



realtimepublishers.com[™]

The Definitive Guide[™] To

Windows Installer
Technology
for System Administrators



Darwin Sanoy and Jeremy Moskowitz

Chapter 4: Best Practices for Building Packages.....	82
Best Practices Formulation	82
Best Practice Is Not Optional.....	84
Darwin’s Law of Technology Sophistication	84
Repackaging Best Practice Recommendations.....	85
Do Not Repackage All Types of Setup Programs	86
Have a Documented Desktop Reference Configuration.....	87
Use Clean System Reloads for Testing and Packaging	87
Why Clean Machines?	88
Additional Management Data for Packaging.....	89
Windows Installer Best Practices.....	90
Invest in Training.....	91
Invest in Good Tools.....	91
Basic Packaging Functionality.....	92
Advanced Functionality	92
Peripheral Features.....	93
Administrator vs. Developer Tools.....	94
Manage Your Windows Installer Engine Version.....	95
Know How Windows Installer Interacts with Other Technologies	95
Configure Policies and Security.....	96
Ensure Source List Management	96
Repackage Existing Packages Rather than Convert Them	97
Use VBScript for Custom Actions and Other MSI Scripting.....	97
Run Package Validation.....	99
Perform a Dry Run with Verbose Logging.....	99
Utilize Windows Installer’s Logging Capabilities.....	100
Formulating Your Own Processes	100
Windows Installer SDK Assumptions	100
Package Classifications.....	103
Package Structure Rules for Administrators.....	105
Component Rules—The Protocols for Sharing	105
Scope of Distribution	106
Code Management Components.....	107

Duplicate Component Definitions	109
Conflicting Component Definitions.....	110
Compounded Problems	112
Upgrade Packages.....	112
Upgrade Processes	112
Package Attributes	113
Update Types	114
Minor Upgrade.....	114
Small Update (Admins Need Not Apply).....	114
Major Upgrade.....	115
Simplifying Upgrades	115
Patch Packages.....	116
Generating Patches.....	117
Patching Reality Checks	117
Conflict Management for Package Structure	118
A Word About Merge Modules	118
Merge Modules in the Administrator’s World.....	119
Merge Modules as a Poor Man’s Conflict Management Tool.....	119
Replacing Repackaged Files with Merge Modules	120
Administrator and In-House Developer Generated Merge Modules.....	121
Summary	121

Copyright Statement

© 2002 Realtimepublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimepublishers.com, Inc. (the “Materials”) and this site and any such Materials are protected by international copyright and trademark laws.

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

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


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

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

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.

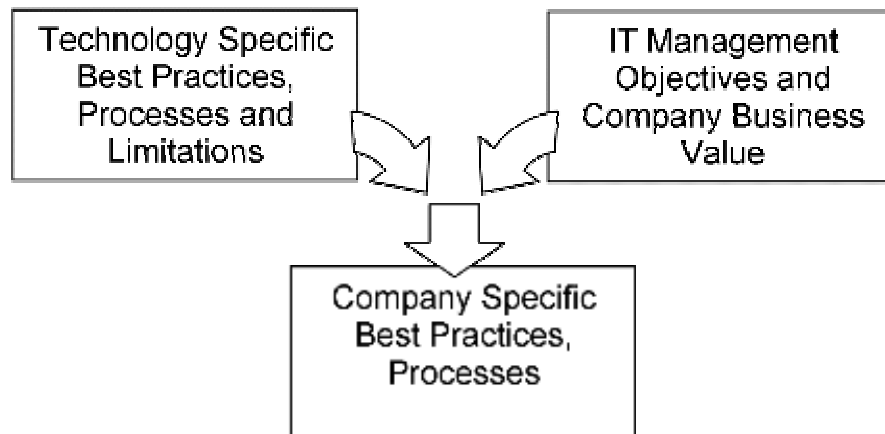



Figure 4.1: Best practice formulation.

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.

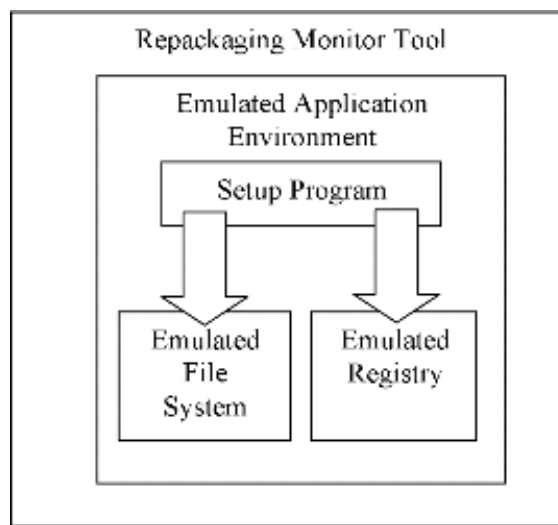



Figure 4.2: Repackaging technology monitors setup changes in real time.

