realtimepublishers.com™

*The Definitive Guide™ To*

# Windows Software Deployment

Making Complex Technology SIMPLE Since 1985

*Leslie Easter*

Chapter 1: The Foundation of Software Deployment ........ 1

Getting to the Point ........ 1
What is Software? ........ 2
Software Is a Collection of Software ........ 3
Software Connects to Software ........ 3
Linking ........ 4
Software Breaks Software ........ 6
DLL Hell ........ 7
Software Evolves ........ 9
Introducing Software Producers and Consumers ........ 9
Motivations and Responsibilities ........ 9
Software Producers ........ 10
Software Consumers ........ 10
Installing Software ........ 10
File Transfer ........ 10
Configuration ........ 11
Autoexec.bat and Config.sys Configuration ........ 11
System.ini and Win.ini Configuration ........ 12
The Advent of Windows ........ 12
Windows' GUI ........ 12
Shared Files ........ 12
The System Registry ........ 13
Unistalls ........ 13
Beyond the Install ........ 13
The Evolution of Software Deployment ........ 15
Repackaging ........ 15
The Growth of Software Deployment ........ 17
Large-scale Deployment ........ 17
Conformance Testing ........ 17
Licensing ........ 18
Distribution ........ 18
Support ........ 19
The Software-Deployment Life Cycle ........ 19
Release ........ 20
Retire ........ 20
Install ........ 20
Uninstall ........ 20
Update ........ 20
Adapt ........ 21
Activate ........ 21
De-activate ........ 21
Summary ........ 21

# Chapter 1: The Foundation of Software Deployment

Spouses of auto mechanics are the first to admit they have the least maintained cars. Often this isn't an idle complaint—it's true. What they mean more than anything else is that their car isn't maintained as well as it should be considering there's an auto mechanic in the family.

I would imagine the same holds true for plumbers, carpenters, lawn maintenance professionals, and software developers. Hah! Caught you off guard didn't I? I admit the last one may be localized to only my household, but it's an example I can share with you. Often, when someone is good at something and they do it for a living, they don't have the time to spare or the inclination to apply that talent within their own lives. Call it human nature. After banging on a computer all day, I'm not eager to jump on the computer when I'm at home.

Last year my wife and I purchased a nice computer. Given all the computer books and computer training classes available, I've been impressed with the computer abilities of an eight-year old and his six-year old brother who have no access to these resources.

With minimal guidance, they've managed to enter my email address into every conceivable online contest, download cartoon screensavers, install desktop themes (usually involving spaceships and aliens), browse the Internet, play online games, and even let me know how much gaming systems are going for on eBay.

The obvious result of all of this tweaking is that user interface design has certainly improved over the past 10 years. These boys also managed to cripple the printer (repeated paper jams) and generate inconsistent hardware errors (the recordable CD drive is no longer reliable). Not to mention all the disk space these activities consume. Amidst the grumbling, I'm trying desperately to rationalize why my own home computer is in such a mess.

Typically, none of these problems have been big enough to disrupt my weekend. After all, I do most of my work in the office. Only when the family computer comes grinding to a near halt does maintaining the home computer escalate to the "A1" (with a bullet) priority on my To Do list. The squeaky wheel gets the attention and all of that.

After a few months of listening to these complaints, I am eager to reduce the impact on my weekend. I've since created separate logon accounts for the boys, the printer became off limits for all activities except printing, downloading software became an "On Approval" only action, and the recordable CD drive was replaced with a newer model. Sure I haven't solved all of all the problems, but the ones I have fixed certainly reduced my workload and I now have more time to enjoy the new DVD player.

## Getting to the Point

What does all of this have to do with software deployment? Good question and I'm glad you've stuck with me. Like my attitude toward the home computer, anything to do with getting an application onto the user's computer has historically been an afterthought. It's the leaky brake line to our successful automobile repair shop. It's the crabgrass in our own yard to our wildly growing lawn-mowing business (sorry, I couldn't resist).

For software manufacturers, software deployment is a necessary evil to reach the ultimate goal of creating software. After all, software deployment has nothing to do with the problems the

software was originally written to solve. It's a by-product of creating the software. It's a customer's need for convenience. Ah, therein lies the rub.

Software manufacturers' lives aren't made easier by software deployment—it makes customers' lives easier. In fact, software deployment is the one aspect of software creation that is completely driven by the customer. While each revision to a software application stems from competitive market analysis, internal feature set creation, and customer feedback; software-deployment improvements are motivated by customer pressure. Typically, customers reduce the cost of deployment by pushing back on the software manufacturer. The software manufacturer in turn, puts in extra effort to make its application easy to deploy.

What then is software deployment? The answer to this excellent question conveniently serves as a great introduction to this book. A software program is first formulated as a result of a need—a potential customer need. From there, the program is pitched to a company's senior staff. If approved, the idea is sent to marketing to verify basic assumptions such as; will enough people purchase this software to make the development costs worthwhile?

If marketing finds that a need exists, the idea is sent to the product team to create a software specification. Development and quality assurance (QA) folks use this document to create and test the software. An install program is created, boxes are made, shrink-wrap is applied, and the software hits the streets. Software deployment encompasses everything the customer goes through to disseminate the software after it has been purchased. By defining what software deployment is not, we've defined what it is.

This description is important and the way it's described is also important. Software manufacturers know what the customer will do with their software; but they don't know how the customer will arrive to the state in which they can use the software. How the customer gets to that point is software deployment. Later in this chapter we'll come up with a more realistic model, but for now this is the path we're going down.

## What Is Software?

Don't groan, but an important part of discussing software deployment is making sure that our definitions are well understood. For this reason, I want to sidetrack a bit and define software. The definition we're going to derive is pretty much what you and I think software is. The only addition is we're going to define software in terms that are convenient for a software-deployment discussion.

Originally, software was monolithic. By that, I mean the executable was self-contained: It used few, if any, external resources and no executable cared whether any other executable existed.

A lot has happened over the past 20 years to change that definition: More people are using computers than ever before. More non-technical people are using the computer. The software market is more competitive than ever before.

In fact, the sheer number of people using the computer today for the bulk of their work has created a whole new economic classification: The use of the computer has increased worker productivity to the point that it impacts the nation's economy. That's simply amazing.

But this productivity wouldn't have developed as quickly if software manufacturers hadn't improved the software-creation process. In addition, we certainly couldn't have become this productive without a high-level of connectivity. These two culprits—rapid development and

higher connectivity—are the primary generators of the software-deployment issues we face today.

Rapid development is a term that spawned from a significant change in software development. Instead of a monolithic executable, software today is a collection of software. Third-party vendors contribute a great deal to the software-creation process. This collaborative effort is a strange paradox: On one hand, this effort lets software manufacturers plug in functionality that would be prohibitive to create internally. On the other hand, as a result of this collaborative effort, third-party software that should have reached the end of its life is still shipped and active because a third-party software manufacturer can retract or modify its software only through the application's creator.

