMS Installer scripts that you want to batch-style convert to MSI files. However, as Figure 2.11 illustrates, not every scriptable option converts to a perfect MSI action. When the SMS Installer converts your packages to MSI, it needs to make decisions about what your intentions are, and thus might perform an action on your behalf that you didn't intend. In other words, you might script an action, but that action doesn't work or work properly when run as an MSI. This behavior is the SMS Installer's biggest shortcoming. That is, although it's a powerful tool that creates useful self-executable .EXE installation files, it doesn't always produce the best-running MSI files. In such cases, you might need to get into the nitty-gritty of the MSI file. To do so, you'll need the Microsoft Orca tool, which the following sidebar describes.

Microsoft Orca Tool

As we've previously stated, each MSI file is really just a database. Microsoft provides Orca, a low-level tool, to let you get into an MSI file and look around. Orca is part of the Windows Installer SDK, which is most useful for programmers. But systems administrators can use the SDK—and Orca—to find the most nitty-gritty documentation for how something works within a Windows Installer executable or an MSI package.

The full SDK can be found at http://www.microsoft.com/msdownload/platformsdk/sdkupdate/. If you're not interested in the full SDK, you can get a subset of the SDK, which includes Orca at http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/001/457/msdncompositedoc.xml.

If you choose to load a file into the Orca tool, you'll be able to see the file's rawest elements: the tables and rows that make up the database, as Figure 2.12 shows.

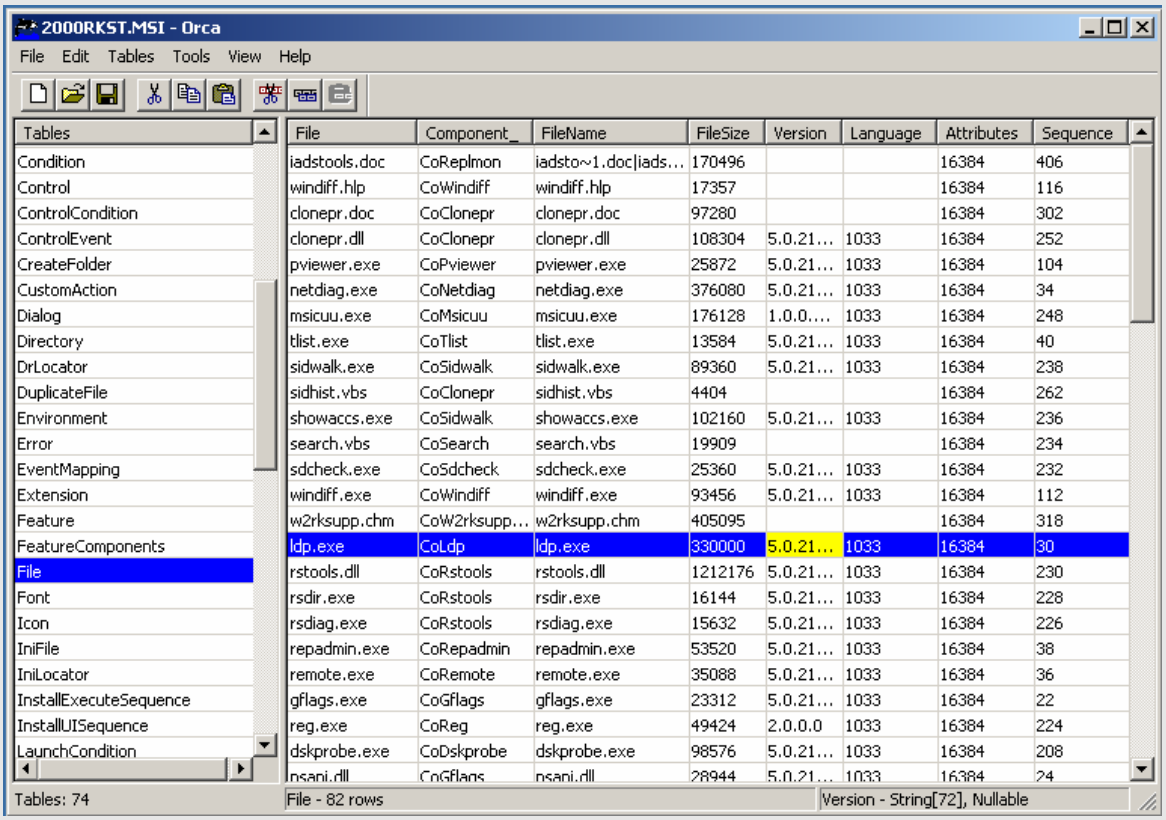

Figure 2.12: An MSI file loaded into the Orca tool.

The frame on the left has the heading of Tables; each table represents a portion of the MSI file. The larger pane on the right represents each row within the table, representing a record or an entry. In Figure 2.12, I've highlighted the table named File, and each row of the File table shows an entry for each file contained within the MSI.

This tool isn't the most user friendly tool for administrators, but with a little poking around you can discover some useful information. Specifically, whenever you use any tool to create and/or edit an MSI file, that tool might veil what's really going on in the file. With Orca, you can be confident that you're seeing what's really going on in the MSI file. However, the beauty of third-party tools is that they provide the information that you need in a user-friendly format—the raw elements that Orca reveals are quite messy. So use Orca only when necessary if you suspect troubles. You can find additional information about the Orca tool (from an administrator's perspective) at http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q255905& .

Commercial Third-Party MSI Tools

Microsoft isn't the only vendor that offers tools that can make MSI packages. Indeed, there are such tools available from third-party vendors. Like the SMS Installer, most third-party tools are dual-purpose; that is, they are capable of building scripts to get your home-grown package packaged and equally able to capture your application and wrap it up into an MSI package. For instance, you can use them to both repackage a setup.exe application into an MSI file and repackage the latest virus definition file components into an MSI that is prepared for deployment.

The feature set of third-party tools is usually quite robust. The best ones offer additional and innovative capture methods as well as advanced conflict management to assist in ensuring that in the case of DLLs that have the same name but are different versions, the best one is used for your MSI. Not every tool has these advanced features, but you might not need all these features. Check out all the tools to see which one fits your needs.

Commercial Third-Party Tools at a Glance

There are quite a few third-party tools that could fit the bill for your environment. To help you decide, Table 1.1 shows some of the most popular third-party MSI repackaging tools as well as a bit of information to help you get started on your third-party MSI tool investigation.

> The tools in this table are commercial programs. I highlight free tools in the next section.

Tool	Vendor	Web site
Wise Package Studio	Wise Solutions	http://www.wise.com
Wise for Windows Installer	Wise Solution	http://www.wise.com
AdminStudio	InstallShield	http://www.installshield.com
Prism Pack (formally Picture Taker)	Lanovation	http://www.lanovation.com
Unity Installer	PriceWaterhouseCoopers	http://www.unitysite.com/
E-Wrap	Novadigm (formerly ChicagoSoft)	http://www.novadigm.com

Table 2.1: Some popular third-party tools.

> There is an additional unofficial list on InstallSite.org, a site dedicated to MSI development (http://www.installsite.org/cgi-bin/frames.cgi?url=http%3A%2F%2Fwww.installsite.org%2Fpages%2Fen%2Fw2k_msiauth.htm). You can find another unofficial list at AppDeploy.com (http://www.appdeploy.com/tools/browse.asp?r=1). Note that InstallSite.org is geared for developers and AppDeploy.com is geared for administrators.

Highlighting every commercial third-party tool available is beyond the scope of this chapter. Instead, I'll review some of the top tools and draw attention to some of their distinguishing qualities.

Wise Package Studio

The Wise Package Studio is a popular tool for system administrators who want to repackage installations into MSI tools. In fact, if you're already comfortable with the SMS Installer, there are places in the Wise Package Studio that are similar to the SMS Installer, such as the Wise Script Editor.

After Wise licensed the code to Microsoft for the SMS Installer, Microsoft didn't develop the SMS Installer much beyond its original inception besides bug fixes, the ISU utility, and the in-program MSI builder capabilities. However, the development of that original code didn't stop for Wise. It became Wise for Windows Installer, and has since evolved to become Wise Package Studio.

As I previously mentioned, there are too many features offered by third-party tools to highlight them all (though I will highlight some specific competitive distinguishing qualities in an upcoming section). Instead, I'll highlight where the Wise Package Studio can help administrators most: in gaining a process around their MSI development process.

The Wise Package Studio wraps its package creation procedure into an approach that contains two main parts: a *process* and a *project*. That is, you start off with a *process*—either a pre-defined process or one that you define. Then, for each MSI package that you want to build, you leverage the process and create a new *project*, say, MSIFILE1.MSI. In other words, perhaps not every MSI file you'll build will require the exact same set of rules to create it, and the Wise Package Studio is flexible in this manner to help you ensure you have a well-defined *process* for your *project*. In the following example, I have modified the standard process to add a reminder for myself to ensure that I won't forget to virus scan before repackaging the application, as Figure 2.13 shows.

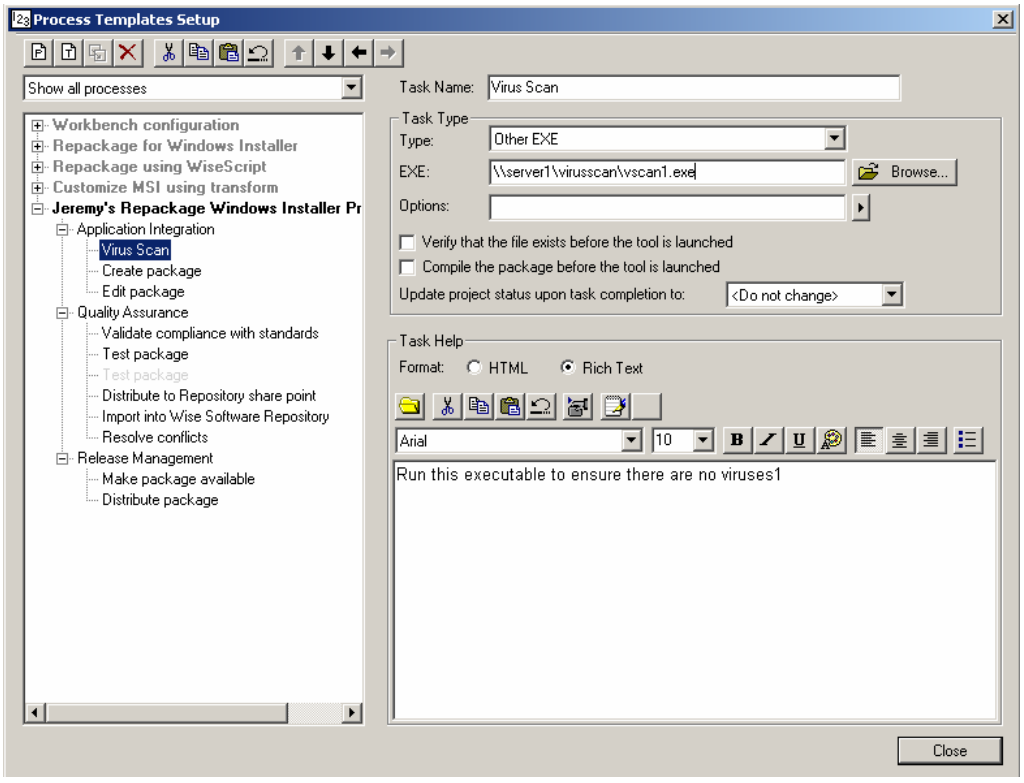

Figure 2.13: You can modify the process with specific steps that you can customize.

Then, when I'm ready to create a new project, I can leverage my own customized process (the one that includes a step ensuring that I won't forget to virus scan). I simply create a new project, and pick the process that I want, as Figure 2.14 shows.

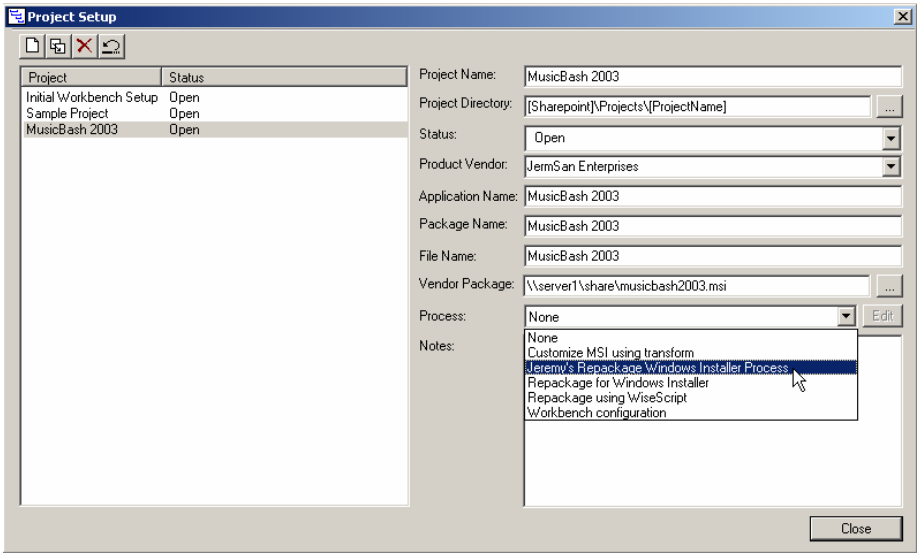

Figure 2.14: Select a process for each project that you want to repackage.

After you've established a process for your project, you're ready to work inside that project. Again, the process is what keeps you on track when wrapping up the MSI. The interface of the Wise Package Studio helps keep you on that track once a project is initiated. For example, the program offers a feature called the Workbench, which Figure 2.15 shows, that helps with the repackaging process.

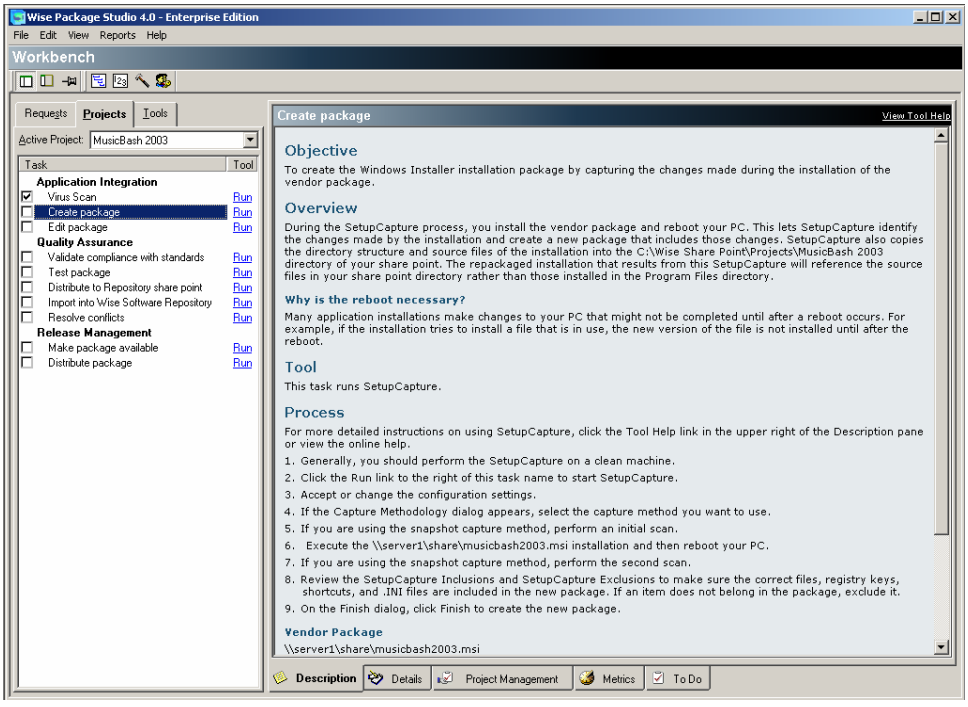

Figure 2.15: Use the Workbench to step you through your specific process for your specific project.

The process steps are seen on the left. Each step in the process should be executed as described in the notes on the right. Simply click the blue Run tags next to each step in a process, and a specific tool will be launched to assist in accomplishing that step in the process. Figure 2.16 highlights the idea that each step can be associated with one or more tools.

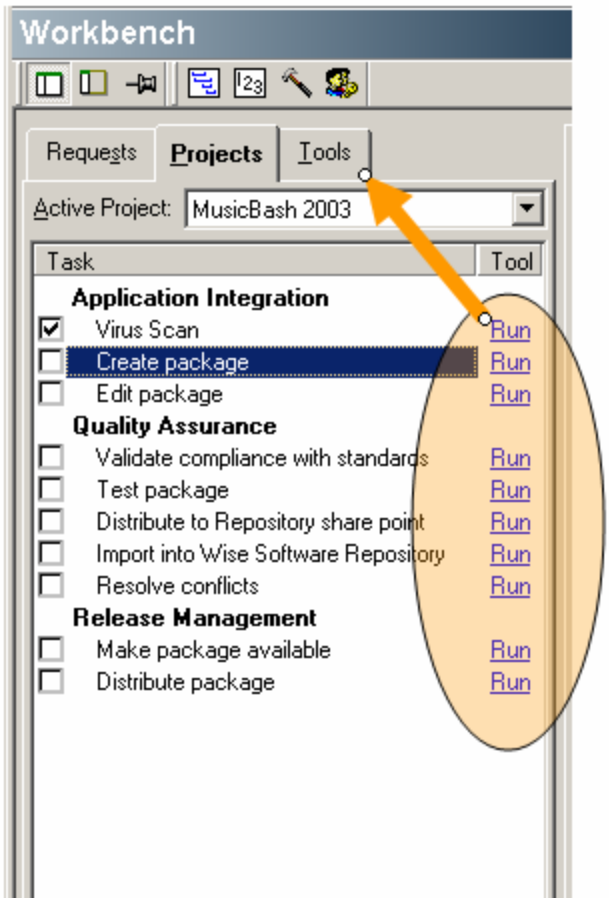

Figure 2.16: Each step in the process can correspond to one or more tools.

Again, selecting Run for a specific task will run a specific tool. Alternatively, you could select the Tools tab to expose all the tools that comprise Wise Package Studio, as Figure 2.17 shows. Each tool has a step-by-step description of how it accomplishes a task as well as a preview of those tasks.

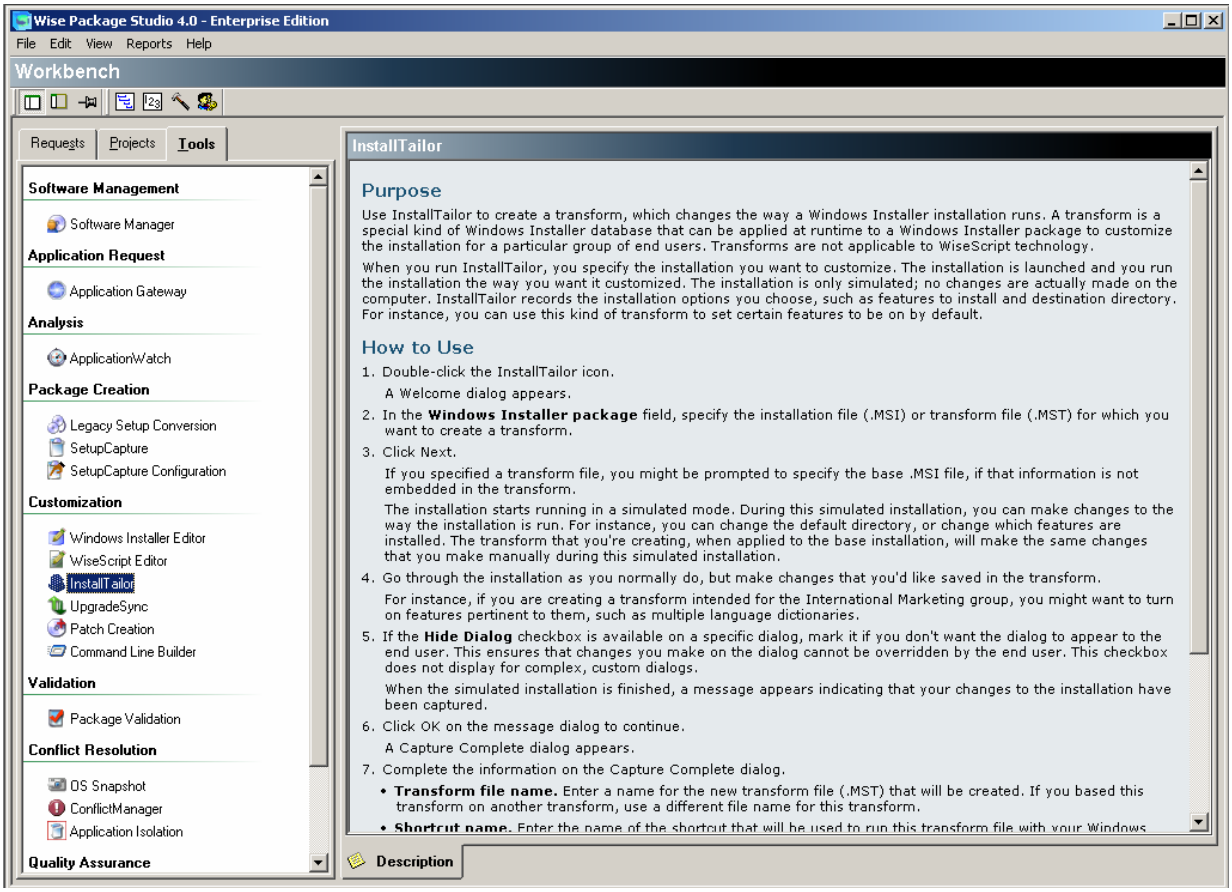

Figure 2.17: Each tool may be selected individually if required.

Wise Package Studio comes in three flavors: Standard, Professional, and Enterprise. Each comes with a different feature set:

- Standard Edition—Performs basic MSI repackaging via "ad hoc" approach. Wraps up packages and validates that they were packaged correctly.
- Professional Edition—Performs more advanced MSI repackaging. Geared for both a single administrator as well as multiple administrators working on the same project. This edition adds increased testing and conflict management and performs the process-oriented approach I've been describing.
- Enterprise Edition—This version is unique because it enables a full end-to-end lifecycle approach for MSI files. Specifically, this version allows for requests to be submitted about a package that an end user might want developed and can track the project's progress through the life of the project.

AdminStudio 3.5

InstallShield is best known in the developer community for its tool that packages *new* applications—called InstallShield Developer. This tool is really a developer's tool. In fact, it even hooks into Visual Studio .NET while the developer is working with it. However, InstallShield also offers an application that is geared toward administrators—AdminStudio. This product allows for both process-and-project driven MSI creation and a manner to launch individual tools to perform specific tasks.

For example, with the AdminStudio's QuickStart Guide (see Figure 2.18) an administrator can simply hover the cursor over one of the proposed questions, and the program will replace the question box with an "answer" that describes which tool will be launched.

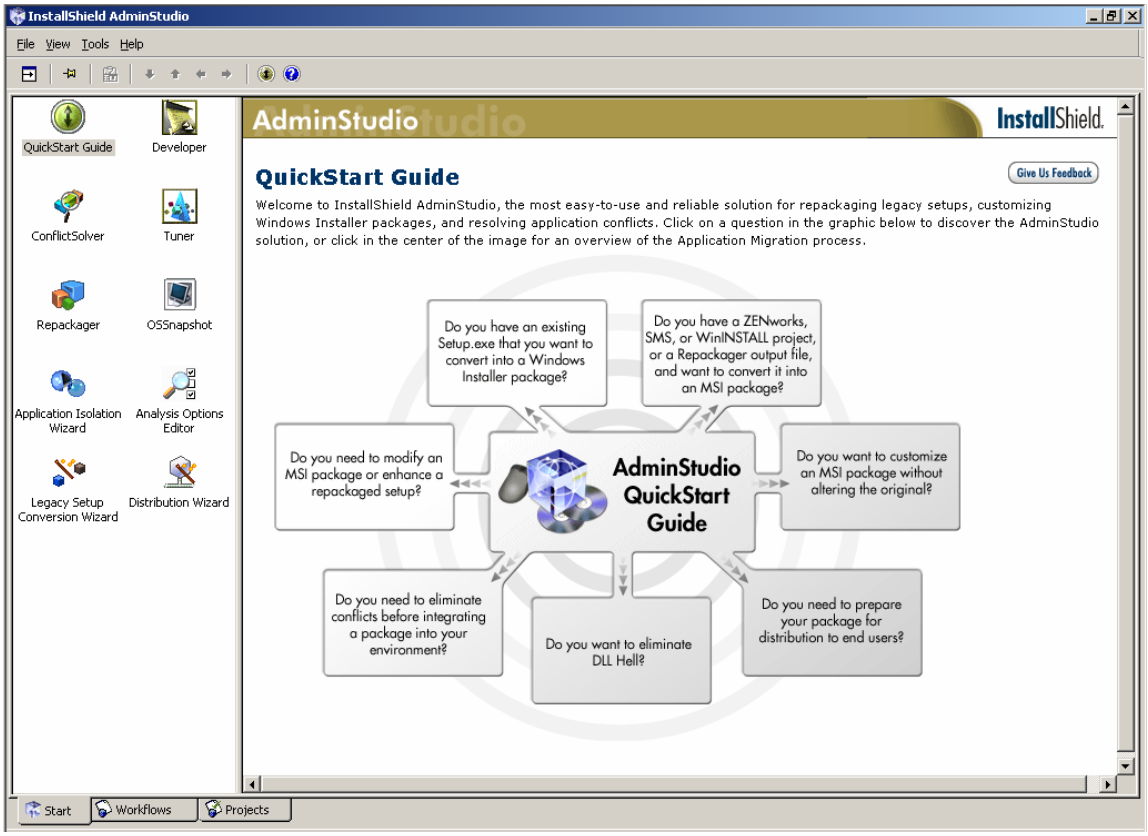

Figure 2.18: AdminStudio's QuickStart guide.

Alternatively, if the administrator knows which tool she or he wants to use, the administrator can simply select that tool from the Start menu, as Figure 2.19 shows.

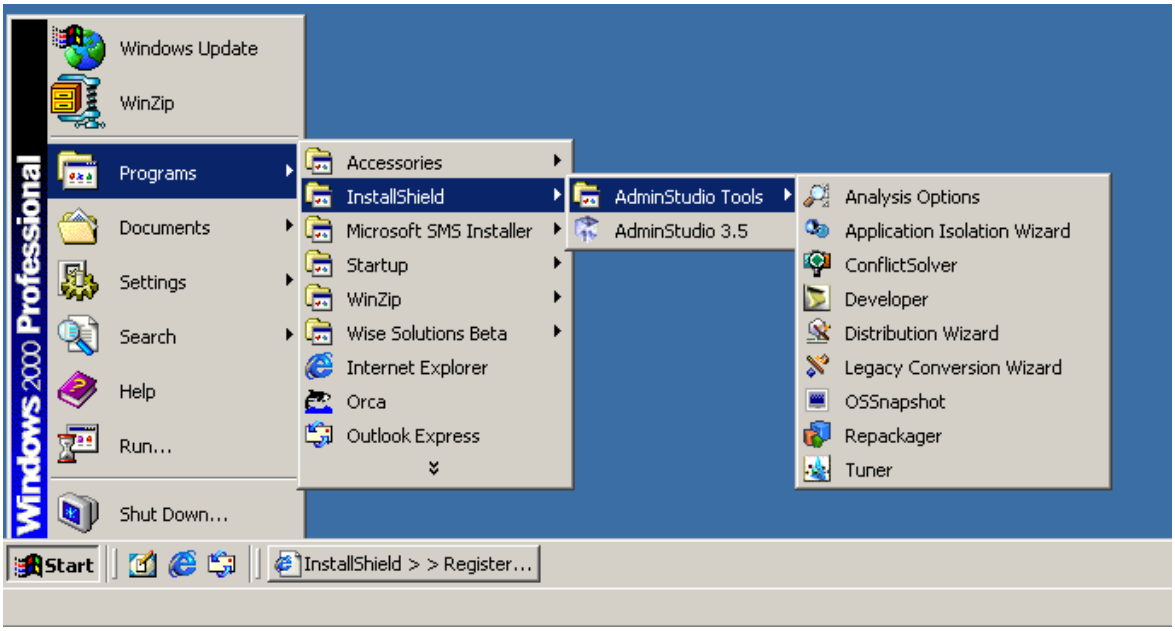

Figure 2.19: The individual tools are also available from the Start menu.

After launching a tool, a tutorial for the tool is displayed. For example, as Figure 2.20 shows, launching the Repackager tool will launch an 8-page tutorial that you can scroll through to get a better idea of what the tool is supposed to do.

Figure 2.20: The tutorial screens warm you up to each tool.

After you read through the tutorial screens, you have the option to actually launch the tool. Alternatively, AdminStudio can be set up to have a customized "flow" process, called a Workflow, as Figure 2.21 shows.

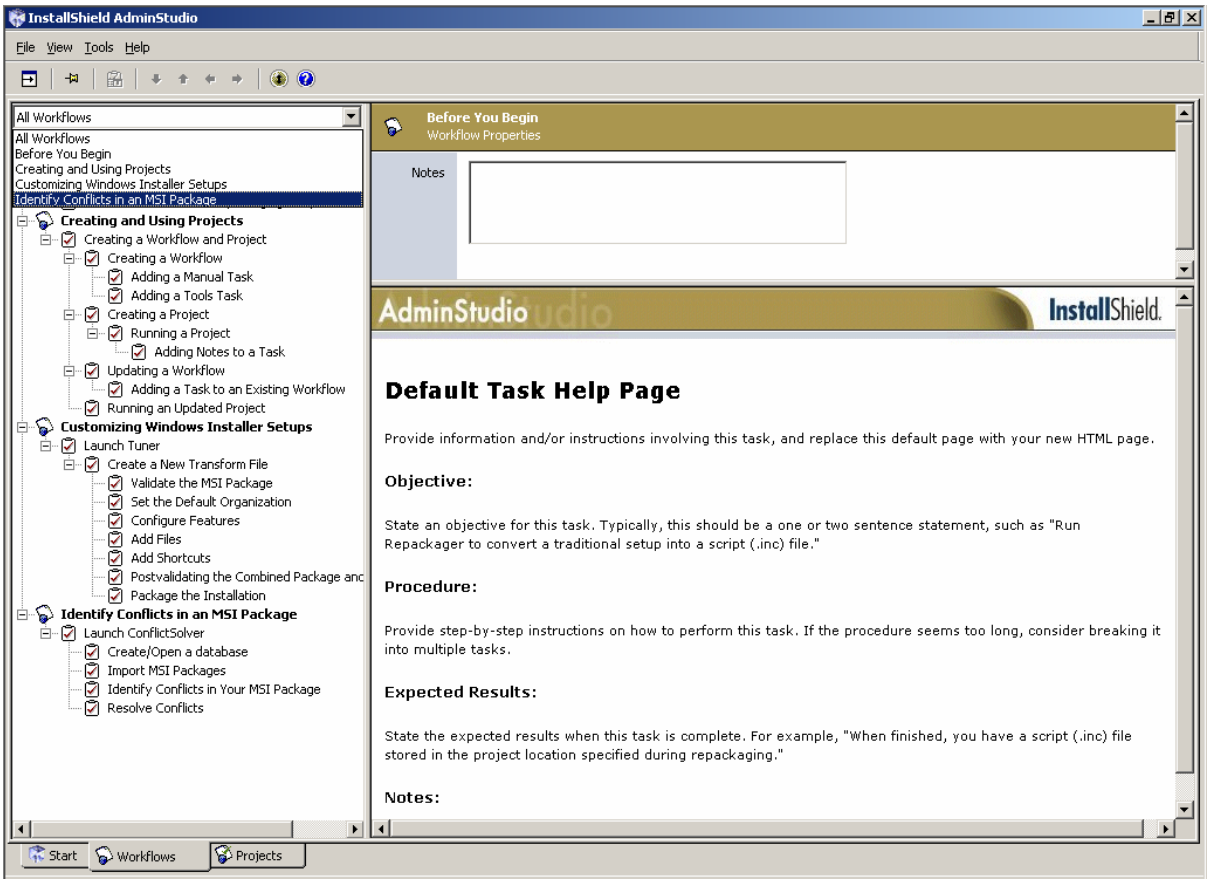

Figure 2.21: AdminStudio offers its Workflow to keep the projects on track.

Workflows have defined tasks and specific tools that can be associated with the task. Administrators can create new projects based on specific workflows.

AdminStudio 3.5 is really meant for a single administrator to perform the repackaging. It is not really a "collaborative" enterprise-ready tool allowing for an end-to-end process. This version comes in two flavors: Standard and Professional. Although the standard version will certainly do the job to get your packages into MSI format, the professional version is much more adept in the automated detection and correction of conflicts.

Prism Pack

Some commercial products are similar to WinInstall LE, but with a friendlier overall ease of use. Such is the case with Lanovation's Prism Pack utility, which Figure 2.22 shows.

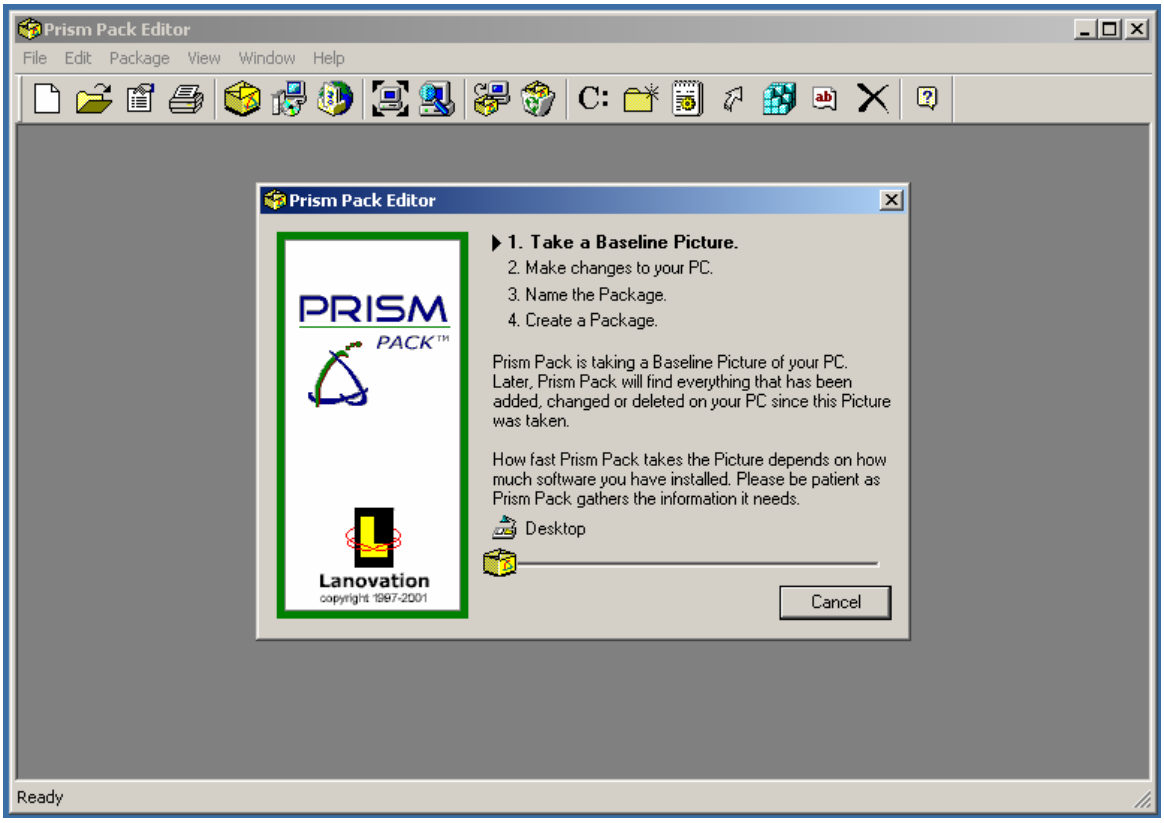

Figure 2.22: Prism Pack "takes off" upon first launch.

Once launched, the tool immediately begins performing a snapshot. It later provides the ability to make basic changes to the created package using the Prism Pack Editor (see Figure 2.23).

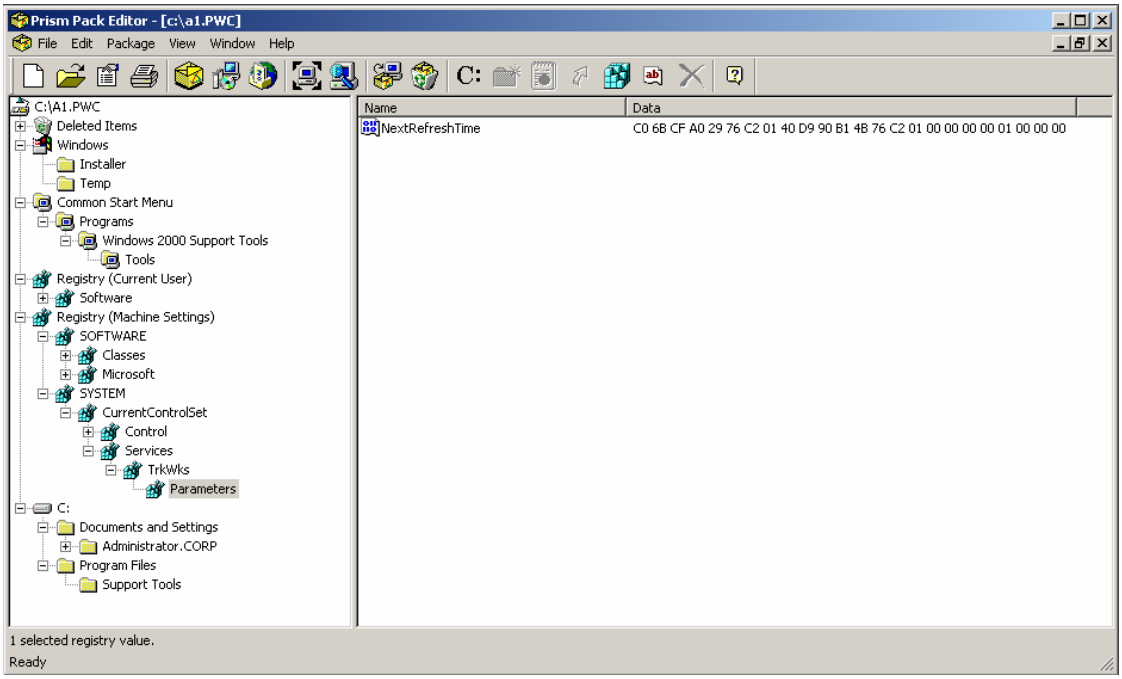

Figure 2.23: The Prism Pack editor is basic and easy to use.

This tool doesn't provide the collaboration abilities, process/workflow/project guides, and elaborate wizards of other third-party tools. However, what Prism Pack lacks in sophistication, it makes up for in ease of use—it's a solid tool that does the job of repackaging well. Sometimes you just need or want a tool that is simple and easy to use.

Added Functionality

As I've already stated, the idea behind third-party tools is to bring additional functionality beyond simple repackaging. Indeed, when choosing a tool, you might want to start with the marketing materials, check out the feature sets, then settle in for a test drive. Most third-party tools will offer the functionality you need to perform day-to-day MSI management functions:

- Snapshot repackaging
- MST (transform) development
- MSP (patch) development
- Easy package transfer to third-party *deployment* tools (such as SMS)
- Customization of the MSI via script, direct edit, or other method
- Ability to manage DLL and other component conflicts
- Ability to key files for self-healing

These features alone might make the ideal tool for you. However, there is still a challenge for the manufacturers of third-party MSI repackaging tools. Today, their challenge is to bring out new innovative features which either augment or replace the traditional "snapshot" methodology. I'll highlight two of the vendors that are doing so and how.

Wise Package Studio 4.0 Repackaging Innovations

Wise Package Studio 4.0 has new functionality beyond performing simple snapshots. This latest version of the Wise Package Studio allows for what Wise calls Virtual Capture (see Figure 2.24). Virtual Capture allows administrators to perform captures on machines that aren't totally clean. Such functionality is a boon for administrators, as cleaning a machine for another repackaging job is a major source of administrative headaches. Usually, administrators simply reformat and reload the machine to get back to a clean machine. But anytime a machine needs to be fully reinstalled, the task can take quite a while. Virtual Capture eliminates this annoyance.

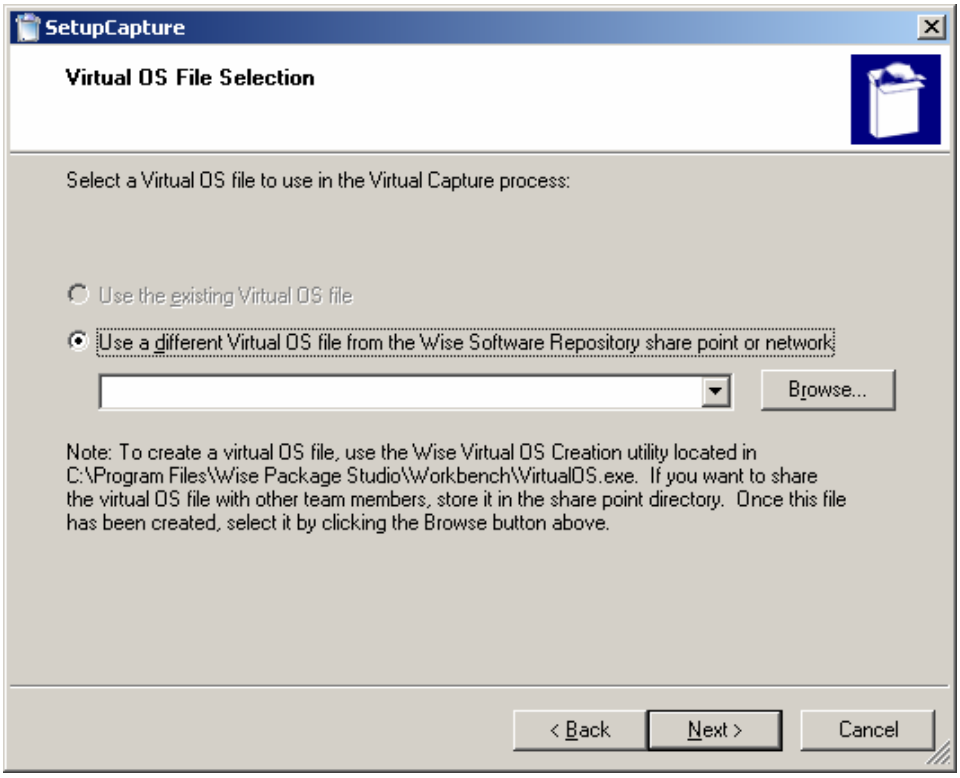

Figure 2.24: Virtual Capture eliminates the need for a clean system.

AdminStudio 3.5 Repackaging Innovations

InstallShield's AdminStudio 3.5 also has functionality beyond simple snapshots. Its latest innovation is called InstallMonitor. This feature is unique because it runs while the application is being installed, monitors what's going on, and actively records the changes occurring on the system. Therefore, there is no before or after snapshot. You simply start InstallMonitor, fire up the setup.exe on the application you want to create the package for, and voila. Over time, the lost minutes waiting for before and after snapshots can really add up. However, it's still best to reimage your machine with a fresh installation after every package capture.

Shareware and Freeware Third-Party Tools

There is an emerging "home brew" market for shareware and freeware tools that help with MSI creation. Shareware tools can be useful, but ensure that if you work with a shareware vendor that you'll get the same level of support as if you went with a commercial vendor. Working with a freeware or shareware vendor one-on-one might produce a relationship you can be comfortable with. And if you're comfortable with the relationship and the product, you've got a good candidate for a useful tool.