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

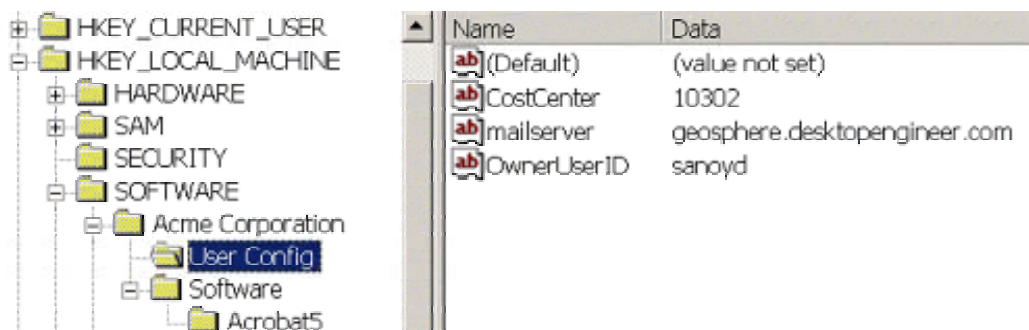


Figure 4.3: Registry storage of management data.

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 handled 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 repackaging 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.

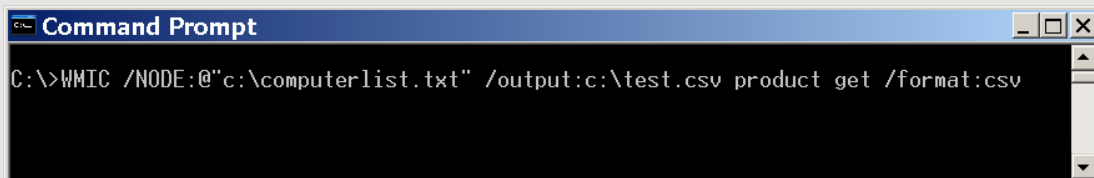
An alternative approach to converting these scripts is to repackage them from the legacy repackaging technology into the new tool. Doing so will ensure that *the results* of all custom coding in the old packaging technology will be captured. Repackaging your repackaged applications will allow you to 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.



```

C:\>WMIC /NODE:"c:\computerlist.txt" /output:c:\test.csv product get /format:csv

```

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.

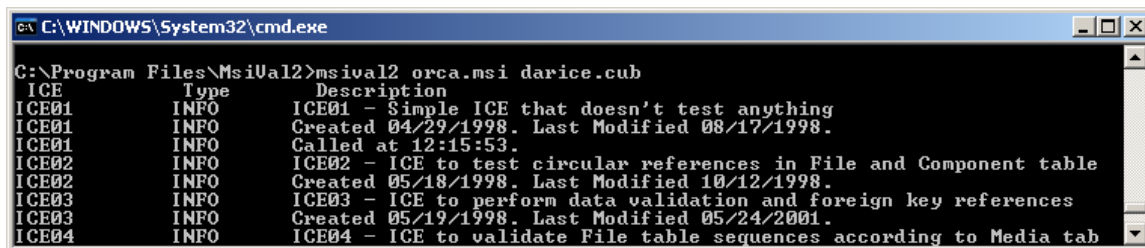
- ❏ If you will be doing MSI scripting outside of custom actions, there are many sample VBScripts in the SDK with full explanations in the SDK documentation. Search the SDK for “Windows Installer Scripting Samples.” In addition, the SDK section “Automation Interface” details all the API calls that can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



```

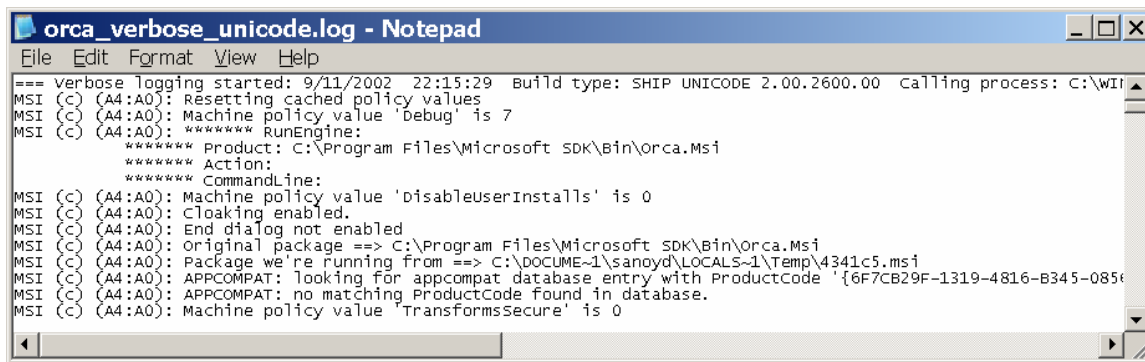
C:\WINDOWS\System32\cmd.exe
C:\Program Files\MsiVal2\msival2 orca.msi darice.cub
ICE      Type      Description
ICE01    INFO      ICE01 - Simple ICE that doesn't test anything
ICE01    INFO      Created 04/29/1998. Last Modified 08/17/1998.
ICE01    INFO      Called at 12:15:53.
ICE02    INFO      ICE02 - ICE to test circular references in File and Component table
ICE02    INFO      Created 05/18/1998. Last Modified 10/12/1998.
ICE03    INFO      ICE03 - ICE to perform data validation and foreign key references
ICE03    INFO      Created 05/19/1998. Last Modified 05/24/2001.
ICE04    INFO      ICE04 - ICE to validate File table sequences according to Media tab
  