Higher connectivity plays a role in software deployment by bringing a higher level of conformance from user-to-user. With a larger exchange of data, software applications share a larger amount of how data is processed. This idea ties into componentized software. Why should I reinvent how to access a specific database if I know someone's software component can do it for me?

### *Software Is a Collection of Software*

The phrase "software is a collection of software" is a euphemism for componentized software. Through published interfaces and a construction cleverly called *dynamic linking*, any library can talk to any other library. In fact, a strong argument could be made that today's software development efforts are primarily spent gluing components together rather than creating new components.

An excellent example for this argument is the Windows operating system.. With the advent of the Windows application programming interface (API) set, developers could leverage common code. This common code is provided by a core set of dynamic-link libraries (DLLs) that ship with Windows. These DLLs make Windows-centric tasks, such as re-sizing, moving, opening, and closing windows convenient. These DLLs aren't limited to graphical functionality—virtually anything done in a Windows application has to go through this core set of libraries.

At the same time Microsoft released Windows, the company was pushing new technologies and new functionality. These upgrades were also made available through a collection of libraries. Since these libraries didn't ship with the original operating system release, they were termed *redistributables*, and software vendors were allowed to ship these redistributables with their software applications.

Today, bundling redistributables with the install program of software applications is expected. Open Database Connectivity (ODBC), Microsoft Data Access Components (MDAC), Visual Basic Runtime, and Microsoft Common Controls are just some of the redistributables available today. The current list of Microsoft redistributable files numbers in the high hundreds. Third-party software vendors are also in on the act. In fact, install programs commonly include third-party redistributable software. For example, Adobe Acrobat and Macromedia Flash are two widely installed software redistributables.

### *Software Connects to Software*

The linking of a Windows application to its underlying libraries is complex. Figure 1.1 illustrates just how complicated things can get. This figure shows how libraries provided by different

vendors rely on the presence of other libraries for correct loading. A missing library generates a system error message. An incorrect version of a library can lead to subtle runtime errors. Contrast this figure to the days of DOS, when an application sat directly on top of the operating system.

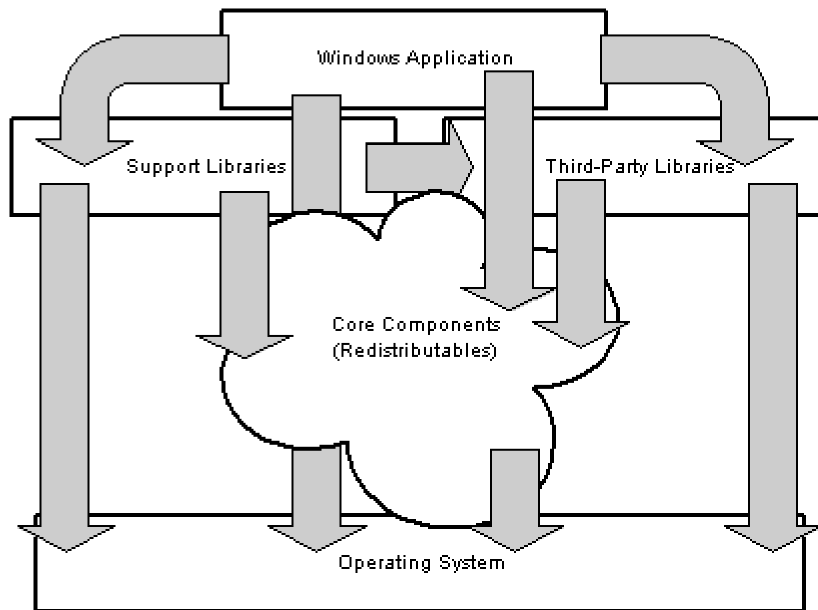



***Figure 1.1: Through library linking, applications strongly depend on the operating system and support files.***

In addition to redistributables, there are always updates to the operating system. These updates consist of hotfixes, maintenance releases, and service packs. These updates usually update the underlying Windows core DLLs.

Just to keep things relatively sane, each DLL was usually accompanied by a numerical version (for example, 4.1.23.6). This version number is usually accurate; however, editing the version resource is the responsibility of the developer. This version number is used during the update process to make sure newer files aren't overwritten. Did you catch the two *usuallys* in the previous sentences? Well, when these tasks aren't done, things get really messy.

## Linking

The software manufacturer divides the development effort into a variety of tasks. This division of labor is done not only from an architectural point of view but also for a nice division of resources. Lisa and Donna can create the computing engine, while Jason and Jesse can develop the database query system. In each case, the functionality results in multiple support libraries. The separate libraries come in handy when there's a spin-off project that also needs a computing engine or a database query system.

The arrows in Figure 1.1 illustrate the reliance the entire application has on external resources. Of course, software deployment would be fairly straightforward if we could always guarantee the correct library is available when it's needed. Unfortunately, such isn't always the case.

The first problem is that determining the complete list of libraries required by an application is an extremely difficult task. The reason this task is so difficult lies within the application itself.

An executable can use a function provided by one of these DLLs in two ways. The first is to actually embed the function inside the executable. This is called *static linking*.

Static linking is much like bringing your own bed on your vacation. Sure, it's comforting to know that it's there, but you're taking on a lot of overhead. If you know there is a bed where you're going, you could take the leap of faith and leave yours at home. Alternatively, if you apply the bed analogy to *dynamic* linking, you'll wait until you need to sleep, then you'll expect the bed to be underneath you when you lie down. Therein lies the problem—sometimes there is no bed and sometimes it's not the bed you expected (a bed of nails versus goose down).

From a software-deployment point of view, things get even a little bit trickier. Not only is there dynamic linking, but there are two types: implicit and explicit. Implicit linking means that a list of libraries an executable requires to run is located within the header of the executable. This arrangement is useful because when the executable loads into system memory, the operating system will track down the necessary libraries. Explicit linking means the application will attempt to load the library when the application needs the file.

The error that occurs as a result of a missing library depends on the type of linking. If an implicitly linked library is missing, the error will be immediately obvious because the operating system won't allow the application to launch. I imagine you've seen the error before, but I've provided Figure 1.2 to jog your memory.