Two tools that fit in this category include MSICreate from Corner House (http://www.cornerhouse.ca/en/msi.html) and Setup2Go for MSI from Offshore Development Software/SDS Software (http://www.dev4pc.com/downloads.html).

There are also two free tools that are in various stages of readiness. One is called izfree and can be found at http://izfree.sourceforge.net/. This program doesn't set out to be a huge application, rather, it just wants to get the job done. This tool is in 1.0 alpha release, so use it at your own risk. The other free tool is called NInstall, and it's not even ready for primetime, but you can find more information about it at http://www.chimpswithkeyboards.com/projects/ninstall/index.shtml.

Summary

Exposure to as many MSI repackaging tools as possible can only be a good thing. Otherwise, how will you know what you like? Although the Microsoft tools do an adequate job, they usually fall a little short in the features category. Third-party tools can be pricey, but ultimately worth it if you take the job of repackaging seriously. Shareware and freeware tools sound great, but be sure that you're comfortable with whatever support relationship you work out with the developers.

No matter which tool you investigate, remember that each tool has a different approach, user interface, feature set, and price point. Getting a trial copy and taking a tool for a test drive is really the best approach to figure out which tool is ultimately right for you.

Chapter 3: Windows Installer Internals

by Darwin Sanoy

Why does an administrator need to be concerned with the internals of Windows Installer? When Windows Installer works correctly, it provides you with some sophisticated features that save you time and enhance your users' productivity. However, when things go wrong, finding the problem will depend heavily on your understanding of how the internals of packages work. This idea is applied equally to debugging vendor supplied packages as well as ones from your own internal packaging operations or in-house developers. A good framework for understanding Windows Installer internals will also give you the foundation for building good packages.

> The Windows Installer SDK and its tools will be referenced throughout this chapter. If you do not have access to the Microsoft Platform SDK, you can visit the SDK online using the shortcut URL http://WindowsInstallerTraining.com/msisdk. This URL has been set up because the URLs for MSDN online are long, cryptic, and frequently move (as the MSI SDK recently did!).
>
> You can also install the SDK over the Web if you want to get the tools and the documentation in Help file format. Visit http://www.microsoft.com/msdownload/platformsdk/sdkupdate/ and click Windows Installer SDK on the left navigation bar.

Application Management Meta Data

As we've already seen and discussed, the Windows Installer technology has many valuable new features such as self healing, improved uninstalls, and customization capabilities. A key element to enabling these new features is recording and referencing information that tracks how software applications should be installed. This information can be thought of as application management *meta data*, that is, data that references or describes other data.

There are two distinct storage areas for management meta data about Windows Installer packages. The first of these locations is in an MSI package file. The internal database in this file stores all the information required to install a software application. The second location in which Windows Installer package data is stored is the Windows Installer (MSI) repository on each computer. The Windows Installer repository is made up of a database within the registry and some cached files on the hard disk.

The Windows Installer repository gives Windows Installer intelligence when performing installations on demand and when self-healing. This information describes to Windows Installer which files, registry keys, and other configuration changes must be installed for an application to work correctly.

The meta data stored in the MSI repository contains a pointer back to the original MSI file. This pointer is generally used to retrieve source files. The data contained in the repository (such as installed product codes, upgrade codes, and so on) is not retrieved from the MSI file; it is stored directly in the repository on client computers.

MSI File Format

Most of the files used by MSI utilize a special Microsoft file technology called *COM Structured Storage*. This storage technology basically creates multiple spaces, called *streams,* within a file. You can think of these streams as files within files, not unlike a Visio diagram embedded in a Word document.

> COM Structured Storage does not use Alternative Data Streams. Alternative Data Streams are a file–system–level technology available on NTFS systems that allows multiple data storage areas in the same physical file.

Three Streams

An MSI file usually has three streams: one for summary information, one for the MSI database, and one for storing the installation files. (Installation files can also be stored externally.) Other streams (such as the AdminProperties stream) might be created by various Windows Installer activities, but these three are the main three to start with. The many other file formats utilized by Windows Installer are generally variants on the MSI file format, such as

- .MST—Transform file
- .MSP—Patch file
- .CUB—Validation file

> Msiinfo.exe is a Windows Installer SDK tool that allows the summary information stream to be queried and updated from the command line.

The Database

The Windows Installer database stream contains the fundamental information required to perform the installation of the software application. Only items inside the database can be customized using transforms. Transforms are essentially database overlays that are used at installation time.

The Windows Installer database is normalized. In database language, this means that to represent any given entity (say all the information to install a file) there may be several linked tables involved.

> Although you will periodically need to view a table, a complete understanding of every nuance of the relationships between these tables should not be necessary if you are investing in good Windows Installer authoring tools.

There are two important types of information that are contained within the database:

- Information about the software application to be installed. This information includes which files, registry keys, and shortcuts should be installed. It also includes information about how the developer organized the package within the Windows Installer rules. Following the Windows Installer rules for package structures results in the software application being represented as Features and Components, which we will be discussing in a bit.
- Information about the actual execution of the package. Figure 3.1 illustrates that the package execution logic is stored in the database along with the tables describing the software application. Windows Installer was not designed as a monolithic script processing engine that can only have a list of files and registry keys fed to it. Instead, many of the subroutines within Windows Installer are configurable by the package developer. A package developer can configure whether the subroutine is called at all and what order it is called in, and the developer can apply if-then statements to these subroutines to have them run only if certain conditions are true.

When the term *software application* is used throughout this book, it refers specifically to the actual files and registry keys built by developers that a Windows Installer package is designed to deploy.

It is important to understand that most of the package processing logic is in the database because this allows it to be customized. As would be expected, an administrator can customize which files and registry keys are copied during a package, however, the administrator can also customize the original package logic built-in by the vendor. This capability provides an unprecedented level of customization of vendor-provided software.

Windows Installer actually has a small subset of SQL within it. It is used for package processing, and can be used to retrieve and write data to and from the database tables. To learn more, consult the Windows Installer SDK document "Examples of Database Queries Using SQL and Script." A Windows Installer SDK script called wirunsql.vbs allows you to easily run arbitrary SQL commands on an MSI file.

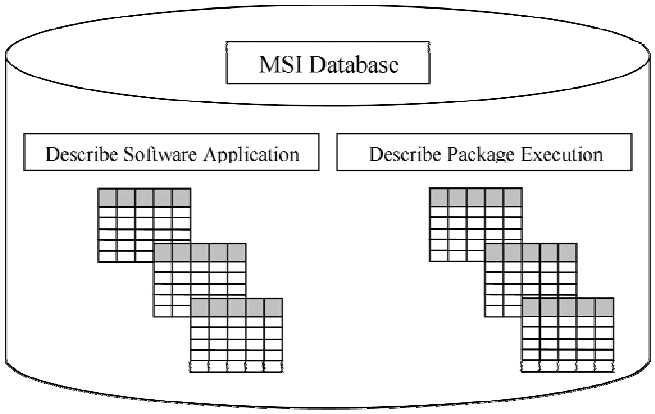

Figure 3.1: MSI database tables.

"Open" File Format

One of the greatest strengths to the MSI file format is that it is "open." This does not mean that the MSI format is tied to the increasingly popular open systems movement; rather there is a specific standard for the format, and it uses existing, well-documented Microsoft file structures. Microsoft provides the APIs to read, open, and change these files. Because no particular tool vendor owns or controls the format, many tools can read and write the same MSI files. Additionally, a package created by a software application vendor is able to be opened by any IT professional. This openness allows administrators to customize vendor packages and gives vendors the flexibility to allow customization.

Package developers can still insert custom functionality and proprietary approaches through the use of custom actions and custom tables—an open file format does not force package developers to expose everything they must do to ensure their software is installed and properly licensed.

How Packages Describe Software Applications and Installation Procedures

Windows Installer logically describes software application and installation procedures with the relational database mentioned earlier. The following section attempts to describe this schema in skeletal detail as a way of providing enough information to proceed on to more advanced topics. Every package engineer and administrator will have widely varying needs for more detailed study of this topic based on their individual experiences and company requirements for package building and troubleshooting.

Your key to mastering Windows Installer is to understand its language. The essence of this language is the framework provided by Windows Installers management meta data. The concepts you learn in this section will continually crop up in the following areas:

- Windows event logs
- Windows Installer logs
- Authoring tools
- The Windows Installer SDK
- Application deployment kits (such as the Office 2000 Custom Installation Wizard)
- Windows Installer command lines
- Group Policy

Software Application Information

Some of the tables in an MSI file store data about how the software application is structured. There are tables that deal with files, registry entries, INI file entries and shortcuts. Windows Installer also introduces a schema that describes the internals of the software application to Windows Installer. This schema defines two main logical entities known as Features and Components. Features and Components are the fundamental constructs that organize all the configuration details of a software application that is installed by the package.

The handy term *component* has been severely overused in the technology industry. When used in the context of Windows Installer, the term *Component* has a very specific meaning. The term *COM Component* refers to compiled executable software that is registered in the Windows registry so that it can be located by many different programs. To confuse matters more, most COM Components will have a dedicated Windows Installer Component to define them in Windows Installer.

Previous setup technologies did not have a way for the OS to know the details of how elements of software relate. (For more information about the benefits of Windows Installer compared with early application management technology, see the sidebar "Application Management Before Windows Installer.") The developer might know that three registry keys, four DLLs, and two INI settings are required for the database view feature to work, but there has not been a way to encode this *management meta data* in the packaging technology or the OS to facilitate intelligent application management.

Application Management Before Windows Installer

Long before Windows Installer, several innovative companies built intelligent application management technology for Windows that included self-healing and other benefits. Understandably, these technologies are expensive and heavily proprietary—sometimes taking a framework approach that requires usage of proprietary distribution mechanisms to take advantage of the packaging engines. Windows Installer has advantages over these approaches in that: it is free, it decouples distribution from packaging (which allows flexibility when building deployment solutions from different technologies), it generally makes packages more resilient for use in many deployment scenarios, and it is owned by Microsoft (which means all newer versions of Windows ship with Windows Installer).

Identification in Windows Installer

For many administrators, this section might be your first encounter with a programming concept known as a Globally Unique Identifier. GUIDs pre-date Windows Installer and have been used in many areas of programming as a result of their ability to create unique identities. You may have seen them in the registry in the CLSID subkey or HKEY_CLASSES_ROOT.

GUIDs are used throughout Windows Installer to identify most elements of a software package. A GUID is a 128-bit integer ("2 to the power of 128" possible values). GUIDs allow *unique identities* to be assigned to objects by many independent developers without a requirement for central coordination.

To understand how GUIDs work, think of 10,000 administrators using a packaging tool to generate 100 product codes each. None of the 1,000,000 products codes would be the same. GUID generation uses a special algorithm with many different seed values to ensure an extremely low probability of identical GUIDs being generated.

Here are some of the package elements that GUIDs are used to identify in Windows Installer:

- Package files
- Products
- Components
- Patch files

GUIDs are utilized directly during many Windows Installer activities. For instance, you might want to trigger a reinstall of an installed package and have the target computer determine from where the original package file was sourced. You can do so using the following command line, which uses the /f switch and product GUID to perform a re-install:

```
MSIEXEC /f {869A369E-6BD5-42e1-B9E9-B3543A46D5F6}
```

Component Structure and Attributes

As Figure 3.2 illustrates, Windows Installer Components are the fundamental unit that define the functionality of the software application. Components can have many types of associated resources. Some resource types include files, registry keys, shortcuts, and INI file settings. Some new attributes that are specific to Windows Installer can also be a part of a Component, including entry points, keypaths, and Component Codes. Although a Component can contain these items, it is not required to contain all of them.

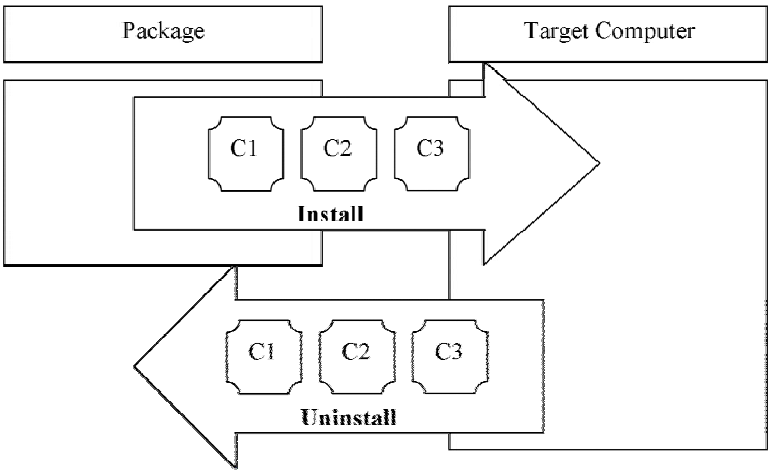

Figure 3.2: Windows Installer operates on lists of Components.

Components are the fundamental unit that Windows Installer manages. Any operation such as installation, maintenance installation, self-healing, and uninstall result in a list of Components that must be operated on to achieve the desired result. Components are also reference counted to prevent uninstallation when more than one application is using a shared piece of software. (For more information about reference counts, see the sidebar "A Brief History of Reference Counts.")

A Brief History of Reference Counts

Reference counts (refcounts) were introduced with Windows 95. All installation programs that follow Microsoft's installation guidelines increment a counter in the registry whenever a DLL is installed to a shared location, such as the system directory.

For example, if four applications had installed abc.dll to the System32 directory, that DLL would have a refcount of 4 in the registry. If one of the applications is uninstalled (again assuming installation guidelines are followed), the uninstall program would simply change the refcount to 3 and leave the file in place because other applications are obviously using the DLL. If the refcount for a file is 1, the uninstall program is free to remove the file because it can assume that the program being uninstalled is the only program using the file. Occasionally, uninstall programs will break other software because they remove shared registry keys required for a DLL to work properly. Windows Installer improves on refcounts by putting them at the Component level. Because a Component contains all the various system resources required for a DLL to operate properly, these related resources will remain on the system if other software is still using the DLL.

Component Name

As Figure 3.3 shows, Components have a friendly name. This friendly name displays in most authoring tools. The friendly name, however, is not how a Component is ultimately identified. A Component is identified by its Component Code. Component names make the processes of authoring and updating packages easier so that we do not have to remember 128-bit hexadecimal integers.

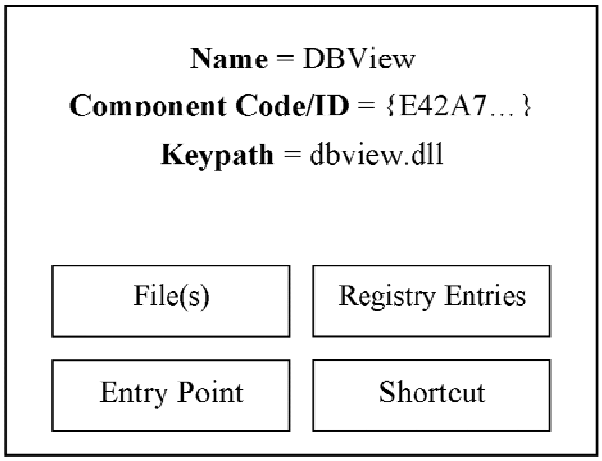

Figure 3.3: Component definition.

Component Codes

Component Codes (or Component IDs) are the identifying attribute for a Component. Component Codes are GUIDs that uniquely identify a Component across the world. In theory, a Component Code should be unique among all Components in the world.

> 🕮 For more detailed information about Component structure and identity rules, refer to the Windows Installer SDK document titled "Organizing Applications Into Components" and its sub-documents.

Keypaths

Through the concept of a Component, Windows Installer uses meta data to model a functional unit of the application software it is describing. This Component definition is placed in the repository of any machine on to which it is installed. However, if the Component becomes broken, how does Windows Installer tell that the Component is not installed as defined in the repository? This is where the keypath comes in.

For each Component that is installed on a computer, Windows Installer checks the existence of a specially tagged resource (known as a keypath) within the Component to determine whether the Component is healthy or in need of repair. If this tagged resource is missing, the entire Component is re-installed. A keypath can be a directory, a file, a registry key, or an ODBC data source.

The reason that Microsoft Word still works when winword.exe is deleted is because winword.exe is the keypath of a Windows Installer Component. A computer with Office XP installed would have a Component definition in its Windows Installer repository that describes winword.exe. When a user attempts to use Microsoft Word, Windows Installer checks to see that the keypath of this Component (winword.exe) exists. If it does not exist, self-healing would be invoked to fix the problem. There are other details to how self-healing works that will be covered later.

Among the uses of a keypath, three are very relevant to administrators:

- Self-healing detection
- Advertising/Install-on-Demand detection
- User profile fix-up detection (special case of self healing)

> ☞ Although not a formal term, *user profile fix-up* refers to a lesser known feature of Windows Installer that lets installed packages properly set up a user profile when the user has not previously used the application. This functionality works even when the user has previously logged on to the computer.
>
> When a user starts an application, standard self-healing checks are performed. If the package is structured correctly, Windows Installer will perceive the lack of user information for the application as being "broken" (even though they never existed) and self-heal the user portions of the package.

Entry Points and Advertisements

Ever wonder how Windows Installer knows to get involved with repairing or installing an application? Entry points allow Windows Installer to proxy the startup of an application and perform application management tasks before the user is allowed to access the application. In other words, when you double-click the icon for a Windows Installer packaged software application, it does not actually attempt to start the application directly. The icon is a special icon that asks Windows Installer to find the software application and start it. This is when Windows Installer can use the MSI repository information, the installed application resources (files, registry keys, and so on), and the original package file to perform the magic of self-healing and install on demand.

> For entry points to work correctly in Windows NT 4.0, you must have a newer version of the shell installed. To update the shell, install Internet Explorer (IE) 4.01, SP1 with the Active Desktop or IE 5.x with the Windows Desktop Update. You can also update the shell when deploying a customized Office XP installation. See the Office XP NT deployment Web site for more details (http://www.microsoft.com/office/ork/xp/one/depd01.htm).

An entry point turns into an *advertised interface* when any Feature that its Component belongs to is advertised or installed on a target computer system. When a Windows Installer package is advertised, advertised interfaces make it appear as though the application is installed and ready to use. When a Windows Installer package is installed, advertised interfaces trigger Windows Installer for self-healing and user profile fix-up checking. An entry point/advertised interface can be:

- A shortcut (special Windows Installer shortcut)
- A document extension (association)
- A MIME type (Internet document types)
- A Class ID (CLSID)—Programmatic identities used for sharing software within and between various applications

> For Windows Installer functionality to work as expected, users must launch applications from Windows Installer shortcuts. If users in your organization are accustomed to creating their own shortcuts by right dragging and dropping application executables, these shortcuts will not trigger self-healing or any other Windows Installer functionality. Unfortunately, it is not easy to prevent users from doing this—it will be necessary to re-culture them through Help desk interaction and other types of communication. Windows Installer shortcuts created by the installation package (on the Start menu or desktop) can be copied to new locations. Windows 2000 (Win2K) and later allow right-dragging shortcuts right out of the Start menu. One drawback is that they are not upgraded when the underlying package is upgraded, so they may not work after a major upgrade to the software application.

Advertisement of document extensions, MIME types, and CLSIDs are all accomplished by configuring the registry on the target computer; however, Windows Installer does not internally store this information as registry keys. Advertising data is stored in special tables and does not become registry entries until the package is installed on the target computer.

> ☞ When first starting with Windows Installer, it can be easy to confuse advertised interfaces with advertising an application to users. Even if you never plan to advertise applications (make them appear as installed, but they actually install on first use), you will still need advertised interfaces in your package if you require self-healing or user profile fix-up to work properly.

The following list provides a summary of information we have covered about Component structure and attributes:

- File resources—Components can contain file resources. If a file resource is the keypath to the Component, it is known as the key file. If a file is not the keypath, it is known as a companion file. There is no practical limit on the number of files or file types that can be in a Component. There are, however, rules about Component structure that define when certain types of files should have an entirely dedicated Component.
- Registry resources—Registry resources are registry keys that are required by the Component.
- Shortcut resources (entry point)—Shortcuts are defined within a Component and must point to a file *within the Component*. Shortcuts can be advertised (entry points) or standard Windows shortcuts.
- Document extension mappings and MIME types (entry points)—Document extensions and MIME types are configured at the Component level and point to a file *within the Component*.
- Additional resources and attributes—Components can have many resources and configuration items associated with them. Some of these include:
    - Controlling and installing services
    - Making INI file entries
    - Creating directories
    - Setting environment variables
    - Configuring ODBC

Typical Components

When I first heard Components described, I thought they would be something like spell check and contain 3 executables, 14 DLLs, 10 registry keys, and so on. It turns out that this type of item (spell check, for example) would be a Feature that contains multiple Components. One of the things that helped me understand Components was learning what typical Components are like, as Table 3.1, Table 3.2, and Table 3.3 illustrate.

Component Item	Typical Configuration
Keypath	The code file.
File Resources	Only the code file and any required data files.
Registry Resources	COM Registration (CLSID) keys and data keys.
Advertisements	Any registry entries, extension mappings, CLSIDs, or ODBC data sources associated with the file (if any).
Service Settings	Any service control and installation items associated with the code file.

Table 3.1: Executable Code Component (EXE, DLL, OCX).

In repackaged applications, most of the registry keys for an application may be contained in a couple of Components (one for HKEY_CURRENT_USER and one for HKEY_CURRENT_MACHINE) except if they are explicitly required for a component to operate correctly. For packages received from a software vendor, most of the registry keys may be with the primary application executable.

Component Item	Typical Configuration
Keypath	A registry key in the relevant hive that should always be present if the Component is installed.
File Resources	None.
Registry Resources	Registry keys for HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER.
Advertisements	Any registry entries, extension mappings, CLSIDs or ODBC data sources associated with the file (if any).
Service Settings	None.

Table 3.2: Registry key Component (HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER).

Package developers have more flexibility in areas such as Components that contain templates for the software application. If templates were critical to this application, each one could be a dedicated component.

Component Item	Typical Configuration
Keypath	The template directory or a single template file.
File Resources	All templates.
Registry Resources	COM Registration (CLSID) keys and data keys.
Advertisements	Any registry entries, extension mappings, CLSIDs or ODBC data sources associated with the file (if any).
Service Settings	None.

Table 3.3: Templates Component (template files for software application).

Features

After you have a basic understanding of Components, Features are quite easy to understand. Features are buckets (container objects) for Components. Features have very few attributes assigned directly to them, they are actually the sum total of the Components contained within them.

Although Features are simply buckets for Components, many of the configuration capabilities of Windows Installer operate on Features. For instance, you can advertise a Feature, but not Components. If you advertise a Feature and 3 advertised interfaces appear, you know that among the Components that make up that feature, there are 3 entry points defined. You can find out exactly which Components contain these items by examining the Components that make up the Feature. Features have some unique attributes. These include:

- Windows Installer configuration commands operate on Features (installing, advertising, uninstalling, and so on)
- Self-healing, install-on-demand and user profile fix-up (discussed in an earlier note) operate at the Feature level
- Features can contain other Features
- Features can be arranged in hierarchical relationships (by being contained by other Features)
- Features contain Components
- Multiple Features can contain the same Component
- Features are **NOT** identified by GUIDs but rather by a Feature Identifier, which is a text string

By contrast, Components do not have these attributes—they cannot contain Features or other Components, they cannot be arranged in hierarchies, and they are not addressed directly through the command line to accomplish installation and configuration activities.

Earlier, we talked about how Windows Installer essentially operates on a list of Components. We can modify this concept by understanding that we specify that list of Components to Windows Installer by using convenient buckets called Features, as Figure 3.4 illustrates.

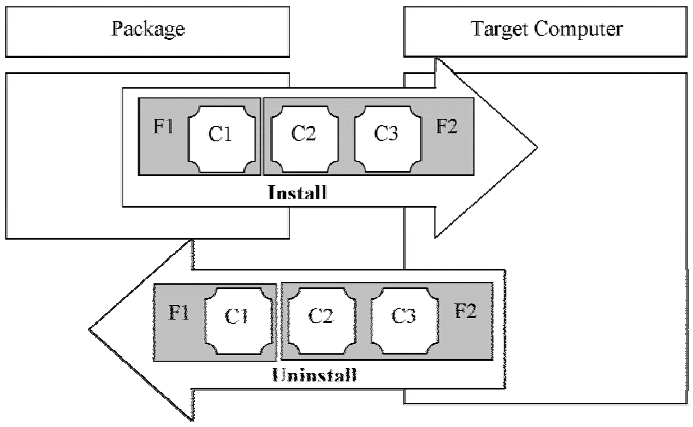

Figure 3.4: Windows Installer operates on lists of Components that are grouped by Features.

Most of the attributes assigned directly to Features are concerned with how these Features are displayed in the Feature selection dialog box presented by Windows Installer during an interactive install. Every package will have a root Feature that is always installed.

Package Execution Information

Even though the package processing engine is built into the OS, much of the engine's functionality can be controlled from within a package file. This allows administrators to customize the actual logic used to install packages, even when packages come from software vendors or in-house programmers. Previous to Windows Installer, setup program processing logic was inaccessible because it was compiled into binary executable files (EXEs) and could not be altered.

Standard Actions

As mentioned earlier, Windows Installer is not a huge block of code that simply processes a package. There are many subroutines within Windows Installer that are called during package installation, configuration, and uninstall. These subroutines are partially configurable through the Window Installer database in a package. These subroutines are called *Standard Actions*. Standard Actions can be configured in three ways, they can

- Be included or not included
- Be reordered
- Have if-then statements (conditions) placed on them to control their execution

Although Standard Actions are configurable in these ways, there are still many rules about which Standard Actions should be included as well as ordering dependencies on other Standard Actions. The SDK's Standard Action reference should be studied before attempting to reorder any of them.

For more information about rules for reordering Standard Actions, refer to the Windows Installer SDK document titled "Standard Actions Reference" and all of its sub-documents. The Window Installer SDK files also include a template for the default set of actions that would be expected in a generic package. This file is called Sequence.msi and can be found in the MSI SDK directory \Program Files\Microsoft SDK\Samples\SysMgmt\Msi\Database.

Custom Actions

Custom Actions allow package developers to extend Windows Installer with just about any functionality they desire. Custom Actions have information available to them about the running installation. Only certain types of items can be called as a Custom Action. Some of the most relevant are:

- Calling DLLs
- Calling EXEs
- Calling a VBScript
- Calling a JScript
- Setting a property

VBScript tends to be the popular choice among administrators who need to create Custom Actions primarily because VBScript can be used for many diverse administrative scripting needs. In addition, VBScript is similar to other scripting languages administrators might already use.

Setup tool vendors also allow you to use their proprietary scripting languages as Custom Actions. For example, Wise Package Studio allows compiled Wise Script to be used as a Custom Action and InstallShield allows InstallScript to be used.

Windows Installer 2.0 has new error logging features for scripted Custom Actions. Previous versions simply reported that a scripted Custom Action had failed and gave the Custom Action name. Windows Installer 2.0 (shipped with Windows XP and Win2K SP3 and is downloadable) will log the actual error and the script line number where it occurred.

Like Standard Actions, Custom Actions can have their sequence controlled and conditions placed on them.

Sequences

We have been discussing how the order of Standard Actions and Custom Actions can be controlled. Windows Installer also supports the ability to have multiple sets of ordered actions. These ordered sets are called *sequences*. Sequences help organize installations. There are two sequences involved in an interactive installation, as Figure 3.5 shows. The Install UI sequence contains all the actions (including dialog boxes) required to gather information from the user during an interactive installation. The Install Execute sequence handles changes to the system such as copying files and updating registry entries. This two-sequence approach is also used for silent installs—the entire Install UI sequence is simply skipped when an installation is run completely silent.

Silent installations are utilized heavily in automated software deployment. Most administrator-authored Custom Actions will need to be placed in the Install Execute sequence to ensure that they are executed during silent installations.

Standard packages (built according to Microsoft templates and guidelines) also have four other sequences. The Advertising UI and Advertising Execute sequences are used when a package is advertised using MSIEXEC or Group Policy deployment. The Admin UI and Admin Execute sequences are used when a package is used to build an administrative install location.

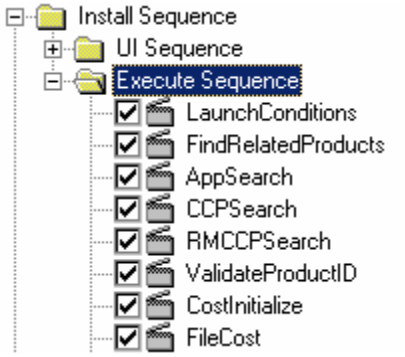

Figure 3.5: Sequences and actions.

Uninstalls and maintenance installs are handled by the Install UI and Install Execute sequences. When specific actions are only relevant to a specific install type, such as uninstall, conditions are used to ensure that those actions only execute when appropriate.

Authoring tools will represent sequences in different ways, but essentially they are interpreting a table that simply has the action name and an associated sequence number. There is a separate table for each sequence. Although a rare occurrence, package developers can create their own custom sequences if desired.

Properties

Windows Installer uses Properties to store package data before and during package processing. They are the equivalent of a variable in a scripting or programming language. Properties are similar to environment variables. As Figure 3.6 illustrates, environment variables provide system information (such as computer name and OS). They can also be used to store data in batch file scripts. For instance, a script might prompt the user to choose a menu item—an environment variable could be used to store that choice for later use.

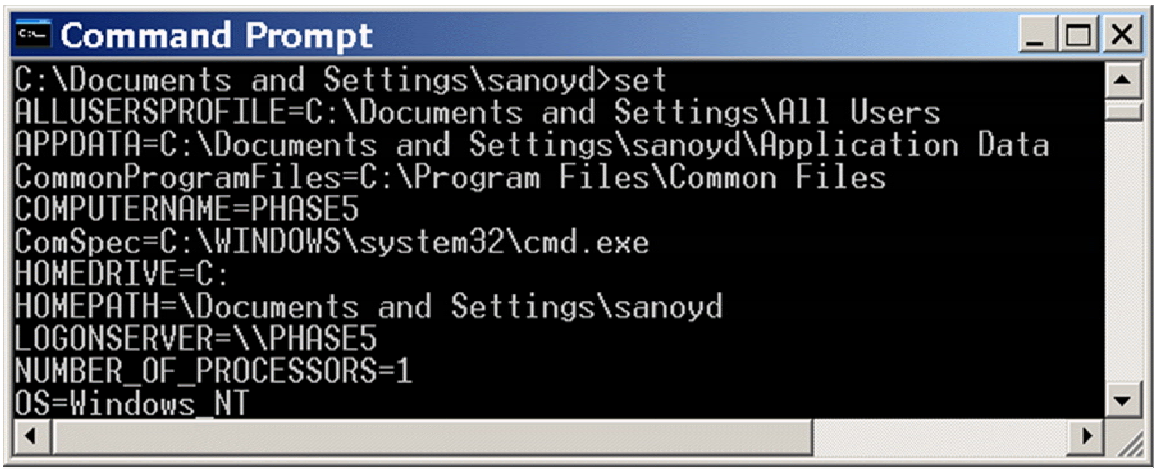

Figure 3.6: Set command output.

Properties behave like environment variables and scripting variables in other ways as well:

- Properties do not have data types, they can store numeric or string data
- Properties do not need to be declared before use—they can be created on the command line, in transforms or by custom actions
- Properties are used to store data about the system

Properties are used store all kinds of data and control parameters. They store data and control parameters such as:

- Installation progress
- Data collected by locator tables (such as registry keys)
- Type of installation activity (such as install, uninstall, rollback, and so on)
- Data about the target system (such as OS version and user profile location)
- Current date and time
- Control information for installation activities (such as the list of features to install or advertise)

Properties can be created on the fly, so do not assume that the property table in a package is a comprehensive list of all properties used or created by the package.

Properties have several classes that determine how they can be manipulated during package operations. The class of a property is determined by the text case of the property name and whether it is in the SecureCustomProperties property or one of the built-in Restricted Public Properties.

- Private Properties can only be changed by transforms and custom actions—they cannot be changed on the command line. Private Properties must have at least one lowercase letter.

- Public Properties can be changed on the command line or in the installation UI in addition to transforms and Custom Actions. Public Properties must contain only upper-case characters.
- Restricted Public Properties can only be changed by administrators or if the EnableUserControl policy is turned on. Restricted Public Properties must contain only upper-case characters AND be either on the list of built-in Restricted Public Properties or added to the SecureCustomProperties property if they are a custom property.

Any of these property types can be built into Windows Installer (known as default) or defined by the developer (known as custom).

Properties are also used by MSIEXEC as command-line arguments. This can be a little hard to get used to because MSIEXEC also uses switches that start with the forward slash character. The following command line shows that applying a transform during package installation is done using the TRANSFORMS property rather than a special command-line switch:

```
MSIEXEC /I package.msi TRANSFORMS=custom.mst
```

In this example, the /I is an MSIEXEC switch and TRANSFORMS is a property.

> ☞ When starting out with Windows Installer, it is important to familiarize yourself with all the built-in properties and the information they communicate or the functions they control. Consider reading through all the information in the "Properties" section of the Windows Installer SDK as a good primer.

Notable Properties

There are several notable properties that will be used many, many times. Most of them control how a package is installed:

- TRANSFORMS—Specifies a list of transforms to apply to an MSI during package installation.
- ADDLOCAL—Lists features to install on the local computer.
- ALLUSERS—Controls whether installations are performed for all users of the computer or just the user running the installation.
- ROOTDRIVE—Controls which drive Windows Installer installs packages on—by default packages are installed on the local drive that has the most free space.
- INSTALLDIR—Controls the exact directory to which a package must be installed.
- REBOOT—Controls whether the package requests a reboot after installation.

> 🕮 When properties are specified in multiple places, Windows Installer has a method for determining which value should be used. Examine the MSI SDK document titled "Order of Property Precedence" for more information.

Self-Healing Overview

Self-healing is the ability of Windows Installer to detect and repair any critical resources that are required for the user to successfully launch and use the application. Every resource of a package is not checked during self-healing. Because self-healing occurs as the application is launched, exhaustive checking of every resource would lead to excessive wait times.

Earlier we discussed how Windows Installer performs basic actions (install, uninstall, and so on) on lists of Components. We also discussed how these lists of Components were specified by a list of Features. Self-healing follows this approach as well.

Self-healing, install-on-demand, and user profile fix-up are all variations on the same functionality provided by Windows Installer. Windows Installer is asked to find the appropriate software application when an entry point is activated by a user (usually double-clicking a shortcut or document type). If Windows Installer finds the software is not yet installed, it will immediately install it. If the software is installed, it will be verified by self-healing. In both cases, this happens at the Feature level.

As Figure 3.7 illustrates, when an entry point is activated, the Component to which the entry point belongs is checked for which Feature it is attached to. *Every* component in that Feature is checked for non-existence of the keypaths. If *any* single keypath is missing, the entire feature is reinstalled.

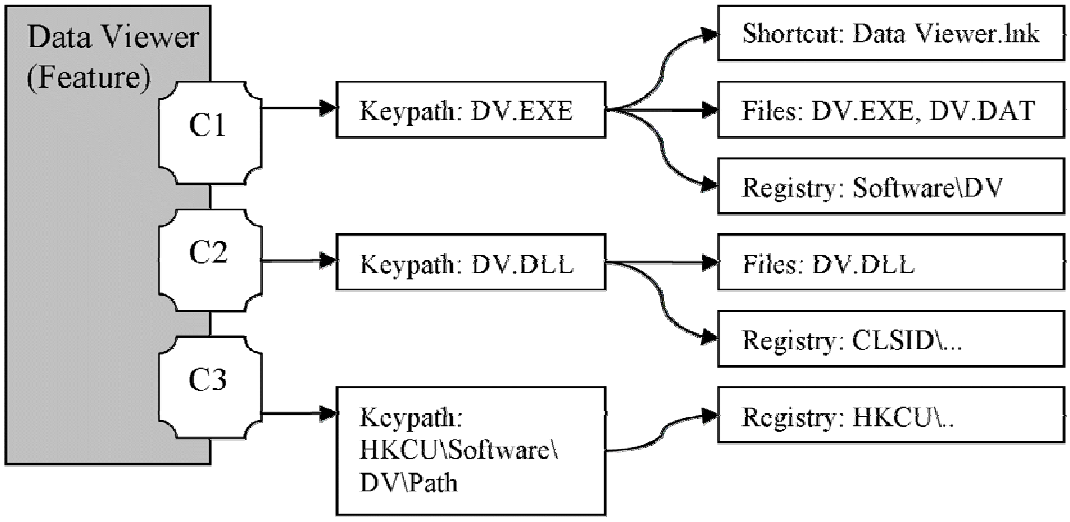

Figure 3.7: Self-healing component structure.

For example, say the Component in Figure 3.7 was installed on a computer. After a couple of months, someone accidentally deletes the file DV.DLL. The next time the user launched the shortcut Data Viewer.lnk, the files DV.EXE, DV.DLL, and the registry key HKEY_CURRENT_USER\Software\DV\Path would be checked for existence. If any of these three resources were missing, the entire Feature (which is made up of the Components C1, C2, and C3) would be reinstalled. This is why self-healing results in much more installation activity than a single component re-installation.

> ☞ Self-healing will not repair resources (mainly files and registry keys) if the keypath of the component they belong to is properly installed on the system. For example, if DV.DAT in Figure 3.7 was missing, it would not be self-healed if DV.EXE was present on the system. To compensate for this, users can be taught to use the Repair option in Add/Remove Programs. This option does a full re-install of the application and will fix problems with missing resources that are not fixed by self-healing.

Summary of Package Structure Concepts

Windows Installer introduces an entire level of application management meta data that is fundamental to creating the many new features and capabilities Windows Installer is famous for. Although it is not a simple task to become familiar with the structure, rules, and terminology of this meta data, doing so unlocks many secrets!

Here are some of the highlights:

- Windows Installer describes software applications using a set of database tables.
- Major aspects of how the package is processed are also described in this database.
- To accommodate the new paradigm for installations, new logical entities are defined by Windows Installer to break down the software application into manageable sub-parts.
- These logical entities are known as Components and Features. Components and Features allow Windows Installer to map the relationships between specific software application resources (such as files and registry entries) for use in management activities such as self-healing, sharing of application resources, and install-on-demand.
- Windows Installer defines additional entities for managing package processing—these entities, known as Actions and Sequences, control the behavior of an installation package while it is performing installation, configuration, and uninstall activities.
- The design of Windows Installer allows package developers to have a large degree of control over Windows Installer's internal functions. This same design allows administrators the same level of control even after a software vendor has built its completed installation package—something not possible with previous setup technologies.
- Variables in Windows Installer are known as properties; they store all types of control information and data for packages. Custom properties can be created by package developers.

Figure 3.8 illustrates how all the internals of a package are utilized to accomplish installations. The numbers correspond to the following discussion.

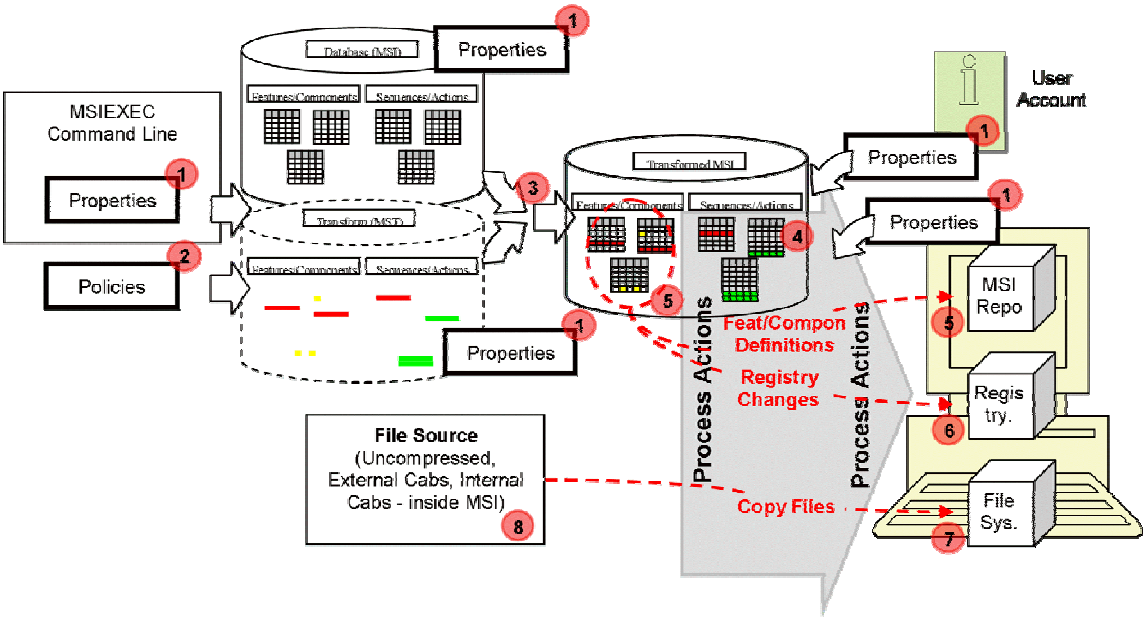

Figure 3.8: Package processing internals.

Properties (1) are a dominant element because they are used to control installations, gather information from the target computer and user account, and for custom functionality. Policies (2) (which are not stored as properties) are read by Windows Installer as needed; some are enforced before any package processing begins. Policies control many behaviors of Windows Installer—they are covered later in this chapter. If transforms exist, they are read and applied to the package file (3). Windows Installer processes the actions contained in the relevant sequence(s) to install the package (4). Package processing causes the package Features and Components to be installed (5). This step includes copying the Feature and Component definitions into the target computer's MSI repository. When Features and Components are processed, all changes are made to the target system, including creating registry entries (6) and copying files (7). Files are copied from the source (8), which can be stored as uncompressed, compressed CAB files, or internal CAB files (inside the MSI file).

Customizing Packages

One of the most powerful benefits of Windows Installer is the ability for administrators to customize installation packages regardless of who built them. Previous to Windows Installer, most software could only be effectively customized through the use of manual installation or repackaging. The following list highlights some of the difficulties in software deployment that result from the inability to customize software installation packages:

- Manual installation of software has very high cost.
- User installation of software creates higher costs due to misconfiguration and end user self-support.
- Repackaging introduces quality risks due to incorrectly installed software applications.
- Repackaging might violate some software vendor's support agreements.
- Enterprise-wide repackaging creates additional costs and requires disciplined processes.

Fortunately, Windows Installer has been designed with these challenges in mind. The primary method for customizing an installation package is known as a *transform*. Transforms are a separate file with an .MST extension. They are specified during the installation of a package.

> 🕮 We briefly explored transforms in Chapter 1.

After our entire installation is modeled in a database, customization is easily accomplished by adding, modifying, or eliminating database rows and cells from various tables. Eliminating a Component from an installation can simply require removal of three rows from three tables and a change to the value of one cell in a row of a fourth table. To add a Feature with two existing Components might only require new rows in two tables.

Transforms use a concept called *overlay* to accomplish customization. Instead of permanently changing a package file's database, overlays are done using a temporary copy of the database created during installation. This allows for many different customizations to be done from the same MSI file on disk because customizations can be picked at install time. Overlays are an extremely flexible method of customization because more than one transform can be used at once.

Transform files are *deltas*, which cannot be used standalone because they only contain the changes you want to make to an MSI file. Anytime you want to create, edit, or apply an MST file, the MSI file it is based on is required to work with the transform.

Because transforms can include changes to the package logic, they cannot be applied to a package that is already installed on the workstation, but must be specified with the initial package installation or advertisement.