```

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.



```

orca_verbose_unicode.log - Notepad
File Edit Format View Help
=== Verbose logging started: 9/11/2002 22:15:29 build type: SHIP UNICODE 2.00.2600.00 calling process: C:\WIN
MSI (C) (A4:A0): Resetting cached policy values
MSI (C) (A4:A0): Machine policy value 'Debug' is 7
MSI (C) (A4:A0): ***** RunEngine:
***** Product: C:\Program Files\Microsoft SDK\Bin\orca.Msi
***** Action:
***** CommandLine:
MSI (C) (A4:A0): Machine policy value 'DisableUserInstalls' is 0
MSI (C) (A4:A0): Cloaking enabled.
MSI (C) (A4:A0): End dialog not enabled
MSI (C) (A4:A0): Original package ==> C:\Program Files\Microsoft SDK\Bin\orca.Msi
MSI (C) (A4:A0): Package we're running from ==> C:\DOCUME~1\sanozd\LOCALS~1\Temp\4341c5.msi
MSI (C) (A4:A0): APPCOMPAT: looking for appcompat database entry with ProductCode '{6F7CB29F-1319-4816-B345-085
MSI (C) (A4:A0): APPCOMPAT: no matching ProductCode found in database.
MSI (C) (A4:A0): Machine policy value 'TransformsSecure' is 0
  
```

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 msixec.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.

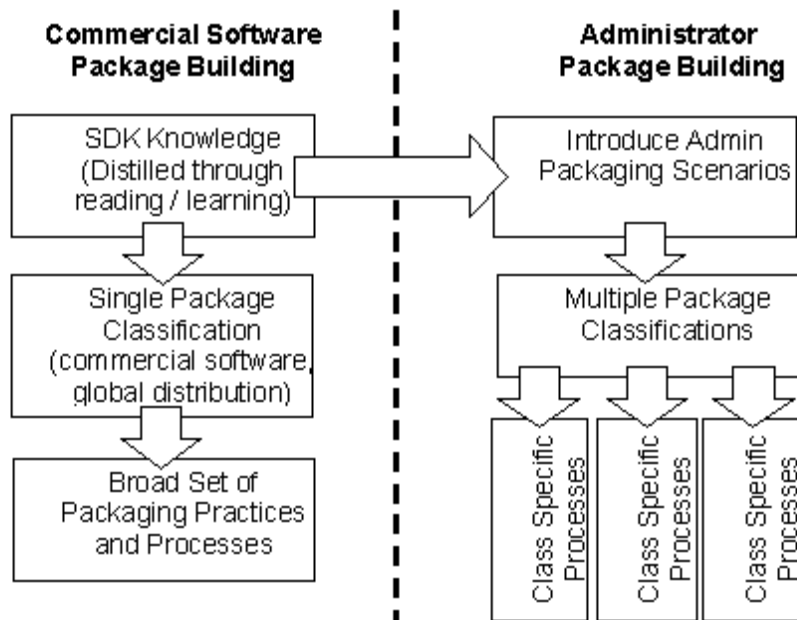


Figure 4.7: Process building guidelines for administrators.