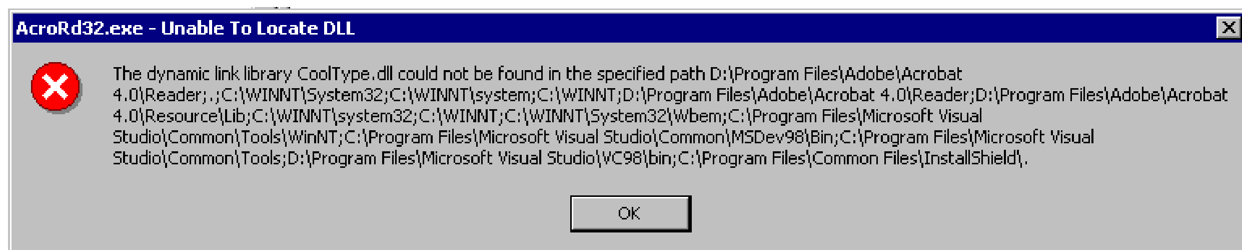



***Figure 1.2: The infamous Unable to Locate DLL error message.***

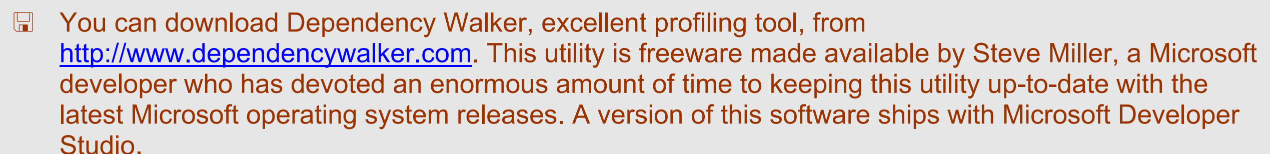
There are very good reasons to statically link versus dynamically link to a library. While static linking guarantees the correct API call for the exact version—after all it's embedded within the executable, dynamic linking reduces the physical size of the executable. A key reason for componentized software, and thus dynamic linking, is so software applications can benefit from updated components. For software deployment, understanding the tradeoffs of static and dynamic linking is important.

There are pros and cons to implicit and explicit links. For example, I'm using Microsoft Word to write this chapter. On a whim, I loaded Word into a profiling utility. Because almost all Windows applications dynamically link at some level, I was curious about the extent of implicit versus explicit linking. Table 1.1 shows the number of implicitly linked versus explicitly linked DLLs.

| Implicitly Linked | Explicitly Linked | |
|---|---|---|
| MSPANDB.DLL | AGENTMPX.DLL | NTLANMAN.DLL |
| KERNEL32.DLL | ATL.DLL | NTMARTA.DLL |
| ADVAPI32.DLL | BLNMGRPS.DLL | NTSHRUI.DLL |
| COMCTL32.DLL | CLBCATQ.DLL | OBALLOON.DLL |
| GDI32.DLL | CSCDLL.DLL | OLEAUT32.DLL |
| MSO9.DLL | CSCUI.DLL | RICHED20.DLL |
| NTDLL.DLL | DNSAPI.DLL | SAMLIB.DLL |
| OLE32.DLL | LINKINFO.DLL | SECUR32.DLL |
| RPCRT4.DLL | LXATSTRN.DLL | SYNTPFCS.DLL |
| SHELL32.DLL | LXATUI.DLL | VERSION.DLL |
| SHLWAPI.DLL | LZ32.DLL | WDMAUD.DRV |
| USER32.DLL | MPR.DLL | WINMM.DLL |
| WINSPOOL.DRV | MSGR2EN.DLL | WLDAP32.DLL |
| WINWORD.EXE | MSI.DLL | WS2_32.DLL |
| ACTIVEDS.DLL | MSSPELL3.DLL | WS2HELP.DLL |
| ADSLDPC.DLL | MSVCRT.DLL | WSOCK32.DLL |
| CRYPT32.DLL | NETAPI32.DLL | MSO9INTL.DLL |
| MSASN1.DLL | NETRAP.DLL | WW9INTL.DLL |
| MSLS31.DLL | NETUI0.DLL | |
| OLEPRO32.DLL | NETUI1.DLL | |
| USP10.DLL | NTDSAPI.DLL | |
| W32TOPL.DLL | | |

***Table 1.1: Implicitly linked versus explicitly linked libraries.***

If you've followed my discussion to this point, you know that the implicit linked entries are complete; however, since I didn't test every bit of Word's capability, the explicitly linked column is probably incomplete. In this example, there are about twice as many explicit as implicit linked libraries. I'll provide more information about implicit versus explicit files in later chapters.


You can download Dependency Walker, excellent profiling tool, from http://www.dependencywalker.com. This utility is freeware made available by Steve Miller, a Microsoft developer who has devoted an enormous amount of time to keeping this utility up-to-date with the latest Microsoft operating system releases. A version of this software ships with Microsoft Developer Studio.

### *Software Breaks Software*

Microsoft has always stored its core libraries within the system folder under the primary Windows folder. Microsoft has attempted to maintain this folder as a repository for the operating system's files; however, nothing prevents software manufacturers from using this folder. In fact, manufacturers place their applications' shared files into this folder because the operating system automatically searches the Windows and Windows system folders when an application needs a library.

This functionality has added responsibility for an application's install process. In addition to file transfer, the install program must configure the application to sit on the operating system. I compare this responsibility to a matchmaker. The install program must take two unacquainted entities and introduce them in such a way that they take an instant liking to one another. As in real life, things can get complicated.

The operating system goes through a well-known search order t to find all the libraries needed by an application. Let's say my application starts from a unique location. It makes a request for a Microsoft Common Control. In this case, let's say the library is ComCtl32.dll. This file usually resides in the Windows system folder and contains some common Windows controls, such as the spin box control and IP address control. (I've selected this control because of its widespread use in Windows applications.) In response to my application's request for ComCtl32.dll, ideally, the operating system finds this file within the Windows system folder, loads the library, and it's ready for my use. During this process, a couple of things can go wrong. First, what if the library isn't in the Windows system folder? The operating system then peruses the rest of the search order. If it's not found through this process, a system dialog box will be displayed saying the system file could not be found.

Second, what if the operating system finds the file, but it's not the right version of the file? There are a couple of options at this point. The file may work just fine if the functionality my application requires is contained in the library. If it's not, any number of errors could occur depending on what is in the library and what my application intended to do with the library.

The biggest question is how could the library be the wrong version? Let's assume a best-case scenario: The install program made sure the correct version of the file resided on the system when it installed the application. Then, another install program came along and overwrote the existing ComCtl32.dll with an older version. My application is definitely broken.

But there are other ways things could go wrong. What if the existing ComCtl32.dll was overwritten with a newer one that wasn't completely backward compatible or wasn't compatible with the functionality my application needed? My application is still definitely broken. What if a second ComCtl32.dll was installed in a different location, but closer in the system search path? Same thing. The wrong version would be loaded before the one my application needs and my application would be broken.

### DLL Hell

As you can see, there are a number of ways these shared DLLs can be manipulated to create havoc for Windows applications, thus the term *DLL Hell.* A classic example of DLL Hell goes back a few years and involved a file named WinSock.dll. This file, while a standard name, was implemented in drastically different ways by TCP/IP stack vendors. (This example takes place in the days before the WinSock.dll came bundled with the operating system.) For an install developer, a physical search of the hard disk was required before a new stack could be laid down. Any chance that a different WinSock.dll could be found had to be removed. Extraneous WinSock.dlls had to be renamed to avoid a collision. These requirements prevented problems from the install side, but nothing prevented another install program from laying down a different and incompatible library.

DLL hell is the primary reason that locked-down systems exist. In fact, large corporations commonly have rather draconian processes in place to keep the operating system shared libraries at a consistent version. These processes range from limiting what the end user can install to physically relaying the Windows system folder when the computer boots.

There are trade-offs with this approach. The more locked-down or standardized the computer, the less flexible it becomes for the user. Then again, the more flexible the computer is for the user, the more likely that version conflicts will occur. It costs money when things break. How much a company is willing to spend is a function of the corporation's motivations. Figure 1.3 graphs these tradeoffs.