Figure 3.9 shows that a transform is just a delta of information that requires the original package to form a complete customized installation. It also demonstrates the overlay concept, whereby loading the transform on top of the MSI file gives the complete picture.

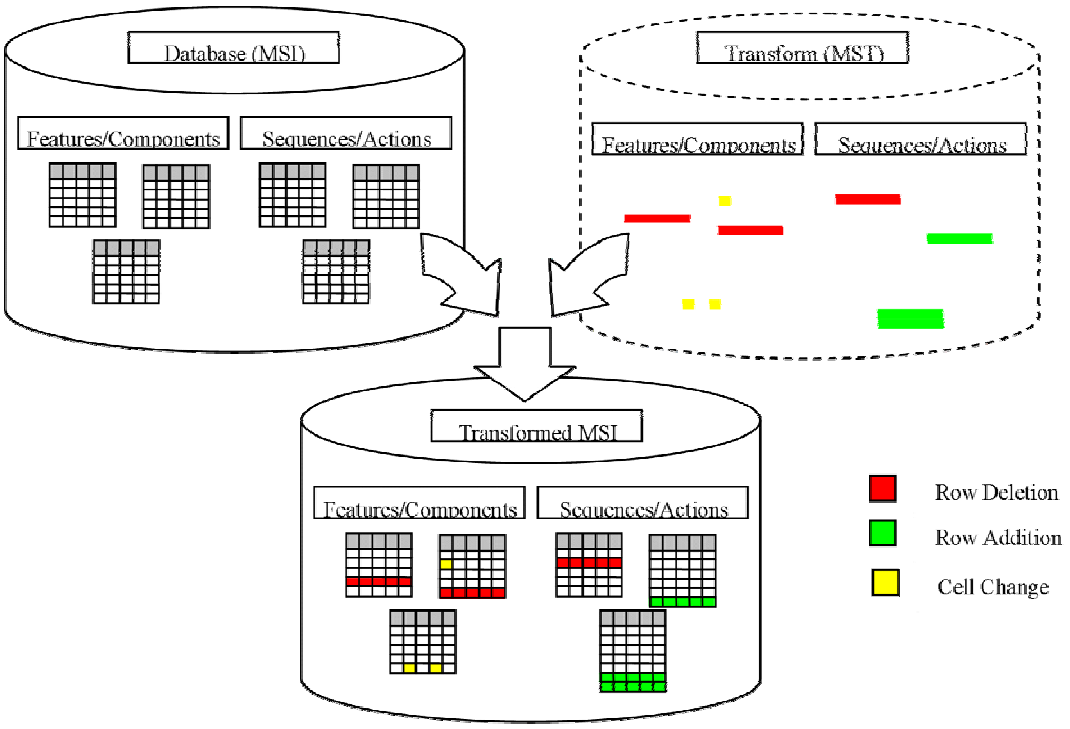

Figure 3.9: How transforms work.

When transforms are applied to a package installation, they are copied locally into a cache. These cached copies are applied to any subsequent reconfigurations of the application so that customizations stay intact.

For packages built by administrators, transforms might be less useful because customizations can be integrated directly into the MSI packages. However, in large enterprises and for repackaged installs that have many possible configurations, transforms are an effective means of customizing repackaged software.

Transforms should be used for customizing all MSI packages received from software vendors—vendor MSI packages should not be directly edited. This is not simply a best practice, but an expectation and assumption of software vendors, Microsoft, and the Windows Installer SDK.

Managed Application Settings

Windows Installer was released with the suite of Win2K technologies known as IntelliMirror. Group Policy is the companion technology that provides deployment and application settings management by way of dynamic policy settings. Compared with Windows Installer, Group Policy provides superior capabilities for managing application settings because they are applied when users log on and at regular intervals afterward.

Since the introduction of Win2K, many organizations have been hampered in deploying Active Directory (AD). Those who implement it rarely burden the directory with exhaustive settings management for all applications in the enterprise. This raises some challenges for package deployment with regard to settings that must be actively managed.

When an application setting is made in a package file or transform, the setting usually consists of one or more registry entries. After the software is installed, Windows Installer will ensure that the same settings are made if self-healing is required or if a new user logs on and uses the application.

A problem arises when one of these registry keys needs to be changed to a different value. Previous to Windows Installer, most administrators would run a simple script to fix up the registry keys on all existing machines. This can still be done, but it leaves out several important scenarios that Group Policies would catch:

- Some self-healing scenarios can set the application setting to the older value contained in the package file.
- New installations of the package after the fix has been set will use the older value.
- Multi-user machines will not have the older value for all existing users.
- New user logons to multi-user machines will have the older value.

To handle these situations purely with Windows Installer technology would require that the application setting be updated in a transform. Either all computers would need to uninstall and re-install the software, or an upgrade package would need to be created and deployed. The later only works if the package is not from a software vendor because you should not create upgrade packages for MSI packages received from a software vendor.

The importance of catching every one of the exceptions is relative to how critical the fix is and how it affects specific user communities. If the problem is blue screening computers on a stock trading floor, it is essential to eliminate any possibility that the old setting is put on any computer. If it creates a minor annoyance to users, it might not be as critical to prevent every case of the old setting being installed. Without some type of policies mechanism, there is no clear path for how to handle this issue, but it is important to be knowledgeable of it and discuss it early in the design of application management and packaging processes.

This is more of a hack than a tip. Windows Installer does not validate that a cached transform file is the exact same file that was used during the original install. For computer-based installations, the location of these files is easy to determine. Administrators can replace the cached transform with an updated copy and simply trigger a maintenance installation or re-install to change application settings *after* deployment. However, there must have been a transform deployed with the original installation for this hack to work.

If an organization has policy setting capabilities of any type—such as NT or Windows 9x System Policies or third-party policy management systems—this problem can be resolved by using the policy mechanism. To prevent overburdening the policy mechanism with application settings, it is prudent to only use it for settings that absolutely must be managed.

Creating Transforms for Application Settings

There are essentially three common types of transform-creation tools. In the previous chapters, we discussed vendor-supplied tools and third-party tools that step through the user installation interface and essentially automate the choices a user would make during a package installation.

In many cases, administrators will need to automate settings that are not exposed using these tools. Detailed settings such as the application data directory or back-end database server are usually only exposed in the most advanced customization tools such as the Office Custom Installation Wizard.

Tier-1 packaging tools generally include a lower-level tool for creating transforms. With this type of tool, the package developer loads the MSI to be customized, then uses the authoring tool as though editing the MSI itself. All the capabilities of editing an MSI are available, but the changes are saved in a transform and the original MSI is left changed.

Although these tools are very powerful, they do not assist in discovering the system changes (registry and INI files) that equate to configuration changes performed from within the software application. A configuration monitoring tool must be used to actually discover the required settings. You might be able to use the repackaging portion of your MSI packaging tool or select any tool that can effectively monitor and report system changes.

> Wise Package Studio Professional has a very helpful HTML-formatted "change detection report" that is automatically created in the same directory as the package. The report does not automatically display, so you have to know its there to use it. This report is a great source of information for finding which registry keys and INI settings were changed while configuring a software application.

Here is a method for creating transforms for advanced settings using the tools and skills you are already familiar with:

1. Start up the software application you want to customize.
2. Start your configuration change monitoring tool (such as your repackager).
3. Proceed to make the desired configuration changes to your software application.
4. Ensure that these changes are "committed". This step may require some experimentation due to the differences in how software applications are programmed. Many applications will save configuration changes when you click Apply or OK in the configuration dialog box. Keep an eye out for software applications that delay the saving of configuration changes until a later time or until you exit the application.
5. Stop your change monitoring tool.
6. Examine the output of the change monitoring tool.
7. If your transform tool allows importing of registry data directly from the computer on which it is running, open the transform tool and create a new transform. Use the output from the configuration monitoring tool to determine which registry entries to copy from the current workstation into the transform.

The last step in this procedure is counterintuitive to many packaging processes. This is because extremely clean packaging processes usually call for the package editing tool to be run on a separate computer from the repackaging tool due to the changes made to a workstation by the package editing tool. In this case, the package editing tool is being used to copy only selected items from the repackaging workstation, not to generate the initial package as it is with repackaging.

There are several interesting transform tools and scripts in the Windows Installer SDK:

Wigenxfm.vbs and Msitran.exe can generate a transform by comparing two MSI files. For scripting, this is the only way to actually create a transform.

Wiusexfm.vbs and Msitran.exe can permanently apply the contents of a transform to a database.

Wilstxfm.vbs lists the contents of a transform in a command window.

Using Transforms

Transforms must be specified during installation or advertisement of an application. The switches used for each of these scenarios is quite different. In Chapter 1, we briefly explored how to apply transforms to an existing MSI package. To review, here is the command line for applying a transform when installing a package:

```
MSIEXEC /i mypackage.msi TRANSFORMS=mycust.mst
```

The special public property TRANSFORMS causes the transform to be applied. Here is the command line for applying a transform when advertising a package:

```
MSIEXEC /j[u,m] mypackage.msi /t mycust.mst
```

For advertising, a special switch and sub-switch are used for transforms. The /j switch indicates that the package will be advertised. The /j switch is directly followed by a *u* for a user-based advertisement or an *m* for a computer-based advertisement. The /t switch is a sub-switch of the /j switch, and can only be used in conjunction with the /j switch.

It can be easy to get confused and attempt to use the /t switch with the /i switch.

Administrative Installs

Chapter 1 briefly discussed administrative installs. In this section, we will dive a little deeper. Administrative installs are not really installations of a package, but rather a special preparation of your package to allow it to be installed from a network. A client installation must still be done for each computer that needs to run the software application.

The following is a list of the main uses for an administrative installation. Knowing this list will help you understand whether they can play a useful role in your environment.

- Pre-setting properties with the AdminProperties property—With normal installations, double-clicking an MSI file results in the full installation interface and no special command-line properties are applied. This can result in leaving out the TRANSFORMS property, which might cause important customizations to be omitted. Administrative installations allow the administrator to specify a list of properties (*not* MSIEXEC switches) to be used when the MSI is double-clicked. This is useful if you have distribution scenarios in which users are either directed to run MSIs from the network or they can easily find them on the network.
- Served applications—Windows Installer natively supports running applications executables from a server rather than from the local hard drive. This is known as *Run from source* in Windows Installer. To do this, an administrative installation must be created. A unique aspect about Windows Installer served applications is that they can be fault tolerant. If multiple administrative installations are made available and specified as sources for the applications files, Windows Installer will check the list of sources until it finds one that is available.
- Pre-activation of Microsoft products—Microsoft Products that require activation can only be pre-activated if they are setup as an administrative install. There are other ways to ensure that activation does not require user intervention.
- Reduce back-end replication—In large networks, the amount of data passed over the network to distribution servers can cause network load problems. Because administrative installs can be patched, using administrative installs can reduce back-end (server to server) replication. Patches have the potential to dramatically reduce the amount of data transfer required because they only contain binary deltas of files that change between two versions of a package. The benefits of reduced bandwidth for maintenance must be balanced with the uncompressed format of administrative installs—which use up to twice as much disk space as an MSI that has the application files compressed within internal or external CAB files.
- Extract only needed files from a software CD-ROM—Many software vendors send out a single, large CD-ROM that has all or most of their software packages—especially if their software is under site or blanket licenses. Some CD-ROMs may contain files required to deploy applications in multiple spoken languages. Such a CD-ROM might contain more than 400MB when all that is needed is a 10MB application. Performing an administrative installation with the desired MSI file will extract only the needed files to the network. This approach only works if the individual applications on the CD-ROM have their own MSI files rather than one large MSI file.

> ☞ Administrative installations are also required to create patches. Most authoring tools will automatically create the administrative installations for you if they do not exist before you start the patch tool.

Building and Using Administrative Installs

During an administrative install, the package is prepared to be installed from the network. Figure 3.10 illustrate that this process extracts all files into a directory structure and copies the MSI (without embedded files) to the root of the administrative install location. If used, the ADMINPROPERTIES value is also embedded as an additional stream at this time. An administrative install share is created by using the /a command-line switch with MSIEXEC. When performing an administrative install, there is usually only one dialog box requesting a network location for the install. Windows Installer does not check whether the location is actually on the network, so this location can be local if you are simply testing a package. Here are a couple samples command lines for setting up administrative install shares:

- `MSIEXEC /a mypackage.msi`

  Prepares the package in the directory specified on the wizard dialog box that appears after this command line is run.

- `MSIEXEC /a mypackage.msi ADMINPROPERTIES = ”TRANSFORMS=mytrans.mst”`

  Prepares the package in the specified directory and embeds the special property ADMINPROPERTIES to be used upon client installation.

- `MSIEXEC /a mypackage.msi /p myfix.msp`

  Applies a patch to an existing administrative installation.

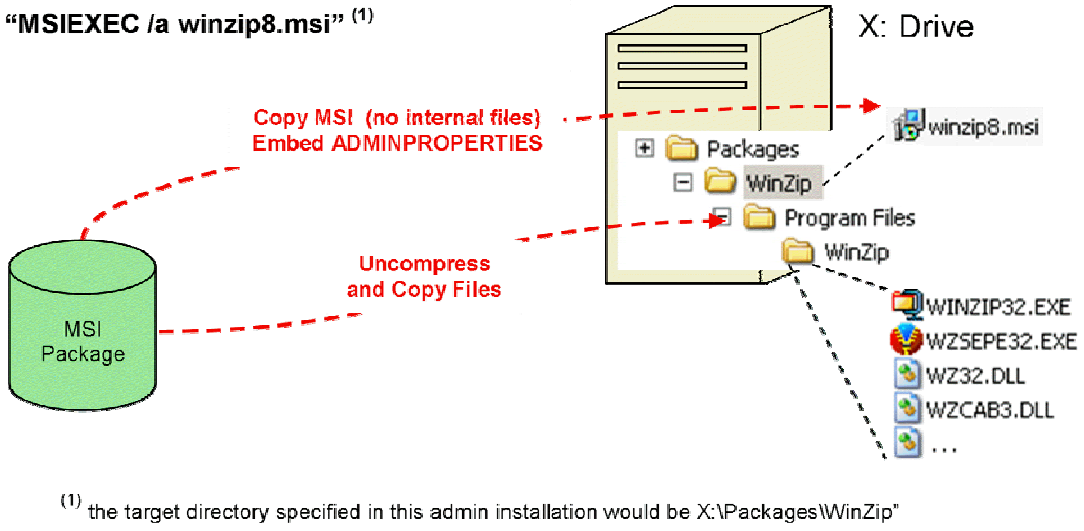

Figure 3.10: Creating an administrative install.

Installing from an Administrative Share

Client installations operate the same as installing from any other installation source. Clients install from the administrative share using the standard command line or by double-clicking the MSI file located on the administrative share, as Figure 3.11 illustrates.

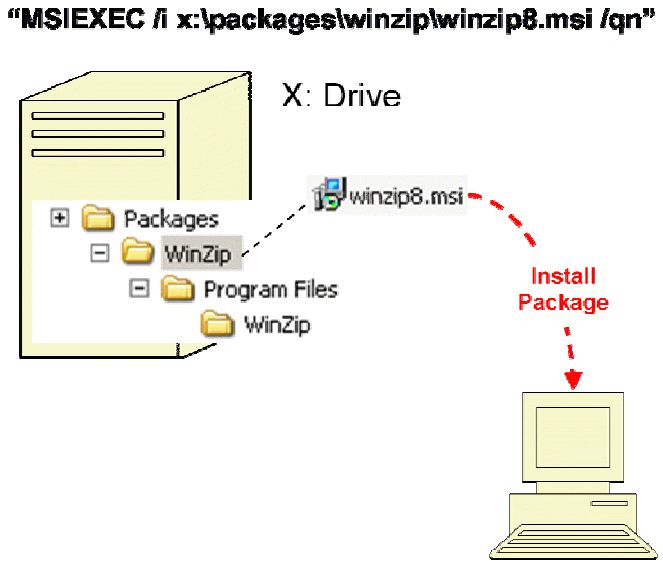

Figure 3.11: Client install from an administrative install.

Serving Applications

Over time, there have been many terms used for the concept of leaving the software application files on the server and having the client execute them from there. For our discussion, we will refer to this as *served applications*. One of the most notable uses for served applications is for implementations that require high availability. Having an application installed on multiple servers allows for fault tolerance when a server fails. Windows Installer supports fault-tolerant served applications.

> Some packagers attempt to build Windows Installer packages for legacy served applications without using Windows Installer's native support for served applications. The legacy approach is to point icons to *existing* binaries located on the network. However, Windows Installer does not allow a shortcut to point to files that are not contained in the current package. A popular workaround is to copy traditional .LNK shortcuts to clients. Although this approach works to a limited degree, the shortcuts will not trigger any Windows Installer activities such as self-healing and install-on-demand. The native support must be used to avoid extensive workarounds and enable the full Windows Installer feature set.

Figure 3.12 shows two key properties used to configure Windows Installer packages for fault-tolerant served applications. The ADDSOURCE property causes the Windows Installer shortcuts to look for the application files at the administrative install location. ADDSOURCE takes a list of features as its value, the special value ALL indicates that all features should remain on the server. The SOURCELIST parameter causes the package installation to include a list of additional locations at which the software application files can be found.

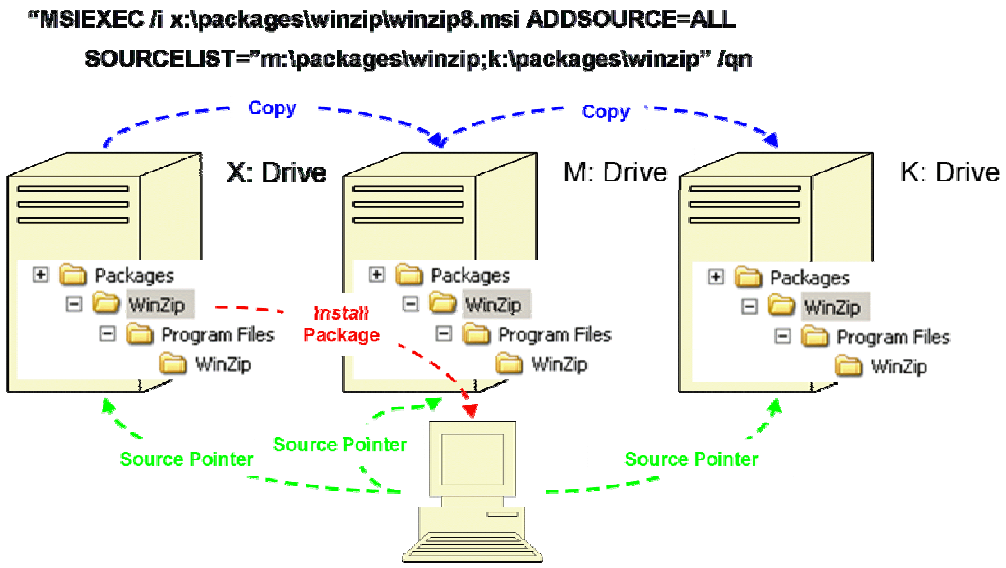

Figure 3.12: Served application configuration (fault tolerant).

Here is the basic process for setting up fault-tolerant served applications:

8. Create an administrative installation of the application.
9. Replicate the administrative installation to multiple locations or perform additional administrative installations with the EXACT same package (package codes should match).
10. Install clients using the ADDLOCAL and SOURCELIST properties.

Windows Installer does not perform load balancing between the various sources for the application. If this is desired, a load-balancing file system technology such as Win2K's Distributed File System (DFS) should be used. Manual load balancing can be accomplished by ensuring that an equal number of clients perform the initial install from each server location.

> 🕮 Chapter 5 will contain a more in-depth look at Windows Installer source lists.

Security and Policies

Windows Installer security and policies is an area of great interest to administrators. Security and policies give some of the flexibility required to design an application deployment approach that is secure from viruses and end-user abuse. Proper attention to Windows Installer security and policies helps address the following significant risks:

- Viruses that take advantage of MSI security capabilities
- Security exploits by users, administrators, or hackers
- Unauthorized software installs on corporate machines
- Software piracy

This section will focus on key policies in Windows Installer and new Windows XP policies for controlling which applications can be installed. As a point of clarification, this section discusses how to configure Windows Installer service settings using policies, not how to deploy Windows Installer packages using Group Policy-based application deployment.

> If you work in a very large organization, it is important to consider that Help desk and first-level administrators might have the technical savvy and physical access needed to abuse elevated privileges. Protecting against these types of exploits requires a different perspective than just having to consider internal end users and external hackers.

Windows Installer Policies

As with most technologies introduced with Win2K, Windows Installer is configurable through policies. However, unlike many Win2K Group Policies, Windows Installer security policies are registry-based. In practical terms this means that AD and special policy processing agents are not required to manage these polices. Any mechanism currently used to mass deploy registry tweaks can be used to effectively configure MSI policies. This includes initial computer build, distribution of .REG files, third-party policy management systems (such as those provided by NetWare), and Windows 9x and NT System Policies.

> An updated System Policy template (.ADM file) is available for download at WindowsInstallerTraining.com. This .ADM file includes two new MSI 2.0 policies as well as the debug policy. This updated policy file can be downloaded from http://windowsinstallertraining.com/msiebook.

The following discussion will focus on the essential Windows Installer policies that should be considered by administrators. These policies generally deal with securing Windows Installer's elevated privileges capabilities.

> For an exhaustive list of Windows Installer Policies, refer to the Windows Installer SDK document titled "System Policy" and all its sub-documents.

Elevated Privileges Implementation

There are some basic concepts of elevated privileges that should be understood before diving into the policies that configure them. Whenever an MSI package is installed, an instance of MSIEXEC.EXE is started in the user's context. This occurs, as Figure 3.13 shows, whether the package is started by double-clicking an MSI or if MSIEXEC.EXE is called via a batch file, logon script, or software distribution system.

> We will be discussing elevated privileges and software distribution systems in more detail in Chapter 5.

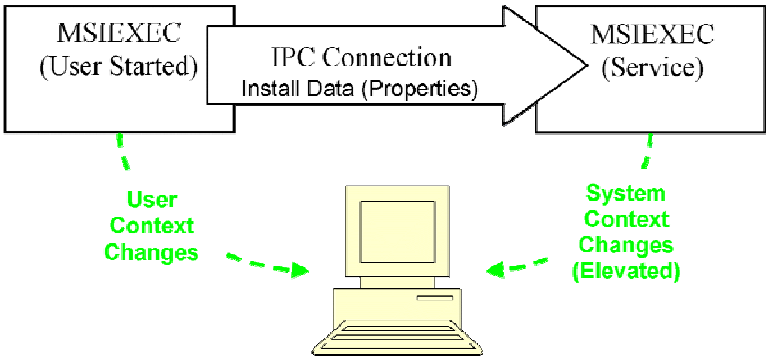

Figure 3.13: Elevated privileges implementation.

An elevated installation is one that uses administrative rights for a portion of the installation. If elevated privileges are requested and approved, an inter-process communication occurs between the instance of msiexec.exe that is started in the user context and the instance running as a Windows service. If elevated privileges are granted, the security rights of the system account are utilized for the activities performed by the service. Windows Installer enforces strict rules about the data that is allowed to cross the IPC connection and what types of commands can be performed on the service instance of MSIEXEC.EXE. This approach is more secure than the user context switching approach provided by tools such as the NT Switch User utility or Windows XP's RunAs functionality.

> It might be tempting to change the account used by the Windows Installer service as a method of preventing abuse of the System Account. This is likely to create difficulties for your installations and should be unnecessary given the built-in and policy-based security controls in MSI.

Managed Applications

Windows Installer gives selected applications Managed Application status depending on how they are installed. Packages that come from any of the following sources are considered Managed:

- Assigned through Group Policy to users (Advertised) or computers (Full Install)
- Assigned using the MSIEXEC command line by an account that has local administrative privileges on the target computer (Advertised or Installed).
- Deployed through SMS 2003 (beta name was Topaz)

Managed Application status gives a software installation elevated privileges during the initial installation and for all subsequent installer operations such as self-heal, install-on-demand, maintenance installs (adding/removing features), and uninstalls. That is to say, packages that are tagged as Managed on a specific computer continue to have elevated privileges for subsequent installation activities on that computer. These elevated privileges continue to operate independent of the original reason that granted the package Managed status.

☞ Unlike traditional setup.exe installers, the Windows Installer engine is not only used during initial installation of a package. The Windows Installer engine is active in all phases of the application management lifecycle, including deployment, installation, configuration (adding/removing portions of a software application), self-healing, upgrades, and uninstalls.

Always Install with Elevated Privileges (AlwaysInstallElevated) Policy

The AlwaysInstallElevated policy is the most permissive configuration of elevated privileges and should be used sparingly. This policy must be set to 1 (Enabled) for the computer AND the user to be completely enabled. This policy allows all packages and installation activities to occur with elevated privileges regardless of their source or the user account that starts them. This policy is intended to permit all installation activities to complete normally for non-administrative users (as they would under Windows 9x) but do so without giving away local administrators rights that grant many more capabilities than application installation.

Managed Application status is NOT given by using the AlwaysInstallElevated policy settings. If packages are installed with this policy turned on, and the policy is subsequently turned off, subsequent install activities are limited by user rights. This can hamper self-healing, application upgrades, and uninstalls.

AlwaysInstallElevated Hacking

Some organizations have used the two AlwaysInstallElevated keys as a method of programmatically controlling elevated privileges. Under this approach, security on these policy keys is configured to allow them to be changed by a wrapper script. The wrapper script will toggle the policy keys on, perform an MSI installation, then turn them off. Although this functionality is convenient, it has a couple downsides that should be taken into account. First, this approach might cause problems for self-healing or when the user attempts to reconfigure the application through Add/Remove Programs because the user will no longer have administrative rights to perform installation activities. Second, security exploits and viruses generally test for "security by ignorance" techniques such as these. There are probably valid scenarios in which using this method is acceptable—just make sure you are aware of the risks if you are considering it.

Disable Windows Installer (DisableMSI) Policy

The DisableMSI policy has three settings:

- 0 (Default) = Always Enabled
- 1 = For Non-Managed Packages
- 2 = Always Disabled

The value 0 means MSI is always enabled. The value 2 means that it is always disabled. There are very few circumstances in which completely disabling MSI is desirable. The value 1 restricts package installs to only be allowed from three sources: Group Policy, SMS 2003, or assignment by an administrator.

The *For Non-Managed Packages* value is usually of interest to organizations that want to restrict users from installing unauthorized software packages. This can be an effective approach for LAN-based environments, but it does create limiting situations for offline package deployment. If you have the luxury of deploying Windows XP you might want to consider software restriction policies (which will be discussed shortly).

> ☞ The Disable MSI policy overrides the more permissive AlwaysInstallElevated policy. If both are turned on, AlwaysInstallElevated is effectively disabled.

Cache Transforms in Secure Location on Workstation (TransformsSecure)

Whenever transforms are used for an installation, Windows Installer caches them on the local computer. This allows them to be applied to all subsequent installation activities. If a transform can be replaced by an end-user or IT personnel, their copy will be executed during any subsequent installation activities. If the application has Managed Application status, replacing cached transforms can allow malicious code to take advantage of local administrative rights.

For packages that are installed for users, transforms are cached in the user profile to support roaming profiles. When the TransformsSecure policy is used, it ensures that transforms are always cached in a secure location regardless of whether a user or computer installation is performed. For security sensitive implementations, this policy should be enabled.

Other Security-Oriented Policies

Most of the remaining security-oriented policies have their most restrictive setting by default. If an organization is deploying a new version of Windows, deploying software distribution or re-engineering application management, a full study of the security focused policies of Windows Installer will provide the background necessary to make wise design choices.

Non-Security Policies

There are several useful non-security policies in Windows Installer. The following section discusses these policies.

Excess Recovery Options

There are two policies that deal with how Windows Installer ensures that failed installation changes are backed out completely. Windows Installer has built-in support called rollback. This support is built-in to Windows Installer and works on all versions of Windows. Windows Installer also interfaces with system restore services on OS versions that have system restore (Windows XP and ME). When system restore is present, Windows Installer requests a restore point before performing installation activities.

There is one key difference between these two recovery technologies: The native rollback support is only used during an installation; if an installation completes normally, all roll back data is deleted. System restore allows the system to be arbitrarily returned to any restore point that is still in the system restore cache—this could be days after an installation.

For many organizations, having both of these options active just consumes extra disk space and extends package processing time. For production use, Windows Installer's built-in support should generally be left on. Windows Installer's usage of system restore could be turned off if desired. Each organization should test install and uninstall times with system restore both on and off and decide whether the impact is significant for typical software installation scenarios in their company.

Windows Installer's use of system restore is disabled using the LimitSystemRestoreCheckpointing computer policy. Setting it to 1 prevents Windows Installer from requesting a system restore checkpoint during installations.

The LimitSystemRestoreCheckpointing policy only affects Windows Installer's usage of system restore. System restore will continue to be leveraged by the OS for all other non-Windows Installer activities.

Windows Installer rollback is disabled using the DisableRollback policy. It is configurable for both the computer or user—setting it to 1 in either location will cause rollback to be disabled.

There is one situation in which you might want to disable both Windows Installer rollback and system restore for package installations. In some large scale deployments of Windows, an extra hour of workstation build time can be a critical cost and project management factor. In deployment scenarios in which computers are formatted and rebuilt, turning off these policies can reduce build time. Because a failed workstation build can be easily restarted, there are no risks to eliminating rollback capabilities.

Logging Policy

Windows Installer always logs information to the Windows event logs. In many cases, this information is sufficient for routine problem analysis. If more detailed data is required, the logging policy can provide it. To say that Windows Installer logging is exhaustive would be an understatement.

The SDK document titled "Event Logging" lists all the messages that Windows Installer might record in the event log.

The logging policy is used to cause Windows Installer to create log files for all of its activities. Although the command-line logging options trigger logging for a specific package installation, the policy covers all installation activities, including self-healing, maintenance installs, and so on. There are 11 single-character switches that can be used to configure logging. Each of them logs specific types of information about the installation. When troubleshooting difficult packaging problems, it is a good idea to put the log in complete verbose mode so that no helpful information is missed.

When configuring verbose logging, the 11 switches can be arranged to spell voicewarmup—this is an easy way to remember the switches, and they can be entered directly in this order.

When the logging policy is used to configure logging, no file location can be specified. All Windows Installer log file names have the following naming convention:

```
"MSI<randomcharacters>.LOG"
```

For user-initiated installs, the log is placed in the user's TEMP directory. For automated installs (such as GPO deployment), the log is written to the system TEMP directory.

> 🕮 We will be discussing more details about logging in Chapter 4.

Software Restriction Policies

Software restriction policies are a new addition for Windows XP and .NET Server. Software restriction policies can enable or prevent execution of many types of files in Windows, including .MSIs and .MSTs. Because these policies are processed before Windows Installer is started, they are a very effective way of preventing unauthorized software installations. Software restriction policies are not a complete substitute for managing Windows Installer policies.

> 🕮 Microsoft has a good white paper that summarizes software restriction policies at http://www.microsoft.com/windowsxp/pro/techinfo/administration/restrictionpolicies/default.asp.

Software restriction policies have four types of rules, discussed in the following sections. Each of these has different implementation considerations when used with Windows Installer.

Certificate Rules

Certificate rules allow restriction of software installations by requiring that MSI files and MST files are *code signed* with the specified certificate. If they are not signed, Windows will not allow them to be passed to Windows Installer for processing. Code signing is extremely powerful, but the following considerations should be taken into account when considering its usage:

- Administrative installs can change structure of the MSI file, so code signing must occur after the administrative install is made. In addition, the "master" administrative install needs to be replicated to preserve the code signing.
- Vendors might code sign their own installations. Removal of vendor code signing can cause problems if the vendor validates their own signing. The vendor's certificate can be added to your software restriction policies if need be.
- Any changes to the package require that it be re-signed.
- Signing certificates are usually accessible by a very few people in the IT organization, which can inadvertently become a bottleneck to the packaging process if a large volume of packages and transforms are expected.

Hash Rules

Hash rules are very similar to certificate rules, except that hash rules do not alter the original file and they do not require a certificate to generate the cryptographic key used by the policies. Hashes can make it easier for administrators to restrict MSI execution without the elaboration of certificates and they may be just as effective at preventing users from installing unauthorized software. The MD5 hashes required for this type of restriction can be easily generated within the Group Policy interface. Hash rules would have the same limitations as certificate rules, except for the possible process bottlenecking. Hash rules would also leave vendor signed packages unchanged.

Path Rules

Path rules allow restriction of software installations by requiring that MSIs and MSTs run only from specific path locations. At first, this sounds limiting, however, path rules can be defined using wildcard characters, environment variables, and DFS share names, making this rule type very flexible. Here are some planning considerations if path rules sound like they will work for you:

- The repository strategy must be well defined to ensure that paths are consistent.
- A strategy for offline installs must be worked out to ensure that it fits with the use of path rules.
- Path rules that are too flexible may allow users or administrators to create a path that mimics the path rule and execute their own package from that location.

Zone Rules

Zone rules are only used for MSI files. They permit or restrict browser-based software installations from occurring based on the Internet zones in IE. The default zones include Internet, Intranet, Restricted Sites, Trusted Sites, and My Computer. These rules can be helpful for building a Web-based, self-service installation system.

Combining Rules

Multiple rules of all four types can be used in combination to create fine-grained control over software installations. Rules that are the most specific to the file being assessed take precedence over rules that are more general.

If you are using Windows XP with Win2K domain controllers, you must load the Windows Server Administration Tools from the Windows .NET Server CD-ROM onto a Windows XP workstation to configure software restriction policies in AD.

Summary

This chapter has laid the foundation for delving into the next level of Windows Installer technology. In addition to covering the basics of the internal structure of a package, we brought out some unique ways of building and utilizing transforms, administrative installs, and policies. Hopefully, the techniques you have learned will help you build more effective and secure packages.

In the next chapter, we will be discussing best practices for building packages. Get set to learn about repackaging, upgrades, and building processes!

Chapter 4: Best Practices for Building Packages

by Darwin Sanoy

In Chapter 3, we laid the groundwork for a better understanding of the internals of a Windows Installer package. Understanding the internal structures of a package and how packages are processed is critical to building packages. I highly recommended that you read Chapter 3 before reading this chapter.

> This book is focused on managed environments because administrators work in managed environments. A backdrop of managed environments has two major implications for the topic matter: Managed environments imply the engineering of practices and processes to ensure that work is done in a manner that is repeatable and high quality. In addition, a managed environment is contrary to the Windows Installer SDK, which must assume the worst case for package deployment (the worst case being a completely unmanaged target environment for packages).

This chapter is roughly separated into two main sections. The first section discusses practical best practices that are generally applicable to administrative package developers in all types of organizations. The second section focuses on critical concepts required for formulating your own best practice in areas that depend heavily on your company's approach to application integration, desktop computing support, and how IT is paid for in your company.

If you are brand new to Windows Installer packaging or need to brush up on the practical nuts and bolts of building a package, there are several good sources for package-building tutorials. One is provided in the SDK, using the Orca editor in the SDK. Others are provided on Microsoft's Web site and in the Help files of popular authoring tools.

> The Windows Installer SDK contains a tutorial—look for the document "Windows Installer Examples" and all subdocuments. There is also a WinINSTALL LE tutorial on Microsoft's site.

Best Practices Formulation

Building best practice is always challenging. It involves the artful mixing of the best practices, rules, processes, and limitations of the underlying technologies with organizational technology management objectives and business value drivers (for example, reduced TCO) to yield company-specific standardized best practices and processes.

The more complex the technology, the more difficult it can be to formulate best practice. This chapter intends to give some guideposts and recommendations to get started in formulating your company's practices and processes. Some of these suggestions will be broad reaching and high-level. Some of them will be more practical. If used as a starting point for your best practices, all of them have the potential to save many dollars and many hours of rework. Figure 4.1 illustrates the combination of technical factors and company specific concerns to yield viable best practices.

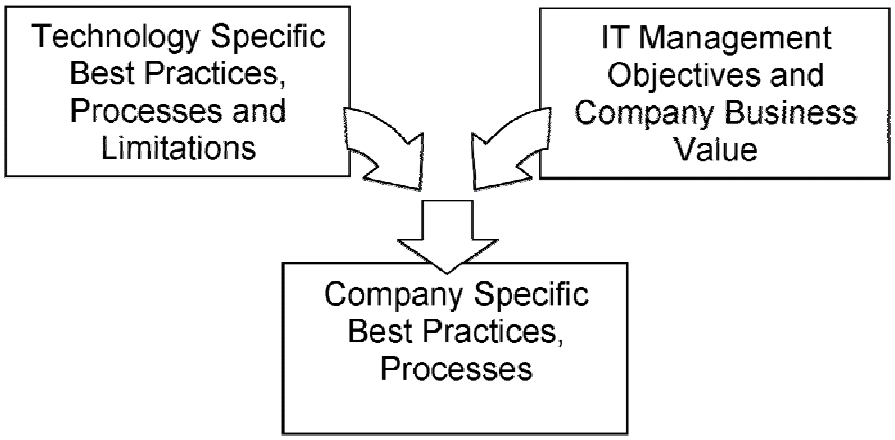

Figure 4.1: Best practice formulation.

For best practice formulation, the Windows Installer SDK is the guiding light. Like any technical document, the SDK makes assumptions about its audience and the environment in which they are working. The SDK does not preclude managed environments or the idea of administrators building packages; however, it lacks qualifying perspectives for helping administrators build packages for managed environments. These assumptions can lead to rules and regulations that require significant effort but yield nominal gains for administrators. In this chapter, we will examine some of the underlying assumptions of the SDK—particularly because these assumptions can create angst for administrators attempting to build best practices and processes for their organizations.

Analyzing the assumptions of the SDK is done by asking "What problem is this feature/function/rule meant to address?" Most of the answers for the Windows Installer SDK have the perspective of building commercial software for potential global distribution by software developers with the possibility of tailoring by administrators. This perspective assumes an unmanaged environment—that there is no repackaging and that administrators do not build packages.

Best Practice Is Not Optional

There was a time when application installation was so simple that best practices could be formulated as you discovered and engineered around problems. In some cases, requirements were simple enough that best practices were less crucial. In many cases, projects are just too rushed or technology professionals are not accustomed to formulating and following best practices. The complexity of Windows Installer requires best practices to be formulated and followed to ensure high-quality packaging. There is a good chance for packaging difficulties for any company that ignores best practice when packaging in Windows Installer. Unlike previous packaging technologies, this prediction will likely hold true for all types of IT environments, even if they are small or simple.

> This chapter assumes that you are using a tier-1 Windows Installer authoring tool for administrators. These tools intelligently use a default initial package structure that saves significant effort for administrators who do not need to learn every nuance of basic package structure before becoming productive in building Windows Installer packages.

Darwin's Law of Technology Sophistication

You are probably wondering *What is the big deal, it's just packaging technology?!* Over time I have formulated the following law that helps me understand the answer to that question: Increased technical sophistication results in increased complexity.

Any technology that makes a leap in sophistication also makes a leap in complexity. In other words, to give life to functionality such as self-healing, install on demand, and transforms requires many more gears, belts, and pulleys working in the background. In the case of Windows Installer, the additional complexity is managed by package developers, while the end user experience becomes simpler.

The essence of the new complexity in Windows Installer is the application management meta data, such as features and components, that must be built correctly for packages to be processed as expected. Microsoft has made this meta data extremely flexible to suit many types of customizations; however, there are strict rules that are base assumptions of the entire technology. The Windows Installer service, which will process your packages, lives by these rules religiously—if you do not know and follow them, your packages might behave in ways that you did not intend.

> In Chapter 3, I mentioned that some companies have built packaging and deployment tools that possess much of the functionality now contained in Windows Installer. Professionals who work with these technologies must also deal with more complexity to gain the sophisticated features these technologies promise.

Repackaging Best Practice Recommendations

Repackaging has a fairly long history in technology terms. The need for repackaging was realized around the time that Microsoft released Windows 95. For most corporate environments, a company's Windows 95 deployment project represented the first attempt at an engineered and standardized desktop computing environment. Initial migrations of the entire desktop computing base required that applications be redeployed. Standardization was the mantra of these migrations, so it made sense to look at standardizing software delivery for the initial migration as well as for long-term software delivery.

Complete automation of application delivery was a very crucial factor in reducing TCO. If every application continued to be installed from the original vendor media by a technician, desktop support costs would remain very high.

It didn't take long to discover that many of the setup technologies used to package software applications could not be automated. The technologies made the age-old assumption that installations were performed by an individual sitting at the computer answering interactive prompts. Attempts at automating the interface of existing setup programs were problematic because users could interrupt the process and some automation technologies could not always identify the dialog boxes and had problems dealing with differing screen resolutions. Repackaging was conceived to solve these problems.

Essentially, repackaging attempts to monitor which changes are made to a workstation. During the years after the Windows 95 release, many applications were still 16-bit applications upgraded to be compatible with Windows 95. Most applications didn't use the registry and were fairly simple installations that in many cases could have been accomplished using file copies and creating shortcuts. As a result, these simple applications required very basic repackaging tools to be successfully deployed.

The reasons that companies have taken up repackaging have changed over time. Initially, the TCO gained through complete automation was the most important driver to repackaging. By gaining control over exact configurations and allowing deployment without human intervention, repackaging dramatically reduced the cost of application deployment. After companies had all of their applications in the same packaging technology, they quickly realized that many DLL conflict issues could be handled proactively. Because the software application's file set was now openly viewable in the same packaging engine, IT professionals could force all packages to use the same DLL version (provided that the standard version was tested for compatibility with all applications). With the advent of Windows Installer, many IT organizations are eager to obtain its TCO benefits for all software applications, even those that have not been packaged in Windows Installer by the software vendor. A significant portion of the overall Win2K TCO proposition rests on the idea that all software applications are packaged in Windows Installer.

Do Not Repackage All Types of Setup Programs

There are several software installations that should not be repackaged. It is always advisable to check readme files and installation instructions to see if they contain cautions about repackaging. The following items should not be repackaged:

- Service packs, hotfixes, and system extensions—This category includes updates to core Windows services such as Windows Media Player or DirectX. These items should not be repackaged because they make extensive system changes and perform special procedures (such as hotfix uninstall directories or binary file edits).
- Device drivers, network protocols, and system agents—All of these items do extensive enumeration of existing settings and resources on any computer before integrating themselves. Enumeration looks for existing configurations before deciding the best settings for the new installation. For instance, if you install Microsoft Office and there is not a previous HTML editor, the installation might make Word your HTML editor. However, if another HTML editor is identified, the installation might not make Word your HTML editor. Some of the installation types in this category use arbitrarily assigned identifiers (such as protocol binding numbers) to install into specific areas of the system. Arbitrarily assigned identifiers can have different values on two otherwise identical systems. A classic example is two machines that have three network protocols installed. Although they appear identical, they might have different binding numbers as a result of troubleshooting that involved de-installing and re-installing a network protocol.
- Anything that updates Windows File Protection should not be repackaged—Although Windows Installer 2.0 is capable of updating Windows File Protection, repackaging tools do not currently monitor for Windows File Protection updates. Only updates from Microsoft can change the files protected by Windows File Protection.
- Any package that comes with a deployment kit (such as the Internet Explorer Administration Kit—IEAK) is best left in its original packaging. When a vendor puts the effort in to building a deployment kit, it can indicate that there are some extensive or tricky configuration activities occurring in the setup.

Windows Installer packages from software vendors should never be repackaged. Repackaging a vendor-provided MSI package will completely restructure the package, lose all package processing logic (for example, custom actions), and make it unrecognizable to future upgrades from the software vendor.