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 unserved.

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.

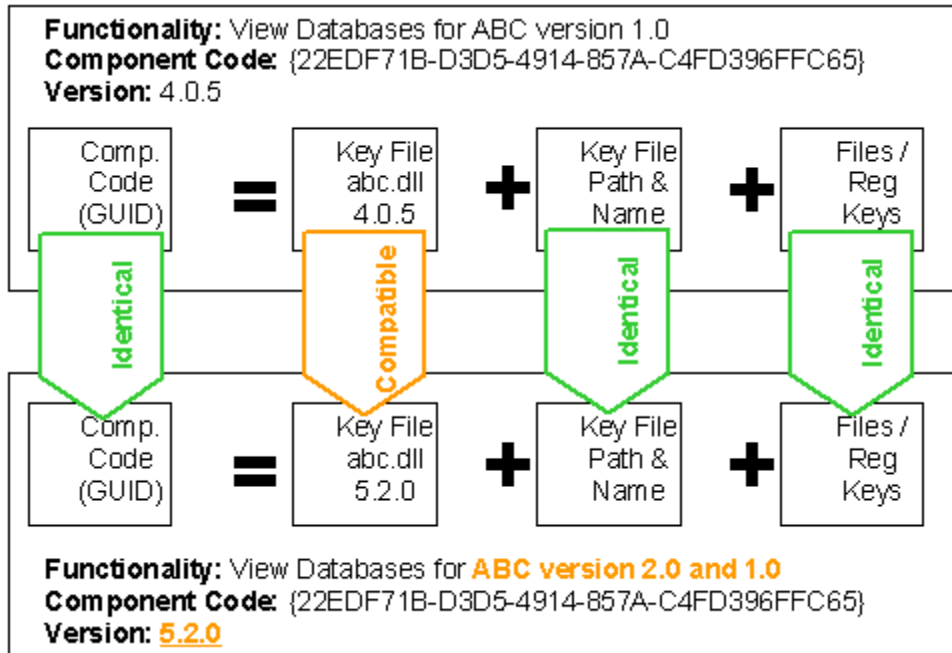


Figure 4.8: Compatible software code change upgrades component.

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.

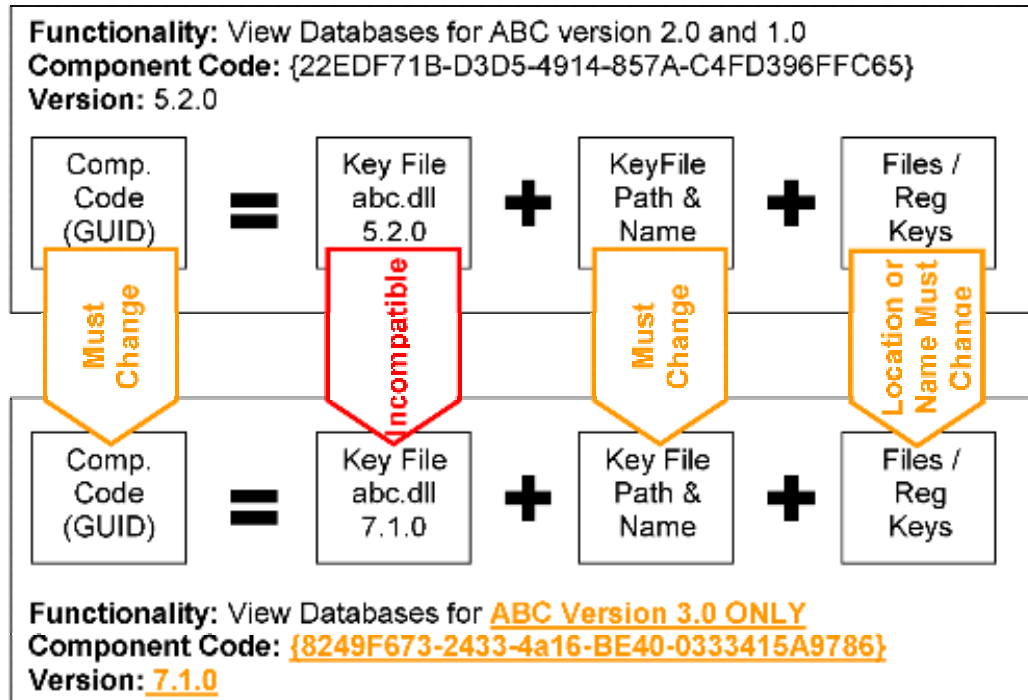


Figure 4.9: Incompatible software code change spawns new component.

