realtimepublishers.com™

The Definitive Guide™ To

Windows Software Deployment

LANOVATION®

Making Complex Technology SIMPLE *Since 1985*

Leslie Easter

Chapter 2: Conformance Evaluation

Whew! Conformance evaluation are big words for such a simple concept: How willing are you to install or upgrade software? For you and I as individual software users, the answer to this question revolves around cost, need, and expectations. Companies releasing software internally must ask the same kind of questions on a much larger stage.

If we're interested in a software package and its minimum requirements exceed those of our 3-year old home computer, how likely are we to upgrade our computer to use the software package? Odds are not good if the need for the software package is half-hearted. We're more likely to upgrade a personal computer to move to the latest version of Microsoft Office than to Wing Commander. But it’s a function of disposable income and other non-related priorities.

If you spend a few minutes thinking about the software you’ve purchased, you’ll probably realize that you invested more than the simple cost of purchase. You’ve spent the time installing the software onto your computer. This process may have involved clearing up some disk space and removing some older software. The install program may have involved tinkering with your system to get the install program to work—installing an updated Internet browser for example. More than likely, you also spent some time learning the software—either through the manuals or online Help. We can safely say that you’ve developed a type of relationship with your new software.

If we look at an investment of time, energy, and money into our software as a relationship, conformance evaluation is the courting phase. It’s the period of time you spend thinking, evaluating, and price shopping before you purchase software. In this chapter, we’ll address the issues of distribution and support of the software.

As individual software user’s, we have it relatively easy. Software consumers that represent large volumes of users—in the order of thousands—have a very difficult task. The process is pretty much the same, but the scale of investment is several orders of magnitude greater and the costs of success and failure are also a lot larger. This task is what the task that we’re going to focus on in this chapter. Even though the quantity of software packages is larger, the concerns of large-scale software consumers are identical to those of individual users.

Large-volume software consumers start by determining the level of need for the software package. They need to understand if the applicable computers meet the minimum requirements for the software. They also need to demonstrate that the investment of time and energy to distribute the software will lead to a cost savings. These factors and more need to be considered, evaluated, and answered before the software-deployment process can start.

In the remainder of this chapter, we’ll follow the trail of software deployment. We’ll look at everything from evaluation to installation to support. This trail will lead us through the following phases:

- Acquisition—Incorporates the process of verifying whether the software is necessary and achieving a plan of deployment.
- Distribution—Includes delivery, maintenance, and reporting of the software packages.
- Support—Encompasses the responsibility of maintaining the existing software in a productive state for the end user.

The remaining sections deal with these phases in much more detail. After reading this chapter, you'll have a complete idea of the software-deployment process. More importantly, you'll know which questions and tasks need to be resolved before beginning the task of deploying software.

Acquisition

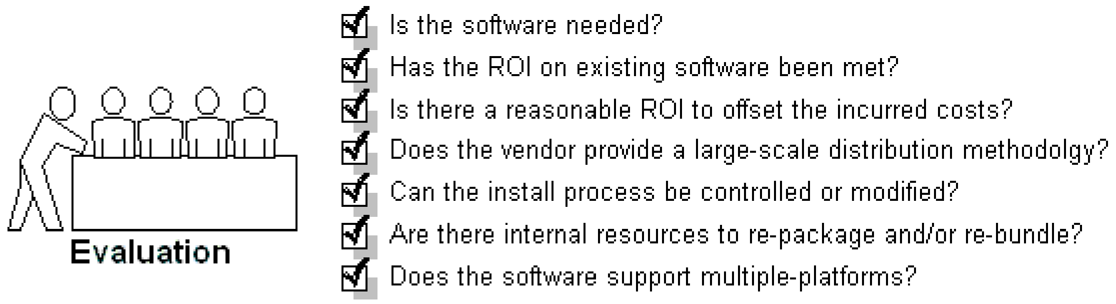
The process of acquiring software isn't as simple as you'd first think. Not only is there a fair amount of work to do before the software is purchased, but there is also an awful lot of work that must be done after the software is inhouse. In any event, the results of the acquisition phase of software deployment determine not only if the software rollout will occur but also, generally, how the rollout will occur.

Let's start at the beginning. Acquiring software is a three-step process that involves an evaluation of need, testing for conformance to standards, and finally a retooling of the delivery mechanism:

- Evaluation—Determines whether the software is necessary in the working environment.
- Testing—Involves testing for hardware and software compatibility.
- Tooling—Incorporates reworking the existing deployment package.

Evaluation

The goal of evaluation is to determine whether the software is necessary and to estimate the cost of implementing the software-deployment process as well as the Return on Investment (ROI). At a minimum, the cost of implementing the software should be offset by its usefulness. In many cases, the actual deployment infrastructure (how the software packages will be distributed) will already be in place. Work done at the evaluation level determines whether the software is worthwhile and how deployment should proceed by answering the questions that Figure 2.1 presents.



- ✓ Is the software needed?
- ✓ Has the ROI on existing software been met?
- ✓ Is there a reasonable ROI to offset the incurred costs?
- ✓ Does the vendor provide a large-scale distribution methodolgy?
- ✓ Can the install process be controlled or modified?
- ✓ Are there internal resources to re-package and/or re-bundle?
- ✓ Does the software support multiple-platforms?

Figure 2.1: Check list of questions for the evaluation process.

Determine Whether the Software is Necessary

The first step is to verify the usefulness of the software. Do you have preexisting software that can do the same job? How about upgrades or extensions to pre-existing software? After all, if you've already got something on the computer that can suffice, the process of deploying software wouldn't be necessary, and a software extension or upgrade would be less complicated.

The need for the software is based on the problems it's expected to resolve. In some cases, the software may be a requirement, for example, to view or edit proprietary files. In other cases, the software may automate tasks that were previously handled by a time-consuming process. The list could go on, but the central point is that the end goal of deploying the software should be clearly defined.