If you determine that you should not repackage an installation, you might consider using a Windows Installer package as a *wrapper*. An MSI wrapper simply uses custom actions to run the setup.exe (and uninstall command) silently. The wrapper script from Win2K SP1 is a good place to start (subsequent service pack wrappers are a little more involved than necessary for most wrappers). MSI wrappers are only necessary if Group Policy is your only available deployment mechanism or if Windows Installer is your only available source of administrative rights during deployment. If you have a distribution system capable of running setup.exe directly, you should use it instead of an MSI wrapper to deploy the application.

The recommendation not to repackage specific items does not mean that it is impossible to repackage these types of setups. However, the cost of doing so can be quite high. To get a complete picture of the total cost of doing so, monitor the effort spent in repackaging any of these types of setups as well as any long-term support issues with the software application being repackaged.

Have a Documented Desktop Reference Configuration

A desktop reference configuration defines how the standard corporate desktop is to be built. In some organizations, these specifications are broad, telling which OS versions and virus software should be used. In other organizations, they document every setting on the base OS. There are many reasons why a documented desktop reference configuration is a good idea; however with repackaging, creating and using this reference is a vital step in ensuring that packages built on the reference configuration will be truly portable to all desktops. Reasons for using a desktop reference configuration include:

- Ensure that the changes detected by repackaging tools will be accurate for all desktops to which the package will be deployed.
- Ensure that integration testing, packaging, and deployment occur on the same assumed configuration.
- Allow large-scale, multi-division or global projects to be on the same reference configuration.

Use Clean System Reloads for Testing and Packaging

Whenever building or testing packages, ensure that the desktop is reset to the reference configuration. You can do so using a drive image product, automated build process, or by using the virtual capturing technologies built-in to repackaging tools, such as Wise Solutions' Virtual Capture and SmartMonitor or InstallShield's InstallMonitor. VMware's VMworkstation is also a very flexible solution for quick "cleaning" of packaging workstations.

Why Clean Machines?

Repackaging technology all works similarly. The basic idea is to detect which changes have been made to a system by a setup program. Historically, repackaging tools have done so using a method known as *snapshotting*. This method involved recording the state of all the existing files, registry keys, and other configuration elements on the package developer's workstation (before snapshot). The package developer would then run the setup program, install the appropriate parts of the application, and configure the application to work correctly. The repackaging snapshot tool would be run again to compare the differences between the stored before snapshot and the current state. The data collected during the snapshotting session was used to generate a package in the repackaging tool's native format.

This approach worked well for simple configurations but did not always reliably detect all the configuration items required to make the application work. For instance, if a DLL file needed by the application was already on the target system with a newer version, this file would not be detected because this file would remain the same for the before and after snapshots.

Advances in Windows APIs, starting with Win2K have allowed repackaging tools to take new approaches to discovering the changes made by setup programs. These new methods actually start the setup program as a child process and watch it as it runs. One method involves monitoring the Windows APIs on the developer workstation for file system and registry access attempts by the child process. All access to these system areas can be tracked, even when the access does not result in a change to the system.

Another approach emulates the file system and registry to the installation program. As changes are made, the emulation layer appears to make changes to the real system; however, it is actually recording the changes that the setup program is making. Figure 4.2 illustrates that these discovery methods can monitor which changes are made while they have the setup program running as a child process.

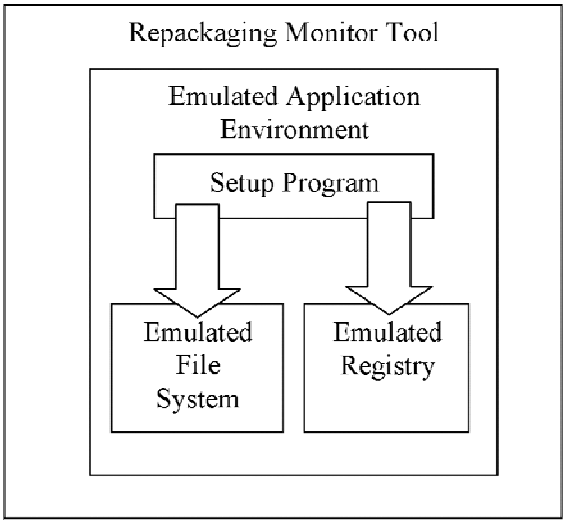

Figure 4.2: Repackaging technology monitors setup changes in real time.

These advanced methods of discovering changes are available in tier-1 authoring tools, such as Wise and InstallShield.

Repackaging tools' new discovery capabilities are a big step forward for the quality of change detection but they do not amount to complete reverse engineering capability. For instance, they cannot detect the internal logic decisions of setup programs. For example, suppose that a setup program will install 20 files and 7 registry keys only if you have Microsoft office on your computer. Repackaging tools' new capabilities will be able to accurately determine that these files and registry keys were installed, but will not report that it was the presence of Microsoft Office that caused them to be installed.

Additional Management Data for Packaging

There are two additional types of management data that may be required when fitting packaging into your application deployment approach. Unfortunately, you cannot directly extend the MSI repository data to include custom elements as you can do with Windows Management Instrumentation (WMI).

The first type is data required to ensure that all packages can be installed without human intervention. Some software applications may require the name of a local database or mail server to be completely automated. To be completely automated, the package will need to receive this data and automatically configure the software with the data. It is important to ensure that this data can be automatically sourced at the target workstation. In some cases, scripts will be able to extract the data or automatically determine the values based on other workstation data. For example, if the first three characters in the computer name indicate the site at which the computer is installed, a package could parse this information and automatically determine the appropriate database server for that site. Some of this data might need to be stored locally as a matter of initial workstation setup, then religiously updated during the change/move/retire process for workstations in your organization. Logon scripts and environment variables are also popular ways to provide this data.

The second data type is tracking and logging data that goes above and beyond what is provided by Windows Installer natively. For instance, many IT departments want to know whether ABC Software was installed by their in-house customized package or directly from the original installation media (potentially missing critical fixes or customizations).

There are many ways to store and retrieve both types of additional data. A fundamental guiding principle is to store it locally with remote accessibility and/or roll it up into inventory. Local storage with central accessibility prevents your packages from assuming that specific network resources or server connections are available during package installation. Such assumptions prevent your packages from successfully running when remote users are offline or network resources are unavailable. Here are some places that you might store this data:

- INI file—Locally accessible, possible remote accessibility, can be stored on the network if necessary
- Registry—Remotely accessible and locally accessible, easily understood (as Figure 4.3 shows)
- WMI—Remotely and locally accessible, roll-up into Microsoft SMS, uses database tables—good for advanced applications, WMI filtering in .NET allows GPO targeting based on WMI data

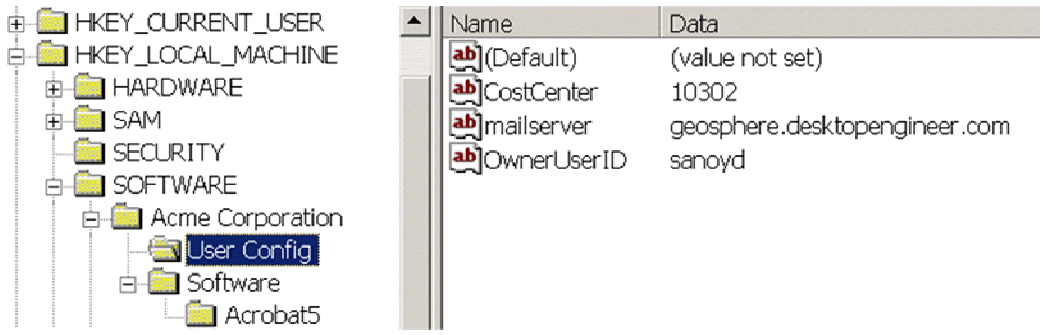

Figure 4.3: Registry storage of management data.

Windows Installer Best Practices

Windows Installer represents more than just a standardization of installation technology by Microsoft. It is a large shift in the idea of how installations happen. It has shifted the basic ideas from a script-driven model to a data-driven model. In the past, script-driven approaches were somewhat similar to writing a batch file—a set of sequential commands evaluated in order within a single process. Now, installations are built by filling in rows in many different tables within a database. In addition, the management meta data (features, components, and so on) introduced with Windows Installer is another shift in thinking about installations. These fundamental shifts enable many exciting new capabilities, but also introduce challenges to the way we think about installations.

In Chapter 3, we talked about some of the essential constructs used in Windows Installer to accomplish installations. To start becoming more familiar with Windows Installer, I recommend reading the following SDK selections. Reading specific sections of the SDK exhaustively is not meant to be a memorization exercise, rather it broadens your ideas of what Windows Installer can do and helps you make mental note of capabilities that you might need down the road.

- Properties—Read all the documents in the SDK section titled "Properties," including all the sub-documents of "About Properties," "Using Properties," and "Property Reference." You might be surprised how many properties there are for configuring packages and controlling installation behavior.
- Standard Actions—Read all the documents in the SDK section titled "Standard Actions," including all the sub-documents of "About Standard Actions," "Using Standard Actions," and "Standard Actions Reference." Make note of the information regarding how specific actions must be ordered and what you can accomplish through changes in the order.
- Policies—Read all the documents in the SDK section titled "System Policy," including all the sub-documents of "User Policies" and "Machine Policies."

Invest in Training

When taking on Windows Installer, you are learning a new way of thinking about installation activities, a new technology, and new tools. This information is a lot to absorb. It is worthwhile to seek out formal training in your preferred format to ensure the best possible experiences as you get started with Windows Installer.

> ☞ A few of the companies that offer Windows Installer training include Wise Solutions, InstallShield, and DesktopEngineer.com.

Invest in Good Tools

For many of the same reasons that training is important, you should invest in solid Windows Installer tools. These tools help in the many difficult tasks involved in administrators' jobs for packaging. Good tools also assist with the learning curve by providing reasonable default settings for tables, sequences, and so on.

Authoring tools also help prevent mistakes in situations in which complex tasks require many changes to many tables in Windows Installer. Tasks such as isolating an application can be very daunting to configure manually. Many authoring tools have integrated best practices throughout their wizards and editing interfaces. The authoring tool will issue warnings and advice when a package developer attempts to perform activities that might not produce the desired results or are in violation of Windows Installer rules.

Chapter 2 explores authoring tools. The following sections talk about these tools from the perspective of what to look for in a good authoring tool.

> ☞ If you are not convinced of the value of good authoring tools, I suggest that you perform the tutorial in the SDK that walks through building, transforming, and upgrading a package using Orca and other resource kit tools. You can search the SDK for "Windows Installer Examples" to find these tutorials.

Basic Packaging Functionality

Any authoring tool chosen for basic package building should include the following functionality:

- Repackager—The repackager should be accurate and reasonably fast. It is important that the repackaging tool allow for file and registry exclusions that can be configured by the package developer.
- MSI editor—A good editor for MSI packages should do much more than allow editing of MSI tables as Orca allows. A good editor should provide a high-level interface that ensures that all tables are properly maintained when the package developer performs functions such as "Add a Feature." In addition, every possible setting in an MSI package should be available—some tools use alternative file formats during package editing and simply compile them into the MSI file format. This approach can be extremely limiting if it leaves out MSI functionality deemed to be unnecessary for administrator-built MSI packages. For example, a repackaging tool might not allow the order of COM component registrations to be controlled; in some instances, the order of these registrations is critical for successful installation of an application because the registrations are interdependent.
- Transform tool—Transform tools should at a minimum allow for the full capability to customize the underlying MSI file. Full-featured editors take an approach of using the MSI editor but make the specified changes to an MST file instead of the MSI file itself.
- Patch creation tool—Patch creation is an involved process that can become downright overwhelming without a solid patch creation tool.

Advanced Functionality

Tier-1 tool vendors such as Wise and InstallShield provide additional value-added tools that are worth consideration. These tools usually carry a higher price tag, but if the functionality they provide is used by your organization, the productivity gains are significant. All tier-1 tools do not provide the following functionality, and the tools' features sets change frequently—be sure to consult the latest version of the available tools to determine which capabilities they offer.

- Upgrade management—When a package is being built to upgrade another package, there are specific rules that the upgrade package must follow. These rules help coordinate package structure and ensure that upgrades behave as expected. Some Windows Installer authoring suites provide tools that examine the previous version of a package and the upgrade package to give warnings, advice, and automatic fixes to ensure that upgrades go smoothly.
- Interactive debugger—When difficult logic problems arise within your package, nothing substitutes for a good debugger. Debuggers help you discover when there are logic errors in your package; generally, a value is not being set as expected. For instance, because you can create Windows Installer properties on the fly, it can be easy to mis-key a property and have critical data put into a misspelled property.

> For those familiar with the Microsoft Script Debugger, Wise Package Studio Professional interfaces with Microsoft's debugger directly. If you have the script debugger installed, the Wise debugger will step right into your VBScript custom actions and use the Microsoft Script Debugger to step through the actions one line at a time.

- Application isolation—If application isolation is a part of your application integration strategy, make sure you have a tool to assist you with this unwieldy task.
- Workgroup management—Features such as package source control (check-in/check-out), security, and centralized tool configuration are available for large teams and geographically dispersed teams.
- Workflow automation, documentation, and project management—Workflow features (such as enabling the authoring tool itself to be scripted for making standardized package edits) reduce quality problems associated with manual editing. Other workflow features include shuttling a packaging project through request and approval processes. Documentation and project management are facilitated through workflow checklists, signoffs, and project status reports.
- Conflict and package structure management—These tools are in a class unto themselves. These tools read all of your packages and provide analysis and resolution services for inter-package problems. The possible problems that can occur between packages include problems with conflicting software application resources (such as DLLs) and conflicting application management meta data (such as component and package GUIDs). If you currently have a requirement to ensure that a large body of packages integrate seamlessly, you will want to examine conflict management tools.

Peripheral Features

Some products offer features that may or may not provide substantial value to your particular packaging needs. The following list gives examples of such features. These capabilities must be analyzed on a case by case basis.

- Legacy script conversion—These tools will convert pre-Windows Installer versions of the same vendor's scripts into Windows Installer packages (for example, InstallShield setup.exe projects into InstallShield MSI projects). Some tools convert scripts from other companies' legacy packaging technology (for example, Novell ZENworks packages into Wise Windows Installer packages). In general, most packaging tools only convert the most rudimentary basics of what was in the previous packaging technology—even if it was their own technology. Files, registry keys, and shortcuts will usually be converted. Any advanced packaging logic or before-and-after procedures will generally not be converted. If you have a significant base of repackaged software in a non-Windows Installer format and do not have the original setup programs, you will most likely get higher quality results from repackaging these legacy scripts than using conversion utilities in the authoring tools.
- Package validation—Microsoft MSI and Windows logo package validation can be done using tools available in the Windows Installer SDK. If a tool provides only basic validation, the tool isn't doing anything more than running the standard validation routines and presenting the results. Some tools, however, are coming up with some innovative value-added features for validation by allowing custom validation scripts to be created, standard validation rules to be filtered, and resolution rules to be set up to correct validation problems.

- Distribution system interfacing—For the most part, tools that help deploy packages into a specific distribution system only provide the most basic job setup. In many cases, the distribution job created by this type of functionality must be customized. In addition, your company change management and/or security controls might prevent direct creation of distribution jobs by package developers.

Administrator vs. Developer Tools

Because administrators utilize Windows Installer in different ways, tier-1 vendors such as Wise and InstallShield have tool suites targeted at administrators. I have had many conversations with individuals who spend incredible amounts of time attempting to work around problems with a developer-targeted authoring tool that are easily handled by the administrator version of the same tool. If you are using the developer version of your vendor's toolset, it is important to take some time to determine whether you are losing productivity to unnecessary workarounds.

A case in point for using administrator versions of tools is how repackaging exclusions are handed in Wise for Windows Installer (developer product) and Wise Package Studio (administrator product). Wise for Windows Installer's repackager is for the convenience of a software developer in building the initial Windows Installer package. As such, it does not exclude any of the captured settings because the developer knows intimately what system elements are part of their software application. Administrators using this product find themselves doing extensive package cleanup and manually building exclusion lists. By contrast, Wise Package Studio has a good set of initial exclusions, offers advanced exclusion management, and allows exclusion of the entire base-build and changes caused by reboot. In addition, Wise Package Studio has a special repackage wizard that allows administrators to choose to include detected changes that were excluded during capture and exclude detected changes that were included during capture before they are formatted as a Windows Installer package and are much more difficult to locate.

Manage Your Windows Installer Engine Version

There are many versions of the Windows Installer engine runtimes. Generally, they all support backward compatibility fairly well. Occasionally, there are problems with packages written for a specific version. You must know the oldest version that should be supported in your environment and it is best to manage the runtimes to a specific version. As with all infrastructure technology, you should test a planned upgrade of the Windows Installer engine before deploying it broadly.

Know How Windows Installer Interacts with Other Technologies

Windows Installer is aware of and coordinates with many of the new technologies in Win2K and later. How Windows Installer interacts with these technologies might have a bearing on your packaging or application-integration strategies. These technologies include:

- Windows File Protection (WFP)—Windows Installer checks with WFP; any files that are protected are not copied.
- System restore—On OSs that support system restore (Windows ME and Windows XP), Windows Installer will request a restore point before making changes.

- Application compatibility services—Win2K and Windows XP (and Windows Server 2003) include technology in the kernel to allow applications to maintain compatibility beyond the release of Windows they were designed for. Application compatibility can be tailored by administrators for aging commercial software and in-house software. Windows Installer will check for compatibility customizations when installing software.
- Application isolation—Windows Installer packages can configure software applications to be isolated as per Microsoft's recommendations on Windows 98 and later and Win2K and later.

Configure Policies and Security

Windows Installer's elevated privileges provide some great new capabilities as well as create additional security risks. Windows Installer policies and related policies such as software restriction policies should be reviewed from two perspectives:

- What functionality can be leveraged to enforce IT policies (such as not installing software that is not on the authorized list)?
- What needs to be done to secure our environment from abuses of Windows Installer?

When formulating an approach to security, it is important that you also consider viruses that can be designed to automatically attempt many different types of exploits. Any security scheme or policy scheme that relies on the ignorance of hackers or users might not protect you against well-written viruses.

Ensure Source List Management

Almost every administrator has run straight into this problem: The classic case occurs when an MSI package was installed from a network location or CD-ROM that is no longer available. The location might be unavailable for a various reasons (for example, the location has been moved or retired, the network share names have changed, or the computer might not be connected to the network from which the software was installed). When Windows Installer attempts to self-heal an application, the user is prompted to provide the original installation source. Upgrading products that share components will sometimes cause this dialog box to display for a software package other than the one you are upgrading at the time. For example, installing a software package that integrates with Microsoft Office might require an optional feature of Microsoft Office, this, in turn, causes Windows Installer to prompt for the Microsoft Office source files because they cannot be found automatically. Adding or removing features from an existing software application can cause this prompt and in rare cases uninstalling an application may cause the prompt to appear as well.

The fundamental difficulty is that the user experience is the opposite of one of Windows Installer's core value propositions—fewer problems due to missing files. Users are possibly more confused than they were by the messages they used to receive about missing files. Now the system prompts them for a file system location that implies they should know how to resolve the situation—and understandably, many of them will attempt to resolve the problem before calling for support.

Whenever Windows Installer requires a file for self-healing or install on demand it will attempt to locate the original package file to obtain the file. The *source list* is a list of locations where Windows Installer should look for the package source files. There is one source list per package. When a needed package is not found at any location specified by its source list, the user is prompted for the missing MSI file.

Source list management benefits from a two-pronged approach. One prong is to design, build, and maintain an approach for where packages will be located. How to design a package repository will be covered in more detail in Chapter 5, but it bears emphasis when talking about source list management. The second prong is to take measures during package building to ensure that packages can be located when needed.

Repackage Existing Packages Rather than Convert Them

Many organizations have investments in legacy repackaging technology. Some have hundreds or thousands of working repackaged software applications. Many MSI tools are capable of converting scripts' built-in legacy packaging technology. For the most part, these conversion filters bring over only the most basic parts of the package. Most legacy repackaging technology has capabilities to provide customization in the form of scripting or proprietary directives configured in the package source.

An alternative approach to converting these scripts is to repackage them from the legacy repackaging technology into the new tool. Doing so will ensure that *the results* of all custom coding in the old packaging technology will be captured. Repackaging your repackaged applications will allow you to handle a significant number of the packages with less loss. There might still be cases in which the repackaging should be done from the original source media of the software vendor—especially if the legacy package was created for an older OS. In addition, you might encounter cases in which custom functionality will need to be ported from the legacy repackaging technology.

Use VBScript for Custom Actions and Other MSI Scripting

The ability to use VBScript is an excellent complimentary skill to Windows Installer packaging. Windows Installer allows custom actions (custom functionality in a package) to be coded using VBScript. VBScript is quite rich in functionality and can be extended with many scriptable components already installed on Windows and available freely on the Web. If you are working with Windows Installer 2.0 (Windows XP and later), errors in your scripted custom actions will be noted in the MSI log along with the line number of the script in which the error occurred. In addition, VBScript can be used for other types of related scripting:

- Custom package validation—For examining packages for validation checking. These can be built as custom Internal Consistency Evaluators (ICEs) or simply scripts that run in WSH.
- WMI scripts that use the MSI Provider—These scripts can be used to remotely manage packages.

WMIC is a command-line processor for WMI that is built into Windows XP. It must be used from a Windows XP workstation, but it can be used to manage any computer that runs WMI regardless of the OS. WMIC is very powerful and allows you to perform very useful management activities from the command line. For instance, the single command line that Figure 4.4 shows will inventory all Windows Installer packages on every computer listed in the file "computerlist.txt" and put the data in a comma separated values file called test.csv. It might take a two or three page VBScript to do the same operation.

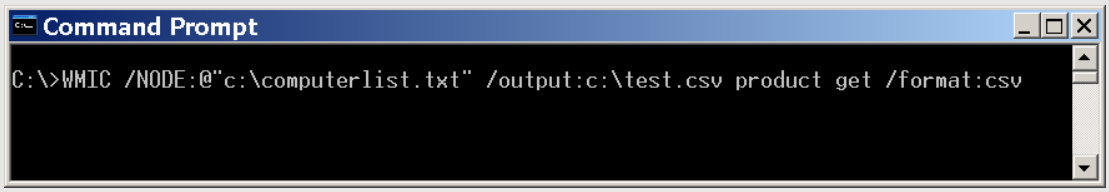

Figure 4.4: WMIC software inventory of many computers using a single command line.

- Package management—Scripts can manipulate package files directly and can be designed to operate in a batch mode that lists all packages in a directory tree and processes each one.
- Installed package/repository management—Scripts can be used to retrieve data from the MSI repository and perform installation and configuration and uninstall activities.
- Package launch—Scripts can be used to manage the launching of an MSI package. This functionality is helpful for ensuring prerequisites are available and for custom logging or reporting solutions.

If you will be doing MSI scripting outside of custom actions, there are many sample VBScripts in the SDK with full explanations in the SDK documentation. Search the SDK for "Windows Installer Scripting Samples." In addition, the SDK section "Automation Interface" details all the API calls that can be made from a script.

VBScript is also versatile for many other administrator needs such as workstation build automation, general utility scripts, Web page scripting, HTML applications, Microsoft Office automation, and so on.

Authoring tools such as Wise and InstallShield allow their legacy scripting languages to be used for custom actions. If you have a significant skill and code-base investment in these languages, you might shorten your learning curve and leverage your current skills by using those scripting languages instead. Keep in mind that these languages do not have versatility beyond packaging and they might tie you to that specific vendor's packaging tools and possibly create additional requirements for your packages.

Run Package Validation

At first, package validation appears to generate a flurry of confusing and semi-relevant information. However, as you become more familiar with the various errors and warnings, the fog begins to clear. Figure 4.5 shows the output from the SDK validation tool MsiVal2. Other authoring tools will use the same .CUB files, so their messages will be the same even if they are presented in a different interface. Package validation of all in-house and vendor packages can save significant time when problems are discovered before packages are deployed broadly. Authoring tool vendors are also putting more effort into filtering and extending package validation to make it much more useful to administrators.

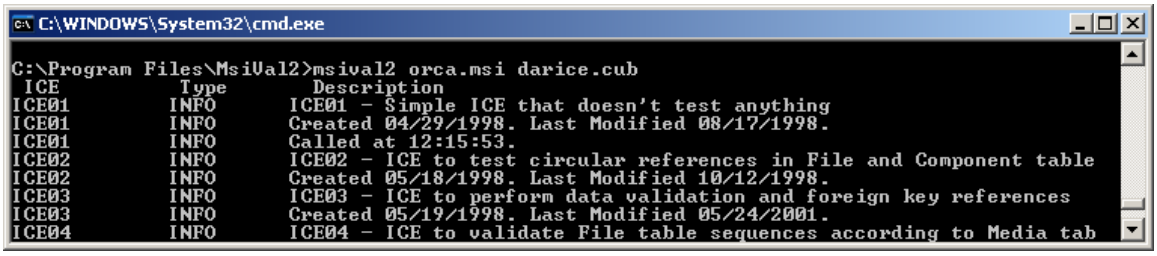

Figure 4.5: MsiVal2 package validation output.

> MsiVal2 is part of the Windows Installer SDK. It must be installed by clicking msival2.msi in the Bin directory of the SDK directories. After you install it, you must use Explorer to locate misval2.exe in …Program Files\Msival2.

Perform a Dry Run with Verbose Logging

When performing package testing, you should turn on verbose logging. There are problems that might not show up in compiling or validation that are shown clearly in a verbose log. The package might have problems with the reference platform that do not show up in testing and validation. Figure 4.6 shows the level of detail contained in a verbose log. There might also be package validation errors that are hard to interpret or find the source of—a verbose log of the same package might give additional clues as to the source of the problem.

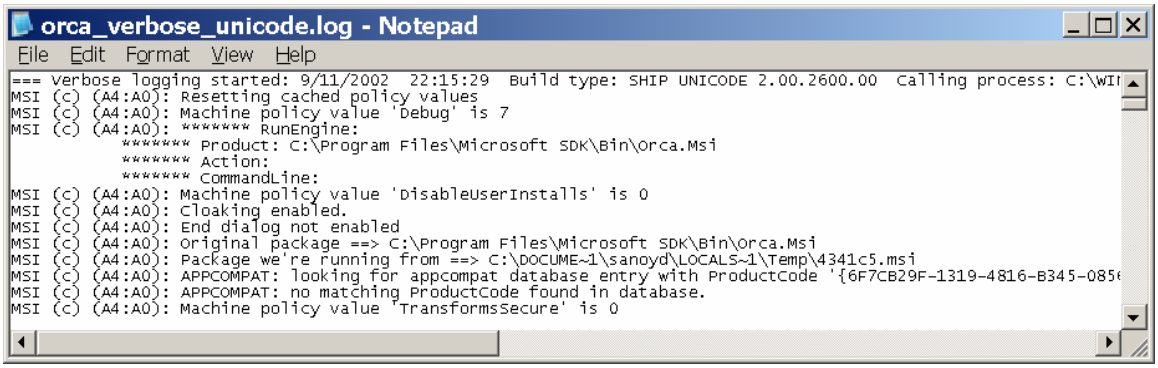

Figure 4.6: A Windows Installer verbose log.

Utilize Windows Installer's Logging Capabilities

It is a good practice to formulate an approach to logging Windows Installer activities to aid in troubleshooting problems in production environments. The following areas of logging should be considered when building this approach:

- Windows event logging—Windows Installer always logs to the event logs. Support personnel might need to be trained to look here and to be familiar with the messages Windows Installer may generate.
- Windows Installer logging policy/registry key—Turning on some level of logging (possibly verbose logging) for all packages ensures that problem diagnosis information is available when needed. Only by configuring the policy will all Windows Installer activities be logged verbosely, including self-healing and other automatic background activities or user chosen activities.
- Windows Installer command-line logging—If verbose logging is not configured globally, it can be configured on a case by case basis depending on the deployment phase of a package (for instance pilot deployments) or how critical the package is.
- Use the status MIFs with SMS—If you have SMS, remember to use the /M switch with msiexec.exe to generate status MIF files. Doing so will ensure that MSI package errors are forwarded into SMS's status reporting system where alerts and reports can be generated.

Formulating Your Own Processes

Now that we've explored the practical best practices that are generally applicable to administrative package developers in all types of organizations, let's shift our focus to the critical concepts required for formulating your own best practices. The following sections explore how areas that depend heavily on your company's approach to application integration, desktop computing support, and how IT is paid for in your company will affect your formulation of best practices and packaging processes.

Windows Installer SDK Assumptions

The Windows Installer SDK is an obvious source for information when formulating best practices. Earlier, we discussed that the SDK has several assumptions about its audience and how they utilize packaging. Figure 4.7 shows how we will refine these assumptions by applying additional administrative uses of packaging technology.

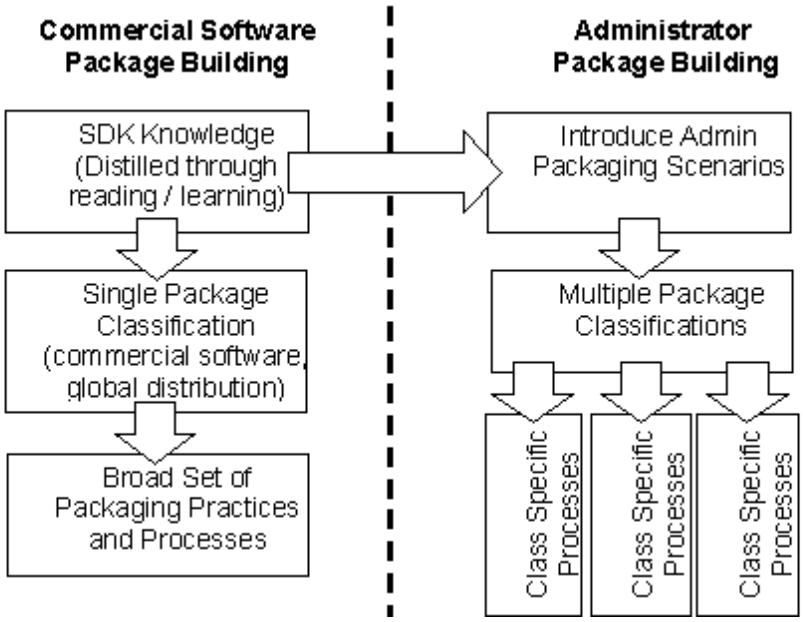

Figure 4.7: Process building guidelines for administrators.

> References to "company" or "corporate" environments should be interpreted to mean any organization with managed IT, including non-profit, educational, government, and military institutions.

The following assumptions must be re-evaluated in the light of how administrators need to use Windows Installer technology:

- Scope of distribution—The SDK assumes that you are building a commercial software package that has the potential to be installed on any Microsoft desktop OS running on any computer in the world. It makes sense that the SDK assumes the broadest case of distribution for a package. It is this assumption of global distribution that can lead to difficulty adapting SDK rules and regulations to a managed environment (with a limited number of OS versions, a limited number of computers, and a known set of applications). No discussion is given within the SDK to alternative approaches in the context of managed environments.
- SLA for installation—Most of us expect that new software setup packages (of any kind) might break the computers on which the packages are being installed. It is simply not realistic to expect that software vendors could successfully integrate with every configuration variation in the world. With administrators, however, the SLA can be ruthless in regard to breaking existing software. Administrators have a more defined scope of integration but the service level might require that packages prepared by the administrative community (as well as the software applications they contain) will never conflict with one another. The looser service level agreement for commercial developers is more of a de facto expectation than an SDK assumption, but it is related to the scope of distribution. Commercial software developers have a vast scope of distribution with a *best effort* service level for breaking existing software.

A defined scope of integration does not mean that the task is simple; making 4000 to 5000 applications seamlessly integrate across 80,000 desktops is still exceptionally complex.

- Control of application source code and development methodologies—This assumption is a significant outgrowth of the assumption that package creation is done primarily by commercial software developers. The Windows Installer SDK might require changes to the underlying software application (for example, renaming or relocating DLLs). These changes cannot be accommodated by administrators who are repackaging software because the administrators do not own the source code or have a relationship with the software developers. In some cases, administrators are responsible for developing the packages for in-house software and still cannot affect such software application changes due to release timing or political boundaries.
- New packages only—The SDK generally assumes that Windows Installer packages will only be built by software vendors for a new release of software they own. Under this assumption, there is clear ownership and sequencing of component code assignments. This assumption does not account for repackaging in situations in which thousands of administrators in thousands of companies assign their own randomly generated component codes for the same software (for example, Adobe Acrobat Reader).
- Knowledge of SDK and the package creation learning curve—The SDK also assumes that the package developer will have a significant amount of time to invest to understanding all the nuances and implications of the rules and regulations that must be followed before building packages. Windows Installer does not easily lend itself to a "learn as you go" approach. This task is challenging even for developers who are accustomed to learning new APIs.
- Role of administrators—The SDK generally assumes that administrators will be involved in the customization and deployment of existing MSI packages rather than building packages. This assumption does not prevent administrators from getting to know Windows Installer well enough to build packages, but it does not acknowledge that the administrator's environment is very different than commercial software vendors.
- In-house developers—In-house developers are also omitted from the SDK in many ways. This omission affects administrators as well because most organizations have a varied mix of in-house development teams, some of which rely heavily on the administration team for packaging and deployment skill sets. Although the SDK can be used by in-house developers, it can be much less cost effective for the in-house developers to build packages with the assumptions required for globally deployed commercial software because in-house developers' environments are generally much more managed.

Most of the assumptions made by the Windows Installer SDK are sensible for the broadest cases, but they can be difficult to adapt to the rules and regulations of managed environments. Reading SDK statements with these assumptions in mind can help you understand where the SDK rules can be adapted to fit your organization.

> In Chapter 3, I mentioned that one of the reasons administrators need to know packaging internals is to diagnose problems with vendor-provided MSI packages. SDK familiarity follows this principle as well—you must be familiar with the baseline knowledge to discover when a software vendor has made a mistake in their package, and you stand a better chance of negotiating an agreeable course of action if you can talk their talk.

Package Classifications

Package classifications are crucial to building processes. Processes and practices become overburdened when there are too many possible scenarios that generate many branches in the process. Creating a classification structure helps ensure that the minimum number of process alternatives is required and that no particular classification of activity is left under serviced or unserviced.

The SDK assumes a single package classification—commercial software with the potential for worldwide (unmanaged environment) distribution. Once again, although the SDK does not preclude in-house developers, it does imply a distribution environment very different from what in-house developers have.

The scheme for classifying packages that I present here is not the only way to build a process for packaging. However, the concept of understanding and defining your package classifications before building your processes is crucial to ensuring that the process reflects reality and will work for your organization. The classification scheme presented here is focused on accommodating administrators' usage of Windows Installer technology and the historic state of packaging in managed corporate environments. The following classifications should be consolidated as much as possible, and understanding them is critical to building a viable process for packaging applications.

> The following classifications are useful for building and customizing packages that come as Windows Installer packages or can be repackaged as Windows Installer packages. As discussed in Chapter 3, there are many setup programs that should not be repackaged at all—these packages must use their built-in silent setup switches and might need to be wrapped in a generic MSI package for specific deployment scenarios.

There are several classifications that should be considered when building a packaging approach:

- Vendor-provided software applications—Software packages provided by a commercial software vendor. These should not be directly edited but customized using transforms. Additionally, no structural changes (feature organization, component codes, and so on) should be made to the package using a transform.
- Repackaged software applications—Packages that the administrator community has repackaged from setup executables. Because the underlying software application in this classification is not owned or influenced by the packager, some of the rules pertaining to package structure are more difficult to manage.

A Windows Installer package received from a software vendor should not be repackaged. I will discuss this scenario in more detail later in this chapter.

- In-house software applications managed with low-end packaging/deployment requirements—These packages contain in-house application software but have no specific deployment logistics that are addressed by packaging technology. Many times developers for this classification already depend on the administration team for packaging and deployment. Generally, it is not a good idea to have the application development teams in this classification learn Windows Installer technology. It is likely that more time will be spent on quality assurance of the packages generated by the application development team than would have been spent simply doing the package for them. If possible, this classification should be consolidated into the *repackaged software applications* classification by treating it as a repackaged setup.exe.
- In-house software applications managed with high-end packaging/deployment requirements—This classification is relevant when the application development team has special deployment challenges or logistics that are addressed by Windows Installer packaging functionality. An example is a development team that deploys to a sales field force that would like to take advantage of Windows Installer patching technology. This classification can be tricky to manage because it brings up the question of whether the development or administration team should be responsible for building the package and managing its structure over time. You might be able to consolidate this classification of packages with the *vendor-provided software applications* class. Doing so would result in the application development team completely managing the Windows Installer packaging with administrators customizing and deploying the package.

It is my opinion that an in-house development team that has high-end packaging requirements should be willing to build and manage the package structure and all upgrades within their team. If the packaging work is attempted by the administration team, it will pull shared administrator resources away from all corporate packaging.

Formulating processes for the in-house classifications is the most difficult proposition. The scheme that I present attempts to keep administrators out of the business of building packages such as those that commercial software developers would build. Doing so prevents administrators from having to learn a deep level of Windows Installer methodology for fairly few packages.

Package Structure Rules for Administrators

As mentioned earlier, there are some strict configuration rules for how the application management meta data in a Windows Installer package must be structured. I will be referring to these rules as package structure rules. They pertain to the content and identity (GUIDs) of components, packages, and products within Windows Installer packages. These rules help allow Windows Installer to solve some age-old packaging challenges. Remember, as with all of the SDK documentation, the perspective of these rules is that of coordinating between all commercial software developers with global distribution:

- To help with DLL hell by ensuring the unique and consistent identification of all executable files (EXEs, DLLs, OCXs) published by all software vendors in the world.
- To allow for efficient and error-free software upgrades.
- To help prevent software uninstallation problems.

Essentially, there are two main areas in which package structure management is required. It is required between your packages and all other packages, and it is required between your packages and any subsequent upgrade packages for your packages. These rules are focused on solving certain software problems for commercial software developers, releasing *new* products for possible global distribution.

> 🕮 I recommend that you read the four main SDK documents that deal with component rules: "Organizing Applications Into Components," "Defining Installer Components," "Changing the Component Code," and "What Happens if the Component Rules are Broken?"

Component Rules—The Protocols for Sharing

The essence of understanding component rules is understanding the problems that the idea of components is trying to solve. Many application integration problems occur due to the sharing of code and other application resources by applications. Here are some classic sharing problems:

- Two applications on the same computer require different versions of the same DLL file stored in the same location on the hard drive.
- Two versions of the same DLL stored in different locations on the hard drive attempt to keep their data in the same file or registry key.
- Installation programs that do not follow version replacement rules unwittingly downgrade a shared DLL.
- The uninstall of an application breaks another application that was using files shared by both applications.
- The uninstall of an application leaves shared files intact but removes resources required by the shared files.
- Commercial application developers in one company use software application code from another company but fail to properly distribute, upgrade, or configure it on target systems.

Microsoft set out to solve these sharing problems with Windows Installer. From Microsoft's perspective, any solution to this problem must be able to work for every application and every desktop computer in the world. With millions of computers and millions of applications, there is no way to put any boundaries around the problem—the solution must be able to cover the innumerable combinations derived by all possible combinations of millions of applications installed on millions of computers.

Scope of Distribution

Using GUIDs components can uniquely identify every piece of Windows code (EXE, DLL, and OCX files) in existence and group each piece of code with all of its required resources. This fundamental idea allows Windows Installer to coordinate code sharing during installations, upgrades, and uninstalls. If developers follow the component rules, incompatible versions of the same code will not be placed in the same location on disk.

Windows Installer made its grand entrance with Win2K. The Win2K application guidelines encouraged developers to place all of their DLLs in the application's Program Files directory rather than shared locations such as System32. So in the ideal world, most components would have been managing software code that was no longer stored in shared locations on disk. However, in the real world repackaging and software development habits have ensured that there is a large body of software code in shared disk locations that is managed by components.

The SDK's assumed scope of distribution is "any software application in the world installed on any computer in the world." However, as we talked about earlier, administrators in a managed environment have a more defined scope of distribution. With repackaged applications, it is necessary to think of these rules with the scope of distribution "any repackaged software application in the company, installed on any computer in the company."

If you work for a division of a large company or conglomerate, you might be tempted to refine the scope of distribution further to read "any repackaged software application in my division, installed on any computer in my division." If this is done, you must be 100 percent certain that no application you generate will ever be installed in another division, including unforeseen division mergers, employees division reassignments (with computer), and corporate restructuring. Unfortunately, Windows Installer will be very unforgiving about overlapping and uncoordinated component definitions regardless of business driven changes in the scope of distribution.

By refocusing the scope of distribution on the boundaries of a company, we can make better sense of the component rules we will be discussing. This customized scope of distribution applies to any repackaged applications because these are the applications for which administrators assign the identifying codes. Windows Installer packages from software vendors come with their identifying codes assigned by the vendor—these should not be changed and are assumed to be unique worldwide.

Code Management Components

Although components are used to solve many sharing problems, their highest calling is to prevent code sharing problems. The remainder of this discussion will focus on components that solve code sharing problems. As with most Microsoft OS-level technologies, Windows Installer intends to solve this problem going forward—as new applications are built on the new architecture. Repackaging is presented with new challenges because multiple component codes (GUIDs) can be generated for the same code or incompatible application code can have the same component code.

In Chapter 3, we learned that each .EXE, .DLL, and .OCX file has its own dedicated component. Although components are identified by their GUIDs, they also inherit some of their identity from the keypath file. The component's "location" is determined by the full path name (path + file name) of the keypath file. The component's "version" is determined by the version of the keypath file.

Components can contain many resources such as files and registry keys, but these resources must remain in the same location and be backward compatible when the component is upgraded. Component resources cannot be added, removed, or change locations. When upgrading the software code contained in a component, the code must be fully backward compatible with all previous versions, and the keypath file must keep the same name and location; if any of these assumption are not true, a new component (and new component code) must be defined. Here are the high-level rules to keep in mind:

- Components are created and assigned a component code when a code file has the same name, location, resources, and *compatibility* as all other existing copies of that component in any software application in the world installed on any computer in the world.
- Component instances (copies installed on many machines) that are identical as per the earlier described rules MUST NOT have more than one component code identifying them (in any software application in the world installed on any computer in the world).

Figure 4.8 illustrates what happens when a component is upgraded with compatible software code.

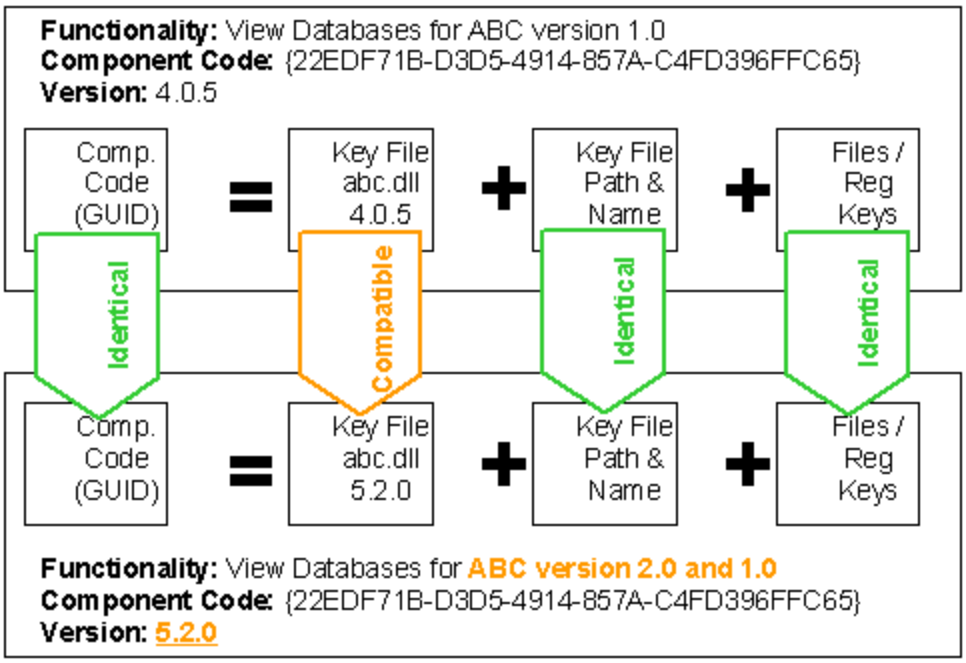

Figure 4.8: Compatible software code change upgrades component.