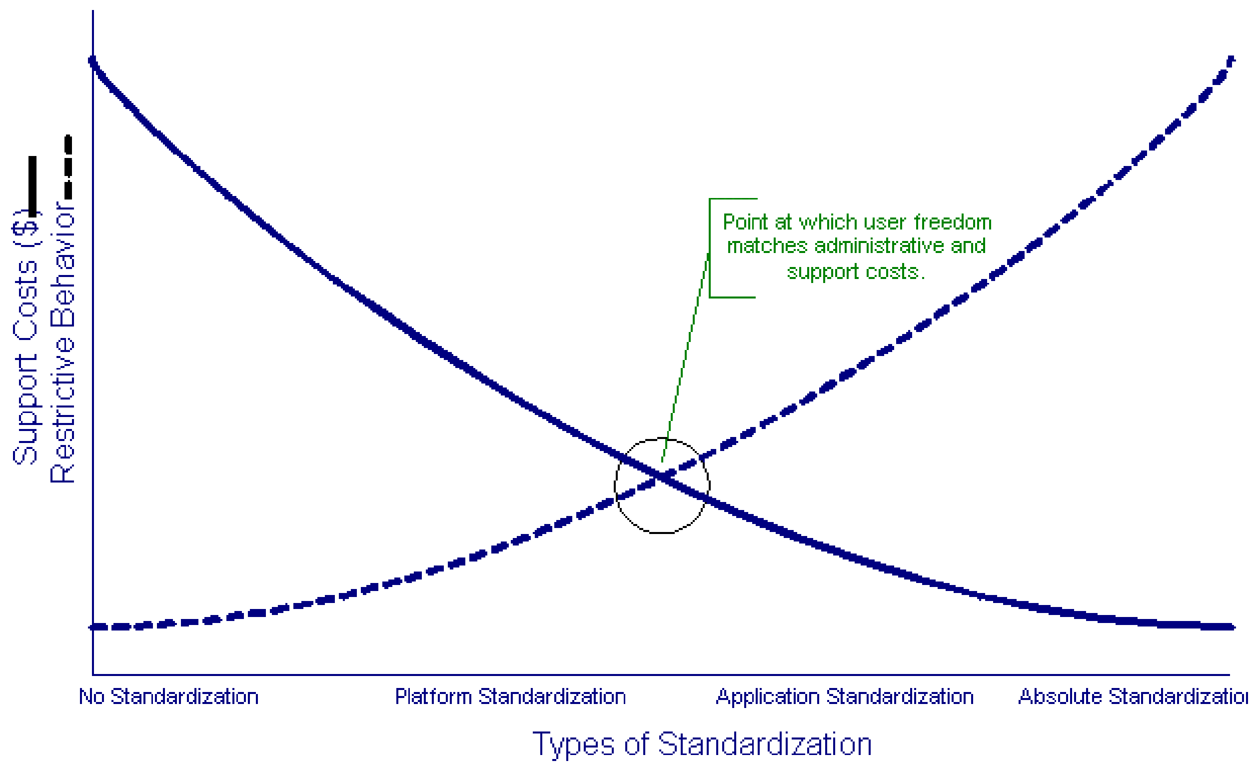



***Figure 1.3: The tradeoffs between restrictive behavior and freedom to modify.***

At some point, a happy medium exists. Most corporations recognize that the computer user requires some latitude. This balance is different for each environment, so the tradeoffs aren't always the same. In other words, the point at which user freedom matches administrative and support costs moves horizontally in this figure.

As a result of DLL Hell and the behaviors of some rogue install packages, a gradual state of system decay occurs after the first application has been installed. In today's world, computers will eventually lose functionality without careful monitoring. This decline is primarily a result of the difficulty in tracking and resolving conflict libraries.

### *Software Evolves*

Microsoft provides two mechanisms to help reduce the impact of DLL hell: Windows File Protection (WFP) and side-by-side sharing of libraries. Both features are completely functional in Windows 2000 (Win2K) or later.

WFP ensures that core operating system files aren't inadvertently overwritten. This functionality is accomplished through the operating system. WFP monitors a select group of files in the Windows system folder. If any of the prescribed files are overwritten, they're replaced with the originals on the next reboot. This feature requires a DLL cache location where the original files are stored.

Side-by-side sharing allows applications to use a specific version of a file even if a different version resides in the Windows system folder. This functionality is accomplished by installing the application-specific version of the file locally with the application. A data file indicates that the local copy of the system file should be used.

## Introducing Software Producers and Consumers

Up to this point, I've been calling the people who make software the software manufacturers. This terminology is a bit cumbersome and not entirely accurate because not everyone who provides software is the creator of the software. Along those same lines, "manufacturer" has some connotations that implicate more niche responsibility. By going up a level, and using the term producer we'll certainly capture all guilty parties.

> ☞ A software producer governs the creation of the core software system.

The same holds true for the computer user. This term is a bit too specific to be useful in many of our discussions. In some cases, it implies the user directly purchased the software or directly deployed the software to the computer. By using the term "consumer," we can capture the responsible audience.

> ☞ A software consumer governs the purchase and deployment of the software.

### *Motivations and Responsibilities*

In one of my favorite movies, the hero is being chased across a barren planet by a rock monster. The hero is in communication with the ship and asks for help to outwit the monster. The calm response, "What's his (the monster's) motivation?" Of course, it's funnier in the context of the movie, but the question is useful to this discussion.

If we understand the motivations of the software producers and consumers, we'll come closer to not only understanding software deployment but also the problems that litter the current software-deployment landscape. After we truly understand the problems, we can begin to look for solutions.

### Software Producers

Let's begin with the software producers. They've got to ship software while schedules and resources are tight. If the software doesn't ship soon, the window of opportunity is lost and market share shrinks. Between product requirements and feature creep (the continuous addition of last minute "must have" functionality), just getting the software built, let alone tested, is an almost insurmountable task.

This daunting task is further complicated by a reliance on third-party components. Will the latest release of their software be compatible? Will they ever fix that one last outstanding bug? Were they being completely honest with respect to bandwidth limitations? Has the legal department finally signed that agreement? These are just a few of the questions that software producers deal with on the race to release.

After the software is released, the motivation of the software producer turns to its true goal—making money. One way to not make money is to release software that generates a lot of support calls. In fact, the ideal software release involves many sales calls, no support calls, and a lot of happy customers. The closer a software release gets to this ideal, the more profitable it will become. Just producing the software isn't sufficient for the software producer to be successful. A successful software producer must be motivated to lower the cost of supporting their software.

### Software Consumers

The software consumer wants software that is easy to deploy and requires little maintenance overtime. In the eyes of the software consumer, keeping the software up to date through maintenance releases, hotfixes, or some other producer-initiated update process is the producer's responsibility.