Calculate the ROI

If current software can't meet the current need, and the need is imperative to accomplishing a corporate goal, the next step involves calculating the cost of deployment. The cost of the software plus the complete cost of deployment (installation, configuration, training, and support) is the investment cost. The money saved by using the new software is the benefit cost. Initially, the investment cost of the software will exceed the benefit cost of owning the software. Over time—the life of the software—the benefit cost should exceed the investment cost.

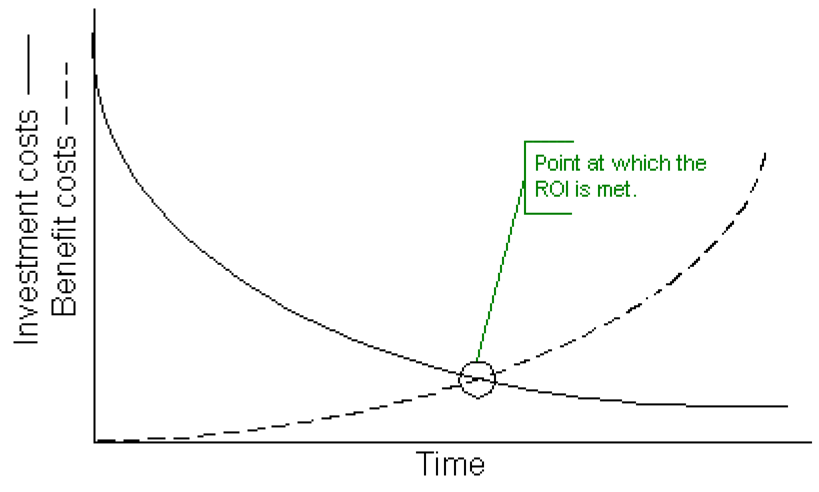


Figure 2.2: The point at which the software deployment effort is financially worthwhile.

In Figure 2.2, the ROI is met at some point after the software has been successfully deployed. In this figure, the complete investment cost is spread out over a period of time. As time goes by, the benefit costs offset the investment cost. While this is a simplistic model, it upholds the basic tenet of software deployment: Deploy only the software that will exceed its ROI. Conversely, don't replace software before it ROI is reached. If the need or the longevity of the software is uncertain, a careful reexamination of the software need is warranted.

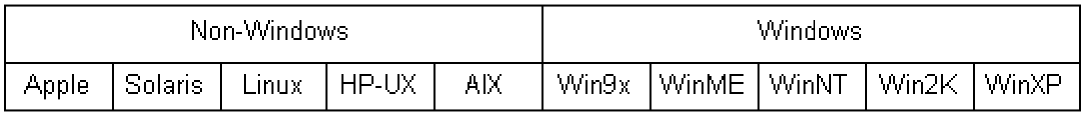
Determine Who Will Require the Software

Once a positive ROI has been calculated, the next step is to determine who requires the software. This number may be fixed initially, but as teams grow or the need becomes more widespread, additional copies of the software may be necessary. At this point, a worthwhile task is to associate the software with specific groups or departments. Some deployment methodologies make deploying across a group of users substantially easier than deploying to specific users.

Some corporate environments structure their system user groups specifically by software requirements. Although this structure covers the majority of cases, keep in mind that there will undoubtedly be exceptions. Deciding how to cover the exceptions should be part of the overall software-deployment strategy.

Develop a Plan Based on Operating System

Now that the need and users have been identified, the next step is to determine the computer system that these users have. All computer systems can be separated into two broad groups: non-Windows and Windows, as Figure 2.3 shows. This separation has as much to do as market share as anything else. Clearly in your work environment one operating system will dominate. Within the non-Windows group there are two distinct types: Apple OS and UNIX variations—although with the latest release of the MacOS, which embeds a FreeBSD variant of UNIX, even these types are becoming blurred.



Non-Windows					Windows				
Apple	Solaris	Linux	HP-UX	AIX	Win9x	WinME	WinNT	Win2K	WinXP

Figure 2.3: Classification of operating systems.

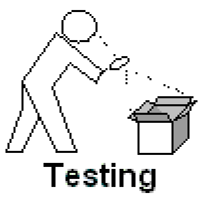
With the exception of software-management systems, there is no best-case solution that deploys to all operating systems. Typically, you'll find yourself having to come up with similar strategies with different implementations to support a heterogeneous platform base.

We'll get to software-management systems later on, but for large software deployments system-management systems offer a cost-effective way to transparently cope with software deployment on heterogeneous systems.

Compatibility Testing

Now that we know the software is required and it's cost effective to deploy the software, the real work begins. Making sure the new software plays nice with the existing software is an important part of software evaluation. Without thorough compatibility testing, introducing new software could be disastrous with effects ranging from intermittent software anomalies to complete operating system failure.

In some cases, compatibility testing extends to the hardware. A complete test would include a range of computer types. Not only would this testing cover brand-to-brand systems but also client computers, server computers, and laptops. If the software has stringent video requirements, for example, testing the software with a large range of video cards and monitors may be in order. If the software has large memory requirements, testing the software on typical corporate systems with a large range of memory capacities would be worthwhile. Figure 2.4 provides a list of questions for you to answer during the testing process.



- ✓ Are resources available for in-depth testing?
- ✓ Is the software compatible with existing software?
- ✓ Are there conflicts over shared resources?
- ✓ Does the software introduce changes to the operating system?
- ✓ Does the software have a negative impact on network traffic?

Figure 2.4: Check list of questions for the testing process.

Compatibility testing starts by defining the typical user computer configuration. In cases in which computers are heavily standardized, this process can be straightforward. In environments in which computers are freely modified, this process is not only complicated but also impossible to test for complete compatibility.

The typical configuration should include standard hardware and software combinations. The boundaries of the configuration definition should be realistic. After all, a theoretical matrix of testing could yield hundreds of test scenarios when in reality there might be 20 or 30. The constraints of the configuration definition should be based on the group of users for whom this software is targeted. After all, it doesn't make sense to include those computers that aren't applicable for the software deployment.