Figure 4.9 shows what must happen when an incompatible software code change is made. Instead of changing the old component, a brand new component is created. This mechanism is powerful because the new application can be coded to look for the new software code using the new component definition while all existing applications in the world on all desktops in the world that share the older software code (whether known or unknown to the software developer) will remain working because both of these pieces of incompatible code (and components) can be on the same system.

Component rules do not have an answer for two sets of **existing** software code that are incompatible (for example repackaged DLLs) that must be installed to the same shared disk location because they assume that the code in the last component defined can be renamed or moved by the package developer.

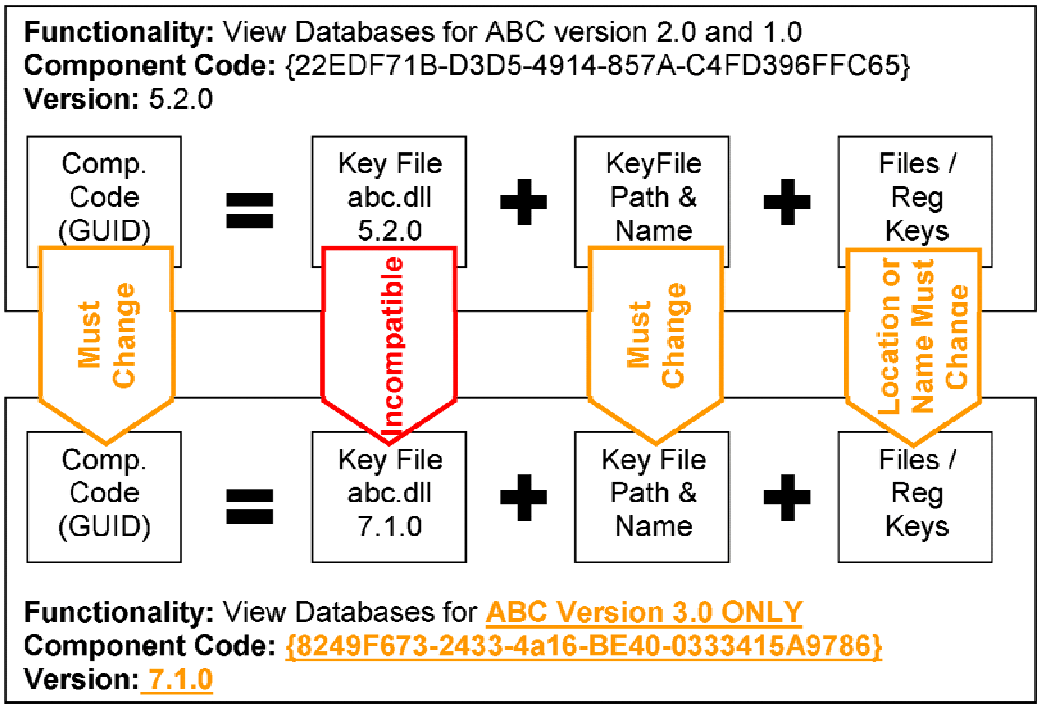

Figure 4.9: Incompatible software code change spawns new component.

Duplicate Component Definitions

In Chapter 3, we touched on the fact that Windows Installer reference counts components based on their component IDs. A reference count simply tracks how many applications are using a specific component. If five applications all install a specific component, the component would have a reference count of five. If one of those applications was subsequently uninstalled, the component would not be removed; instead its reference count would be decreased to four and it would be left on the system.

> In the past, some IT organizations have prevented uninstall problems by making a rule that managed packages will never be uninstalled. For legacy setup technologies, this rule is sensible. Windows Installer upgrades, however, perform *component level uninstalls* during upgrade operations. Thus, upgrades of the same application will indeed remove files and system resources rather than layer on top of them like legacy repackaging and setup programs.

Repackaging tools assign new GUIDs to the entire package structure each time the repackaging tool is run. You can observe this behavior by repackaging the same application three times, then viewing the component code for the same file in each package. This behavior can lead to multiple component codes for the same file when multiple corporate repackaging labs package the same software or even if the same package developer repackages a software application from scratch multiple times and sends both versions into production.

Duplicate component definitions occur when two functionally identical instances of a component have different component codes. Figure 4.10 illustrates how duplicate component definitions can result in sharing problems. If Package 1 in Figure 4.10 were to be uninstalled, the file abc.dll may be removed, which would break the software applications in Package 2 and Package 3. Windows Installer performs some additional checks when determining whether to remove a component; however, if the component has a duplicate definition, there is a much higher risk that it could be removed when it is still needed.

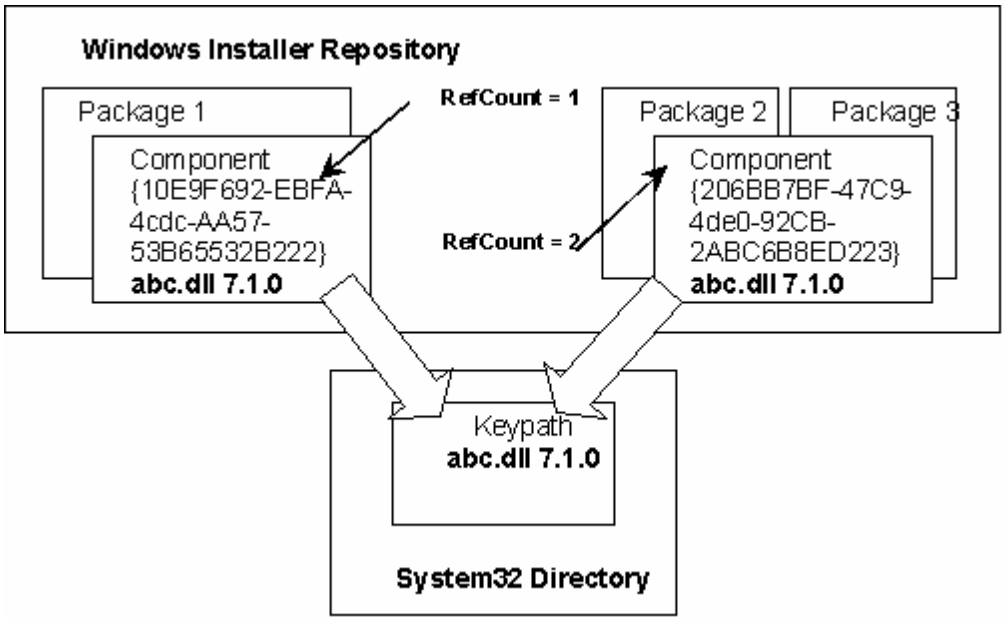

Figure 4.10: Duplicate component definitions.

A frequent misconception about accidentally uninstalled components is that they are simply self-healed by Windows Installer when another dependent application is started up. As we covered in Chapter 3, self-healing is dependent on feature structure, so it is possible to have missing files that will not self-heal when components are unintentionally removed.

Conflicting Component Definitions

Another related problem occurs when the same component definition refers to two sets of resources that conflict when installed at the same time. Figure 4.11 shows the same component defined with two different DLLs. In this case, abc.dll version 8.5.2 is not fully backward compatible with version 7.1.0 and it breaks the software application in Package 1. If the DLL were compatible, this illustration would actually be the correct configuration for this component.

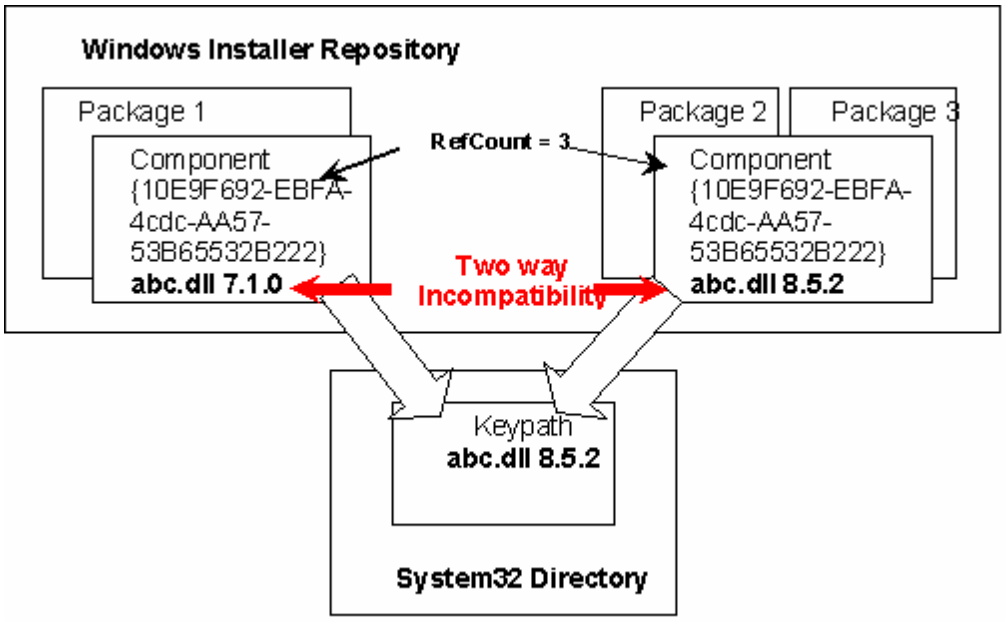

Figure 4.11: Conflicting component definitions.

If you were the developer of abc.dll, you would create a new component for the functionality contained in abc.dll as well as rename and/or move abc.dll for the newer version of the software that requires it. When you are repackaging software, you do not have this luxury and must resort to extensive application integration testing. After you complete the testing and find a compatible version of the DLL, it must be set up as the standardized component in all packages.

Windows Installer does follow DLL replacement rules in part because components inherit their version number from the DLL. A DLL file is never downgraded during a default package installation. Special parameters and values authored into a package can cause packages to force their version of a DLL onto the system; however, in practice, this configuration is not frequently used.

Self-healing can unintentionally "downgrade" a component when the keypath file is missing. This occurs because self-healing files are sourced from the first package that triggers a self-healing event. For instance, if abc.dll in Figure 4.11 was deleted and Package 1 triggered self-healing, when the user started up the software application, version 7.1.0 of the file would be copied into the system32 directory. When the software applications in Package 2 or Package 3 were started, self-healing would see that the file existed and no self-healing would occur, leaving abc.dll at version 7.1.0, which would break the software applications in Packages 2 and 3.

If Windows Installer packages from software vendors are repackaged, they will have many problems with duplicate component definitions because the repackaging tool will re-assign all component codes.

Compounded Problems

Duplicate and conflicting component definitions create some big problems, even in a simple illustration using three applications installed on one computer—consider this problem multiplied by hundreds of applications with hundreds of thousands of DLLs. If a popular runtime support DLL is used by many applications, it might have multiple duplicate component definitions and multiple conflicting component definitions across the many packages that utilize it.

In the face of this level of complexity, conflict management tools take on a new level of importance. Conflict management tools, which will be discussed later in this chapter, allow administrators to manage component definitions across all packages in your scope of distribution.

Upgrade Packages

We have been examining how package structure management is important for application sharing—that is coordinating between your package and all other packages. Package structure management is also critical for building upgrade packages—that is coordinating structure between your package and any of the upgrade packages that are eventually built for it. Because patch packages are a special kind of upgrade package, we will discuss them after upgrades.

From a Windows Installer perspective, upgrade packages are fully functional packages that can install software applications on a clean machine. Package developers might require previous versions before installations can proceed, but this requirement is simply a licensing control—the package itself contains all the information necessary to install the package on a clean workstation. An upgrade package includes additional information that helps it identify upgrade candidate packages on the target workstation. Its internal structure is also designed by the package developer to coordinate with previous versions of the package.

Windows Installer is intimately involved with performing upgrades, enforcing specific rules at the component level during an upgrade. If the package structure is not managed appropriately, Windows Installer might do unexpected things with your package. Even if you only ever had three Windows Installer packages to build for your environment and were able to successfully ignore package structure rules without any problems, you would still need to adhere to them if you intend to upgrade your own packages.

Upgrade Processes

Before we can discuss the SDK upgrade rules and how they apply to administrators, we must understand how Windows Installer prefers to perform upgrades. A standard Windows Installer package will perform upgrades steps as follows:

1. Identify upgrade candidates (installed packages that can be upgraded by the currently installing package).
2. Install new and updated components.
3. Remove unneeded components.

The last two steps in this sequence might seem backward. This sequence is meant to address the issue of ever-growing software applications. Take Microsoft Office for instance. Say that a given configuration of Microsoft Office takes 950MB on disk. Further, imagine that an update to Microsoft Office requires 10MB of new and updated files and the deletion of 4MB of files. If a full de-install and reinstall is performed, then 950MB of files are deleted and 960MB of files are copied. Windows Installer's method of installing new and updated components and then deleting unneeded components reduces this load to 10MB of file copies and 4MB of deletions.

From an administrator's perspective the significance of these file transfer savings depend on whether software is deployed while users wait. In many organizations, software is deployed during off hours, so the length of time waiting is not as significant to the end user experience. In addition, it depends on the average size of software packages. If most software is smaller than 10MB, the difference between the two approaches may be negligible even for interactive installations. You can configure Windows Installer to perform the uninstall of the package to be upgraded before installing the new package.

For more information about configuring Windows Installer to uninstall first, see "Sequence Restrictions" in the SDK document "RemoveExistingProducts Action."

Package Attributes

There are three key attributes of a Windows Installer package that are used to designate what type of update a package is. These attributes are:

- Package code—The package code is a GUID that is stored in the summary information stream. A package code indicates that two Windows Installer packages will perform identically when executed. Only functionally identical packages should share the same package code.

Package codes function similarly to hash codes, which ensure that two file versions match exactly (like CRCs). Windows Installer cannot use hashes to determine identical functionality because MSI files contain a database that might vary in physical organization between two identical copies of a file and Windows Installer packages can be functionally identical but structured differently. For instance, an MSI file with compressed source files and an administrative install share from that package have very different physical file structures but perform identically when installed.

- Product code—Product code is a GUID stored in the property table. Just like a component code is the authoritative identification of a component a product code is the authoritative identification of a software product. Two packages should only have the same package code if they are the same major release of the software package. Two instances of the same identical packages should always have the same product code. Sloppy or uncoordinated repackaging processes can cause identical repackaged applications to have multiple package codes deployed to production computers. This can result in upgrade packages not recognizing upgrade candidate packages installed on computers to which they are deployed.

- Product version—Product version is a period delimited numeric value stored in a property that usually coincides with the version number of the software application. The format is w.x.y.z where w is major version, x is minor version, y is the build number, and z is a further revision number. If product versions are not properly managed, upgrade packages might fail to apply to upgrade candidate packages on computers to which they are deployed.


Although Windows Installer allows four positions of the version number to be defined, it only pays attention to the first three when comparing version numbers.

Update Types

The Windows Installer SDK defines three types of update packages. These types have very specific meanings within Windows Installer. Although these update types are more specific than the generic update categories that have evolved as best practice in software development, the user expectations of what might change in a given update type are similar.

Minor Upgrade

Minor upgrades allow additions to existing software as long as the addition is in the form of new components. Features can be added or removed, but the features cannot be reorganized. The MSI file name must remain the same as well.

Minor upgrades must increment the product version to a higher version. The product code is unchanged. Because the upgrade package is not functionally identical to the package it upgrades, the package code must be regenerated.

This update type roughly equates to a minor release of a software package that might fix some bugs and add a few improvements and a minor feature or two. Strictly speaking package developers could make some pretty large-scale changes to their packages and still fit within the SDK rules for a minor upgrade.

Small Update (Admins Need Not Apply)

A small update is identical to a minor upgrade in regard to the rules about what can be changed within the package. The only difference is that the product version is not incremented.

When the product version is not changed, it is impossible to tell which installations of the original package have had a given small update or multiple small updates applied to them. The Windows Installer meta data (product version and product code) for the packages looks identical. Presumably, this update category is meant to facilitate policies in application development organizations that require extensive authorizations or regression testing when version numbers are changed. This can be overburden when a single file needs to be changed or the initial package has been distributed to an extremely limited number of computers. Because a small update package is not functionally identical to packages it upgrades, the package code must change.


I advise strongly against administrators using small updates or allowing in-house developers to use small updates. The inability to distinguish two installations of application software that are distinctly different does not follow generally accepted practices for managed computing systems.

A small update roughly equates to a hotfix to an existing application with which very few changes are made to an existing application. However, even when following Windows Installer SDK rules, much more could be changed.

Major Upgrade

There are no limits as to what can change with a major upgrade. The package could for all intents and purposes be a complete rewrite that does not use a single file or registry key from the previous version. In most cases, however, a major upgrade will share package structure elements with the previous versions it upgrades, but specific types of changes require the package to be classed as a major upgrade.

> 🕮 For complete details about what can change during minor and major upgrades, see the SDK document "Changing the Product Code."

A major upgrade is equivalent to a software application release that changes the highest version number (for instance, ABC version 2.5 to ABC version 3.0). However, Windows Installer package structure changes required to deploy small scale changes to the software application might trigger SDK rules that force the package to be classed as a major upgrade.

Simplifying Upgrades

Upgrades can be simplified by using the package classifications we established earlier. Here are some general approaches that you can use as a launch point to establishing your upgrade practices:

- Vendor-provided software applications—Because the vendor is the one building the packages, you simply deploy whatever types of updates the vendor delivers to you.
- Repackaged software applications—In general, an upgrade package for a repackaged application should be assumed to be a major upgrade. Vendors who deliver setup.exe applications are not bound to any of the Windows Installer package structure rules and therefore can make software changes that automatically trigger the major upgrade classification. Rather than spend many hours sorting through a package to determine whether it is a major upgrade, streamline the process by treating all updates as major upgrades. You might want to test whether moving the RemoveExistingProducts standard action to cause a complete de-install before package install gives higher quality upgrades for your upgrade packages that are built for repackaged applications.

> 🕮 For more information about configuring Windows Installer to uninstall first, see "Sequence Restrictions" in the SDK document "RemoveExistingProducts Action."

- In-house software applications managed with low-end packaging/deployment requirements—Generally, these packages should be managed as repackaged applications, which would put them in the major upgrade category. However, in the case of in-house applications, you can generally learn from developers whether the changes between two releases are significant and make a judgment between a major and minor upgrade. The caution here is that this creates a new fork in your process and involves more coordination and potentially more familiarity with package structure than would otherwise be needed by an administrative packaging team.

- In-house software applications managed with high-end packaging/deployment requirements—This classification should generally be managed like the vendor-provided software applications, which means that you would deploy whatever is given to you by the in-house development team. However, it is wise to always discourage the building of small updates by another in your company. You might be unfortunate enough to be an in an administrative community that has no choice but to manage this type of packaging for an in-house development team. If this is the case, ensure that you build up an expert on package structure from a developer's perspective. Also ensure that a team member is intimately familiar with the in-house software application's structure and that the team member can exert influence on this structure to ensure that packaging rules can be followed as intended. You might want to test whether moving the RemoveExistingProducts standard action to cause a complete de-install before package install gives higher quality upgrades for packages in this classification.

Patch Packages

Windows Installer patch packages provide efficiency during deployment by allowing a much smaller package to update existing software. Patches require that an existing version of the software application be present because (unlike upgrade packages) they do not include a fully functional version of the package or software application. Patch packages achieve smaller file sizes through two methods:

- Only files that have changed are evaluated for inclusion in the patch.
- Only the actual binary changes between the old file and new file are included in the patch.

Special differencing technology is used to generate binary pieces of the files to be patched and to use the pieces to alter older versions of the file during patch application. Patching has the ability to patch multiple previous versions of the target files. For instance, a patch may be able to update versions 1.01, 1.37, and 1.42 to version 1.50.

Generating Patches

To generate a patch, a full upgrade package must be built. The Windows Installer patching routines (shipped in the SDK and included in many authoring tools) are run to generate the patches.

Patching Reality Checks

Many administrators become enamored with the potential benefits of patching. Here are some points that should be kept in mind when considering the use of patching in your practices:

- Patching is not faster than generating a full package—Administrators might have experienced the concept of patching as building small scripts to fix up application deployment problems. Generating these scripts may have been much quicker than building a new package because they adverted some change and quality control processes and were much quicker to build than a full package. Because Windows Installer requires a complete update package to generate a patch, it is actually extra work, processes, and quality testing to build a patch package. It might be more efficient to use the upgrade package that must be generated to deploy the upgrade.
- Patching primarily helps with bandwidth management—Patches are smaller and can provide bandwidth relief in two specific scenarios. Constrained bandwidth between servers can benefit from patching packages on administrative install shares. This allows patch packages to be distributed and applied to servers rather than sending an entire upgrade package. The decision to use administrative install shares has other considerations, such as higher disk space usage, that must be factored into the decision about whether to use them.

  Constrained bandwidth to client computers can benefit from patching packages on workstations or deploying to client computers using administrative shares that are patched. Either method reduces the amount of data transfer for updates.
- Patch packages are more difficult to manage—Because patch packages use a different file structure, they cannot be opened and examined as easily. In addition, the application of patch packages uses very different processes and generates different log entries and troubleshooting scenarios.
- Patch packages might require original source package—Previous to Windows Installer 2.0, patch packages required the original file source to extract files and patch them rather than patching files directly on the target system's disk.

These cautions do not mean that patching is inappropriate for administrators. Upgrade packages should generally be the first approach for updating software applications with the costs of patching being weighed with the additional benefits that it can provide in specific deployment scenarios.

Conflict Management for Package Structure

There are many aspects of package structure that must be managed and kept straight to ensure high quality software distribution over the long term. Package structure management is further complicated by the fact that Windows Installer was not intended for use in repackaging. There are many conflicts that can occur in package, product, and component contents and identification. These challenges combine with the tighter service level that requires that software packaged by administrators not cause problems with any other installed software.

Windows Installer tool vendors have recognized administrators' need to gain tighter control and have responded by creating conflict management tools within their administrative tool suites. Essentially, these tools import every package in the scope of integration (usually your company) and analyze them for potential problems such as duplicate and conflicting component definitions and duplicate and conflicting package and product codes.

You can also use conflict tools to find problems with Windows Installer packages received from software vendors. Software vendors can create packages that inadvertently duplicate package structure elements in other vendor packages. In addition, conflict tools will reveal if your packages contain component definitions that are also in vendor packages. In these cases, the vendor packages should be given preference whenever possible.

Conflict tools generally have some capacity to resolve conflicts. For example, Wise's conflict management tool allows package developers to specify which version of a DLL to use for a given component and what component code should be assigned. The conflict manager can then re-write this component definition in all affected packages and re-compile all affected packages. The re-compiled packages are configured as upgrade packages so that they can be used to fix previously deployed versions that have incorrect GUIDs.

A Word About Merge Modules

Merge modules are a mechanism in Windows Installer that allows software companies to prepackage and share standard component definitions. For example, take the infamous Crystal Reports runtimes. The files, DLLs, and registry entries that make up these runtimes can be defined as Windows Installer components and placed in a merge module.

When any developer in the world wants to distribute these runtimes, they use the vendor merge module to ensure that a complete set of runtimes files is included with the vendor assigned versions, component codes, registry keys, and other Windows Installer elements that the owning software vendor assigned. Obviously one of the biggest publisher's of merge modules is Microsoft. When not merged into a package, merge modules are contained in .MSM files (another variant on the .MSI format).

Merge Modules in the Administrator's World

There are several scenarios in which administrators might find merge modules useful or necessary. The following list provides some examples:

- To use vendor-built merge modules instead of the files repackaged with a setup.exe. Software vendors may build a setup.exe installation package that includes third-party runtime files that are also provided by the third-party as merge modules (for use by software vendors who are building Windows Installer packages). Using the merge module instead of building your own components ensures that reference counts of vendor-provided components are accurate when vendor-provided MSIs and repackaged MSIs both use the same versions of third-party runtimes.
- To alert in-house developers that they should be using vendor-provided merge modules if they are authoring packages that include runtimes that are distributed as merge modules by the runtime vendor.
- To engineer merge modules for in-house packaged software. As I mentioned earlier, if an application development team in your company wants to build merge modules, they should probably be doing their own Windows Installer packaging—so hopefully you will not need to be involved in this scenario.
- To engineer merge modules for repackaged software.

Although engineering merge modules for *repackaged software* is a topic of much debate, it is my opinion that if a company is considering engineering their own merge modules for repackaged software, they may be better served with a full conflict management tool.

Merge Modules as a Poor Man's Conflict Management Tool

Many administrators have wondered whether merge modules might be leveraged to help with coordinating a managed set of DLLs among *repackaged* software applications I am not in favor of using merge modules as a form of homemade conflict management because

- Merge modules are difficult to distribute in a timely fashion to maintain a synchronized version among all package developers.
- Additional Windows Installer expertise is required to engineer merge modules.
- Merge module–based schemes focus on developer component coordination requirements and might not catch other possible problems and conflicts with package structure.

By contrast, conflict management tools are designed for application integration needs. The benefits of conflict management tools include the ability to test packages before excessive integration changes are introduced, track DLL versions which allows reverting to the shipped version, and report on and re-release all affected packages when managed DLLs are upgraded.

Merge modules may be used effectively in environments that can be tightly controlled or that are simple (single site or few package developers), but the cost of additional engineering, expertise, and risk should be weighed against the cost of formal conflict management tools.

Replacing Repackaged Files with Merge Modules

Many administrators might want to replace repackaged runtimes with the vendor's official merge modules or internally generated merge modules during the packaging process. You should be familiar with the following cautions when replacing repackaged files with merge modules:

- As of Windows Installer 2.0, merge modules can alter package logic (such as adding custom actions), which might result in unintentional package behavior.
- Merge modules may contain errors.
- Merge modules may add many more files than your setup.exe package originally contained due to dependency information that was not tracked by the developer who built the setup.exe program.
- Merge modules may have different versions of supporting files; although they should be the vendor's specified matching versions, in some instances they may create compatibility difficulties with the software application if the original setup.exe contained mismatched file sets on which the application vendor built their software.
- Merge modules will need to be run through your conflict management tools if you are using them.
- Be sure you are familiar with the rules followed by automatic merge module scanning/replacing tools and ensure that they meet your guidelines for application integration and testing.
- Application testing can be made difficult when large portions of an application are replaced by merge modules. If merge modules replace 50 DLLs and there is a problem with integration testing, it can be very difficult to know where to start diagnosing the problem; you might want to keep a reference copy of the MSI as cleanly repackaged before merge modules are substituted.

Administrator and In-House Developer Generated Merge Modules

If used, administrator or in-house developer–generated merge modules should follow a set of simplified design rules. Since version 2.0 of Windows Installer, many packaging items (such as actions) can be included in a merge module. For the sake of management simplicity, you might want to start with the following merge module guidelines and adapt them as your requirements dictate:

- Read up on merge modules before making design decisions in your company.
- Do not use the configurable merge module features.
- Do not include package processing elements such as custom actions or dialog boxes.
- Only use dependency information if it is applicable to all possible uses of the merge module.
- Always check to determine whether files are already contained in a vendor-provided merge module.

- Only include a single code component within a merge module. If a merge module must contain multiple components, make absolutely sure that you are not guessing as to which code file components to include and which versions match each other.
- Only include the registry entries required for COM registration of code files or other registry entries required for operation of the code.

In any organization, the approach to merge modules should be discussed and defined during the definition of packaging standards. The usage of merge modules can prove to be very difficult to engineer as an afterthought to the design of packaging in your company.

Summary

In this chapter, we have covered a lot of ground in regard to best practices and process formulation. Windows Installer packaging and processes are extremely flexible to be able to accommodate global coordination between all Windows software. The flexibility as well as the underlying assumptions for this new paradigm are not suited to simplistic best practices and processes. Hopefully, the principles and practical recommendations in this chapter will provide a good launching pad for your efforts to build packaging practices and processes at your company.

Chapter 5 will focus on the infrastructure required to successfully deploy and maintain packages. This upcoming chapter will discuss how to build this infrastructure if you don't have Active Directory (AD) as well as provide pointers for those of you who do have AD.

Chapter 5: Windows Installer with or without Active Directory

by Darwin Sanoy

Although Windows Installer saw the light of day as the technology that installed Office 2000, its grand entrance was with the release of Win2K. Many new management technologies were also released in Win2K. To the casual observer, it might appear that the management technologies in Win2K are only available if a company is willing to deploy Win2K to all of their servers and desktops and to use AD for authentication and directory services. In fact, Windows Installer can actually be used very effectively in the absence of Win2K servers, AD, and even Win2K or later desktops. This chapter will examine several topics with two major themes in mind:

1. If a company does not have AD, what plans must be made to ensure that Windows Installer has the necessary supporting technologies to be effective. This perspective is important for companies who will never have AD as their primary directory service and companies who will have a delay or extended AD deployment.
2. If a company has or plans to deploy all the Win2K technologies (server, directory, desktop), there are still some significant challenges to building software distribution by Microsoft's IntelliMirror playbook. This perspective will point out these areas and highlight alternative approaches.

Beware the Tide of Windows Installer

Packaging is an expensive and time-consuming activity that software vendors are not willing to do twice. As software vendors change to Windows Installer packages for their latest software releases, vendors will cease to provide setup.exe packages. Many vendors are delaying the switch to Windows Installer until Windows Installer has a much greater install base. However, once the tide turns, it might catch many organizations off guard.

This chapter is somewhat presumptive that your company is taking a planned approach to the introduction of Windows Installer technologies in your environment. However, even if you do not have the luxury of deploying Windows Installer as part of a formal IT project, it will be important to manage the items highlighted in this chapter to ensure that Windows Installer does not create more problems than it solves for your company. You can do so by gradually phasing in some of the new support technologies with the first Windows Installer packages that need to be deployed.

Services Provided by Win2K Technologies

Management technologies deployed in Win2K provide some key services to the IntelliMirror desktop management model. Most notably are technologies that support the package distribution repository by allowing it to be universally accessible, locally sourced, load balanced, and fault tolerant. These technologies also help to replicate the files to multiple locations throughout the network. The primary technologies at work here are Distributed File System (DFS) and File Replication Service (FRS).

> In Chapter 4, we talked about the fact that Windows Installer relies on Group Policy for managing application settings that must be changed after package deployment. This chapter does not discuss replacements for this functionality in detail. Any third-party policy management system can replicate the basic registry tweaking abilities that are used in Group Policy for managing application settings.

Win2K technologies also allow packages to be installed on client workstations. The primary technology here is Group Policy Objects (GPOs), which require Win2K Server, AD, and Win2K or later desktops, as Figure 5.1 illustrates.

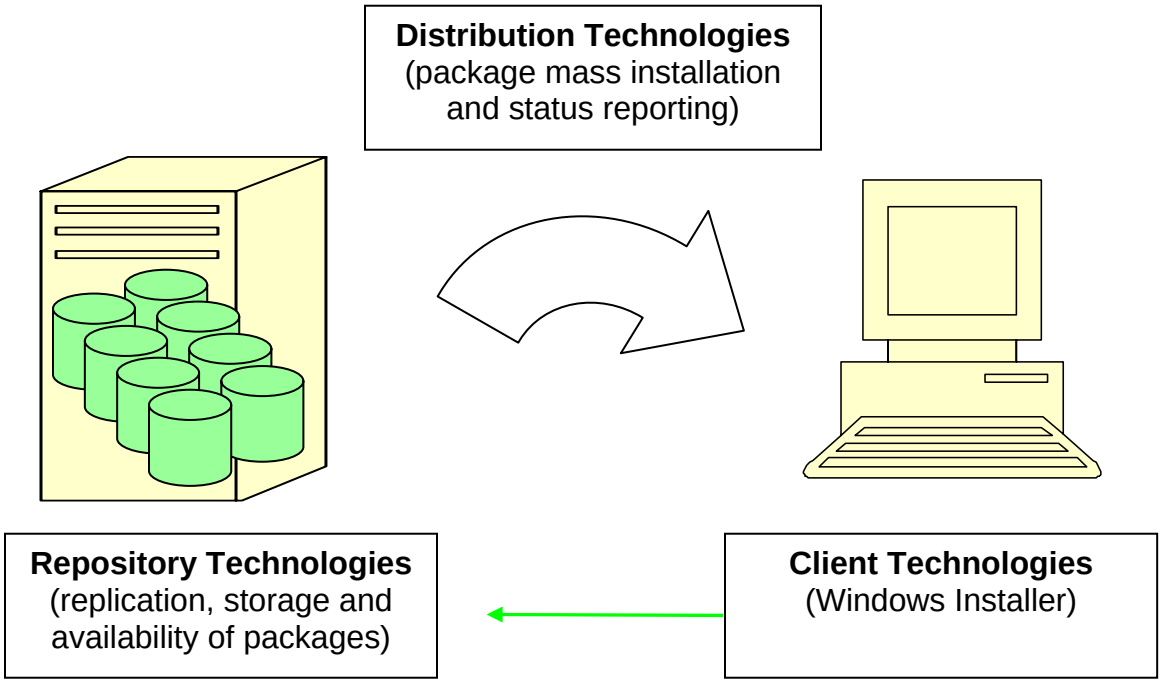

Figure 5.1: Win2K technologies needed for Windows Installer package distribution.

IntelliMirror Repository Technologies Overview

The out-of-box IntelliMirror desktop management solution uses DFS for repository services in large-scale enterprise implementations. DFS allows Windows Installer packages to be sourced from a single domain-level share name. Because DFS operates at the file-system level, its services are transparent to the client applications that use them. When designed and configured properly, DFS provides the following benefits (see Figure 5.2 for an illustration of these concepts):

- Local sourcing—Requests for files are automatically taken from a server that is in the same local site as the computer requesting the files. Figure 5.2 does not indicate the location of the client computer; however, DFS would favor a server in the same site as the client when finding a source location for the files.
- Universal availability (file location abstraction)—Requests for files are made to a share name that appears to be attached to the domain rather than a specific computer. Win2K DFS then determines from which server to actually retrieve the files. Figure 5.2 shows that the share name \\domain\share (where *domain* is any AD domain and *share* is a DFS share name) can be requested, and DFS finds a server location for this reference. This type of approach is also known as *abstracted* because the client computer is requesting a logical location and DFS finds a real location.
- Fault tolerance—If a request for a file encounters a server that is not operating, a new server is tried until a working server is found. The requesting application does not need to be aware of the multiple locations where these files reside. If any of the servers in Figure 5.2 were not available, DFS would find a different server for the client to use.
- Load balancing—Requests for files are automatically balanced between identical directory trees on multiple servers, which provide those files to the network. In Figure 5.2, this functionality would manifest itself by dynamically allocating half of the client requests at Site 1 to one of the servers and the other half to the other server.

Although these benefits are all desirable, the two that organizations of all sizes will be affected by are local sourcing and universal availability. Local sourcing ensures that packages are not pulled across WAN links, which is critical for the user experience due to potential impact to shared WAN bandwidth.

Universal availability lets Windows Installer packages be installed on computers with little or no concern for where the computer might be moved—either permanent moves during system provisioning or temporary moves constantly made by mobile computers. Universal availability is related to local sourcing for mobile computers whose “local site” changes depending on where they are located.

FRS provides replication services for DFS. FRS is the technology that replicates AD objects. FRS is a simple replication system for synchronizing the contents of DFS shares. It automatically configures a replication topology, tracks when files change, and automatically copies updates to all replicated copies. As Figure 5.2 shows, each server that participates in replicating a set of files will map to the other servers and push files to them as changes occur.

FRS is also used to replication logon scripts, GPOs, and directory changes; however, FRS acts very differently when performing these replication activities than when it replicates DFS shares. I will discuss this behavior in more detail later in this chapter.

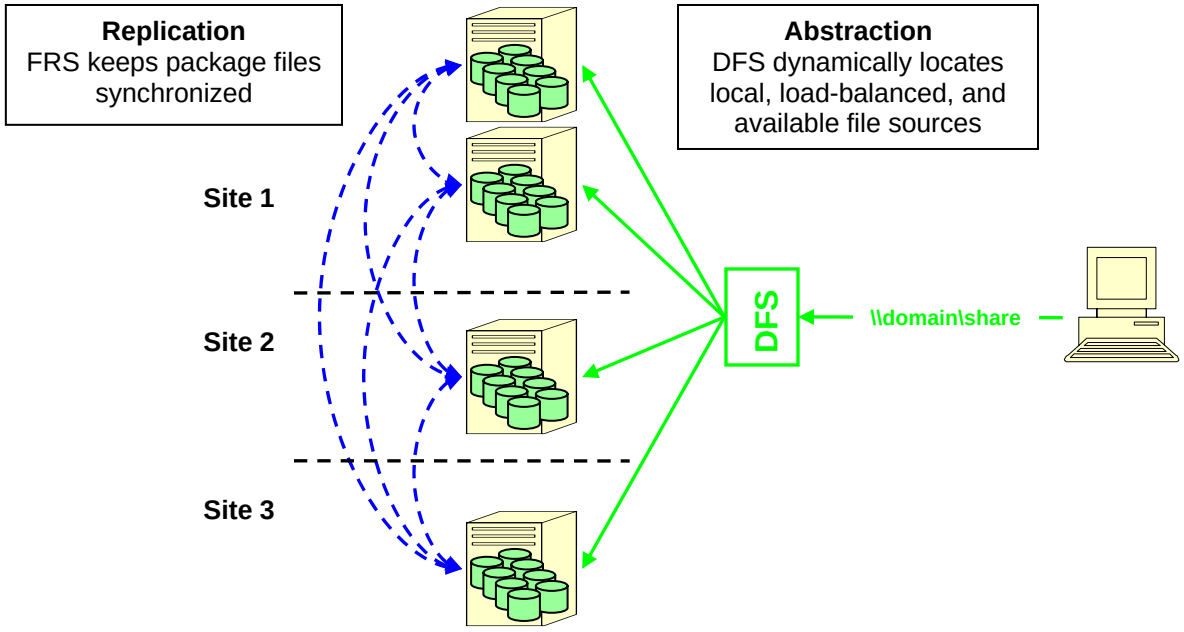

Figure 5.2: IntelliMirror repository technologies.

IntelliMirror Deployment Technologies Overview

GPOs are the main technology that allows Windows Installer packages to be distributed directly to clients without the use of an Electronic Software Distribution (ESD) system. As Figure 5.3 shows, Group Policy uses a client agent that reads AD objects to determine whether software packages need to be installed. This same agent reads the package installation information from the directory and triggers the actual installation of the Windows Installer package on the client. One of the biggest limiting factors in utilizing this technology is that clients must be running Win2K or later and AD must be used for directory services. However, even if AD is available, there are other limitations of Group Policy deployment that should be considered—we will discuss these limitations later in this chapter.

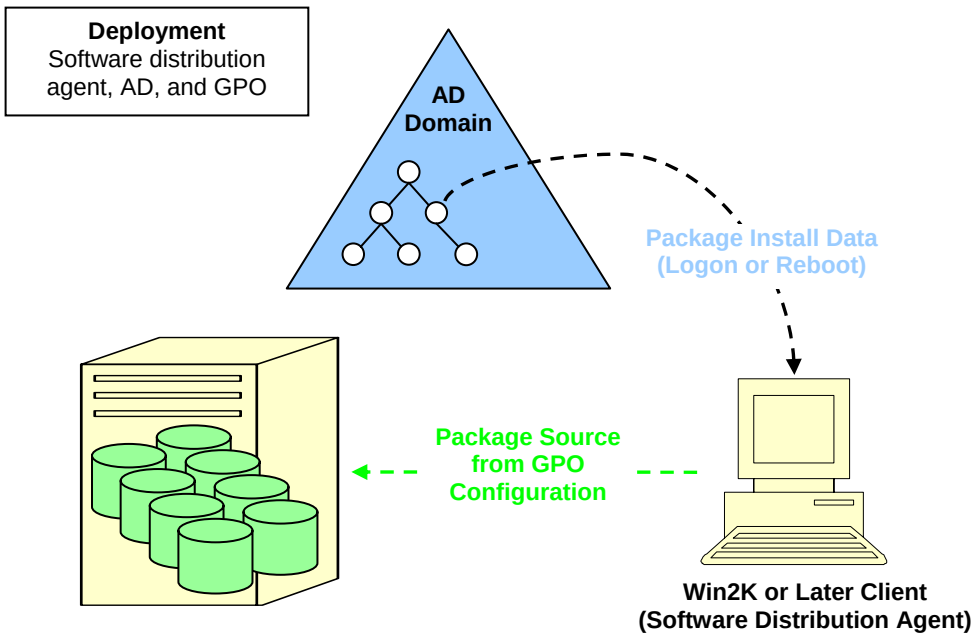

Figure 5.3: IntelliMirror deployment technologies.

Source Lists—the Good and the Bad

In the last chapter, source list management was mentioned as a necessary best practice. As mentioned in that chapter, each package installed on a computer keeps a list of places where the package source files can be found. Package source files include the package file itself (MSI), any needed transforms (MST), and the actual software application files. The software application files can be embedded in the MSI, external from the MSI in .cab files, or external from the MSI and uncompressed. Aside from the installation of the application itself, they are also used for self-healing, install-on-demand, patching, and for server-based client applications. Source locations are also used to re-cache missing transforms if they are unavailable in the local Windows Installer file cache.

> 🕮 Microsoft has enhanced Windows Installer 2.0 so that it does not need to touch the source files as frequently as previous versions do. In addition, Windows Installer 2.0 supports building patches that do not require access to the original package source files.

Source lists are intended to provide fault tolerance in finding source files, so they are processed sequentially until a valid source location is found. Source lists do not inherently provide any type of load balancing. A poor man's load balancing can be accomplished by randomizing the source list when the package is installed.

Source lists can be managed more loosely in a by-the-book enterprise IntelliMirror implementation because packages are generally installed from a DFS share. The DFS share handles fault tolerance as well as several valuable features. A by-the-book implementation will need to manage source lists if the overall approach includes any mobile machines that will be installing from removable media.

When no efforts are made to manage the source list for a package, the source defaults to the location of the package file when the initial installation begins. A casual approach to installation locations was the norm with setup.exe technologies, and most administrators do not discover this painful Windows Installer behavior until they have quite a mess on their hands.

There are three types of source lists—*removable media sources* for packages located on CD-ROMs, floppy drives, and so on; *network-based sources* for packages located on network drives or UNCs; and *URL-based sources* for packages located on Web servers.

Drive imaging is a popular method of deploying ready-made desktop system configurations. If Windows Installer packages are deployed on the desktop image, special care must be taken to ensure that the embedded Windows Installer source lists will be relevant in all the physical locations where the image will be used. Because decisions to use drive imaging might occur in a different part of your IT organization, this nuance can be lost in the shuffle of desktop engineering.