The consumer is well aware of the cost of owning software, which isn't simply the initial purchase cost. This expense consists of all the additional costs associated with owning the software, including the cost to invest in better hardware to run the software, the cost associated with the loss of productivity when the software doesn't work according to expectations, the expense of additional training required on non-intuitive software, and the cost of negotiating license or site agreements.

## Installing Software

Crucial to a software producer's success at lowering support costs is the ease of the software's install process. This point is an important aspect of software deployment. After all, the early models of software deployment were predominately installing the software.

A deployable software package contains the physical application files and the embedded knowledge to not only transfer those files to the user's computer but also configure the computer such that the application is usable. Ideally, the file transfer and configuration is done in such a manner that all other applications still work.

### *File Transfer*

File transfer is the cornerstone of software deployment. If the deployment process can't handle getting the application binaries from the media to the user's computer, all hope is lost.

In the early to mid 1980s, most applications were typically on the order of a handful of diskettes. As applications grew larger, the size of physical media also increased. Early versions of Microsoft C shipped on less than five diskettes. Later versions (in the early 1990s) took 24 diskettes. Unfortunately this number of diskettes fell within the disk-failure rate and generated a substantial number of support calls related to failed media. Don't forget that phone support was free at that time.

Compression and joining algorithms were the hot commodities. Typically the application resided in a single folder structure. Compressing this folder and splitting the compressed file across the diskette media was the media-creation process. The application was extracted into a single folder on the user's computer. From almost the beginning, some file-integrity checking was performed.

All reasonable efforts were taken to reduce the number of diskettes. After all, shipping on one less diskette lowered the overhead and provided one less reason for media failure. In those days, the install program was little more than decompressing and joining software.

Often the ReadMe.txt file that accompanied the first diskette gave directions about any additional configuration. Usually, this configuration process involved a manual edit of the autoexec.bat and config.sys files. The end user would reboot the system manually.

During this period, large-scale distribution entailed little more than placing the image on the network. End users launched the image from the network to install locally. For smaller distributions, the system administrator had to visit each computer and install the software directly, especially if the install required any changes to system files.

The file-transfer process today has experienced little change. Most install packages compress the application files into a single image. Although this image can be split across physical media, this process isn't often used because the space required by the distribution image is well within the space available on a CD-ROM. In addition, compression algorithms are more efficient and provide better file-integrity checks.

File transfer usually encompasses removal of the application. In the pre-Windows days, uninstalls were virtually non-existent. Only within the past 5 years has removal of application files become commonplace and within the past several years that a true application uninstall has been around.

### *Configuration*

Configuration is the other half of the application-install process. After all, we've long ago left the days when simply transferring the files to the user's system was the only action necessary for the software to be usable.

#### Autoexec.bat and Config.sys Configuration

When the install software had the smarts to edit files directly, autoexec.bat and config.sys were the first files to benefit. The first several releases of Windows didn't impact the install process substantially because most applications at the time were DOS based. The ability to edit these files gave the install program more responsibility over the install process. As Windows grew in popularity, file editing became a requirement.

And with this added responsibility of file editing came opportunities to fail. Because the operating system provided little more than reference documentation about the install process, software manufacturers were left up to their own devices when the time came to create the install program. Although there were a few dominant install tool providers, the install programs themselves ran the range from excellent to downright dangerous. For example, the PATH environment variable commonly exceeded its allotted length or for system drivers to be replaced without the user's knowledge. Discovering these shortcomings was all part of the learning curve for install developers.

### System.ini and Win.ini Configuration

Like modifying autoexec.bat and config.sys, editing the initialization files for the Windows layer was commonplace in the Windows 3.x days. Although editing of config.sys and autoexec.bat required a system reboot, editing Windows initialization files required a restart of Windows itself. Today it's hard to remember back when Windows was distinct from the operating system.

Today these files are no longer used in the operating system. As the Windows layer permeated the operating system, the system registry replaced these files.

## *The Advent of Windows*

With the discussion we've had so far, you can see the distinct line between pre-Windows and post-Windows software deployment. That line occurs with the introduction of a graphical user interface (GUI), which in turn coincides with the widespread acceptance of Windows 3.1x. The Windows' GUI has driven the current software-deployment trends, so I'll delve into this area a bit deeper.

### Windows' GUI

The large-scale acceptance of Windows 3.1 was the turning point for software deployment. Windows 3.1 made the computer easier to use for non-DOS users. The graphical interface was made easier to navigate with the addition of the mouse.

The Windows interface opened new possibilities for developers. File extensions could be linked to applications. Double-clicking on a file could launch an associated application. Drag-and-drop functionality supported embedding of application objects. Graphics could be added directly into documents. What you see is what you get (WYSIWYG) became the buzzwords of software.

The graphical nature of Windows drove the graphical interface of the install process. This fact may seem trivial at first, but it opened doors to an entirely new user: the novice. The novice user rapidly accepted working on a Windows platform using a Windows application. The graphical nature was typically non-threatening. Designed correctly, the software install program provided safe default choices. Configuration changes, which historically had been left up to the systems administrator, were now handled automatically.

### Shared Files

All those shared components that a software application requires are lumped under the generic title of shared files. Not only do these files have to be updated based on version number, but (on Windows) the installation of these files requires an update to an internal shared DLLs count.

The concept of shared files and the internal counter is a construct that first appeared in Windows 95. This functionality was an important step in maintaining system state. Previously, shared files were always left on the user's computer.

If you've ever seen the Windows system folder of a Windows 3.1 system, you know what I'm getting at. Literally hundreds of files were left orphaned on the computer with absolutely no way to determine whether they were actually still in use. Adding to this frustration was the fact that disk space was a fairly expensive commodity.

While the Shared DLLs count is a big step forward, it's not a complete solution. Because the install process updates this counter, the count can easily become inaccurate. In the worst-case scenario, a shared file could be removed prematurely.

### The System Registry

The system registry truly came to life in Windows NT. Although a variation existed in Windows 3.1, it didn't exhibit the depth of system reliance that NT and later versions of Windows 9x exhibited. The system registry is the repository for hardware and software configurations—for individual user accounts and for the computer itself.

The vast amount of energy in deployment is expended on updating the system registry, especially for computers with multiple user accounts. The registry maintains a profile for each local account, so any software that requires initialization of specific user profiles has to be installed multiple times—once for each logged-in user account. This requirement is true even if the files go to the same physical location. The reason is the user's registry data needs to be instantiated.

Current trends in application design attempt to make this process easier by relying on the software application to populate the user registry data on first launch. In this case, not only are all user accounts taken into consideration, but all future users are as well.

The system registry is centralized storage for ActiveX controls. Each ActiveX control creates a series of CLSID registry entries during a process called self-registration. By invoking self-registration, the ActiveX server registers each of its controls. Amongst these registry entries is the location of the ActiveX library or executable.

### Uninstalls

As Windows grew in acceptance, users developed some demands of their own. Key among them was the ability to uninstall an application. As a result, uninstalls gradually became commonplace (granted, it took 5 years). They even became quite good; however, it took some work. Completely uninstalling an application from a computer is not an easy task. It's not simply a matter of removing the application and it's configuration—it's a matter of returning the computer to a known working state without the application or its configuration present.