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.

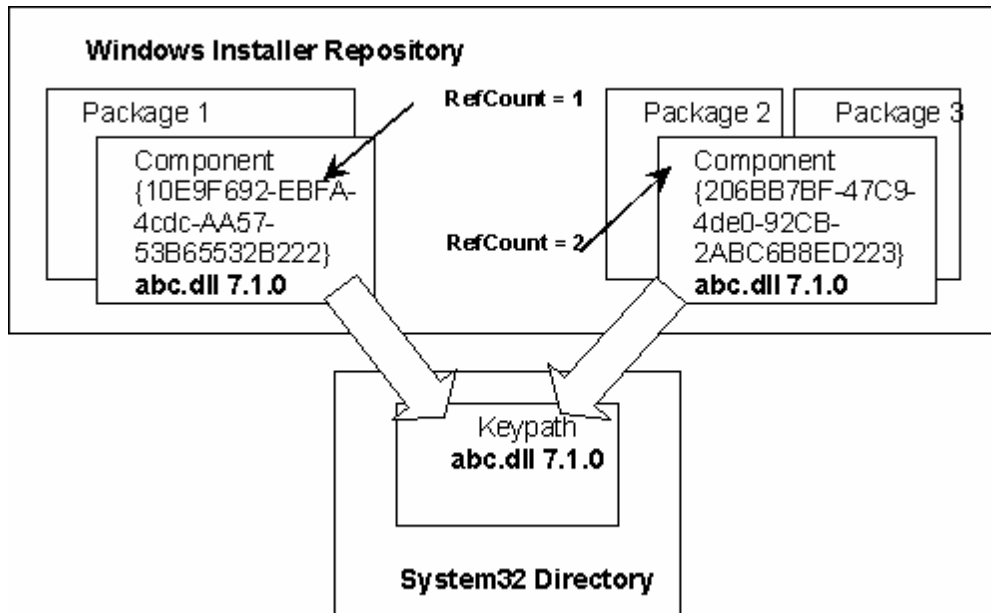



Figure 4.10: Duplicate component definitions.

 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.

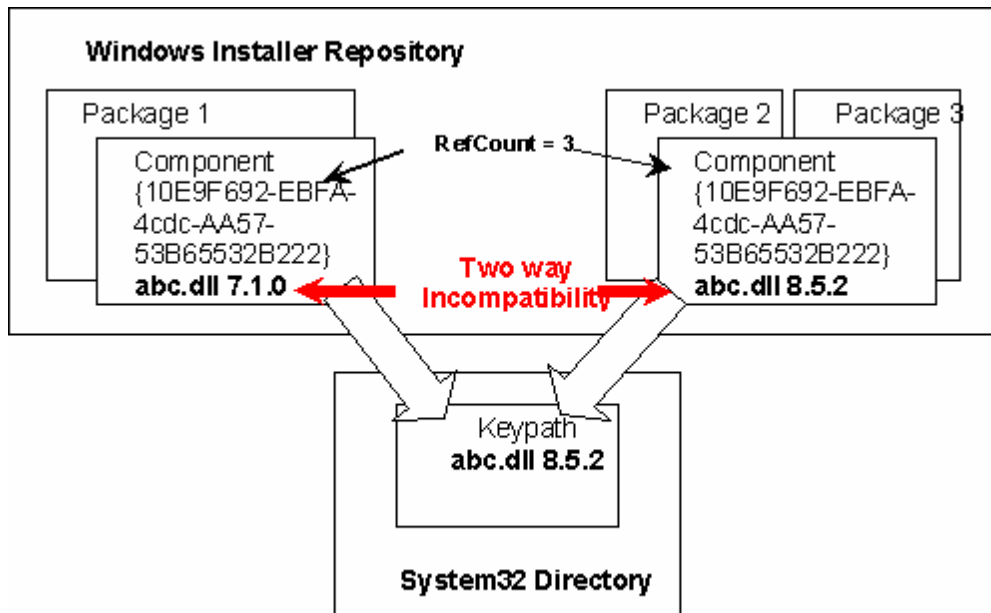



Figure 4.11: Conflicting component definitions.

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.