The following list provides the basic actions that can be taken to manage the source list:

- Manage which type of source lists are scanned first—removable media-based, network-based, and URL-based (SearchOrder policy).
- Add source list entries one or more at a time in a specified order (SOURCELIST public property or scripting API call).
- Force the SourceDir private property value so that the *recorded* initial location is different from the *actual* initial location (custom action). Doing so can only be accomplished if the forced location is available to the client at the time of installation.
- Use a custom action to make MSI API calls to clear the source list and add new sources.

WindowsInstallerTraining.com publishes a script called MSICAResetSources (MSI Custom Action Reset Sources) that inserts a custom action into existing MSI files that will implement the approach in the last bullet point. This script uses the default public property SOURCELIST to populate the source list—essentially overriding the normal behavior of making the install location the first source location. If SOURCELIST does not exist, the custom action does nothing, which ensures that your package defaults to the normal expected behavior. You can download this script from http://windowsinstallertraining.com/msiebook.

You can combine these methods to do things such as remove the original install location and add managed locations or add network sources and reorder scanning so that network sources are preferred over media installs (such as with laptops that load software from CD-ROMs).

Environment variables can also be used in source lists to make the lists dynamically point to site-level repository locations. If you are using an environment variable strategy rather than a managed UNC or drive location, the source list must also be cleared of the initial installation location. The reason is that any environment variables used in the initial installation location will be resolved to a literal location before Windows Installer processes the package. This literal location then becomes the first source list location—which does not include the environment variable to make it dynamic. This occurrence is in contrast to a managed UNC or drive location, which does not require source list management because the embedded literal location is made dynamic through DFS or dynamic drive mapping.

Using environment variables causes the "dynamic" nature of your source lists to be distributed by being embedded in every package configuration on every computer. Using a managed UNC or drive letter leaves the "dynamic" nature of source lists in the hands of a centrally manageable infrastructure.

Trickle Services, CD-ROM Distribution, and Source Lists

Many distribution systems, including SMS 2003 (code-named Topaz), support the ability to trickle a package down to a client slowly over time, then execute it from the local drive. These types of services generally do not attempt to manage the source list for your package, so the first and only source list entry will point to the locally cached copy. Locally cached copies can be deleted by the distribution system on some clean up interval. In some companies, mobile users who are always remote use a CD-ROM for software distribution. In both of these cases, it is important to manage the source lists for installed packages. You can use several approaches to manage the source lists.

One approach is to include a couple of network-based source locations in the SOURCELIST property when launching the package. This approach causes the client to search for network locations first and then for the media it was originally installed from—if the mobile user keeps the media handy, they can use it. If they do not have it, they can connect to the network and attempt the Windows Installer operation again.

It might also be worthwhile to set the SearchOrder policy to *nm* to ensure that Windows Installer searches network sources (*n*) before media sources (*m*). *nmu* is the default behavior for this policy when it is not configured (*u* being URL-based sources), but setting the policy explicitly helps others to know that it is a managed policy value. It should be set high in your OU hierarchy to prevent overriding at lower levels.

Another approach is to clear the source list of all sources, then add only network sources. Doing so forces mobile users to connect to the network to gain access to package files. You might want to do so if you have multiple versions of packages and want to ensure that the mobile user does not attempt to use an old CD-ROM as the source for an installed package. You can implement this setup by using in-house maintenance scripts or the MSICAResetSources script mentioned earlier.

If you've always wanted to use trickle-type distribution, you might already have it for free! Windows XP and Win2K SP3 include the Background Intelligent Transfer Service (BITS) for supporting trickle down of Windows Update files. BITS works very well—detecting when to resume, limiting bandwidth usage, and so on. By using the BITSADMIN.EXE utility on the Windows XP CD-ROM, you can schedule your own files to deploy from a Web server (using HTTP) to a client through BITS. For more information about how to use BITSADMIN, visit http://desktopengineer.com/bitsadmin.

Fixing Existing Unmanaged Sources

Many of you might be feeling a sense of doom in regard to package installation locations because you have not been managing them for the past 2 years that installations have been occurring. You might have thousands of computers that have installed packages from many different locations that might no longer be valid. Or perhaps you are going through a process to consolidate locations or migrate servers—in all of these instances, package source lists might be pointing to incorrect locations.

MSISources is a script written for just this purpose. It allows you to identify which packages you want to manage source locations for in a control file. The control file is then used by MSISources to find packages that should be managed and point them to new locations. MSISources allows drive letters or UNCs and allows absolute paths to be used (for example, \\Server\Share\acrobat\version5). In addition, MSISources allows re-rooting of existing source list paths. This feature facilitates easily consolidating or moving existing source locations without manually coding all the paths into the control file. When source lists are re-rooted, each existing source list is retrieved, and the drive letter or UNC is replaced with the new root location.

MSISources allows flexible remapping of the source lists of multiple installed packages at one time. For example, by running it in the logon script, you could use it to ensure that every installed copy of Visio on every computer in your company has its source list pointed to a managed location. You can download MSISources from http://windowsinstallertraining.com/msiebook.

Designing the Package Distribution Repository

In this section, we will talk about the considerations in designing the package distribution repository where packages will reside so that clients can access them for installations, self-healing, patching, and upgrades. It is actually the scaling of the package distribution repository that creates the challenges and solution alternatives we will be discussing. If you work at a small, single-site company at which all packages can be put on one well-managed server, simply installing all of your packages from this server location would handle most of your repository challenges. However, as soon as the repository must take into account multiple sites, mobile computers, slow links, global organizations, and other similar factors, the repository must be designed and deployed correctly to effectively support enterprise-class software distribution.

In Chapter 3, we briefly touched on the importance of these design activities. The following initial discussion will identify some reasons why organizations that have the option to use DFS might elect to choose other technologies. The remainder of the section deals with alternative configurations to DFS—this information will apply equally to those who could have DFS but choose not to and those who cannot have DFS due to technology or business limitations.

The guidelines given here should be used carefully. A mix of DFS and non-DFS file sources is a viable solution. In fact, larger sites (as judged by the number of seats) that generally require fault tolerance and load balancing might also have the IT resources to justify a site-level DFS solution. Smaller sites could go without DFS if it posed design, deployment, or scalability difficulties for enterprise-wide implementation.

When to Choose Something Other than DFS

DFS deployment can be challenging to deploy or inappropriate for your computing environment. You might want to consider alternatives to DFS if your situation is characterized by any of the following statements:

- You will not be deploying AD.
- You have few sites or very small sites—DFS deployment in these locations might be overkill simply to support software distribution.
- Global DFS can be difficult to design and deploy. DFS has some roots in the NT 4.0 version of this technology which generates some limiting design recommendations such as 16 DFS servers per DFS namespace and only one DFS root being hosted on any given server. In many global environments, these limitations can prove difficult when DFS is being leveraged for multiple purposes. If you organization is planning multiple AD domains, DFS planning can become even more complex.
- Not all sites in your organization are moving to Win2K and AD before Windows Installer packages need to be deployed. In companies with large overseas contingents or companies with regional budget responsibilities, technology spending tends to be need-based. If desktop management is the only requirement for a given site to move to AD and DFS, there might be push back on these costs.
- An extended desktop deployment to Win2K or Windows XP is planned. During this time, package repository services will still be needed for Windows Installer at all sites and/or for pre-Win2K desktops. For example, if you need to distribute the latest release of your MSI packaged virus software when your company is only halfway through a Win2K deployment, you will potentially need to deploy Windows Installer packages as sites without DFS and to clients that are not Win2K.
- Delayed plans to move to AD can cause Windows Installer software distribution to become a higher priority as software vendor packages and internal packaging initiatives continue to develop. In this situation, the repository must be built before AD is available and be compatible with AD once it is in place.
- Large environments might never be able to support AD for all desktop clients due to the sheer diversity of their networks or server environments.
- A mass project to move to AD and Win2K or later clients is planned, but the project planning is unnecessarily hampered by DFS dependency on AD. In this case, requiring that AD deploy (even in a limited fashion) before desktop computers are upgraded can create costly dependencies for infrastructure migration projects.

DFS Functionality Alternatives

Earlier we talked about four main functions provided by DFS. Table 5.1 shows possible alternatives for these functional areas. A complete discussion of these alternatives is not possible in this book, but specific solutions are discussed in more detail.

DFS Functional Area	Alternatives
Local sourcing (Getting files from the local site)	• Managed drive letter • Managed environment variable • Source list management
Universal availability/file location abstraction	• Managed drive letter • Managed environment variable
Fault tolerance	• Source list management
Load balancing	• Complex source list management

Table 5.1: DFS functionality alternatives.

As with DFS, these alternative solutions have some infrastructure requirements:

- A managed drive letter generally requires logon script changes so that the same drive letter is pointed to a local copy of the package repository at all locations. If this approach is extended to mobile machines, additional logon scripting is required to ensure that mobile machines map their drive letters to local resources when they are away from their home sites.
- A managed environment variable might require similar logon scripting changes to ensure that the environment variable used is set correctly for every computer, including mobile machines if traveling support for local sourcing is desired.
- Source list management might be as simple as including additional sources during package deployment, then using the policy to change the search order to network-based sources first. This solution is more static and generally cannot be easily adapted for mobile computer needs.
- Complex source list management can provide a method of mimicking load balancing using an install-time script that ensures that each client install occurs from a pool of possible locations. Instead of using a static source list property, a script must be devised that can dynamically randomize a site-specific list each time a given package is installed. To accommodate mobile users, your list should always include a standard corporate location or site-specific location (environment variable) to ensure that mobile users can get to package sources while traveling to other locations.

Figure 5.4 illustrates that you can use a combination of both source lists and file system approaches to effectively manage the package distribution repository. The illustration is showing all possible methods—any given company should be utilizing the same method for all packages. The Microsoft Office package in Figure 5.4 demonstrates the generally assumed model for IntelliMirror. DFS is set up to handle the main repository requirements—the source list on the client then only needs one entry. If the package is originally installed from the DFS location, the installation defaults for the package source lists work without need for changes or maintenance scripts.

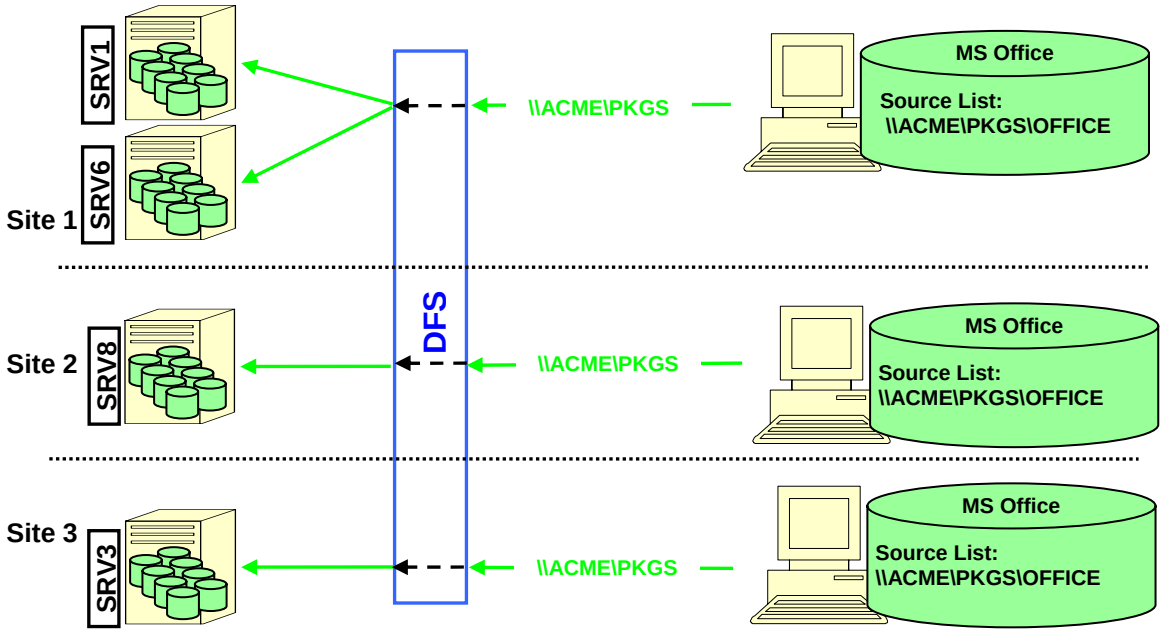

Figure 5.4: DFS for abstracting package source location.

Managed Drive Letters and Managed Environment Variables

Long before Windows 95 came on the scene, many companies accomplished local sourcing and file system abstraction by using a managed drive letter. Each physical location would run specific logon scripts that would set one or more drive letters to local locations. Users at every site would know that the Y drive was for data and that T contained software to install. Drive letters can still bring the benefits of local sourcing and abstraction to the current Microsoft world, and indeed, many companies have never abandoned their drive letter strategies for this reason. Figure 5.5 shows how drive letters can be used to abstract the repository to a local location. As long as packages are initially installed from Q, the source lists can remain unmanaged.

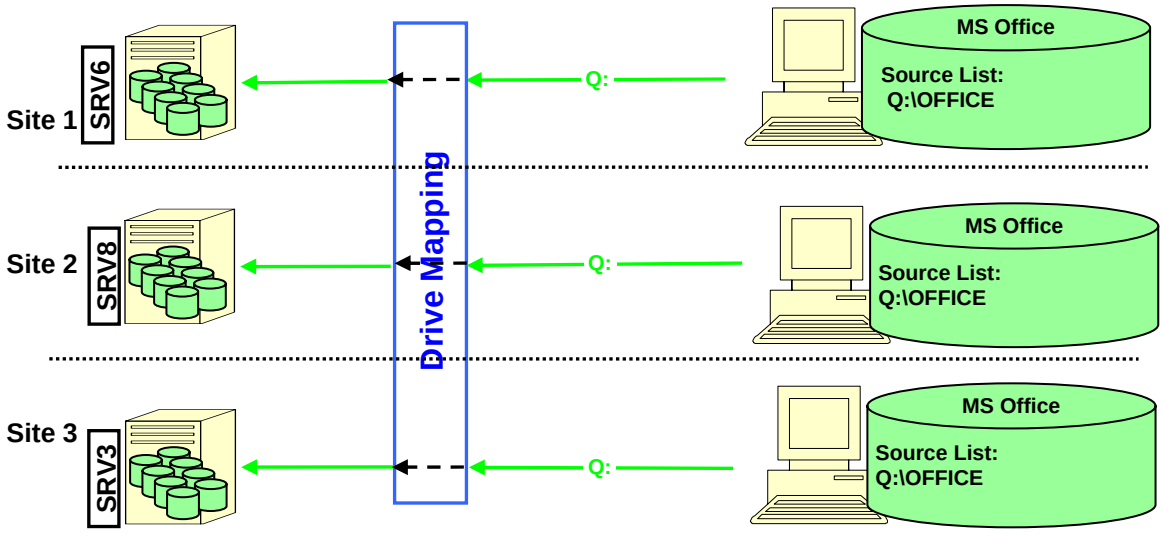

Figure 5.5: Drive letters for abstracting package source location.

Drive letters are a well-established method of mapping to network-based file systems in Windows. Many different file systems are supported. The following list provides some of the benefits of a managed drive letter:

- Eliminate DFS deployment concerns—Earlier we discussed several key concerns with the deployment of DFS. Using a drive letter solves many of these problems, including multiple domain AD designs (with possible multiple DFS namespaces), design constraints in scaling DFS, and small sites at which resources for supporting all infrastructure services might be scarce.
- Relieve DFS domain bounded nature—DFS namespaces are in the context of a domain. Organizations that will be using multiple domains in their AD design will be faced with either creating the same software distribution DFS share in each domain (and deploying packages within each domain to that specific share) or using the DFS namespace of one of the domains in all the other domains, which would necessitate a domain controller with a DFS server running in close network proximity to each client that must use the share.
- Eliminate deployment project dependencies—By using existing managed drive letters, extended Win2K deployments do not hamper the building of a distribution repository. This benefit addresses the problem in which desktop clients cannot deploy until enough DFS infrastructure is present to support package deployment. It also handles scenarios in which Windows Installer packages will need to be deployed to pre-Win2K clients while Win2K infrastructure is trickling through an enterprise organization.
- Enable offsite, offline, and OEM builds—Some workstation build processes require that packaged software be called during the build. When the build depends on resources that are on your company's private network (such as DFS), builds cannot be done while disconnected or by third-party system-building organizations.

- Eliminate source list management during system provisioning—Packages can easily be loaded from any file source connected to the managed drive letter. When the client is deployed to the final location, source lists will work fine without clearing or rebuilding the list because the same drive letter is mapped to the relevant local resources.
- Eliminate drive imaging concerns—Using the same drive letter for software eliminates concerns for deploying drive imaged systems because the drive letter can easily be made available in every location.
- Work with latest technologies—If necessary, drive letters can also work with technologies such as DFS by simply mapping a drive letter to the DFS share name. Drive letters can be used by Group Policy deployment as well.

> ☞ Group Policy will automatically resolve the UNC of a network mapped drive and embed it in the software distribution configuration. To override this behavior, use the SUBST command (built-in console command) to map the drive.

- Work with all client OS versions—Domain-based DFS share access is limited to Win2K and later desktop OSs. Additional DFS functionality can be achieved by previous desktop OSs by installing the AD client; however, an enterprise-wide deployment of the AD client can pose its own challenges.
- Work with any network client type—Whether you are using Microsoft or the Novell network client (or a third-party client), a drive letter mapping will work.
- Use any back-end server system/file system—Any file system that a drive letter can be mapped to could be used as a repository, including UNIX systems, mainframe systems, mid-range systems, and any other type of server that can have a drive mapped to it in Windows.
- Turn off drive letters—By unmapping a drive letter, the repository can be made unavailable. When using DFS, if a local repository server is not available, DFS will randomly select one from the list and map to it over whatever network links are between itself and the first successfully contacted server for the DFS namespace. In some environments, this over-the-WAN behavior has a much higher impact on system and network resources than the problem of individual desktops not being able to load software.
- Allow for a universal approach—Because drive letters can be mapped to any network resource as well as any local device (such as CD-ROM, zip drive, hard drive, LS 120 drive) the same drive letter can be utilized for local and remote package loads—keeping source lists consistent across online and offline storage.
- Enable site-level responsibility for repository—Some companies operate with independent IT resources at each site. Standards are used to ensure interoperability. Using drive letters for the repository allows the standard to be loose—letting the site determine how to store repository files. The site can implement local repository services by simply replicating the files from an agreed location and mapping the repository to an agreed drive letter.

- Integrate with many ESD systems—Using a drive letter allows the distribution repository to be utilized by many software distribution systems. The drive letter can also be leveraged via custom pull menu systems and by technicians who can browse the distribution repository directly. SMS 2.0 and 2003 (code-named Topaz) can be configured to use such a repository. Any ESD that can pull files from a drive letter can most likely work with this configuration. Some ESD systems use specially prepared file sets and cannot source files directly from the file system mapped by the end user.
- Use drive letters at times when DFS cannot be used—Suppose you wanted to use the repository for building the OS on workstations and you are using a DOS boot disk—the DOS boot disk can access a drive letter-based repository. You can use drive letters when DFS is not an option.

Figure 5.6 illustrates that the repository can be leveraged for many software distribution activities, including the initial machine build.

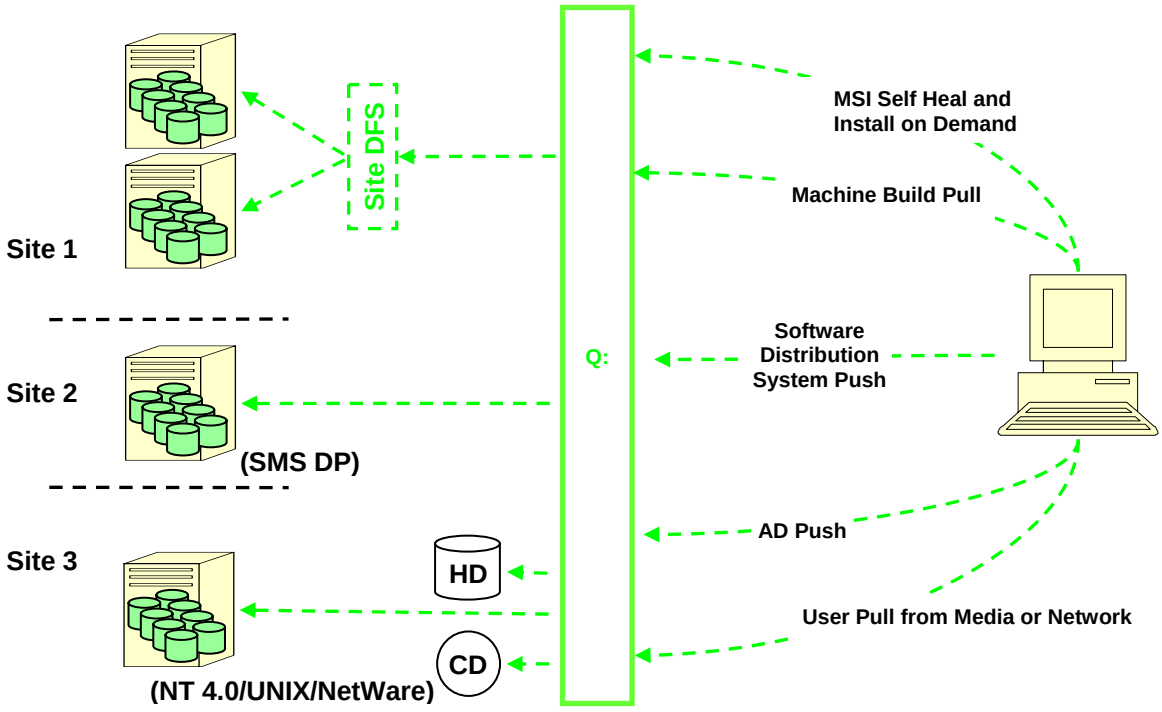

Figure 5.6: Multiple use repository.

Use an Existing Drive Letter

It might be difficult to clear a single drive letter across your organization just for distribution packages; however, you might be able to use an existing drive letter. A new subdirectory can be created on an existing shared drive. With Win2K server disk management, this subdirectory could be served by new, dedicated disks. Server-level DFS can also be leveraged to make the subdirectory point to another server.

Use an Environment Variable in Source Paths

If a dedicated managed drive letter or a directory on an existing managed drive letter does not work for your organization, it is possible that an environment variable could provide the flexibility needed to manage source lists. Source lists will allow an environment variable to be embedded in the source reference, and Windows Installer will properly retrieve the environment variable value before scanning source locations. An environment variable can provide a single reference in all packages that can be set individually for specific sites, regions, or even individual computers. Environment variables can also be set dynamically for special deployment jobs such as packages located in a non-standard location for security reasons.

The flexibility benefits of environment variables come with some challenges that are not present with a drive letter solution. Some ESD systems might not be able to invoke a package using environment variables in the command line. In many cases, you can work around this drawback by having the distribution system call a batch file. Figure 5.7 shows how an environment variable can abstract package source locations to point to any type of file system resource at local sites. Note that the source list must be cleared of the initial install location and the environment variable location added.

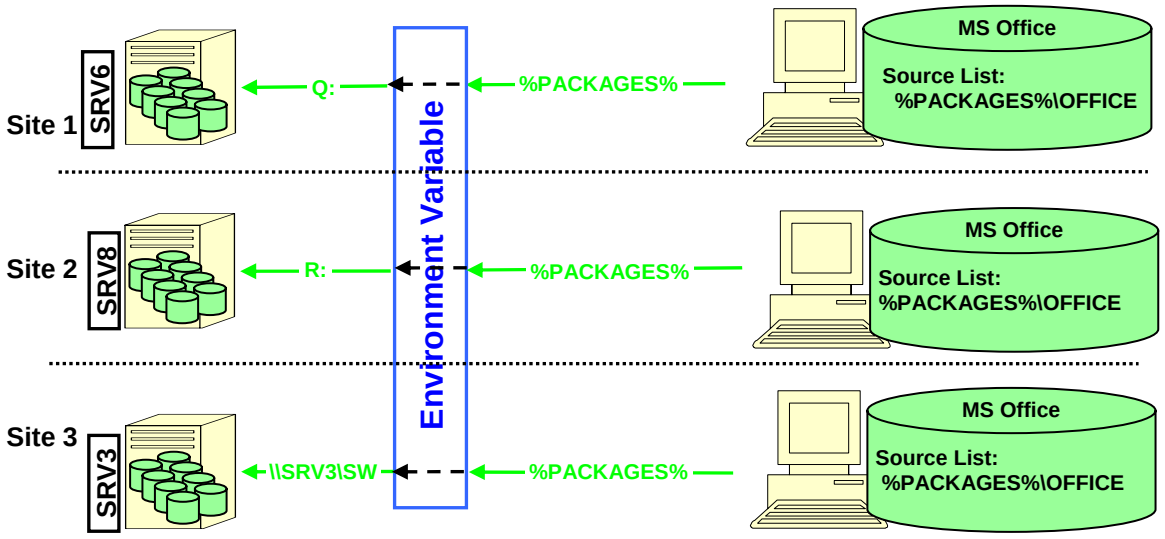

Figure 5.7: Environment variable for abstracting package source location.

The use of environment variables also requires some source list management through which a drive letter strategy can be devised that does not require source list management at all. Source list management is required because the first source list location is the location from which the package was originally installed. This original location must be resolved to a literal disk location by Windows Installer in order to get the package installed on the system; it will always contain a direct reference to the location of the package, not any environment variables used when calling the package location. The absolute location must be cleared from the source list in order to rely completely on the environment variable.

Drive Letter and Environment Variable Infrastructure Requirements

Universal drive letter and environment variable solutions have some infrastructure requirements that DFS handles automatically. DFS automatically provides information about where in the physical network a client is located. DFS does so dynamically, regardless of how the client connects or whether logon scripts run. DFS then maps the client to local servers for the given domain-based share. When creating a custom solution, some or all of these dynamic capabilities might need to be mimicked if your environment requires them.

Many organizations already have well-established logon script designs that can easily suit repository mapping needs as well. If you have deployed AD, you can use ADSI scripting to learn the physical location of the computer and map the local repository based on that information. Without AD running, a custom solution must be devised to assist with pinpointing the physical location of computers on your network. The IP-based cool tool solution that will be discussed shortly has an advantage over ADSI scripting in that it can match multiple hierarchical location contexts such as Europe AND France AND Paris and Mercer Building. The follow two cool tools can help you by providing some key elements to a scripted solution.

- SiteSense.vbs is a script for detecting a computer's exact physical location by determining which subnet the computer is on. Once the computer's physical location is pinpointed, it can be used to map a local version of the package repository. This functionality is similar to AD's built-in sites functionality. However, SiteSense only uses the computer's IP address, so you can use it with any version of Windows, any network OS, and any directory service. You can download SiteSense from http://windowsinstallertraining.com/msiebook.

- VPNRASLogonHook handles several cases in which logon scripts do not reliably execute. Logging on to the network from a VPN or RAS generally does not run the logon script. As of the release of Win2K, dynamically plugging in a network card also allows users to access network resources without running the logon script. VPNRASLogonHook uses WMI to detect when a network connection occurs and runs the logon script if it is available. You can download VPNRASLogonHook from http://windowsinstallertraining.com/msiebook.

Package Source List Management

Active source list management can help ensure that source lists correctly point to local package repository locations. This approach does not dynamically allocate local sources for mobile computers, but can provide an effective approach for desktop machines.

Some organizations use MSISources, which was introduced earlier, on the first boot of a newly built computer. A primer script detects the computer's physical location and remaps all managed packages to the local repository server. This script could be run during computer moves or scheduled periodically to correct source lists for desktop computers that change physical locations.

When to Choose Something Other than FRS

The Win2K version of FRS is capable of replicating reasonably sized logon script directories. Replication loads that are more intense in their size or replication frequency can be quite difficult to manage with FRS. The following list provides the primary attributes of FRS that make it unsuitable to the loads generated by large-scale replication:

- Multi-master replication methodology—A multi-master model means that each server copy of the replicated files can be independently changed and the changes will be automatically replicated to all other servers. This model is a big plus in the area of directory services but creates difficulties in software distribution where there is generally assumed to be a single master copy of software distribution packages that should be replicated to every other location.
- Full mesh replication topology—I have referred to this topology as "full mess" replication topology. When a set of servers are configured to keep a single, replicated copy of files, each server maps directly to every other server, regardless of physical network topology. Servers at end sites of the physical network will attempt to communicate with each other to exchange files that have changed.
- Unrestrained automatic operation—Once configured and turned on, replication happens automatically, immediately, and has no performance governors between physical sites. Thus, replication cannot be constrained in terms of bandwidth usage or scheduling times of day when distribution should occur.

Rudimentary editing of FRS replication topology and replication schedules is possible but a brief encounter with the editing utilities reveals these capabilities to be ill fitting in enterprise environments.

- Lack of human-readable logging—Although FRS has logging for its internal purposes, these logs use globally unique identifiers (GUIDs) to track multiple versions of the same named file. This system essentially makes them unusable for tracking replication problems. In addition, there is no audit tracking for finding out which user IDs might be responsible for accidental replications that can have a high impact on network and server resources.

Even when used in a LAN environment, the flurry of activity created by FRS replication might generate uncontrolled peak loads for networks and distribution servers.

FRS Alternatives

There are several alternatives to FRS that might be considered for replication of the package distribution repository. Specialized replication software manages replication by making sure that bandwidth usage can be controlled through scheduling and throttling. Replication topology can also be designed to coincide with physical network layout and individual link utilization.

Many enterprise-level ESD systems have robust built-in replication. Bandwidth can be scheduled and throttled and topology can be managed. It is unlikely that an ESD can be cost justified based on replication needs alone; however, if you already have one, you might be able to leverage it for your repository needs.

Any replication system (or the underlying network) can be overloaded if your package release strategy does not account for long-term network bandwidth requirements and server disk space requirements. For instance, if your methodology allows software application owners to request package updates at any time and then build and replicate an entire upgrade package for every change, it can generate big strain on replication bandwidth and the replication management system. If you also keep older copies around for too long, disk space can be strained.

Many companies have used NT or later shell scripting (.CMD or .BAT) and the resource kit utility ROBOCOPY to implement their own replication scheme. Implementing replication with scripting rather than a product is a challenging exercise—especially in a large organization.

Tim Hill's *Windows NT Shell Scripting* (New Riders Publishing) contains a complete and robust sample script called REPL.BAT that uses ROBOCOPY to emulate logon script replication. This script could serve as a starting point for creating a script-based replication system.

BITS was mentioned earlier as a cool tool to perform trickle replication to remote desktop clients. It might also be used a method of low-impact replication between servers on your network. You will need to be running Win2K SP3 on your servers to use BITS directly on the server. In addition, the file source must be an HTTP server. If Win2K SP3 is not an option, you could script a Windows XP desktop to transfer the files and then move them to a server. BITS might not be a good alternative if you are transferring a large number of files—for instance, if all your package sources are administrative installations.

Directory Structure Issues

Up to this point, we have been discussing the repository location in terms of the root location—the location where packages are stored. This root location would be a UNC or drive letter. We could simply put all our packages at the root of this location if they all had different names. However, it is important to choose a directory structure for the repository that will be flexible and be scalable.

For pre-Windows Installer package repositories, some organizations have used a convention of having the latest package version in the same directory so that it is easy to locate. The location of the latest version might need to be known by technicians browsing the repository and by automated build processes and pull menu applications. Windows Installer adds a new wrinkle for this approach. When a package is installed from a network location, Windows Installer will return to that location to find the MSI file and the associated software application files. If Windows Installer determines that the MSI file is not the same as the one that was installed from that location it might refuse the location as a valid source.

This occurrence might not present a problem for applications that have a limited distribution scope (number of seats, number of sites) because they can all be updated to the latest version in a fairly short time. If, however, you are updating software in an organization of 30,000 desktops, the sheer time it takes to update every one might result in problems with self-healing or new computer builds if the new version is pushed to all distribution servers as the first step in deploying the updated package. If local sites have responsibility for triggering desktop installations, the time lag between a centralized push replication to distribution points and local site installation on clients might be unpredictable. This creates the same problem with regard to self-healing and other Windows Installer activities not completing due to the wrong version of the source.

You might want to consider a scheme that allows a distributed package to reside in its destination directory indefinitely and use another approach to identify the latest version. For instance, a batch file could be created called current.bat that would contain the command line to use for the current production version of the package. This batch file could be read and parsed or simply executed by automated processes and administrators.

Directory Structure Considerations

This section will consider several key Windows Installer issues when designing a directory structure for the package distribution repository. The following list offers some of the high-level considerations:

- If you are using administrative install shares, multiple versions of the package cannot have the same directory root. The reason is that each version will have files of the same name and directory location—the install share would end up being a mix of files from multiple versions. Windows Installer does not check the actual version of the file on disk, so an installation could be performed from this share, but the software might be inoperable and self-healing behavior could be erratic.
- Schemes that are too shallow could take a long time to enumerate, which causes delays when browsing the directory or enumerating it with maintenance or pull menu applications.
- Schemes that are too deep might not fit on offline media (if replication to offline media is a part of your scheme).
- If you plan to use *path rules* with software restriction policies, make sure that your chosen directory structure does not result in overly complex rules.

The following example provides a good place to start with directory structure:

```
<root>\<pkg_ident>\<version>\<filename>.msi
```

This structure allows each package version to exist simultaneously, which prevents problems with self-healing and install-on-demand during lengthy upgrade cycles. It also allows multiple administrative installs of the same package because each administrative install uses the <version> directory as its root. Some type of batch file or other file can be stored in the <pkg_ident> directory level to ensure the current version. This scheme can also work well for storing informational data such as instructions, documentation, logs, and so on. The following example shows an alternative to the previous scheme:

```
<root>\PACKAGES\<pkg_ident>\<version>\<filename>.msi
```

This scheme allows more portability and flexibility for where the software share is placed because it does not assume that it owns the root of the file location. For instance, if you need to place the distribution repository on an existing drive letter or DFS namespace, this scheme would be helpful. Even if you currently have a dedicated root location, consider what happens when you scale up your implementation. Also consider whether other IT departments might want to piggy back their needs on your repository strategy; using the PACKAGES directory allows multiple functional sublocations under the root locations.

You might be tempted to store the command line for the "current version" in SMS or some other distribution system configuration. This information will be much more flexible and accessible in a batch file because it can be utilized by less capable automation activities such as initial computer builds and by technicians without the need to rifle through the distribution system to learn the appropriate command line.

The following schemes are based on traditional non-Windows Installer approaches, so they are bound to come up during repository design. These schemes might not work well:

```
<root>\<pkg_ident><version>\<filename>.msi
```

When the version is attached directly to the directory level for the package identifier, the number of directories under the root location might become difficult to manage. This approach also makes it more difficult to establish a method of identifying the current version of a package because the files that do so must reside in the package root directory.

```
<root>\<pkg_ident>\<version><filename>.msi
```

When the version is appended to the file name and stored in the same directory as other versions, administrative installations cannot be used.

Administrative Installs

Chapter 3 covered administrative installs; in this section, they will be explored specifically in regard to considerations of building the repository. Most likely, you will have some type of administrative installs in your repository. The consideration is whether to make it a standard practice to make all of your network deployed packages into administrative installs. There are several issues revolving around using administrative installs in your distribution repository.

In Chapter 3, we covered some reasons why you might want to create administrative installs. The following list provides a quick review:

To present properties (such as TRANSFORMS) for execution when the MSI is double clicked.

To use served applications.

To pre-activate Microsoft products (and to deploy patches to them).

To reduce the required server-to-server replication load by patching admin shares with updates.

To extract only the needed files from your installation media.

Using administrative installs in your distribution repository can be a good long-term implementation decision if you have site-to-site bandwidth constraints because you can patch administrative installs to reduce the server-to-server replication load. Only patches need to be replicated and executed. There are some qualifiers to this approach:

- The initial replication load is much higher than compressed source files.
- The extra effort to build reliable patches is significant.
- An automated or manual activity must execute the patch on all distribution points—something that does not need to be done with straight replication of complete packages.
- If you want to keep previous versions intact for self-healing to work properly, you must locally copy the administrative directory to a new location before applying the patch.
- Offline media replication (sending a tape or CD-ROM) might be less work than a full-scale patching approach.

The following list provides additional considerations (aside from a patching strategy) that should be a part of your discussion when designing a distribution repository that includes administrative shares:

- You will most likely have to deploy some administrative installs, even if your strategy is to favor not using them. For instance, Microsoft Office products assume that network distributions are always done from administrative installations.
- If you use administrative installs for all of your packages, disk space requirements will be much higher—potentially double that of compressed .cab files or .cab files stored inside the MSI.
- The number of files being replicated will be much higher with administrative installs. It will be important to ensure that your replication technology is up to the task.
- If you are using code signing of MSI packages for your packages, you will need to replicate the signed version. Patched administrative installs will need to be re-signed.
- Directory structures for a repository might become too deep to fit on offline media.

This list might make it sound as though administrative installs are not a good idea. Such is not the case. Rather, because most traditional package repository schemes use some type of compressed source (setup.exe files), it is easy to overlook the many additional considerations of using Windows Installer administrative installs.

AD cannot deploy patches directly to clients, which seems to have led Microsoft's software deployment guides to emphasize administrative installs for network-based deployments. If patching administrative shares is not a part of your deployment strategy, you can effectively use compressed source files (external or internal .cab files) to manage your distribution repository.

Repository Availability Service Level Agreement

The magic of Windows Installer self-healing implies to many end users and business departments that their applications will never break, no matter how far mobile computers travel from their home base. As you build a repository strategy, you will learn that it would take an immense amount of human and technological resources to keep that promise in every single usage scenario in a global company. It is, therefore, important to establish a service level agreement (SLA) that sets reasonable bounds around when users and business departments can expect magic and when they will receive a prompt for a package location.

For example, perhaps self-healing can be expected to work when mobile users are on their home continent. For some organizations, self-healing might only be expected to work when mobile users are at their specific physical site. Whatever the case is at your company, be sure to go through the design activity to balance your implied or formal SLAs with the IT resources that are required to get the job done.

Package Deployment Technology Planning

AD and Group Policy provide the capability for an out-of-box Win2K implementation to perform software distribution to desktop computers. IntelliMirror technologies can provide a workable software distribution solution for workgroup environments or companies that have fewer sites that operate fairly independently. There are two high-level indicators that you might need more than IntelliMirror can provide:

- If AD and/or Win2K or later clients are not in your future or are in a slow-burn deployment plan, AD deployment capabilities might not help you in the immediate term.
- If you are accustomed to a full-featured software deployment system, AD might not be able to meet your current SLAs.

Many companies assume that they can eliminate the cost of their current software distribution system when moving to a full deployment of AD and Win2K or later desktops. Once this idea gains momentum, it can be difficult to turn back. I advise that you insist on a thorough feasibility study of whether IntelliMirror is up to the challenge of your current distribution system requirements before this momentum starts to build.

IntelliMirror has limitations, some of which stem from the fact that it is piggybacking on a directory service, others result from the simple model of deployment scenarios that IntelliMirror targets. It can be difficult to get a comprehensive view of how these limitations stack up against existing and future SLAs.

SLAs are a critical part of technology design activities. SLAs can be formal or informal. For instance, you might have a signed document stating that a user will receive a software distribution within 24 hours of requesting it. Informal SLAs can carry as much or more clout than formal ones. It might just be a foregone conclusion in your company that software distributions always occur at night to prevent user interruption—even if there is no signed document to back this assumption, everyone intuitively understands what would happen if distributions were to be done during working hours. Assessing whether a technology can meet your formal and informal SLAs is a critical step and can be an early indicator of whether your implementation will be ultimately accepted by all stakeholders.

IntelliMirror Fine Print

IntelliMirror has limitations with regard to how it can classify objects because it uses the directory service to do so. It also has very limited functionality for non-MSI packages. In the areas of reporting, logging, and scheduling distributions, IntelliMirror is lacking. The following bullet points provide a fairly complete list of these limitations and their potential effects on your software distribution design:

- The ability to target users and computers for software installation is limited to AD's OUs. This hierarchical container classification only allows a single container membership for each user and computer. Many ESD systems allow for objects to be classified in multiple groupings or containers. In AD, container membership is arbitrary, while many ESD systems allow targeting by inventoried hardware information. IntelliMirror allows filtering of distribution targets by security groups, and in Windows XP, IntelliMirror allows filtering by WMI queries executed on clients; however, this flexibility comes at the cost of multiple layers of target filtering that are evaluated at different stages of distribution.
- Most of AD's inheritance is within the bounds of a domain. Distribution targets, security groups, GPOs, and delegated authority are all relative to the domain they are configured within. If your company will have multiple domains, many distribution configuration activities will need to be repeated for each domain. In addition to being a lot of work, this requirement can lead to quality issues when detailed work such as this must be repeated many times.
- AD's classification scheme (sites, domains, OUs) is leveraged by a company for many IT management and business unit purposes. With so many requirements that stem from outside of software distribution, the directory structure naturally becomes inflexible to all but the most pressing needs. Because ESD systems focus on software distribution, many times their underlying classifications schemes can be adjusted or reworked with no implications for other parts of the IT or business organization. In addition, excessive fragmentation of directory nodes (OUs) for the purpose of distribution targeting can lead to an unmanageable directory configuration.
- Non-Windows Installer package distribution is difficult and severely restricted in IntelliMirror. Non-Windows Installer packages cannot use elevated privileges and they can only be published to the user. Thus, a user must visit the Add/Remove Programs applet in the Control Panel to kick off the distribution. Although MSI wrapper scripts have been developed by Microsoft and others to distribute items such as service packs, they are definitely a workaround use of Windows Installer technologies and can be difficult to learn and maintain.
- IntelliMirror uses a simple methodology for triggering distributions. During the boot up and logon process, users are not using applications and cannot have locks on key system resources or applications that need to be updated. For this reason, IntelliMirror only distributes software at that time. This activity can, however, result in peak loading for distributions because many users logon or boot up at common times.

Software distribution policies differ from other Group Policies in that they DO NOT refresh every 90 minutes—they only occur at boot up (for computer-targeted distributions) and logon (for user-targeted distributions).

- The inability to schedule distributions to desktop clients is a result of the simple methodology mentioned in the previous point. The implications of this limitation vary by company. Some companies use scheduling to prevent interruption of business activities (for example, overnight distributions), other companies use it to offset the load placed on the network and distribution servers. To help with this, IntelliMirror only advertises software when it is assigned to users. However, as discussed in Chapter 3, there are several key reasons why software should be installed per computer rather than per user.


Windows Server 2003 will allow users to have software completely installed when assigned rather than only being advertised.

When to Consider Alternatives to IntelliMirror Deployment

The previous considerations might be difficult to evaluate for any given organization. The following list enumerates attributes of your technology environment and key distribution system features that might indicate that you need to choose another technology for software deployment:

- Large-scale implementations with large numbers of clients and/or many sites might find that the lack of logging restricts their ability to deliver distribution services in the same fashion traditionally expected by business units and end users.
- Desktop management and software distribution activities have formal SLAs associated with them that require a distribution success rate or exception reporting. Environments with formalized service levels might find IntelliMirror unable to provide needed detailed and summary reporting to prove the service level is being met.
- Homogenous environments that will always contain many different directory services, desktop OS, and server OSs might find the limitation to AD to be too restrictive.
- The inability to schedule distributions can have a surprising number of cascading effects on other distribution activities. Ensure that this factor is evaluated thoroughly.
- Organizations that want to manage servers as well as desktops with the same system will find that IntelliMirror's focus on interactive use (reboots and logons) and Windows Installer technology make it a difficult fit for server distributions.
- Companies that are coming from distribution systems that have a mature set of distribution capabilities (including SMS, Novell's management products, or any of the integrated desktop management suites available) might be surprised by what they would be giving up to move to an IntelliMirror-only system.