## Beyond the Install

Not surprisingly, the install process alone doesn't completely define software deployment. To be truly representative, software deployment takes a bigger view. Let's take a stab at a preliminary definition of software deployment. After the discussion so far, I'd go with something like the

following: Software deployment is the integration, installation, and support of software in a known environment. This definition includes how the media is shipped, the mechanism used to distribute the image, the conformance testing that occurs before large-scale distribution, the process by which adequate license counts are maintained, the decision of who gets the application and which parts, how the product will be supported internally, and so on.

The audience is a strong function of software deployment. Clearly, deploying software in a large corporation is different than in a small company. Technology is also a function of software deployment. A company that uses a software-management system is better suited for deploying software than one that doesn't. A company that can support the latest deployment features is one step ahead of one that can't.

Software deployment relies heavily on experience. Success relies on not repeating mistakes made by others as well as knowledge of the operating system. Success in software deployment requires an understanding of how Windows applications interact.

Figure 1.4 illustrates the role software deployment takes in the software-creation process. The user assumes the process and expense of software deployment. However, software manufacturers can make this process less painful.

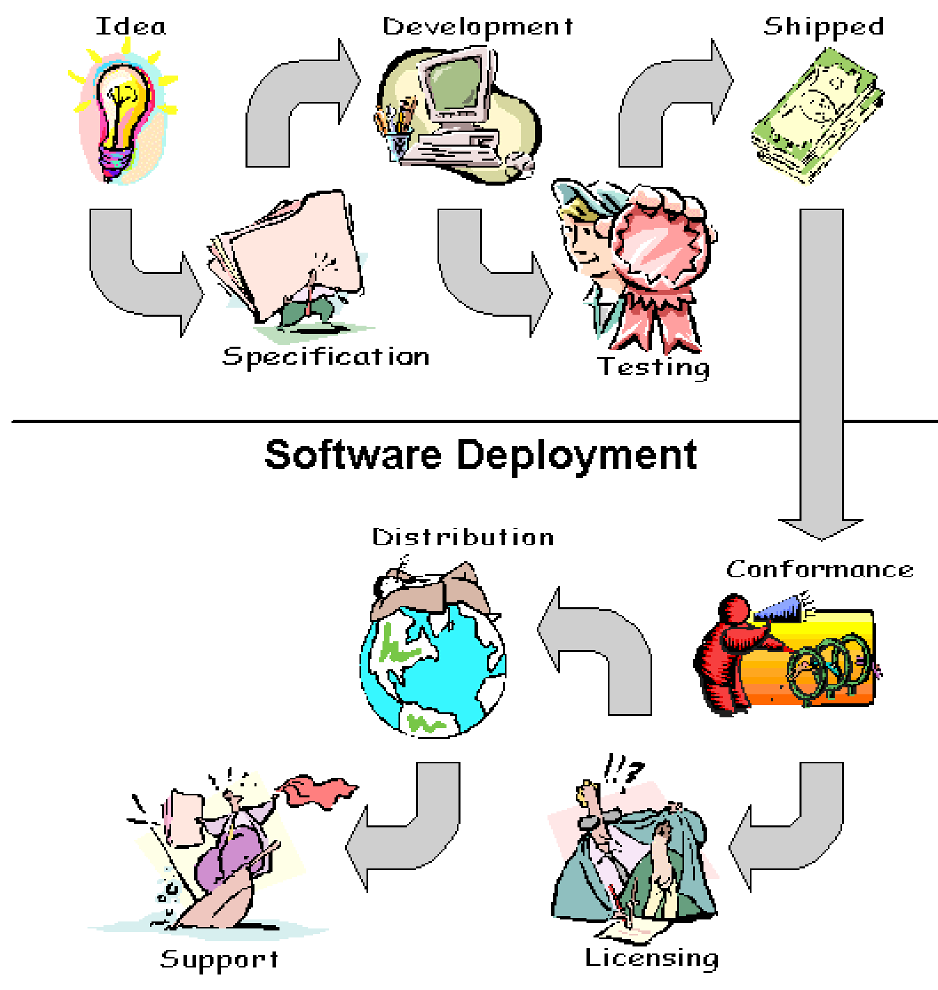



***Figure 1.4: Portion of development cycle that encompasses software deployment.***

### The Evolution of Software Deployment

As this figures shows, software deployment is a big area. But this hasn't always been the case. Given the Windows timeline, software deployment for the bulk of Windows history has been concerned with simply installing the software. Only over the past few years has the area of software deployment matured. Originally, getting your software to work on a user's system was the biggest goal. After you got that to work, the goal became getting your software to work with existing applications. Once we were happy with that process, the issue became removing your software. And of course that was followed with removing your software without destroying anyone else's software. The rest of the software-deployment areas were picked up along the way.

Initially, software deployment was little more than getting a single executable onto the computer. Those were the early days of DOS, and most users were either technically savvy or had direct access to systems administrators. There was no GUI to the operating system, and applications were launched from a DOS prompt. The install process was fairly straightforward. It involved decompressing files from a removable media to the user's hard disk. To put you into the right mindset, these were the days of 5.25-inch floppies (300Kb and 700Kb!). Any configuration was handled through a ReadMe.txt file. Most users were technically knowledgeable to make changes to autoexec.bat and their config.sys files. For the most part, that's all that was required of software deployment.

A DOS-based delivery mechanism was more or less predominant through the late 1980s. The advantage was its simplicity. The primary disadvantage was its limited functionality—there aren't many things you can do from within a batch file. In addition, a DOS batch file didn't lend itself to mass distribution.

Within the past 5 years, software deployment has made great strides. Today's large-scale distributions involve thousands of computers—all administrated remotely. Management software can remotely deliver virtually any standard software package to a client. Remote administration is a foregone conclusion.

## Repackaging

An excellent example of the importance of software deployment is the growth of repackaging. For about every three to five install programs written from scratch, there's a corporation repackaging of the same software. Why? Because the company's software-deployment requirements demand it, and repackaging someone else's software is cheaper then using the existing install program. There are strong advantages to repackaging software: additional software can be bundled, components can be removed, the user interface can be suppressed, and file conflicts can be tuned to the client's environment. By repackaging the software, these users can exert fine control over the deployment process. In some cases, repackaging leads to better integration with their software-management tool.

The evolution of software development closely mimics my situation at home with the computer. Initially, software was thrown to the end user with very little forethought. As the Windows operating system matured and penetrated more of the corporate environment, how software was installed, accounted for, and maintained became a concern for the software companies.

Those within the software industry know all too well that the critical gate to shipping software is the install program. Typically, it's the last on the list and sits on the back burner until shortly before the ship date. Seasoned developers have been known to run when the topic of the install

program is broached. To be honest, this reaction is understandable. There's not much glamour in writing an install program. If you're successful and the install program is perfect, you won't get any kudos. After all, that's what it's supposed to do. However, if the install program is buggy or exceeds the allotted time to complete, the world crumbles, schedules slip, and project managers camp outside your office.