When setting up the initial test scenario, be sure to use the most constrained security settings most likely encountered. This setup will force incompatibility issues between security settings and the examined software. You might need to revisit the exact security settings for the application to work effectively. An example is access to the specific registry keys on a Windows system or read/write privileges to data files.

Table 3.1 illustrates a sample matrix. This type of table can go a long way to ensure all possible configurations have been thoroughly tested. It's also an excellent reference document for the support folks. This table should be replicated for each operating system as appropriate. In addition, a similar table should be compiled for the hardware requirements.

Existing Software	**Test 1**	**Test 2**	**Test 3**	**Test 4**
Microsoft Office	×		×	
Corel Suite		×		
WordPerfect				×
Netscape Navigator 5.0	×		×	
Netscape Navigator 6.0		×		×
Internet Explorer (IE) 5.0	×			
IE 5.5		×		×
IE 6.0			×	
MDAC 2.1	×		×	
MDAC 2.5		×		×
NT Service Pack 3 (SP3)	×			
NT SP4		×		
NT SP5			×	
NT SP6				×
Status (Success/Failed):				

Table 2.1: Sample configuration-testing matrix.

Not shown in this table, but just as important, are the variations introduced by notebooks and laptops versus desktop computers. While less of a factor today, older models may have more of a discrepancy. Older models of notebooks and laptops have distinct hardware—different video, smaller hard drives, and less memory—which may not be compatible with the testing software.

Create a Reproducible Test Environment

In practical terms, configuration testing starts by creating a reproducible test environment. At a minimum, this environment would be a single computer whose hardware could be reconfigured to match the results of the hardware-matrix table. In addition, the operating system itself could be reinstalled multiple times—even for multiple operating systems.

The importance of reproducibility cannot be overstated. The ability to quickly and accurately reproduce any single test scenario is imperative. Although hardware changes would require physically adding or removing components, software changes could be instantiated through replication software. There are a number of software packages that capture a system's binary state and store it to a data file for later use. For a range of configurations, these system image files may require large amounts of disk space, although burning the image to a CD-ROM is certainly an option for smaller test scenarios.

A well-known imaging software package is Symantec's Ghost. You can find this software at http://www.symantec.com.

Some failures in reproducibility are a result of subtle differences in hardware. Just because two computers have the same model number and external case doesn't mean they're identical. Some hardware manufactures will substitute similar internal devices when original sources become scarce. Opening up the computer and logging its exact hardware contents may be worthwhile.

The reproducibility of tests is important for a number of reasons. First, as a sanity check, reproducible tests are useful to verify issues or validate later assumptions. Second, for software support, they're ideal to quickly reproduce a client's configuration locally even though the actual problem computer may be hundreds of miles away.

Once a test environment is created, the process is quite simple. Set up the hardware and software for each of the individual matrix checks. Install the to-be-deployed software, and test. After a single matrix test, you'll quickly see why this phase of software deployment is the most time consuming.

DLLs and Shared Resources

Remember the discussion about static and dynamic linking of libraries from the Chapter 1? Well, bad news. From that discussion you'll see why the entire functionality of the new software must be exercised. In addition, the entire functionality of the pre-existing software (from the software matrix table) must also be fully exercised. To avoid DLL Hell, which I also discussed in Chapter 1, you can rely on a sufficiently thorough software matrix to uncover any version incompatibilities.

In addition to these two very important items, there are other things to consider. The bulk of these considerations have to do with shared resources. Shared resources include file extensions, user-document workspace, ActiveX controls (through self-registration), ODBC databases and drivers, fonts, shortcuts, and so on.

A classic example of a shared resource is file extensions. In a Windows–operating–system environment, applications can be linked to specific document types. The document type is identified by the file extension. On most systems, the extension .txt is associated with Notepad.exe. This association means that double-clicking a document called Readme.txt will launch Notepad with the selected document active. Although this association isn't usually an issue, when two applications vie for the same extension or any shared resource for that matter, you'll have to make a decision about which one wins.

A classic example is Web browsers. When installed, Internet Explorer and Netscape want to be associated with .htm and .html extensions. However, only one application can be associated with an extension at a time. Another example, are common picture extensions such as .gif, .jpg, and .bmp. These extensions are usually associated with the same viewing software. The same holds true for video data files with extensions .avi and .mpg.

Another often-forgotten test point is uninstalling applications. I'm not specifically talking about the new software—any software will do. Sometimes a resource that is shared is inadvertently removed during the uninstall process. This removal has the nasty tendency to break the software that is staying behind. This kind of event should be noted for latter conflict resolution.

Software-Automation Tools

For some companies, investing the time and energy into software-automation testing tools may be worthwhile. Although these tools can be expensive and time consuming, they compensate by being reusable. In this case, you create an initial automation test script to verify compatibility of the new application. You run the new automation test in tandem with all the existing tests. After the application is accepted and deployed, its automated test is saved and used for compatibility tests for future software. Over time, the savings in test time and resources will be substantial.

The Results

Are you going to be able to achieve 100 percent test coverage? Absolutely not. But the closer you get, the better you'll be able to predict the success of the software-distribution process; not to mention the reduced number of support issues, which should be enough incentive to gain as much test coverage as possible.

If you've created a thorough hardware and software matrix, most of the work is done. Any issues should be adequately documented for later conflict resolution. The testing and verification phase is time consuming and tedious, but it's in no way challenging.

The results of compatibility testing determine the configuration steps necessary to use the software. What you'll end up with is a hefty document that specifies all the detected conflicts. The next step is to gain sufficient resolution on each of the conflicts to proceed with the deployment effort.

Now that you've seen what compatibility testing entails, you may be interested in hearing more about standardization. After all, the closer all computers are in pre-existing hardware and software, the easier compatibility testing will be. The smaller we can make the hardware and software matrix tables, the less compatibility testing needs to be performed.

Standardization

As we saw in the previous chapter, there is a way to reduce the amount of work required during the evaluation phase. You could simply lock down users' computers. Restricting what users can change on their computers drastically reduces the differences between systems. And, because compatibility testing is a function of standardization, testing for compatible software is easier.