This last section paints a restrictive picture for organizations that choose IntelliMirror for their package deployments. IntelliMirror is a very respectable solution for the class of businesses that have traditionally been targeted for Microsoft's Small Business Server offering. For these companies SMS is definitely overkill.

However, this ideal implementation target for IntelliMirror distribution has been stretched by unrealistic expectations of upward scalability and cost effectiveness for enterprise environments. If you feel that IntelliMirror might be suitable for software distribution in your environment, be sure to do your homework.

Summary

Platform 2000 (Win2K servers + AD + Win2K desktops) was presented to the market with intricate interdependencies that can make it difficult to understand how to extract value from specific subassemblies such as Windows Installer. Hopefully, this chapter has unraveled a few mysteries and given you some alternatives to think about if you are implementing Win2K management technologies. This chapter is the last to be written by me, as Jeremy will be wrapping up the book with a final chapter about distribution systems. I hope to cross paths with you at conferences, in my training class, and on the Internet!

Chapter 6: MSI Deployment Roundup

by Jeremy Moskowitz

They say that getting there is half the fun—if such is the case, you've already experienced half the joy of your MSI deployment journey. Up to this point, you've been through the ins and outs in getting your MSI file "just right." In Chapter 1, you learned about what an MSI file is and why you would want to use one. In Chapter 2, you discovered the tools you can use to create MSI files, and in Chapter 3, became familiar with the internals of the files. In Chapter 4, we explored the best practices for building MSI files, and in the last chapter, you learned how they work inside and outside AD environments. Now, you're ready for the last leg of the journey—the deployment to your client systems of the MSI files you've created. In this chapter, we'll explore myriad free, cheap, and third-party deployment options for getting the MSI file from the administrative workstation where you developed the package onto the plate of each of your systems. Let's start with MSI deployment methods that are free.

> ☞ While reading this chapter, try to determine how and where to standardize. You will be much happier with your day-to-day MSI deployments if you can find one "tried and true" method for your environment and stick with it. All of the following options are approaches to solving the same problem—getting the MSI file onto the user's desktop and installing it. With so many possibilities, you'll benefit from standardizing on one method, if possible. If that's not an option, you can standardize on an option for each specific type of problem. For instance, you might use one method for the home office and another for branch offices.

MSI Deployment for Free

When it comes to deploying your MSI file to your client base, you'll need some sort of deployment mechanism to get the MSI tool to the desktop. Free is good, but be wary as oftentimes you get what you pay for. The free deployment methods each have pros and cons (the pros generally being that the method is free), but on a limited budget, these methods might just be the best or the only option.

Sneakernet

Sneakernet is the tongue-in-cheek description of, basically, running around to each workstation to perform the same task. After you have developed your MSI file, you could, theoretically, burn a copy to CD-ROM and roam the halls, hopping from desktop to desktop to perform the installation. This method is as simple as popping the CD-ROM or floppy into the user's system, and double-clicking to start the installation. However, if the user does not have Administrative rights on his or her workstation, a simple MSI package could generate a host of miscellaneous installation or runtime problems, as Figure 6.1 shows.

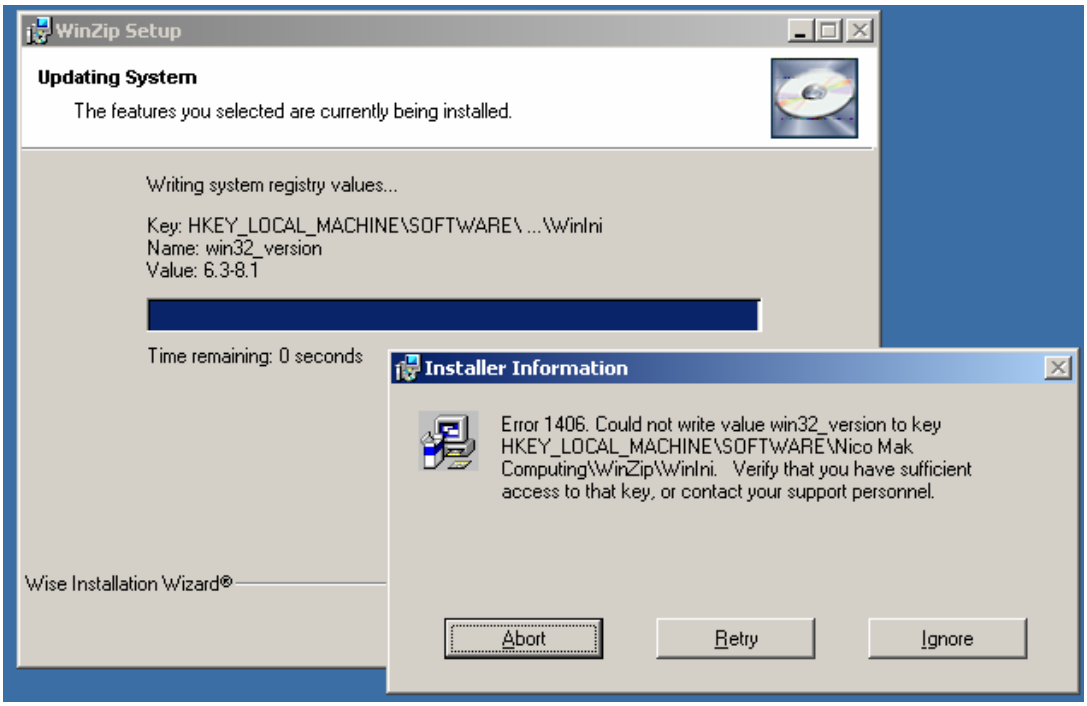

Figure 6.1: Most packages require Administrative privileges.

This scenario represents most environments, in which users don't have administrative rights on their workstations. Thus, you will need to perform the installation on behalf of each user through Sneakernet. To do so, you'll need to elevate the user's installation rights, and perform an installation on behalf of a user by using the runas command. For information about runas, see the following sidebar "The Runas Command."

The Runas Command

The `runas` command has the ability to allow a particular process to run in the context of another user; say, the administrator of the local workstation. Once the rights are in an elevated state, the administrator has the control required to accurately perform the installation. The runas command takes the following form:

`runas /user:{DOMAIN}\{username} command.exe.`

After you run this command, the system then prompts you for the password of the user to elevate. If the command you want to run has any spaces, you need to put the command into quotation marks.

To launch an MSI file in an administrative context, you can't simply specify the .MSI file on the command line; instead, you need to call the parent application for an MSI file—MSIEXEC. As you might recall from Chapter 1, one of the command-line syntaxes of MSIEXEC—specifically msiexec /i—will perform an installation. You can use runas and msiexec together, as Figure 6.2 shows.

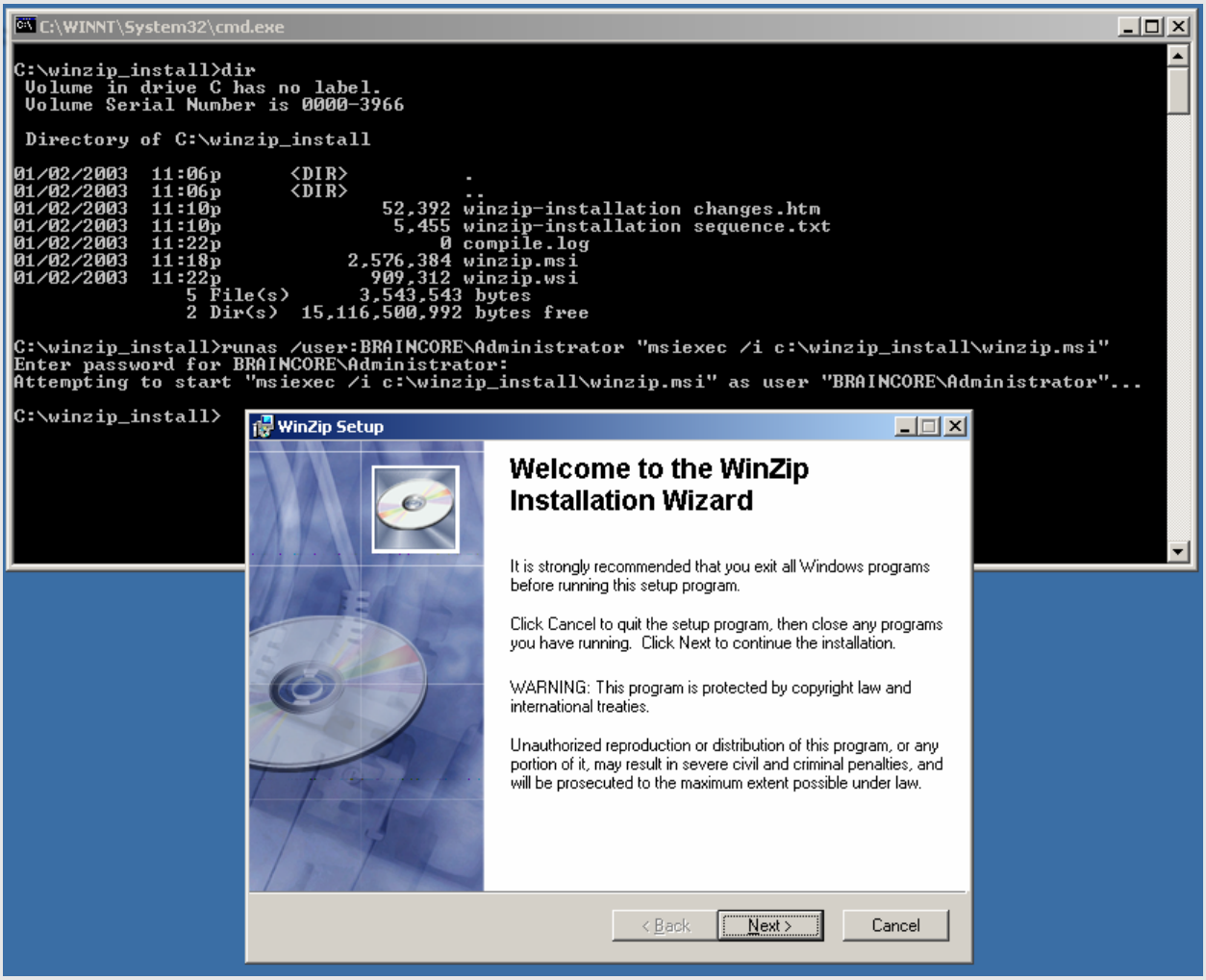

Figure 6.2: The runas command can help install packages that require administrative privileges

Sneakernet will work best for only the smallest environments. After about the tenth workstation, the installations will become very tedious. In addition, only the most finely tuned MSI packages will be successful by using Sneakernet because, in general, your MSI packages will usually ask for qualifying information, such as the directory to install to, the features desired, and so forth. If there is more than one person performing the installation, there's likely going to be more than one method of installing the software. More methods equals more errors equals more problems. Even with the best documentation, the best staff, and the best of intentions, there is still the great chance of error during installations. Unfortunately, even with the best of intentions and direction, when you give an .MSI file to 100 people, you could potentially wind up with 100 different installations.

Sneakernet has the following cons:

- No guarantee that the population to which you're distributing has the software installed in the same manner
- No guarantee that the population has the correct hardware to run software
- No guarantee that the staff loading the software has the correct knowledge to do so

However, Sneakernet does have pros:

- Increased contact between IT staff and end users
- Free one-on-one training opportunities after post-load

Batch File Installation

One method that does the job of deploying MSI packages is the good ol' batch file. Although this technique is a bit raw in practice, many systems administrators have honed it to a fine art. The idea is simple: modify your network logon scripts to determine whether a user is in a particular group, and have the batch file kick off the installation for the user.

To successfully pull off this trick, you can choose from many tools that will help you determine the group membership of the currently logged on user. One such tool is easy to find as it's available in the Win2K resource kit—IFMEMBER.EXE. You can call this tool in a basic batch file or use the more powerful KiXtart, which is a full scripting language and can be substituted for normal logon scripts.

KiXtart 95 is located in the Win2K resource kit. For Win2K, it has been updated to allow for conditional branching based on Win2K AD sites. For instance, if the computer that runs the batch file is in the New York site, then perform some actions, and if the computer that runs the batch file is in the Philadelphia site, then perform some other actions. An even newer version of KiXtart, KiXtart 2001, is available at http://www.kixtart.org/.

For an example, we'll use IFMEMBER.EXE, which will raise the environment variable error-level code to 1 if the membership is met. For instance, suppose you had an NT or Win2K domain-based group named ITGROUP with a member named John, and John logged on, then the error-level code would be set to 1. With this little bit of knowledge, you can get much accomplished. Listing 6.1 shows an example logon script addition that begins to achieve our goal.

```
ifmember DOMAIN\ITGROUP
if not errorlevel 1 goto NOT_ITGROUP
   rem load IT Group's software
EXIT
:NOT_ITGROUP
   rem load other groups' software
EXIT
```

Listing 6.1: Example IFMEMBER logon script addition.

In each part of the branch, you would run an MSIEXEC command with the software you wanted to install. However, if you ran this script upon next logon, once again, MSIEXEC would kick off and reload the software. To prevent this from happening, one option is to enhance the batch file to place "flag files" in strategic locations of the file system for each loaded piece of software. For instance, if you were loading DogFoodMaker5.msi, you might use the logon script to place a 0 byte DogFoodMaker5.txt file on the C:\ drive. In addition, we need one more line of code to jump over the MSIEXEC installation if we've placed the file there before, and presumably we've performed the installation. Therefore, we enhance our logon script a little bit by adding the lines that Listing 6.2 shows.

```
If exist c:\DogFoodMaker5.txt goto BRANCH2
Echo software_loaded > c:\DogFoodMaker5.txt
MSIEXEC /I \\server\share\dogfoodmaker5.msi
:Branch2
rem check to see if next flag file exists
```

Listing 6.2: Additional example text to add to a logon script.

This code will use the command interface's `if exist` command construct to verify that the flag file exists. If it does exist, then jump to another branch of the batch file or exit. If not, create the flag file by using the `echo` command, and create a flag file on the fly—in this case DogFoodMaker5.txt with the works "software_loaded" inside it. Finally, execute MSIEXEC with the /i switch to load the software.

> ☞ You might also choose to make the flag file a hidden file (via the attrib command) to ensure that it's not readily seen by prying eyes.

However, if the MSI installation fails, this addition to the logon script could be a disaster because the flag file is already written but the MSI installation didn't fully complete. You might use other additional checks, such as the known path to the .EXE file, or use additional tools to search the registry for the MSI's GUID.

As we learned in the Sneakernet section, you might run into situations in which you need to launch the MSIEXEC command in the context of the administrator. To do so within a batch file, you would need to expose the administrator password in the batch file with a command such as SU (found in the resource kit), causing a security risk by having the password totally exposed. To combat that, you might want to inspect myriad batch file compilers, which will wrap up batch files into .EXE files, which cannot be read as the logon script goes by. Although this method isn't the most secure for accomplishing this task, it at least presents a deterrent.

Both NT and Win2K domains can optionally execute .EXE files instead of just .BAT files for the logon process if desired.

One such batch file compiler is called Winbatch + Compiler. You can find this tool at http://www.winbatch.com/wb-compiler.html.

As you can see, the process of creating a batch file is easy but not simple. Every time a package changes, you need to keep on top of the script or scripts that are called to do the work. You have a new package? You change the logon script. However, most troubling, is that if something goes wrong during the MSI installation, users might not be able to interpret the feedback to help you assist with troubleshooting. The free nature of batch files might make them a tempting option, but they are usually difficult to work with when you have many MSI files and many machines to deploy to.

Batch file deployment has the following cons:

- No guarantee that the population has the correct hardware to run the software
- No centralized reporting to administrator if something goes wrong
- Batch file maintenance could be burdensome

However, batch file deployment has the following pros:

- Batch file creation is easy
- With enough elbow grease, you could have one logon script controlling all software deployments
- Theoretically, guarantees that a specific population of users (NT/Win2K group) has the same software

Microsoft MSI Deployment with Group Policy

In Chapter 1, we discussed the Windows Installer service built-in to Win2K and newer clients. As a refresher, the Windows Installer service installed in Win2K can be seen in Figure 6.3.

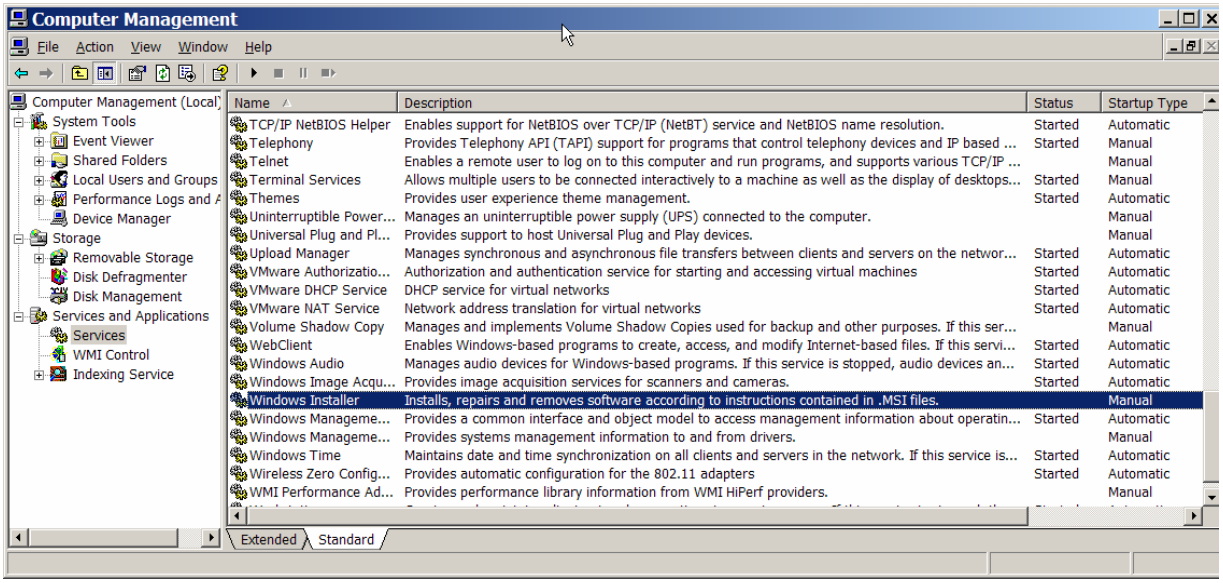

Figure 6.3: The Windows Installer service is built-in to Win2K and Windows XP.

It's not the service itself that's exciting, rather what can be done with the service. This service really comes to life when the Win2K client is used within a Win2K AD. The reason is that the service enables you to centrally deploy MSI files without having to worry about the administrative context of the user.

To connect to and leverage the Win2K service, you'll need to work a little bit. For small to midsized environments, Microsoft suggests that you check out a new technology built-in to Win2K AD. That technology is software deployment via the Group Policy mechanism. This mechanism is part of the Win2K feature set called IntelliMirror, whose goal it is to maintain system state from machine to machine as well as maintain overall system state health.

The software deployment features are very cut and dry—that is, they work under a very specific set of circumstances. First, Group Policy software deployment is typically meant to work with MSI files, though .EXE files are supported under very limited circumstances. At this point, you've wrapped up your applications as MSI files, as we covered in Chapter 2. To use Group Policy to deploy these MSI files, you must have a Win2K AD domain (this environment is likely an uphill battle to achieve). In addition, Group Policy applies only to Win2K and Windows XP clients—so the clients to which you want to deploy software must be running Win2K or Windows XP. Again, this requirement might prove difficult in larger environments. However, if all these conditions are met, you can start to experiment with Win2K Group Policy software deployment.

> 🕮 For an in-depth view of how to deploy each IntelliMirror feature, learn how Group Policy works (and how to troubleshoot it if it doesn't), and additional helpful tips and tricks for software deployment with Group Policy, check out *Windows 2000: Group Policy, Profiles, and IntelliMirror* (Sybex).

In this section, we won't explore every Group Policy option with regard to software deployment. For your beginning practice, you'll want to start simply with a Win2K AD and a single Win2K Pro workstation.

After you create a new Group Policy object (GPO), you can edit it. When you do, you'll see that there are two software installation settings sections (as Figure 6.4 shows)—one for computer configuration and one for user configuration.

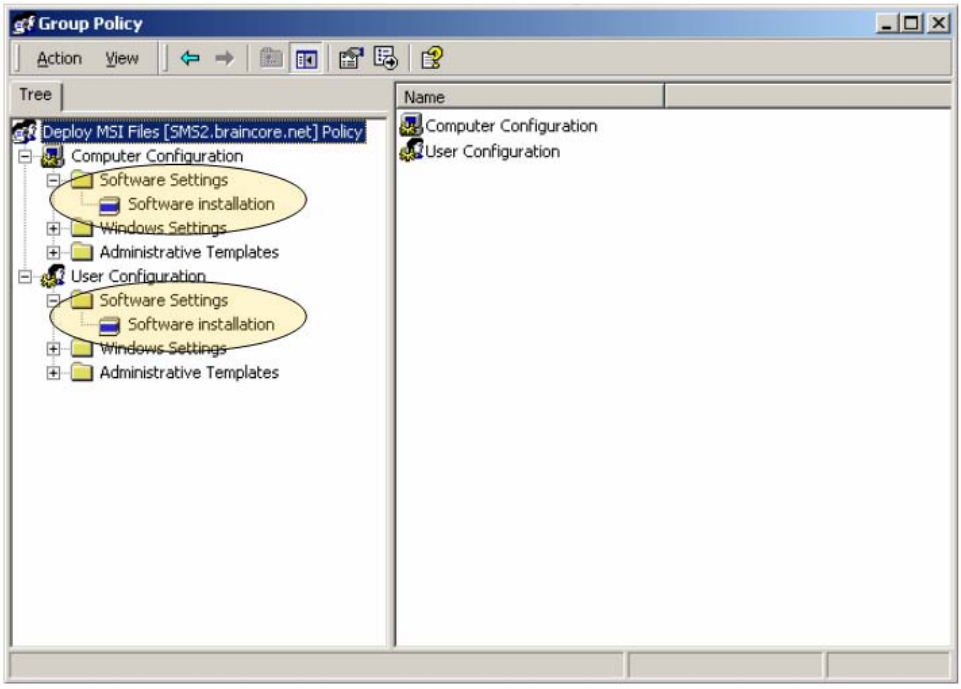

Figure 6.4: You can deploy packages to either computers or users.

At this point, you need to make a determination about how you want your software to be deployed. You can select to distribute a new Package, as Figure 6.5 illustrates.

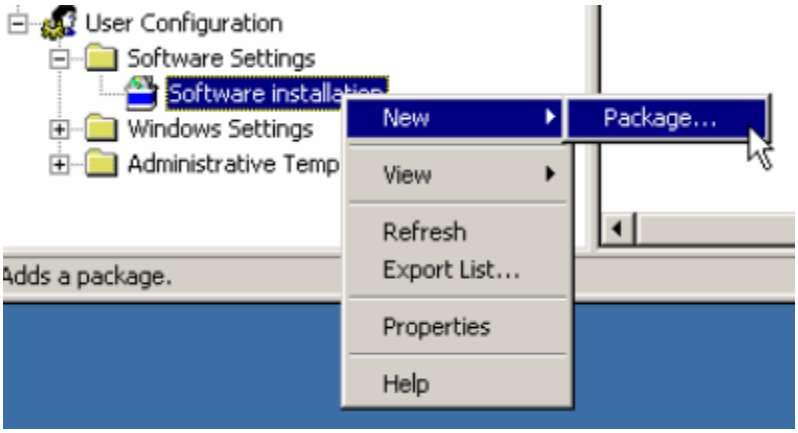

Figure 6.5: Choose either computer or users, and right-click to select New, Package.

You'll then be prompted for the star of the show—the MSI file that you created in the previous chapters. Figure 6.6 shows this window.

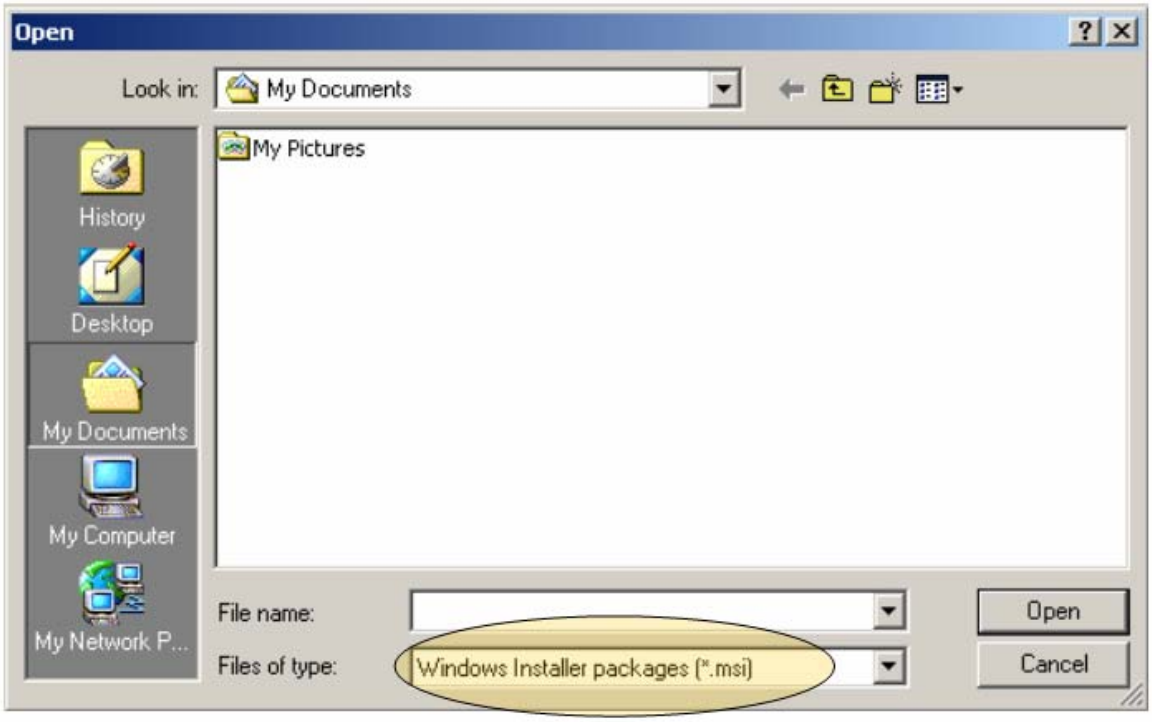

Figure 6.6: Select your MSI file for deployment.

For example, suppose you complete this set of tasks to deploy an MSI package to a user. In such a case, only the icon representing the software will make its way to the user's desktop. The icon is representative of a component that triggers the installation of its parent feature. A single feature can contain any number of advertised shortcuts. Once the user selects that icon, the required components to support that feature are installed in a just-in-time fashion. This installation usually incurs some delay from when the client clicks the desired icon and the application starts. However, this delay is only present for the first installation of the software; subsequent clicks of the icon result in quick loading the installed version of the software from the hard drive.

If you deploy your MSI package through the computer option (rather than user) all features in the MSI are loaded, and they appear loaded for each user who uses the computer. This method gives whoever is using the machine no options about which components are to be installed.

Group Policy software deployment has the following cons:

- Requires Win2K AD
- Requires Win2K or Windows XP clients
- Can be difficult to manage with a large user population as a result of a complex OU structure
- No choice in which components are potentially available

Group Policy software deployment does have its share of pros:

- Software can be deployed to either users or computers via Group Policy in AD
- Deployment is easy
- Theoretically guarantees that a specific population of users (an AD OU, for example) has the same software

MSI Deployment with Third-Party Tools

Another option for deployment of your MSI packages is to use third-party tools. The following sections explore how you can use the assistance of your MSI repackaging tool to hand off your package to a third-party deployment mechanism. We'll also discuss how to deploy packages even if you don't have a third-party package-creation tool to assist you.

Deploying with the Assistance of Third-Party Repackaging Tools

In Chapter 2, you learned about the various tools you could use to create your MSI packages. In that chapter, we discussed several free or cheap options to get the job done; however, to get the job done with finesse, I suggested that you look into purchasing a third-party tool. Indeed, the initial cost of a third-party tool can be mitigated if you use enough of the features it has to offer.

When we discussed the tools, we talked mostly about how tools wrap the packages into proper MSI files for future deployment. Some third-party MSI wrap-up tools have something special: the ability to deliver that MSI in a format that's "ready" for a third-party deployment mechanism to pick up. In other words, the third-party MSI creation tool doesn't directly have the ability to *deliver* the MSI it wraps up; rather it can pass the wrapped file directly to your company's third-party deployment choice. The goal is to have a seamless transition between the tool you use to create your MSI package and the tool you use to deploy your MSI.

To accomplish this task, you can use any number of tools, including Wise Package Studio and InstallShield AdminStudio. InstallShield has partnered with Marimba to offer repackaging and deployment functionality. In this example, I'm using the Wise Package Studio to transition my package to any number of supported third-party tools. To do so, I'll use the Package Distribution wizard, which Figure 6.7 shows.

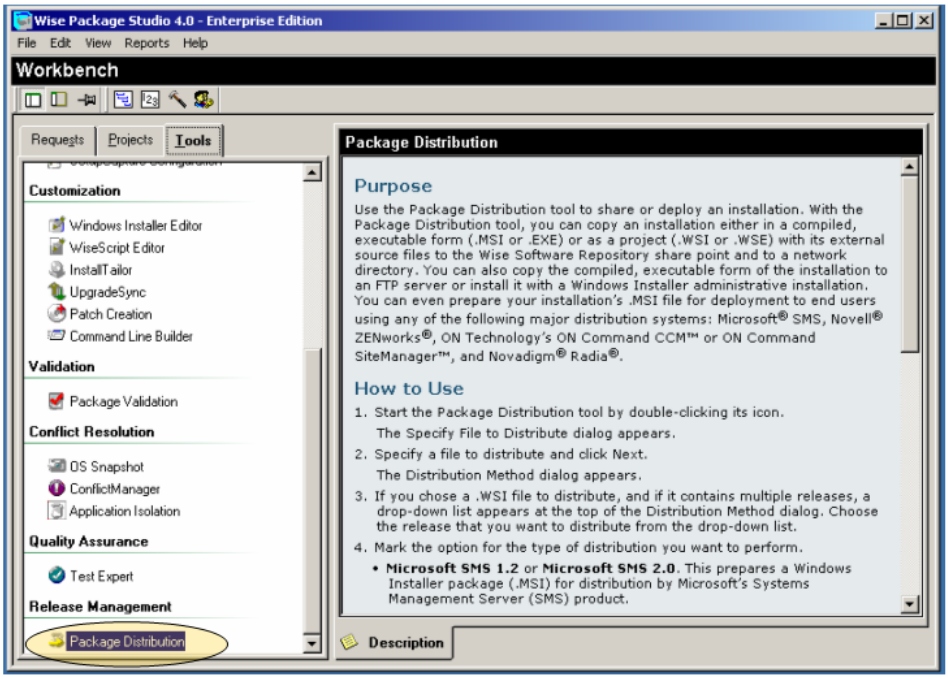

Figure 6.7: The Package Distribution wizard hands off the MSI to any number of distribution applications.

Once launched, the Wise Package Studio automatically detects which third-party distribution programs you have installed. It then presents you with the ability to push to the various third-party programs or select alternative methods to get the package out the door (see Figure 6.8).

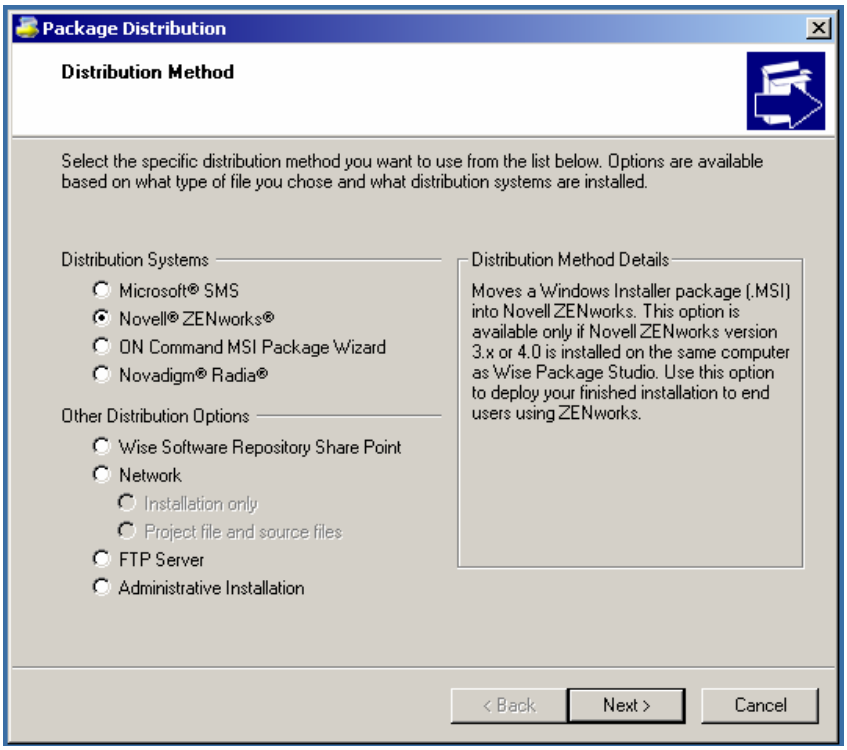

Figure 6.8: Popular distribution systems and other deployment options are presented with this MSI repackaging tool.

For example, selecting Novell ZENworks brings up the ability to easily migrate the package to that distribution method. This feature supports all the normal Novell features, such as Tree Name and Context, as Figure 6.9 shows.

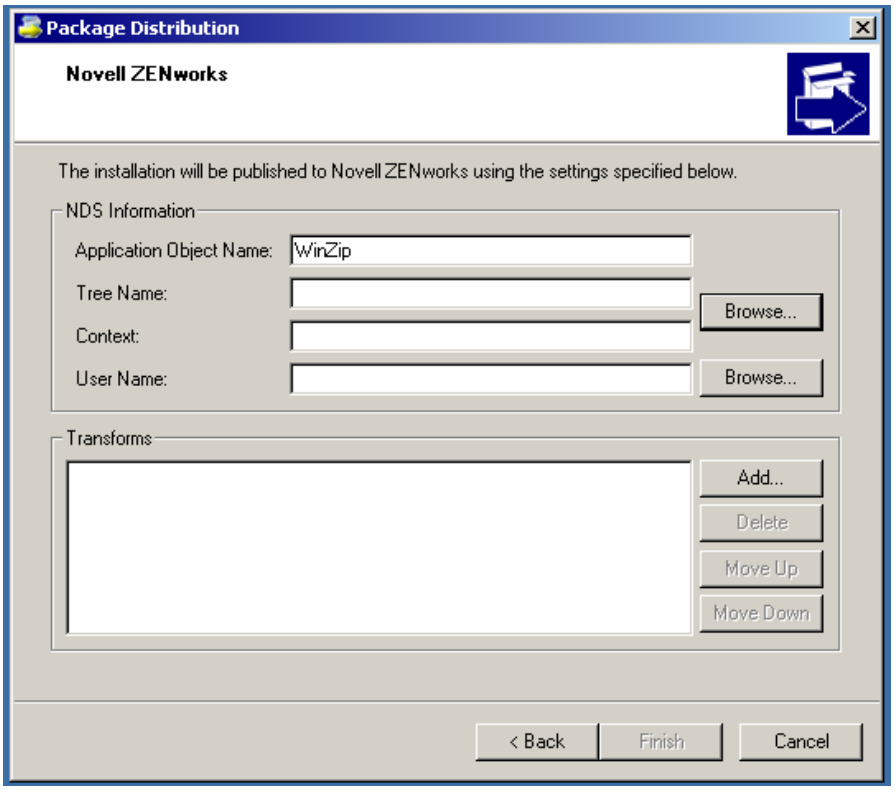

Figure 6.9: One possible way to distribute your package is through a third-party tool such as ZENworks.

Other hand-off functions are similar in that they target the specific third-party tool with exactly what's required. Each hand off is a bit different, so be sure to test each option if you have multiple third-party distribution methods in conjunction with an MSI repackaging tool such as this.

Third-Party Distribution Methods

You might not have a third-party MSI creation tool to assist you in handing off to your third-party distribution method. For instance, you might have used one of the free MSI creation tools—such as WinINSTALL LE or the SMS Installer—that don't offer the ability to hand off the resulting MSI file to any distribution mechanism. In either case, you need to learn about how to deploy your MSI file. This section examines several third-party distribution methods for deploying MSI files.

As I talk about third-party tools, my goal is educate you about some of the possibilities available to your for deploying MSI files. The programs we'll explore offer myriad functionality—software inventory, software metering, remote control, and even OS deployment features! However, I'll stick to discussing the MSI-centric features of these solutions.

Each tool takes a slightly different angle toward how they handle MSI file deployment and management. Hopefully, one of the tools I discuss will give you ideas about how to best deploy MSI packages in your organization.

> 🕮 I simply don't have enough space to cover each tool that can deliver MSI files. However, you can find a list of deployment (and repackaging) tools at http://appdeploy.com/tools.

Microsoft SMS and MSI Deployment

SMS is Microsoft's offering to deploy any and all files to your Windows desktop. SMS is robust in many ways, including the ability to deploy files to any 32-bit Windows desktop. I've categorized it as a third-party MSI distribution solution because it is an add-on product (you cannot do most of what SMS does "right out the box" with just Win2K).

A little background about SMS: Out of the box, SMS 2.0 wasn't originally intended to deploy MSI files. Rather, it was mostly meant to deploy .EXE files. Indeed, for years, the SMS repackaging tool (the SMS Installer tool) didn't create MSI files, it only created .EXE files. Only recently, as we explored in Chapter 2, did the SMS Installer repackaging tool offer MSI file creation functionality.

> As mentioned in Chapter 2, not every MSI function works correctly when creating MSI files using the SMS Installer tool. Therefore, you might want to consider using a full-featured third-party repackaging tool to create your MSI files. Then you can use the information here to continue to deploy that repackaged MSI file using SMS.

Deploying MSI files with SMS is somewhat different than deploying .EXE files with the tool. You need to prepare a little before doing wholesale deployment of your MSI files with SMS. First, you need to deploy the latest Windows Installer bits to your PCs, which can be a monumental task. Like an MSI, you must deploy Windows Installer as an .EXE deployment.

> 🕮 Get the bits you need for Windows 9x and NT 4.0 at http://www.microsoft.com/msdownload/platformsdk/instmsi.htm. The bits for Win2K and Windows XP are already installed, but you can update them if you want. Check out Chapter 1 for a refresher of which versions of Windows Installer are available.

When it comes to actually deploying your MSI files, you've got a little more work to do. First, you need to set up a share point with read access to everyone. Next, you need to perform an Administrative Installation of your MSI file into that share. Recall from Chapter 2, that an Administrative Installation unpacks the guts of the MSI file and dumps the necessary contents to a directory.

> ☞ Recall that you can run an Administrative Installation using MSIEXEC /a such as
>
> ```
> MSIEXEC /a myfile.msi
> ```

In the example that Figure 6.10 shows, I've performed an administrative install in the directory called e:\csav_distry, and created a new package pointing at the distribution source.

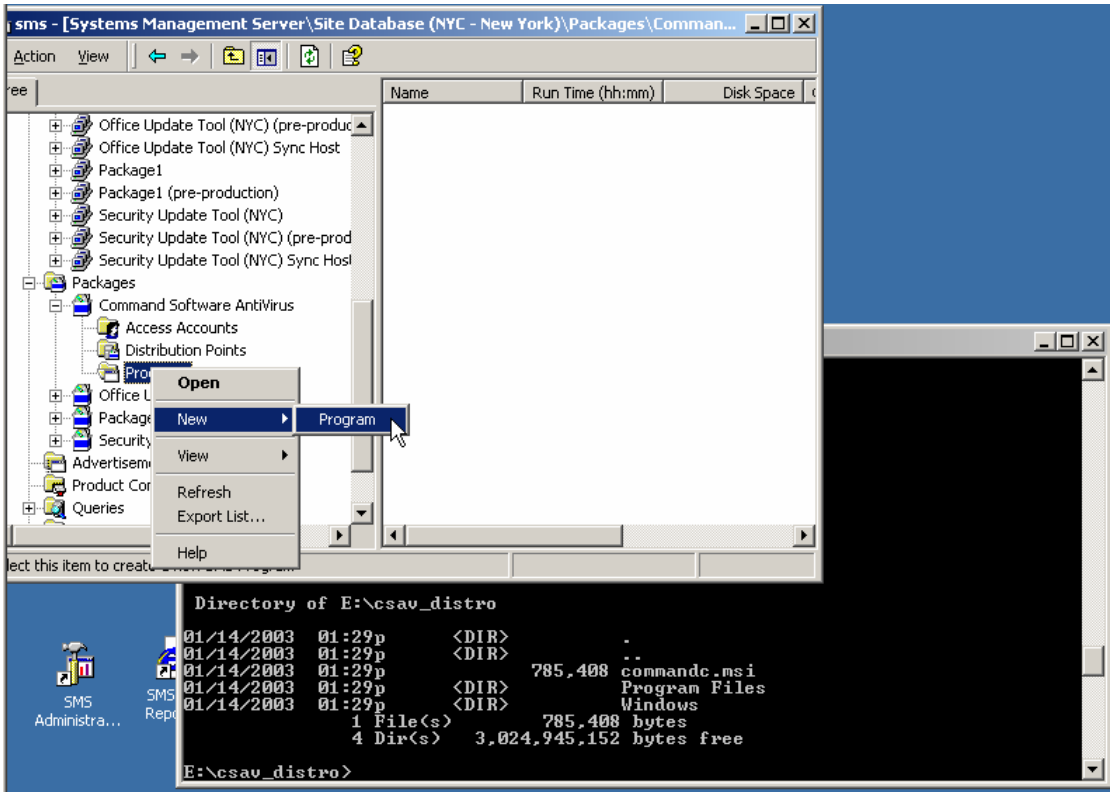

Figure 6.10: After creating your Administrative Install, you're ready to deploy MSI packages with SMS.

Once the files are in the directory and the SMS package is defined, you can continue to leverage your knowledge of the MSIEXEC syntax to deploy the packages to your SMS clients. You'll create a new program, as Figure 6.10 shows. Recall that to perform an installation of a particular MSI file, you use the msiexec /i <filename.msi> syntax (you must also specify a transform and/or command-line switch to automate the installation). This syntax is the same syntax that you pump into an SMS program to perform the installation, as Figure 6.11 illustrates.

> In SMS, a "program" is defined as the command line used to kick off a package.