There is a distinct lack of enthusiasm when it comes to writing the install program. This lack of eagerness may be due to the high profile that comes with taking on the task. In moderately sized project teams, most software developers are assigned key areas of responsibility. Most have limited impact on the user interface of the application. If bugs are generated, testing can continue around the affected area. Rarely is the entire application crippled. The install program, however, is critical. One bug can cause the software not to install, and a bad bug can cause the user's system to be corrupted. Given those scenarios, which end of the software do you want to work on?

Figure 1.5 illustrates the explosion of software-deployment functionality within the past decade. For the bulk of the 1980s, a simple file transfer would do the trick. Within the past 10 years, customers have been much more demanding about how software gets onto their computers and what happens while the software is on their computers.

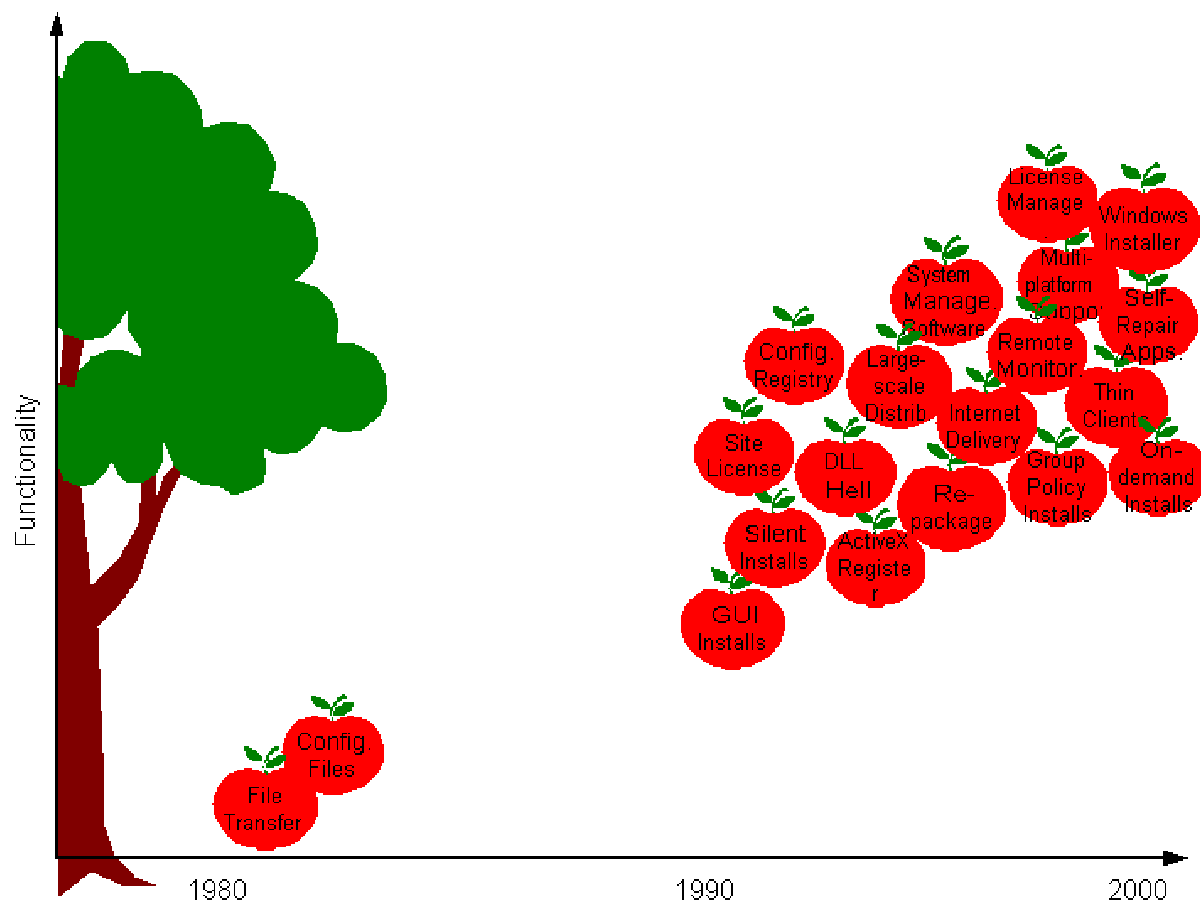



***Figure 1.5: A timeline of software deployment's evolution.***

### *The Growth of Software Deployment*

Software deployment started out as little more than install development. As I previously mentioned, the install-development process was primarily compressing the application, splitting it across physical media, and transferring the files correctly to the user's computer.

As the operating system grew and customers demanded more, the install-development process matured. As the number of Windows computers grew within corporate environments, distributing software internally added more requirements to the install process.

This trend continues today. The advent of system-management software to oversee the distribution process has saved corporations money.

### *Large-scale Deployment*

The entire discussion to this point culminates in large-scale deployment. Starting from such humble beginnings as DOS batch files, through the GUI, including the bells and whistles of Windows 9x and NT configuration, we arrive at the landscape of today—large-scale deployment.

The true test of software support for software deployment is large-scale deployment. Rolling out software to thousands of client computers is not a trivial task. Sure, clicking Next through a single install seems easy enough, but no one wants to repeat an attended install for thousands of computers—even a handful can be a test in persistence.

Beyond the annoying aspects of attended installations, systems administrators are concerned with what the install process will do to the existing and working computers. Is the install process going to break the stuff that is already working? Is the software application going to require a system update? Is the system update going to break working software?

For a large number of installations, systems administrators want to be able to roll out the software in a way that minimally impacts their day. They would prefer an install package that supports an unattended mode, can be remotely installed, and works with their system-management software.

Software deployment has come a long way since its DOS roots. The fact that most software applications can be deployed through system-management software is testament to this evolution.

The process of deploying to a large number of computers is separated into four tasks:

1. Conformance Testing—Verifying the software application will play nicely with existing software.
2. Licensing—Ensuring there are sufficient licenses for the expected user base.
3. Distribution—Getting the application out to all applicable users.
4. Support—Dealing with the inevitable.

## Conformance Testing

Left to their own devices, new applications will interfere with existing applications. A process to eliminate, or at least substantially reduce, this interference is required in a productive environment. This process costs money. In fact, there are cases in which the cost of implementing a software release is prohibitively expensive.

When productivity gains exceed the costs associated with implementing the software, the software consumer is seeing a positive Return on Investment (ROI). Consumers need to be assured of a positive ROI before updating to new software is even considered. A large part of that assurance is conformance testing in which the new software is tested in small rollouts to detect incompatibilities. An investigation into incompatibilities determines how the software will be deployed. The likelihood for incompatibilities is largest when shared files are encountered. In many cases, the original install delivery method isn't the one used internally. There are a couple of reasons for performing conformance testing, but the main reason is to adapt the existing corporate computer's environment to accept the new software.

This adaptation process is primarily spent updating libraries or configuring the system to reduce the amount of file sharing. Deriving the conformance testing is the most time-consuming task. Commonly, conformance testing of one version of a software package wraps up just as the next release hits the streets.