The basic premise is simple: the effectiveness of software deployment is largely dependent on having an accurate picture of the target platform. In fact, follow me closely here, software deployment is a cost-reducing benefit that is directly related to the cost-reducing functionality of computer standardization. Although that fits nicely in the sentence, it may leave you scratching your head.

At the extreme level of standardization everyone has the exact same computer—hardware and software included. Keep in mind that this discussion is completely theoretical. If you can buy that premise, you can easily see that deploying software in this kind of environment is very simple. By completely removing the matrix table, compatibility testing is limited to a single computer! Before you roll your eyes, this description is exactly the scenario for the home consumer—that's you and me. The only luxury the home consumer doesn't have is testing the software before installing it.

Range of Standardization

Standardization doesn't even have to be so clear-cut. There is a full range from completely open systems to locked-down environments.

For starters, computers can follow a platform standard. In this case, each computer abides by defined standards. These standards limit the computer to a list of available operating systems, client computers, laptops, servers, and network communication devices.

Computers may be required to follow specific application standards. In this case, each computer abides by defined software standards. Typically this standard is a list of version-specific, vendor-specific applications with standards established for specific users and user groups. Application standardization includes setting up purchasing policies and procedures as well as monitoring and managing application distribution, installs, and updates to assure that compliance is maintained.

Enforcing Standards

Enforcing standards is accomplished in a couple of ways: managed environments or locked-down environments. A managed environment keeps users from making changes to their systems, such as introducing unauthorized software or changing settings that may cause conflict with other system resources. In addition, a managed environment controls the ease of use of the desktop, providing a common set of applications and access for groups of users or individuals. In this manner, users are presented with only the tools they have been trained on and need for the job. This environment assures that changes are managed.

A locked user environment is a limited version of the managed user environment. It precludes users from changing settings and installing unauthorized software. It is different than a managed user environment in that it is machine specific and is local to the device. It is not typically managed or synchronized with a server profile. The primary reason for a locked user environment is that it makes a predictable target for software distribution.

Conflict Resolution

Okay, after a couple of months of testing, we've got a stack of incompatibilities. They could range from the serious to the trivial, but we still must sort them out and arrive at workable solutions. In every case, additional work is required to discover the exact cause of the conflict. Maybe an existing application breaks when the new software updates to MDAC 2.6. Does the new software absolutely require MDAC 2.6? We don't know and we'll have to investigate further.

For each incompatibility, we'll have to attribute the incompatibility to a piece of software. From there, we'll need to clearly define the exact problem—having imaging software helps greatly in reproducing the problem. Then a couple different solutions will have to be applied and retested.

During this process, you'll benefit from playing with each of the software packages in different configurations. You may discover the incompatibility is caused by an optional component of the software. Removing that component removes the problem and resolves the conflict. However, every incompatibility may not be so easy to resolve.

Let's take a deeper look at some of the standard issues. The most common incompatibility is versions of system files. This incompatibility usually involves a software package installing a new core component. This new core component may or may not be required. Depending on how the install program was written, a developer may have just grabbed the latest and greatest from Microsoft's Web site and slapped it into the install package.

The conflict-resolution process involves a series of tests. First, replace the new core component with the oldest possible component for your environment. Once again, test each software application for complete functionality. Chances are you'll replace one incompatibility for another. Keep updating the core component—perhaps while the original core component wasn't sufficient, a version in between resolves the incompatibility.

Part of the conflict resolution may require retooling the install package. This requirement is especially necessary for some incompatibilities: If you remove shared resources during an install, retooling the install package is the only solution.

If the incompatibility is less serious, a simple runtime script run before or after the install program may be a remedy. Of course, this solution would imply that the existing install program meets all other requirements.

When the work is done, you'll have documented each conflict resolution with an applicable fix. Starting the compatibility testing again with the documented conflict resolutions is an excellent way to verify the software-distribution process. The results of the conflict-resolution phase determine the configuration steps necessary to use the software. Additional lab testing verifies the fixes. This testing is where the imaging process from compatibility testing will come in handy.

In the cases in which nothing works, a reevaluation of the software need may be warranted. It may be simply that existing software must be removed for the newer software to be installed. The complete documentation set should be made available for the support team.

Tooling

In virtually every case of conflict resolution, changes must occur within the software-install process. Because the software application typically already has an install wrapper, retooling or recreating the install program is a necessary part of software deployment.

The conflict-resolution process may all come down to the simple fact that the install package doesn't conform to the deployment objectives. In this case, an install program that conforms to those objectives must replace the existing install program. Figure 2.5 illustrates the questions to answer during the retooling process.

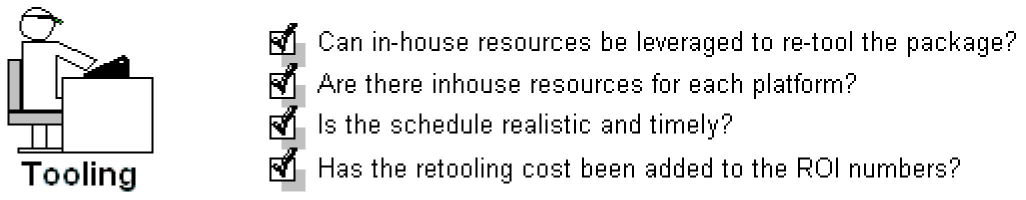

Tooling
Can in-house resources be leveraged to re-tool the package?
Are there inhouse resources for each platform?
Is the schedule realistic and timely?
Has the retooling cost been added to the ROI numbers?

Figure 2.5: Check list of questions for the tooling process.

Don't get me wrong; retooling a software application isn't limited to conflict resolution. The following list provides several other just as important reasons to retool:

- To resolve incompatibilities and conflicts
- To bundle additional software
- To add or remove components within the distributed software
- To eliminate or customize the user interface
- To facilitate a proprietary software-distribution mechanism
- To provide additional install configuration

Although resolving conflicts and removing the user interface are the most common reasons to retool, it may also be a way of bundling additional software. This bundling could include other software applications or simply proprietary extensions to the software. Instead of using three separate installs, you could retool them all into a single package.