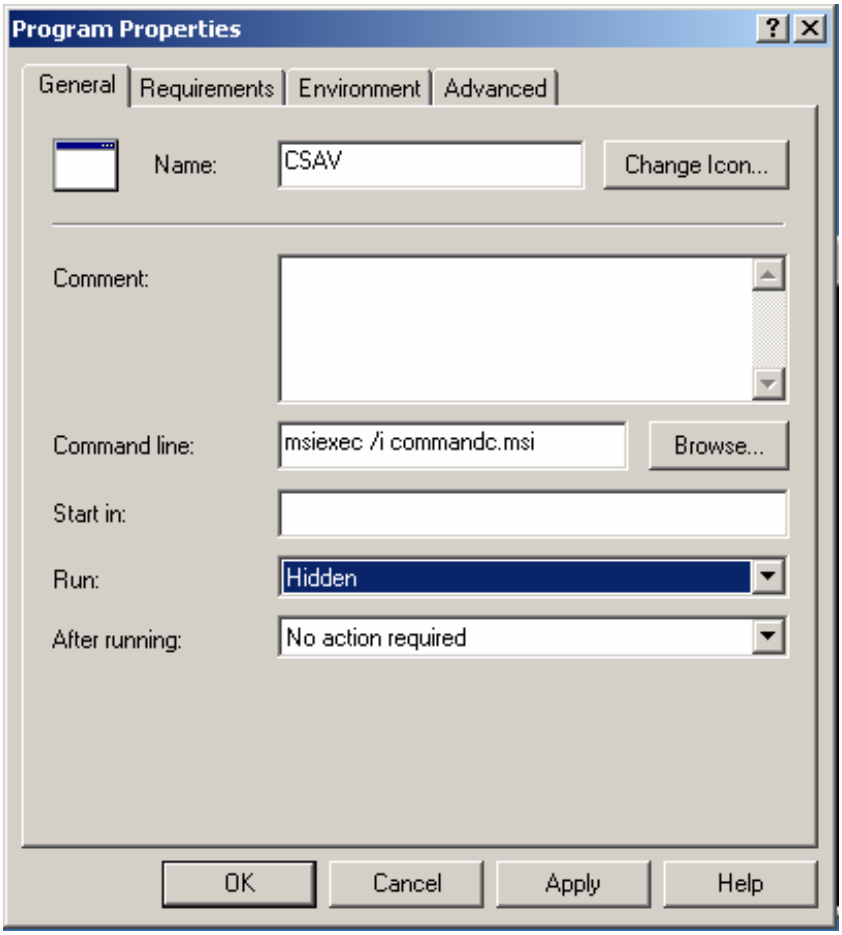

Figure 6.11: Use the MSIEXEC command line to assist with your deployment using SMS.

There are many additional options at your disposal for configuring how to run an MSI file at an SMS client. For example, if you supply an /M switch, you can generate a status MIF file, which SMS will pick up and inventory to let you know what happened during the install. Using the information you already know about MSIEXEC and Windows Installer, it's a lot easier to perform the task.

> ☞ You might want to check out Microsoft's additional notes about deploying MSI files when using SMS and Windows Installer at http://www.microsoft.com/smserver/techinfo/deployment/20/deployosapps/deploymsi.asp.

MSI Deployment and Management with Altiris Client Management Suite

Altiris' Client Management Suite is another third-party method to distribute and manage your MSI files. The Client Management Suite has many layers; we'll focus our attention on its MSI hot spots. Three parts of the Client Management Suite deal with MSI files:

- The deployment solution can send out an entire OS image. Once the image is deployed and finished, the deployment solution can pull an MSI package and install it before the user logs on for the first time.
- The software delivery solution is the suite's main software delivery mechanism. It's similar to SMS in functionality and terminology. For instance, to deploy MSI files, you use the MSIEXEC command when you set up your packages and programs.
- The application management solution lets administrators scan machines to determine the health of their distributed packages (either distributed via Altiris or some other distribution mechanism). This process can perform a scan of the machine, then report about the general MSI health of the system, as Figure 6.12 shows.

Altiris also offers an MSI/EXE repackaging tool similar to the SMS Installer that creates both .EXEs and MSI files. This tool is called Rapid Install, and similar to earlier versions of the SMS Installer, it requires an external .EXE to .MSI conversion utility.

Figure 6.12: Altiris' suite can help manage and fix MSI files once you deploy them.

If the deployment of one package has broken another package, the Windows Installer might not know about this mishap and won't fix it until the original program is launched. The suite's application management solution can scan the system and learn which MSI packages are currently installed on the machine. It can then run through all the MSI interdependencies and links to determine what is broken. Finally, it can trigger the Windows Installer service to perform an automatic repair. Either quick or full scans can be chosen, as Figure 6.13 shows. A quick scan examines the MSI's key path and attempts a fix. A full scan tracks down all associated components and DLLs of an MSI and attempts to ensure that every bit is back in order.

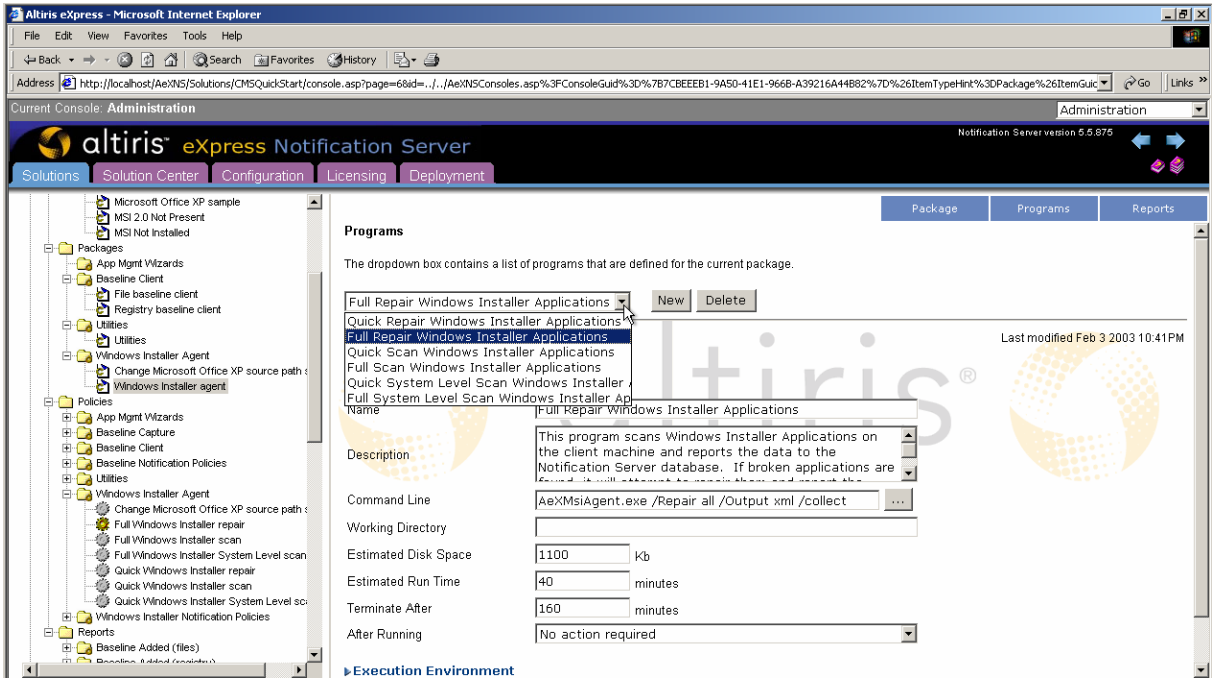

Figure 6.13: The suite's repair option allows for "Quick" or "Full" MSI scans.

Also available is the ability to create a baseline of a known good local machine, then do comparisons against a broken machine. You can compare which MSI components are registered in the registry. Simply find the broken component, and you're off to the troubleshooting races.

Altiris also provides several reports that can help explain how packages are interacting with the Windows Installer service on each client system, as Figure 6.14 illustrates. For instance, you can get a quick view of *Most frequently broken products* or *All repair attempts in the last N days*.

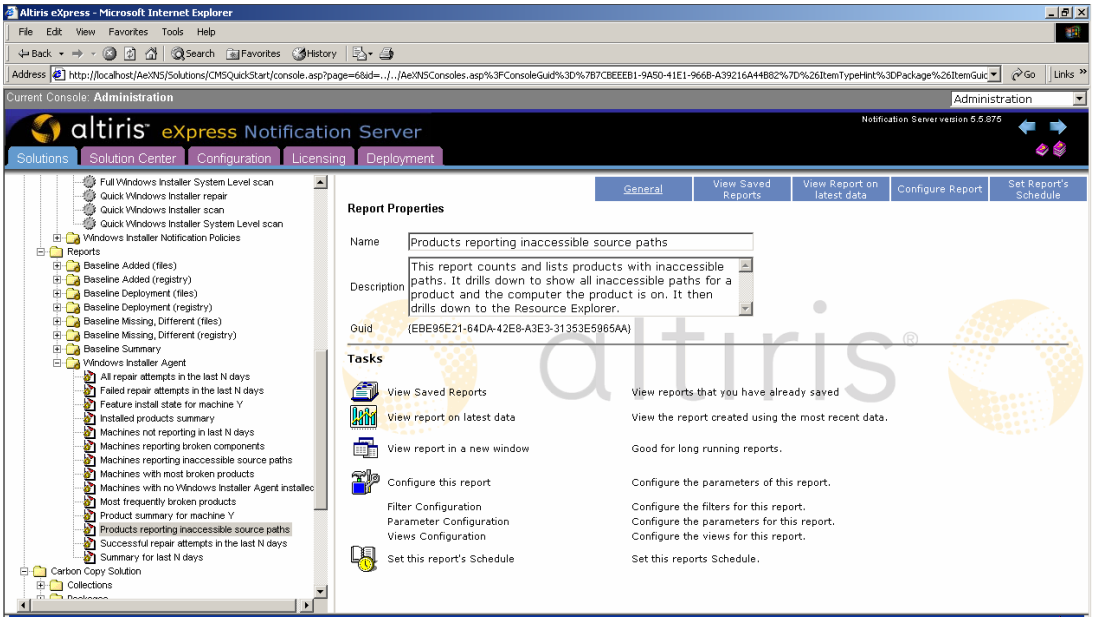

Figure 6.14: Altiris has facilities to run reports about the status of the Windows Installer service.

Altiris' suite continues with a rigorous MSI inventory. For instance, it can obtain a list of all MSI features not installed. For example, perhaps when a user loaded an MSI file, the user chose not to install a particular feature—say, the Help file. Altiris can examine which machines are missing the MSI feature, then simply send a new job to the client computers to pull the feature from the source to ensure that all clients are the same.

MSI Deployment and Management with ON Technology's ON Command CCM

Another third-party technology that has the ability to push MSI files is ON Technology's ON Command CCM (change and configuration management). This product provides unique MSI deployment functionality. For instance, CCM has the unique ability to bypass the entire requirement for an Administrative Installation and lets you distribute complex MSI packages directly from the source.

For example, in the windows that Figure 6.15 shows, I'm telling CCM to deploy Microsoft Office XP with FrontPage. The MSI file is called PROPLUS.MSI (in this example, I'm grabbing it directly off the CD-ROM).

Figure 6.15: CCM allows for you to bypass the administrative installation step.

CCM goes one step further: It has the ability to add a transform file you create right into the definition as well as essentially create customized transforms on the fly! If you do not choose an MST file, CCM will crack open the MSI file and locate the changeable variables. Instead of going through the lengthy customized MST creation process for your application (or by using a third-party MST creation tool), you can bypass it all and use CCM (see Figure 6.16).

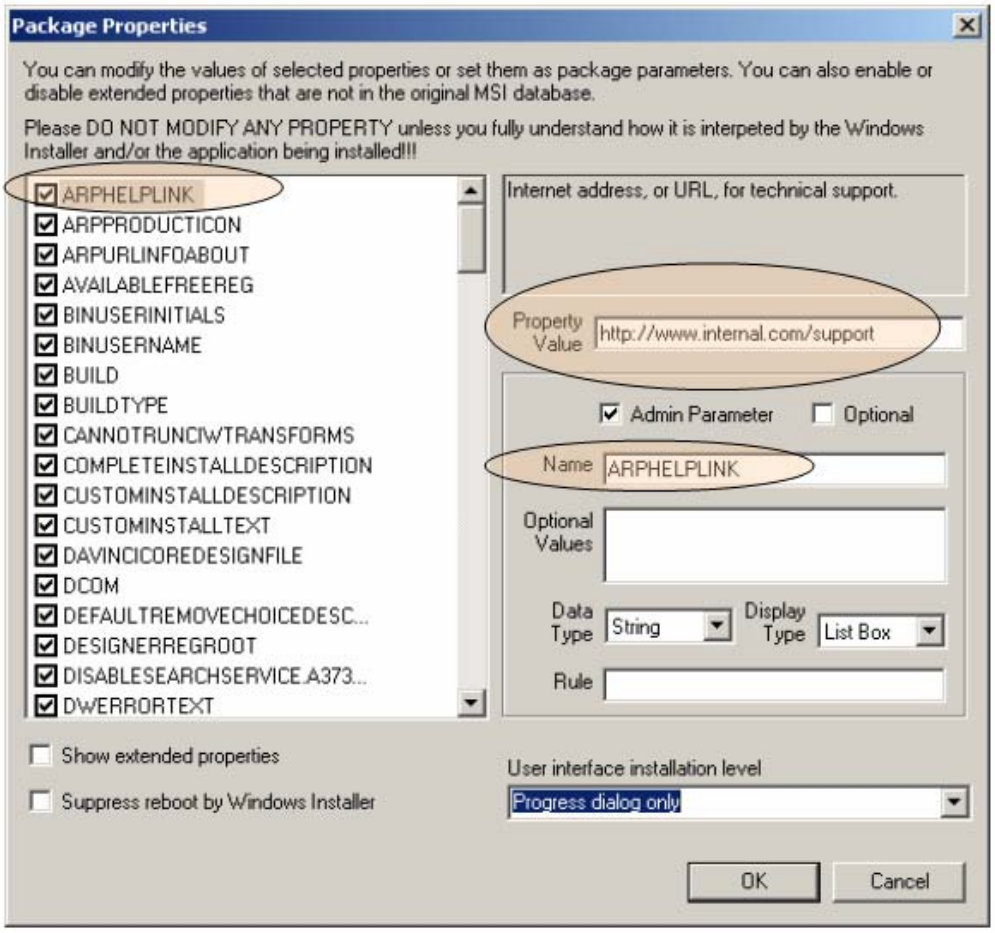

Figure 6.16: You can tap directly into an MSI's properties and change them on the fly.

In my example, I've changed the value of Office's ARPHELPLINK, which defines the location of online Help. Usually, it has a link to Microsoft. However, using CCM, I'm changing the location to my own company's internal support Web site.

When it comes time to target the package to your client systems, CCM has an additional notable feature. That is, it allows you to target the machines you want, and, in spreadsheet fashion, simply select which systems will get which MSI features once they hit the target machines. As Figure 6.17 illustrates, each of these three systems (gx150x, nx1, and MINI) will have the Office XP features in one of the MSI potential installation states.

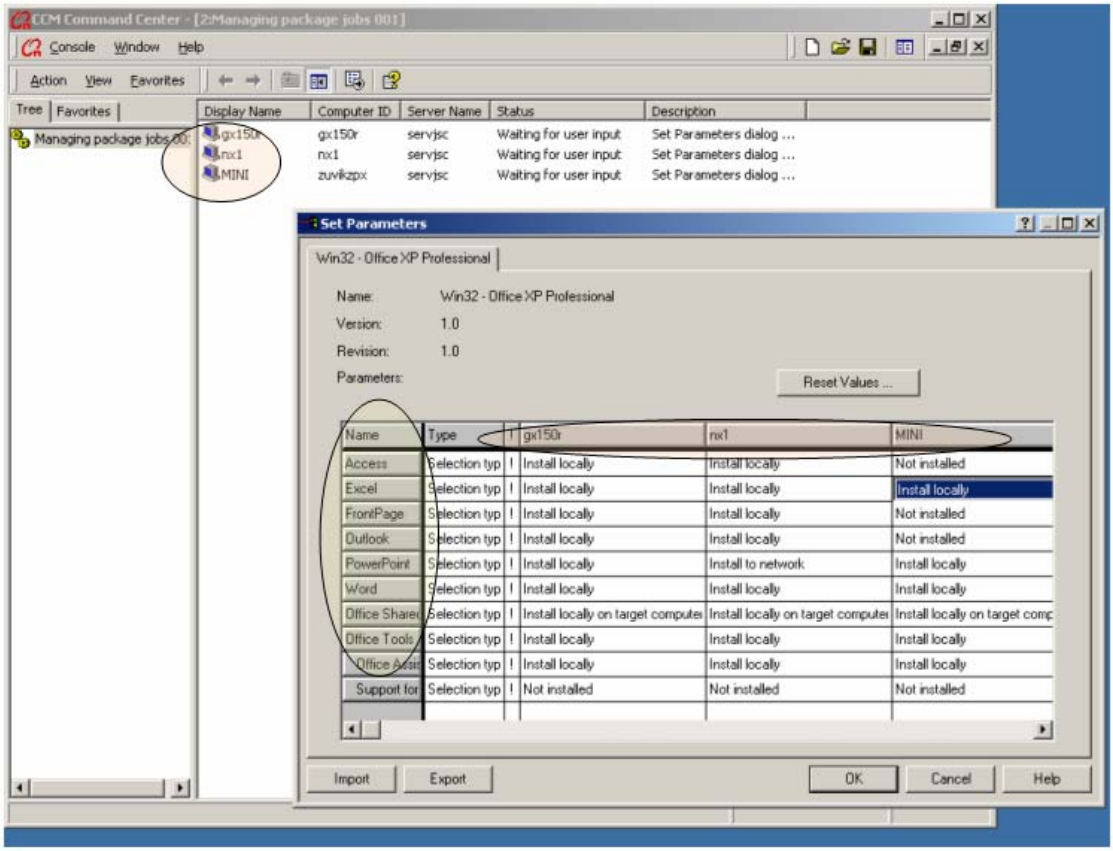

Figure 6.17: You can specify which computers get which options on the fly in spreadsheet form.

MSI Deployment with ONDemand Software's WinINSTALL

Another product that can help deploy your MSI packages is ONDemand Software's WinINSTALL (formerly owned by VERITAS). As we saw in Chapter 2, WinINSTALL has a companion product called WinINSTALL LE that can help create MSI packages, which you can then deploy via any distribution mechanism you desire—not just through WinINSTALL. Like its companion product, WinINSTALL has gone through a long history of ownership. It was developed by ONDemand Software, then Seagate bought WinINSTALL and WinINSTALL LE, then VERITAS bought Seagate. Because VERITAS wanted to focus more on backup and recovery, WinINSTALL and WinINSTALL LE were sold back to a newly reformed ONDemand Software—and the products are back in development. WinINSTALL's strength with MSI packages is that you are able to view and manipulate all the features and components of an MSI package that you create, as Figure 6.18 shows.

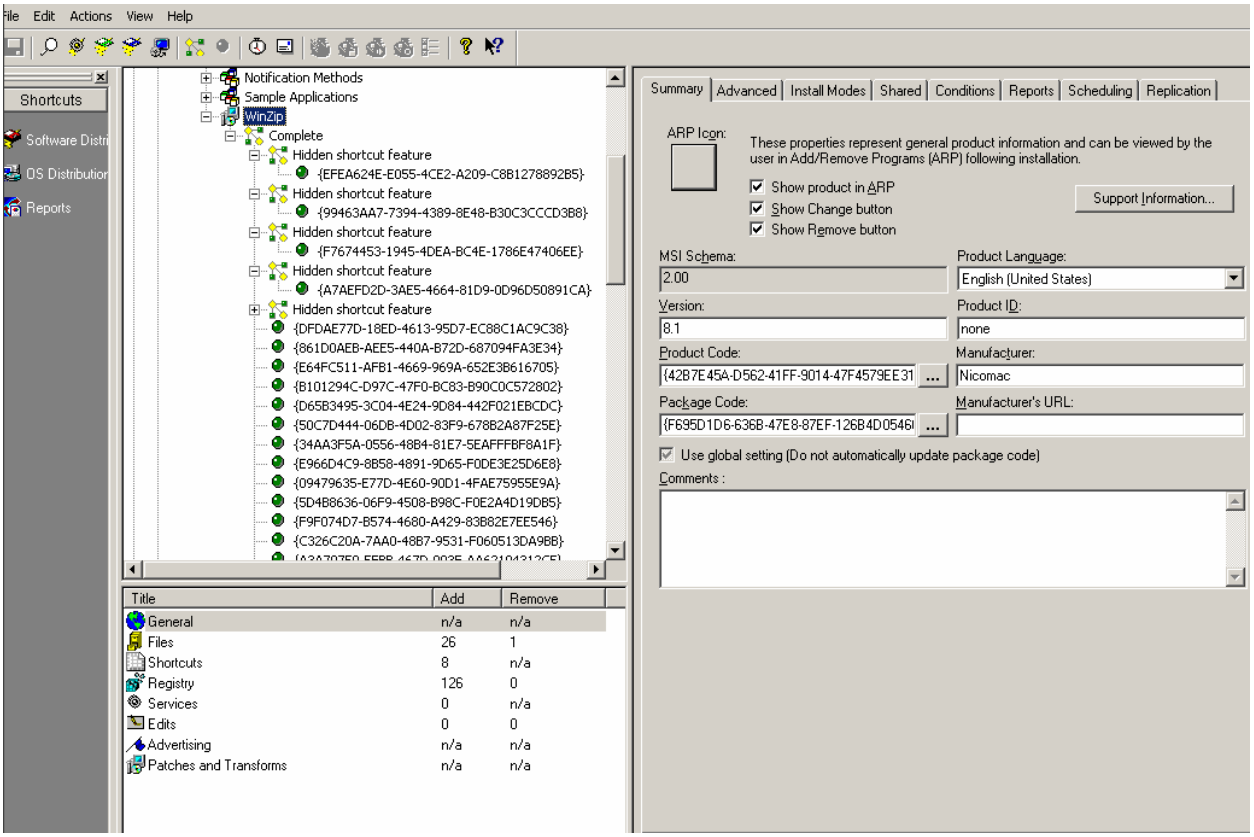

Figure 6.18: WinINSTALL displays all features and components of each deployable MSI file.

Once an MSI file is loaded into the WinINSTALL Console, you can see each of the features (represented in Figure 6.18 with the little DNA-type icon) and each component (represented with a green dot). In addition, at the feature level, you can easily make the initial determination whether a feature should be installed on first use, loaded and run from the hard drive, or an alternative option (see Figure 6.19).

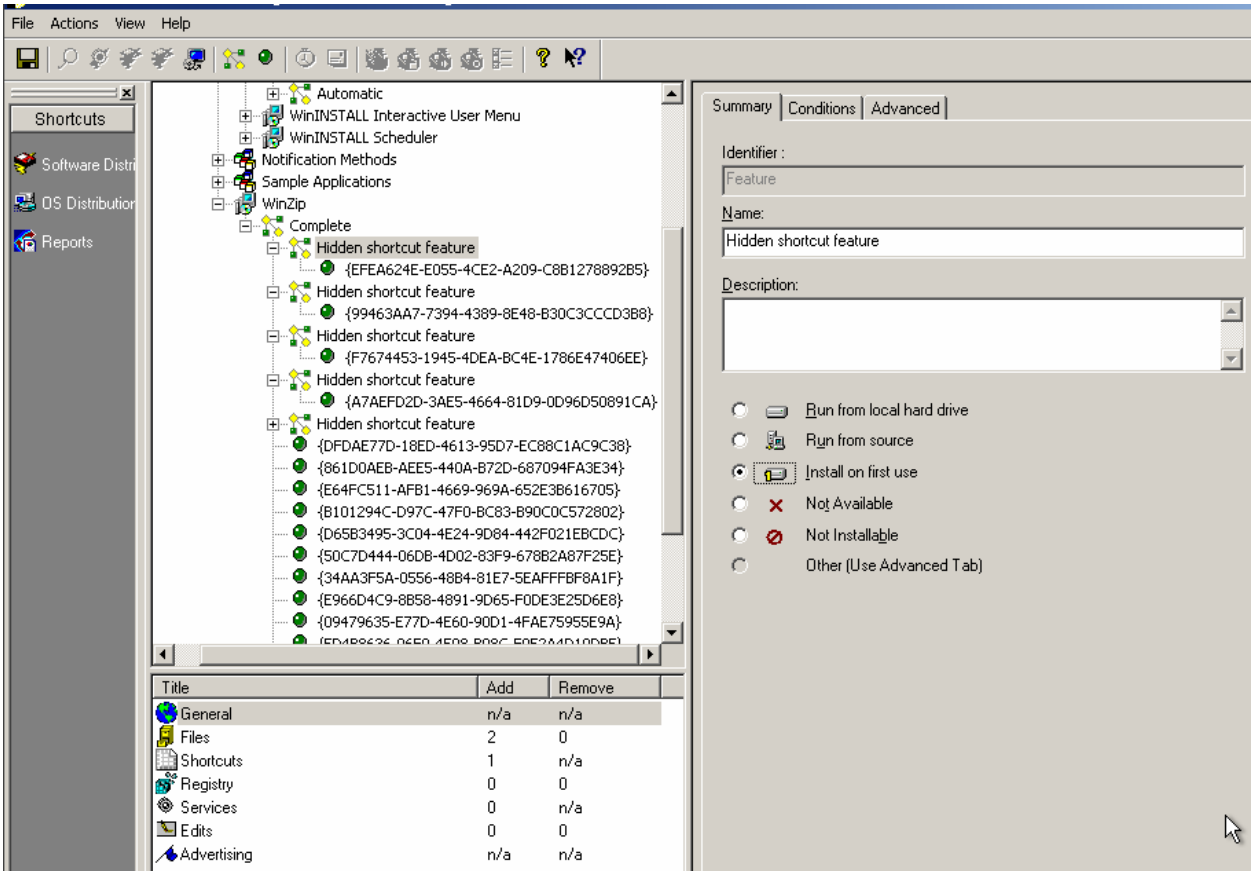

Figure 6.19: You can easily control the MSI installation state.

You can even dive in and take it a step further by determining which features and components are going to be installed in what manner. For instance, you might only decide to install the component related to the Help file (for example, to save space or eliminate confusion) of an application if the ProductLanguage was English and if the machine was at SP1 or higher. By drilling into the component and specifying the criteria you want to match, you can determine which components will be installed (see Figure 6.20).

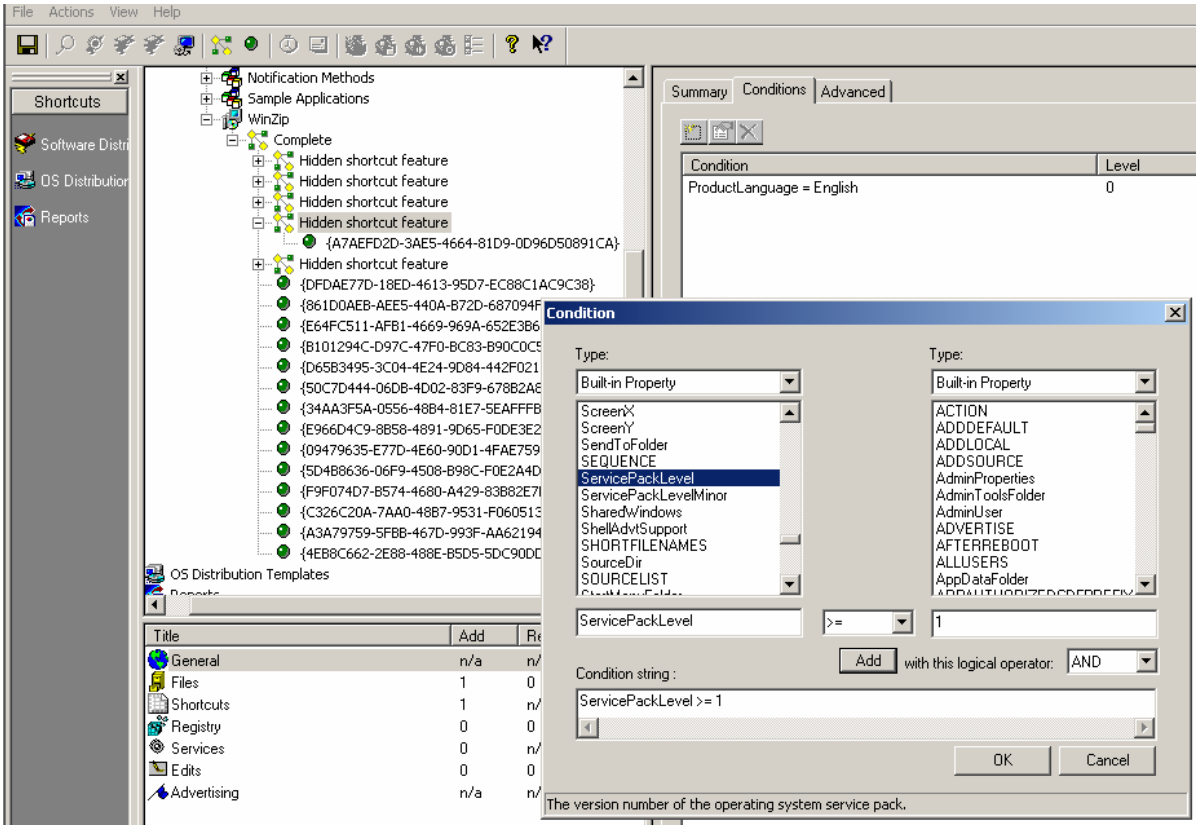

Figure 6.20: You can control how each feature and component is installed.

In addition, WinINSTALL can do a compression upon your MSI file based on the features you chose to install, as Figure 6.21 shows. That way, you're only sending the necessary data to the clients that need it; not the entire package.

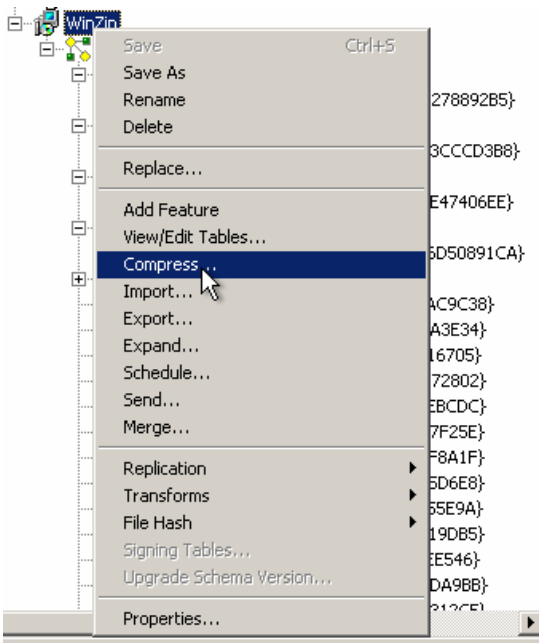

Figure 6.21: You can compress your MSI file before shipping it off to destination servers.

MSI Deployment with Mobile Automation 2000

Mobile Automation 2000's claim to fame is that it was the first kid on the block to perform a "trickle down" deployment, through which users on slow links can download a package to install in "drips and drabs," then once fully downloaded, the job kicks off and runs. Imagine trying to download, say, Office XP over a 56k modem in one fell swoop, and you can understand how this feature might come in handy. Mobile Automation 2000 also has the ability to send software to all sorts of Microsoft and non-Microsoft devices, such as Windows CE, Palm, and Blackberry, in addition to standard Windows-based PCs. Mobile Automation 2000 builds upon its history and is now a robust MSI package deployment mechanism.

Mobile Automation 2000 allows for a full script of any sort of install—including MSI files. In the example that Figure 6.22 shows, the Execute MSI Package command is pulled off the Command List as a possible first command to start an MSI deployment.

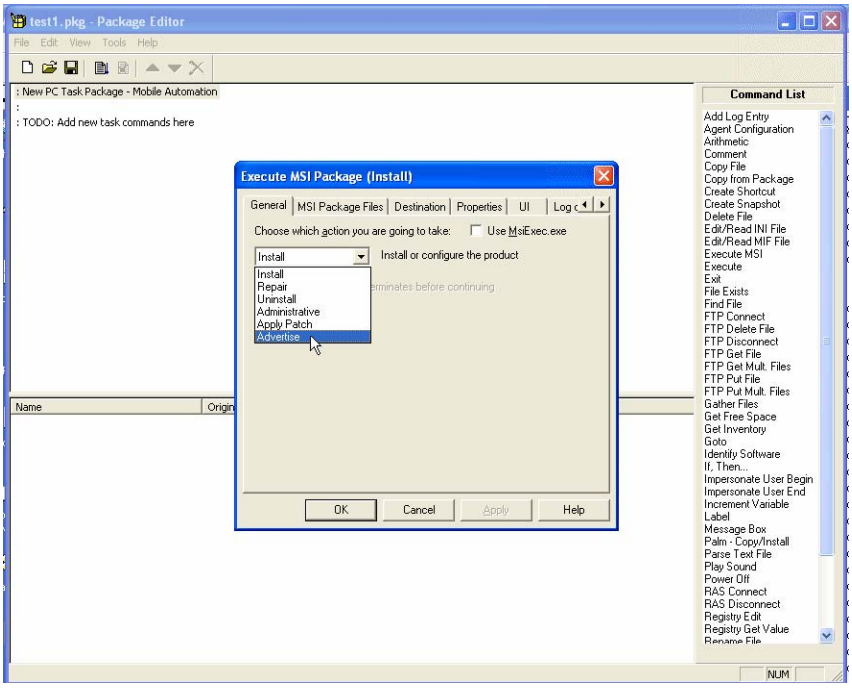

Figure 6.22: Mobile Automation 2000 allows for pre and post scripting during an MSI installation.

Note that the Use MsiExec.exe check box is clear by default. The reason is that Mobile Automation 2000 has the unique ability to install packages in two ways. When the check box is selected, Mobile Automation 2000 will use MSIEXEC to install the application. With the check box clear, Mobile Automation will use the direct Windows Installer APIs to perform the installation.

Mobile Automation 2000 also has the ability to grab the state of certain attributes of the package, such as if the package requires a reboot. In the example that Figure 6.23 shows, if the package requires a reboot, you could make some additional changes before actually going forth and performing that reboot. Doing so could save time during each and every deployment. Imagine how much time you could save if you could squelch every MSI reboot until absolutely necessary, perform the changes you wanted (which could also require a reboot), then reboot just once when you're ready.

Figure 6.23: You can inspect the state of several MSI variables, such as whether a reboot is needed.

Finally, because Mobile Automation 2000 has the ability to use the direct Windows Installer API calls (instead of just calling MsiExec.exe) to install the package, there is a huge jump in the level of detail provided about precisely what happens during the install time of a package. Every possible status code result is directly returned from the API function call and handled as a branch condition, as Figure 6.24 shows. (The API error code list is quite long and only a fraction can be seen in the drop-down box in the figure.)

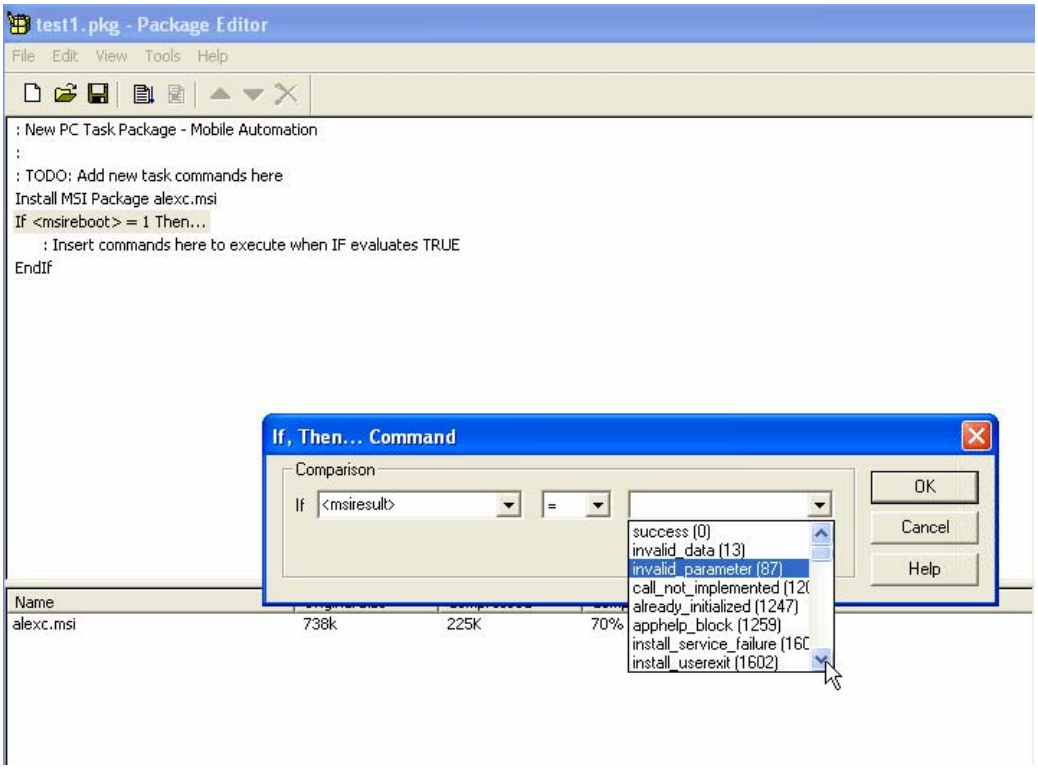

Figure 6.24: Mobile Automation 2000 can branch on the precise error returned from the Windows Installer API function for installation.

Once you know the error code, you can conditionally branch to handle specific conditions or simply make note of the error in a log file to help with MSI deployment troubleshooting. In the example that Figure 6.25 shows, I'm adding a warning flag to my Mobile Automation 2000 log file if the error condition known as <msiresult> is anything other than "success (0)."

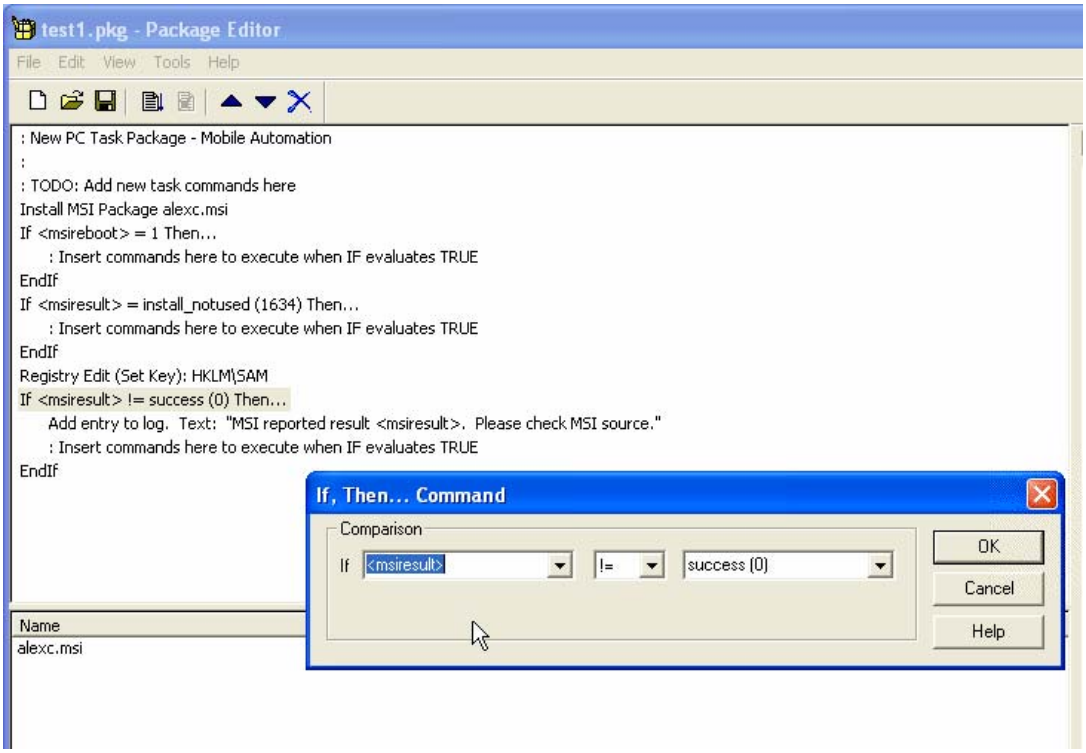

Figure 6.25: You can customize the error logs to demonstrate specifically what went wrong with an MSI installation.

Additionally, Mobile Automation 2000's inventory function can grab MSI data directly from the desktop—such as how often certain MSI packages are being used (see Figure 6.26). Finally, when the package is ready for uninstall, it can be uninstalled with a proper MSIEXEC uninstall string.

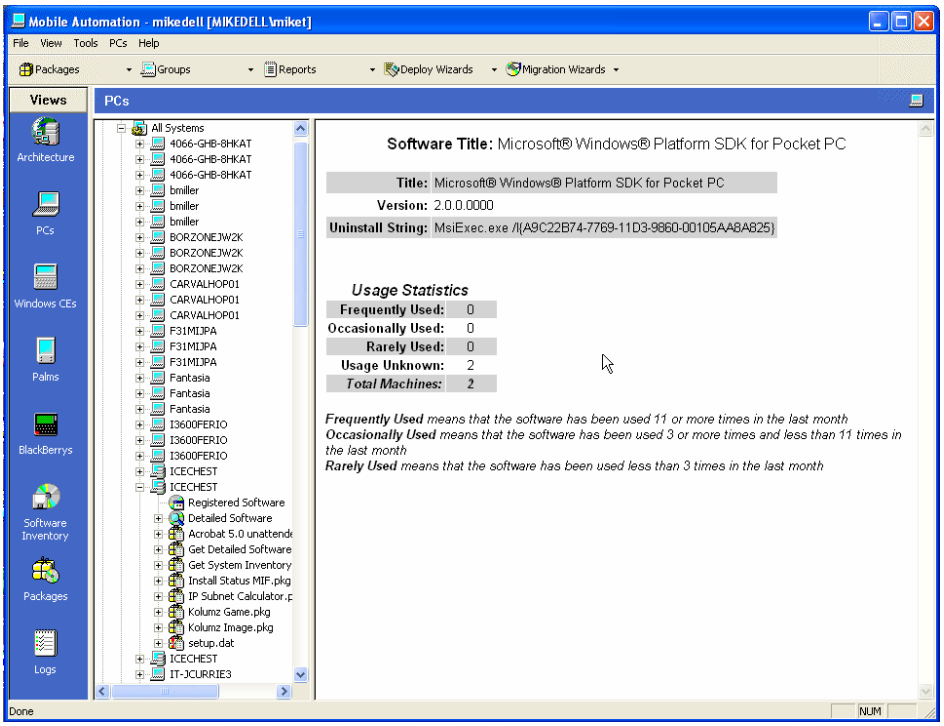

Figure 6.26: You can find out how popular the packages you deployed are by inspecting the usage statistics.

Summary

In this chapter, you learned about the free, the cheap, and the third-party ways to get your MSI files out the door and onto your users' plates. If you have a small environment, it's possible you can get away with Sneakernet or batch files. Midsized environments could make decent use of Win2K's Group Policy deployment features. However, the largest environments will likely need an industrial-strength solution to get those MSI files out the door and fully managed day to day.

Windows Installer technology has been out and about for several years now, and much has been written about it. Administrators leverage Windows Installer from a unique perspective, and they need resources that directly address their IT management concerns with sensitivity to their learning styles. To that end, we've tried to address Windows Installer technology management issues with practical tips and techniques you can apply immediately to your packaging efforts.

It is our hope that this book will be such a valuable resource that it will become very "virtually" worn and tattered as you reference it again and again.

Content Central

Content Central is your complete source for IT learning. Whether you need the most current information for managing your Windows enterprise, implementing security measures on your network, learning about new development tools for Windows and Linux, or deploying new enterprise software solutions, Content Central offers the latest instruction on the topics that are most important to the IT professional. Browse our extensive collection of eBooks and video guides and start building your own personal IT library today!

Download Additional eBooks!

If you found this eBook to be informative, then please visit Content Central and download other eBooks on this topic. If you are not already a registered user of Content Central, please take a moment to register in order to gain free access to other great IT eBooks and video guides. Please visit: http://www.realtimepublishers.com/contentcentral/.