For example, suppose that you're delivering Microsoft's PowerPoint. By creating your own install package, you can include corporate templates and sample presentations. You could also choose to remove unnecessary PowerPoint components and create corporate-specific shortcuts.

The useful thing about retooling is that you seldom have the same constraints that the original software manufacturer did. When the original install program was created, the range of supported computer configurations was quite large. Retooling for your own environment is usually a more manageable task. For example, although the original software may have supported all flavors of the Windows operating system, your inhouse requirements may be just to support Win2K.

> There are two methods of retooling: repackaging and creating an install program from scratch. Although creating an install from scratch is functionally the same as repackaging, the term repackage is associated with a specific install-package creation process. Thus, we'll stick with that usage here.

Repackaging

Repackaging is the process of extracting an application from one delivery vehicle and inserting it into another. Repackaging involves capturing the changes made to the computer while the original install program is running. (Although this functionality is what is advertised for the sake of simplicity, every repackaging tool on the market compares the changes before and after an install is performed to generate a package and doesn't actually capture the changes made to the computer while the install is running. I elaborate this point in Chapter 3.) There are specific software packages that provide this functionality for you automatically. Some even go so far as to recreate the install package automatically. Of course, you'll want to get into the package to make your modifications.

A classic use of repackaging is to remove the user interface from the install program. In this case, perhaps the original install program didn't provide an unattended or silent install. By tooling our own install program by repackaging the software, we can create a suitable solution.

There are limitations to repackaging: Repackaging software detects and replicates only changes to the system. Thus, if a file is already on the machine on which the repackaging task is being performed, the file will not have changed during installation and won't be included in the repackage. Therefore, the file will be missing if pushed to a machine that doesn't already have the file installed.

In addition, there are some technologies that the repackaging system is not familiar with—on Windows systems, configuring Internet Information Server (IIS) for a virtual directory or Web server is a common limitation.

For the most part, the fundamental limitation to repackaging is that it doesn't capture the logical evaluation process of the original install. Some repackaging software provides more options to you than other repackaging software—shopping around and comparing features will help you greatly. The next chapter deals exclusively with repackaging.

Create an Install Program from Scratch

Alternatively, you can write the install program from scratch—a much bigger task than repackaging. In fact, if at all possible, repackaging is the preferred option. Writing an install program from scratch, although giving you greater flexibility, incurs the most responsibility.

A very rough time estimate comparison makes repackaging the quicker option. In fact, a typical repackaging process may take the most part of a week, including testing. Creating an install package from scratch on a moderately difficult application may take the bulk of a month. That estimate also includes testing. Creating an install program from scratch is discussed in detail in Chapter 5.

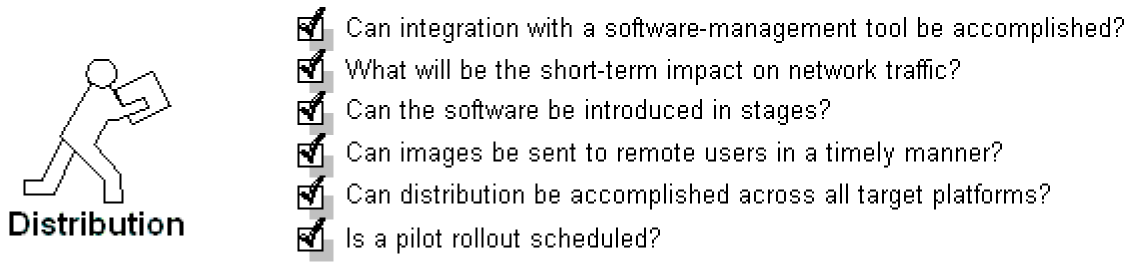
Lab Testing

After retooling the software install package, thorough testing will have to be done to ensure proper installation and configuration. After all, you don't want to introduce new problems in the process of resolving existing ones. There's nothing like success, and lab testing is a way of determining how close your newly tooled package is to success.

Distribution

Now we’ve done it. We’ve determined that the software application is important. We’ve resolved all outstanding software conflicts. We’ve even retooled the install package for the software—making install-specific changes along the way. The next step is to derive a strategy to get the install package to run on users’ computers.

There are a number of ways to accomplish this goal. They range from making the install image accessible on a networked drive to administering the install process through a software-management framework. Which method you select is a function of cost and your final objectives. In this section, we’ll spend a little bit of time talking about each one. Along the way I’ll point out the pros and cons of each method. Figure 2.6 outlines the key questions you should answer for the distribution process.



- ✓ Can integration with a software-management tool be accomplished?
- ✓ What will be the short-term impact on network traffic?
- ✓ Can the software be introduced in stages?
- ✓ Can images be sent to remote users in a timely manner?
- ✓ Can distribution be accomplished across all target platforms?
- ✓ Is a pilot rollout scheduled?

Figure 2.6: Check list of questions for the distribution process.

There are some key components to the distribution process that are secondary to the physical distribution of the software:

- Provide warehousing support (catalog and store the package for delivery)
- Verify software-package integrity
- Report success and failure status of distribution attempts

Within this section, we’ll also touch upon the topic of software-management environments. These are prepackaged software bundles that automate much of the software-deployment process. I’m introducing the topic here because this software is a must-have solution for large corporations. Chapters 6 and 7 go into much more detail.

Image Delivery

Delivering a software image can be done in a number of ways. The image could be a physical CD-ROM that is passed around the office. A link could be made on an Intranet Web site. The software could be installed directly to the computer without user intervention. Or an email could be sent to each user identifying the software location and requesting that the user install the software.

There are pros and cons with each delivery method. In this section, we’ll talk about some of these tradeoffs. You should be able to get a general idea of the software-distribution mechanism appropriate for your environment.

The primary factor in determining your distribution method is the method that's currently in place. The second important fact is the number of computers that require the software. While my prioritizing doesn't do the "right solution" any benefit, I'm assuming that the implementation you're dealing with is correct for your environment.

If a distribution mechanism is already in place and working effectively, there's not much else to consider. Software-management systems are expensive and time consuming to implement. If your company has made such a selection, there were very good reasons for its selection. There's not much else to do but learn how the environment works. However, distributing software in a fresh environment is primarily a function of the number of computers that require the software.

Independent of how you choose to deliver the install package to users' computers, there are primarily two delivery modes: push and pull. A push install delivery mode is one in which the server drives the delivery, whereas a pull delivery mode is one in which the user (or client software) initiates the install action.

Push Delivery

By far push delivery is the preferred method of choice. It ensures that the distribution process is started for the targeted system and can usually record a success or failed status. A push delivery can also make sure that a valid license exists for the targeted computer and keep track of the complete license use of the software.

There are many remote options available. In general, the targeted computer must be powered on (although most system-management software and hardware options provide a remote boot option). Usually, push installs are accomplished at night when most users aren't active on the computer and network traffic is minimal.

In addition to the standard mode of delivery, a push install can also advertise the availability of the software application. In this model, the initial advertisement is pushed to the computer, but the actual installation of the corresponding software is done through a pull install. Advertisement is supported natively in Win2K and Windows XP; however, the mechanism for implementation is well documented and could be extended for use with other operating systems. There are two forms of advertisement: assigning and publishing.

Assigning

Assigning the availability of an application gives the appearance that the application is completely instantiated on the user's computer. All shortcuts are present, all self-registered actions are present, and all file extensions exist. Basically, everything exists but the application itself.

Assigning is useful when an application isn't required by a user, but is available when needed. The strength of assigning is that it doesn't consume disk space until some request for functionality is made.

When a request for use is made, the operating system makes a request to the Windows Installer service. The Windows Installer service tracks down the source media image and initiates the installation of software files. Note that this process occurs silently behind the scenes. To the user, there is a delay between the selection of an advertised application and the actual launch of the application. During the delay, a status dialog is displayed to show activity.

Publishing

Publishing is a subset of assigning. In this case, the software isn't directly available through the user interface. Instead installation is triggered when other software makes a request for the software. Similar to assigning, when a request is made, the operating system makes a request to the Windows Installer service and the install proceeds as the previous section describes.

Pull Delivery

A pull software install is the easiest to implement. It's also the least expensive. It typically makes a network, URL, or intranet location available to the user and relies on the user to trigger the install process. This process has the drawback that it's difficult to force the user to install the software. A much needed security patch may languish because the user fails to trigger the installation. In addition to not being able to enforce installation and configuration management, a major reason for not implementing pull software installations is security. By default, users often don't have sufficient rights to perform software installation.

A variation of the pull install is to create an entry in the boot process. In a Windows environment, you could create an entry in the boot process through the RunOnce registry subkey, a logon script, or the Startup folder. The next time the computer boots up, the install process is kicked off. While the original entry in the boot sequence was made remotely, the fact that it's still client-initiated (in this case by the reboot action) makes this method a pull delivery. This method suffers from the usual pull delivery drawbacks.

Pilot Rollouts

An important facet of distributing the software is testing the distribution methodology on a representative sample group. The point of the pilot rollout is to test our assumptions about the target systems, the install package, the bandwidth, and so on. Not to mention that recovering from disaster is easier when it affects a small number of computers.

The first requirement of the pilot rollout is to find a good variety of target computers, groups, and users. This variety allows the rollout process to be exposed to the largest number of distribution circumstances. If remote and mobile users are a concern, they should be included in the pilot rollout.

If a pilot rollout detects a substantial drain on network resources, distribution to intermediate deployment servers may be in order. A deployment server has the effect of offloading network traffic through different connection points. In software-management terminology, these servers are known as software distribution points (SDPs). Another alternative is to stagger the software distribution so that different parts of the company receive the software at different times.

Any modifications that need to occur as a result of the pilot rollout should be documented and included with compatibility-testing and conflict-resolution documentation. The support process requires this kind of documentation to resolve issues after the final distribution.

A final part of the pilot rollout is to test accessibility of all groups to the software package. This testing includes distributions that occur through firewalls, virtual private networks (VPNs), and dial-up connections.

Software Management

A good software-management tool distributes and installs software. It compliments these functions with a sweeping range of support modules. These may include reporting, license monitoring, and network monitoring to name but a few. The clear advantage to using software-management tools is the automation and black box approach to software distribution—I mean distributing a software package to a user may be a simple click-and-drag action—the underpinning of what is really happening is not apparent.

Software-management software comes in several forms. It can be a focused software-deployment solution or it can be bundled within a larger enterprise-management framework. The distinction is that software deployment is a subset of the enterprise-management framework—typically a plug-in module. The larger enterprise-management framework includes many functions in addition to software deployment. We'll go over the details in later chapters. This section introduces the terms and concepts commonly associated with the software-deployment side of software management.

At a basic level, the role of software-management software encompasses the following areas:

- Inventory—Keeps a record of what is currently installed and by which user
- Delivery—Provides a means of getting the install package to users' computers
- Licensing—Keeps a record of in-use versus available licenses
- Reporting—Creates a variety of status reports
- Warehousing—Physically provides storage space for the distribution images
- Managing components—Performs a limited automated support role for the distributed software

The primary goal with this type of software is to provide a centrally managed location for the bulk of software-distribution tasks. Centrally managed implies ease of administration and clearly defined status of installed software and pending software releases.

Inventory

A software inventory is an up-to-date listing, preferably electronic, that contains detailed information about the client, network, and server software installed within an organization. Inventory information is used for troubleshooting, license compliance, and strategic planning.

Ideally, the information should also be automatically collected, updated, and maintained. An ideal inventory would include license terms and conditions, date of acquisition, user names and location, system installation details, maintenance agreements, usage monitoring, history, and other relevant data.

Without a software-management tool, software inventory would be limited to keeping track of purchased software. For small companies, this inventory could be a spreadsheet or hard copy. In this case, the inventory is manually updated and contains minimal information about the software. The objective is to meet minimum requirements to avoid breach of license challenges, and provide upgrade proof and volume purchase leverage.

A more advanced software inventory includes regular inventory of software from purchasing through distribution. At this level of detail, the inventory process would be used with asset-tracking and system-discovery tools.

Delivery

Of course the delivery aspect is the most crucial to our focus. In this case, software-management tools typically provide their own version of user and group profiling. This functionality allows software to be deployed based on policy management.

By creating a new user and associating the user with a specific computer and appropriate user groups, the corresponding software is automatically deployed. The software delivery vehicle in most cases is an unattended push installation to the client machines. Because of the extensive reporting, success or failure is often automatically indicated to the administrator.

My personal favorite is the ability of these tools to automatically schedule software deployments based on network traffic. When network traffic falls below an administrator-specified amount, software distribution is kicked off. If for whatever reason, network traffic picks up to a point that exceeds the administrator-specified threshold, software distribution is temporarily halted. This ability to monitor the network and automatically schedule software releases is truly impressive.

Licensing

Licensing is a key component of software distribution. The bulk of software-deployment tools support some form of integration with licensing mechanisms. The advantage to having licensing centrally managed is that the software-distribution administrator has immediate record of license status. Typically license status is part of the built-in reporting process. Licensing functionality includes software metering to manage the use and control the number of licenses. Many solutions also support the dynamic balancing of licenses across licensing servers.

Reporting

With built-in reporting, systems administrators have immediate access to the success and failure of software rollout. Additional reporting functions include auditing and management reports as well as network status and loads over time.

Warehousing

Warehousing is the physical storage of distributable software images. It usually incorporates the added benefit of license integration. In this case, not only does the software store the software-media image but also lets you know how many licenses are currently used and how many remain.

Policy Management

Policy management allows administrators to assign individual users to specific groups. Simply by belonging to a managed group, the user has access to job-appropriate software. Users can become members of multiple groups if their job function overlaps or to make software distribution simpler.

Manage Components

The manage components feature of a software-management tool provide automatic self-healing of distributed packages and catastrophic system recovery. Self-healing is usually provided by monitoring the client systems and keeping a record of the install status. A significant change in the client computer status would trigger a redeployment of the affected software.

Catastrophic system recovery is similar but on a larger scale. In the case of software failure, the recovery entails a reimage of the system followed by redeploying all subscribed software applications.

The manage components feature additionally tracks and manages software compliance according to specific rules. Events that fall outside these rules trigger a report to the systems administrator.

Finally, all managed software is included in regular reports. These reports usually include such details as software type, revision, user name, user number by software title, and a record of valid software installs versus failed installs.

Support

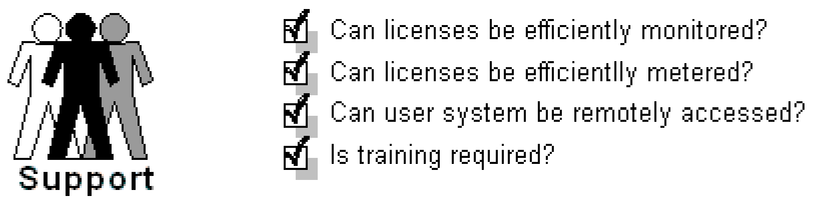
After a software application is installed to users' computers, there is an inherent responsibility to monitor the software to make sure it continues to function correctly. Added to this responsibility is that of educating the user about the software.

In addition to these two important tasks, support is the central repository for the process history of the software. The results of compatibility testing and any conflict-resolution processes should be documented and made accessible to the support team. This kind of information goes a long way to quickly resolving implementation issues as they arise.

The support facet of software deployment encompasses the following roles:

- Ensure software and computer integrity through file management.
- Manage the software-licensing schema.
- Resolve software issues through the documented deployment process.
- Serve as a repository for hotfixes, maintenance packs, and upgrades.
- Manage deactivation and decommissioning of out-of-date software.

Figure 2.7 provides a check list of questions for the support process.



- ✓ Can licenses be efficiently monitored?
- ✓ Can licenses be efficientlly metered?
- ✓ Can user system be remotely accessed?
- ✓ Is training required?

Figure 2.7: Check list of questions for the support process.

Software Repair

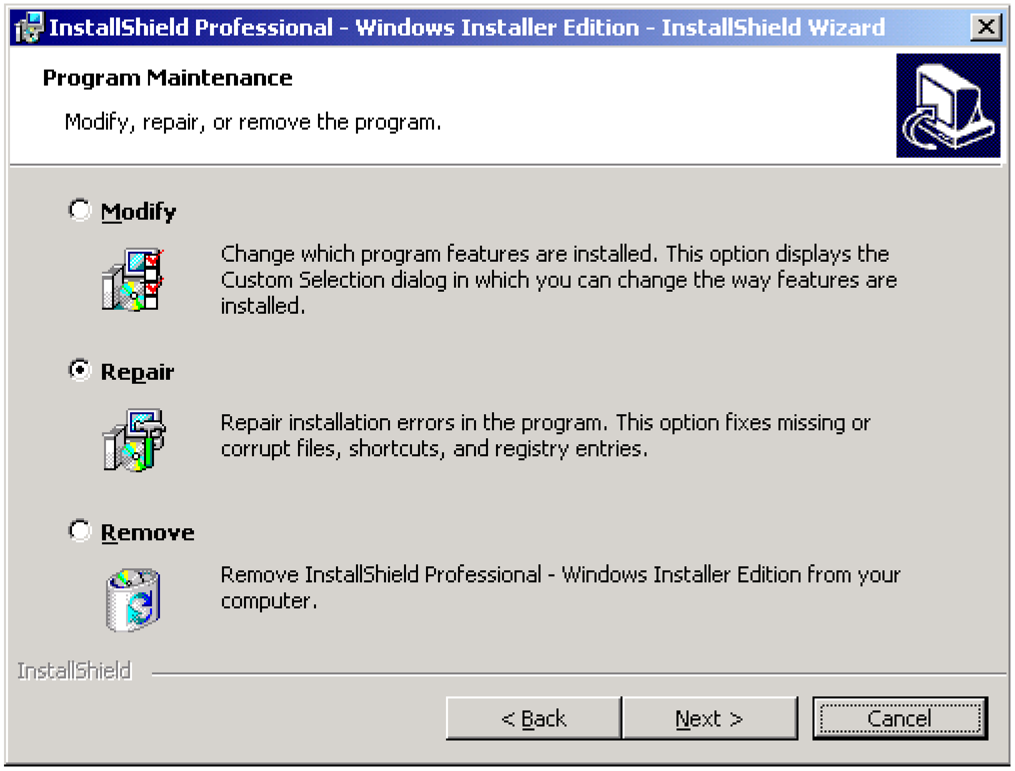
An interesting aspect of software support is repairing software that becomes damaged. The key is how to implement the check and subsequent repair before the damage resource is encountered by the application. In a locked-down scenario, a record of acceptable system files is compared against those on users' computers. Using a bit-pattern detection scheme, the files are compared on a case-by-case basis. When an invalid file is encountered, its acceptable equivalent is pulled from a central repository.

With Windows 2000 and XP, the operating system itself protects system files through Windows File Protection. Whenever the operating system detects an attempt to replace any of these special system files, a corresponding restore sequence is initiated to restore the system to its acceptable state.

While this process works fine for system files, it doesn't help the application binaries out too much. Here again the Windows operating system is on the forefront. Through runtime resiliency, the Windows Installer service can detect missing, corrupt, or the wrong version of key files or other installable resources. When corruption is detected, the Windows Installer service restores the appropriate component (technically this component may entail far more than an individual file; especially if repackaged, the entire application may be seen by the Windows Installer service as one big component)and associated installable resources from the originating source image.

Short of operating system intervention, there are limited avenues to incorporate true runtime software repair. In this case, repair amounts to warehousing the installable media images from which the user can initiate the repair process through the application install. This process works because the newer Windows install programs mark the system as the application is installed. On a subsequent running of the install program, maintenance mode is triggered in which case the user can elect to modify the current application status, repair the application by reinstalling the application files and configuring the system, or to simply remove the application.

The process has become fairly standard on the Windows operating system. A typical sample of the maintenance mode detection is shown in Figure 2.8.


InstallShield Professional - Windows Installer Edition - InstallShield Wizard

Program Maintenance

Modify, repair, or remove the program.

[ ] Modify

Change which program features are installed. This option displays the Custom Selection dialog in which you can change the way features are installed.

[✓ Repair]

Repair installation errors in the program. This option fixes missing or corrupt files, shortcuts, and registry entries.

[ ] Remove

Remove InstallShield Professional - Windows Installer Edition from your computer.

Figure 2.8: Sample maintenance mode dialog box with Repair option selected.

License Conformance

Just because we've created an implementation process for our software, doesn't mean we can freely distribute it. After all, there may be a fixed number of licenses allotted and all are assigned. In this case, either software must be removed to make a license available or additional licenses must be purchased.

A key aspect to support in the software-deployment process is to ensure that the company maintains license conformance. As soon as all licenses are consumed, an alert should be triggered to investigate the software's status.

In some cases, license metering may be in affect. For this unusual situation, a fixed number of applications can be in use at a specific time. As soon as that limit is reached, any other instances of the software can't be run. Usually, the user will be asked if he or she wants to wait for a license to become available or to try again later. A job within the support team may be to automate the availability of these licenses.

The type of software license has a lot to do with how much flexibility is available. Some software is licensed per user. Thus, the software can be installed any number of times; however, only the assigned user can use the software. Other software is licensed per computer. This licensing scheme means that the software can be installed on only one computer—no matter who is actually using it. Another form of software license is a site-license. This license method is the most flexible arrangement and allows any number of installations as long as each user is within the same company.

Understanding the scope of the licensing arrangement is imperative to staying within conformance. It may also provide latitude in distributing the software.

A few years ago, I ran into a global company that circulated its licenses from time zone to time zone. When the employees in New York were done for the day, they would uninstall the software application. At the beginning of the day, those in the Belgium office would install the software—thereby consuming the licenses made available by the uninstalled software in New York. This process would continue around the globe, eventually leading back to the New York office. Keep in mind that this practice may be expressly forbidden by some software licenses.

System Recovery

Although system recovery isn't typically part of software deployment, it's worth mentioning here. Because the support mechanism already houses the distribution images, taking on the role of system recovery makes sense.

System recovery is slightly different than software repair because system recovery is the singular case of catastrophic software repair. In this case, the operating system as well as the computer applications has become inaccessible.

Ideally, system recovery is accomplished by storing a base image of the computer. Here again standardization becomes an important component. If there were a fixed (and hopefully small) number of standard systems, the system-recovery mechanism would have to warehouse a few images.

Based on the base computer image and the group that the user is a member of, all software (including the operating system) could be distributed through the standard software distribution channel. Excluding hardware failure, this process would quickly restore the computer to a working state. In conjunction with regular system backups, a minimal amount (if any) of user data would be lost.

Decommissioning Software

In much the same way that the software can be pushed onto a computer, it can also be removed. This removal is often done when upgrading existing software or simply removing software that is no longer useful. Software could also be removed because a specific licensing scheme has expired. In this case, instead of remotely pushing the software, a remote uninstall is performed.

As I previously mentioned, potentially, trouble can arise when removing software. Shared components or software that relies on the software being removed are common reasons not to remove software—or at least to do so very carefully.

Summary

Through this chapter, we've defined the fundamental basis of software deployment. With this information, you should feel comfortable starting a deployment effort. Perhaps you've even managed to put together a list of questions to present to the management and the deployment teams. Keep in mind that a thorough evaluation of the software and the distribution process goes a long way to ensuring an effective deployment strategy.

Copyright statement