### Licensing

A large concern for software producers is ensuring each copy of their software is legitimate. Anti-piracy measures are easier to enforce within corporate environments than for individual consumers. Current deployment schemes usually include some method of accounting for license schemes.

Licensing implementations vary from the restrictive to liberal. A restrictive licensing scheme usually involves some method of verification before the application is usable. This verification might be as simple as a hardware-software handshake along the lines of parallel port dongles to client-server verification.

An example of a liberal licensing scheme is a site license. In this model, a software producer grants unlimited use of an application within a corporation.

Of course, there are licensing schemes that fall between these two extremes. Rolling licenses allow a fixed number of software users at one time. This implementation requires metering software to maintain a user count. When the number of users matches the license count, additional users are restricted.

### Distribution

The true test of software deployment is distribution. After conformance and licensing issues have been worked out, the last big hurdle to jump is getting the software onto the user's computer. Typically, this process involves an investment of resources to design and repackage the software application.

Repackaging is done for a number of reasons:

- Provide a silent or unattended install method
- Move to a supported distribution scheme (for example, IntelliMirror)
- Remove conflicting shared files or resources
- Add or remove optional parts of the software application

- Bundle other software packages or augment the software application with internal components

### Support

Support in the case of software deployment is largely concerned with implementing updates to existing software. Certainly software training is part of the investment that a corporation makes when rolling out new software, but it's not strictly part of the software-deployment cycle.

## The Software-Deployment Life Cycle

Up to this point, we've seen that software exists on the user's computer almost as a living organism. There is growth and there is reduction. Adding software and removing software changes the environment for existing software.

Figure 1.6 illustrates the possible paths of software deployment. We'll be using this life cycle flow throughout the remainder of this book. It's important to understand that true software deployment must embrace each aspect of the life cycle.

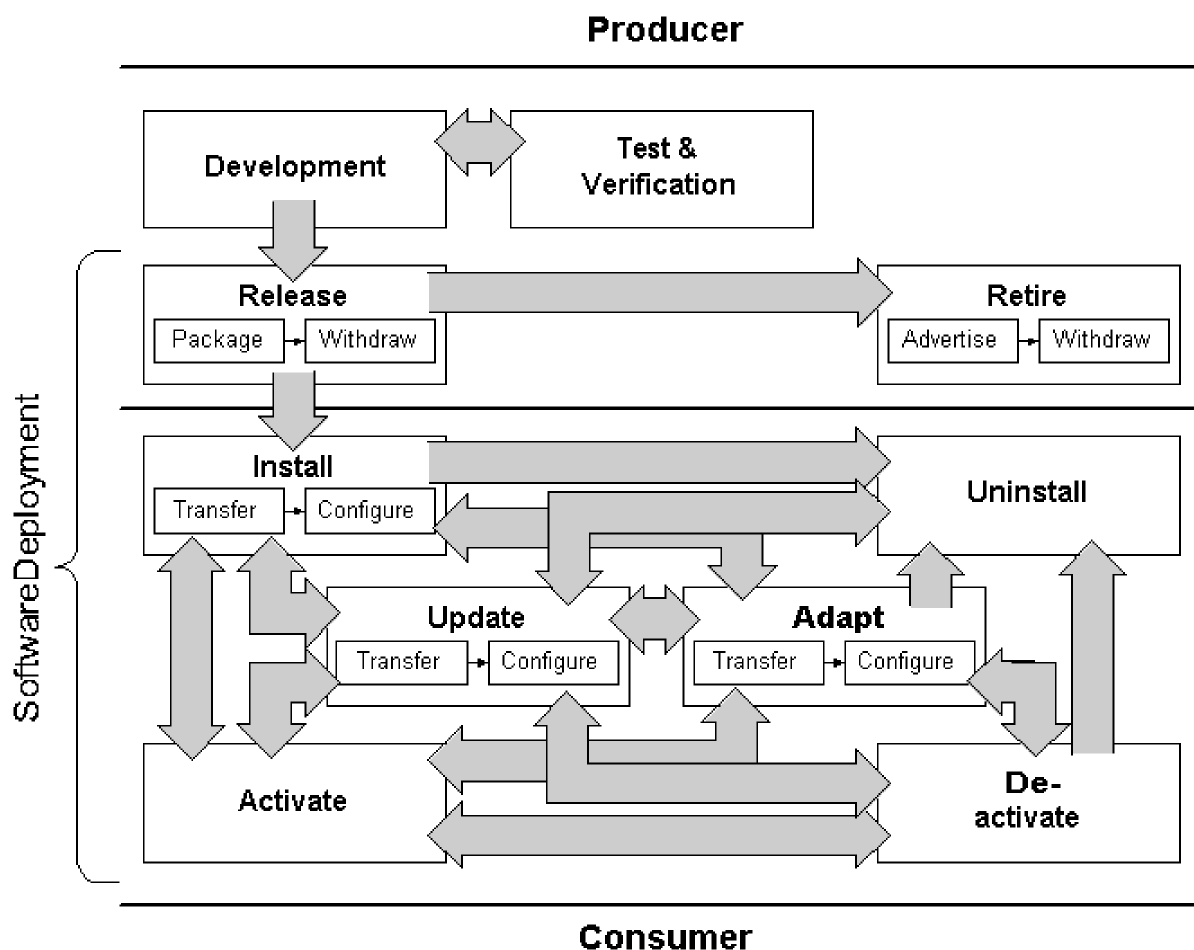



*Figure 1.6: The complete software-deployment life cycle.*

As Figure 1.6 shows, software deployment isn't just installing software. It's the transactional relationship between different software states.

### Release

As the figure illustrates, the release stage is a producer concern. It's also the connection between software development and software deployment. The successfully released package embodies not only the requisite application files but also the embedded knowledge of the install and configuration process.

A release occurs for any change in the original package. Changes include maintenance releases, hotfixes, and any other change from the original software state.

### Retire

Much like release, retiring software is a producer concern. When software is retired, it's no longer available for further deployment. More than likely, it's still maintained at many locations, but the producer would ideally like to stop it from propagating further.

### Install

We've talked about the install process, but now we can see its role in the software-deployment life cycle. The definition of the install process remains the same as earlier. It embodies the knowledge of instantiating the software application on the computer. Part of this knowledge is evaluation. The other part is configuration. An interesting way to think of the install process is as an upgrade when the application doesn't exist.

### Uninstall

When the software is no longer needed, uninstall provides the mechanism for removing the software from the system. In our model, we'll assume that uninstall is a complete and calculated task. After uninstalling the software, the remaining applications must retain their current state. Uninstalling should remove the application files and any resources that are no longer shared.

Along these lines, an effective uninstall examines the current state of software dependencies and constraints and removes the targeted software in such a way as to not violate the state of dependencies and constraints. In our model, an install isn't a simple matter of undoing everything that was done during the installation phase. Uninstall may first require a de-activate process.

### Update

The update process modifies software that has been previously installed. By its nature, an update isn't as complex as the install—it involves a subset of the original install process.

In our model, the complete update requires a de-activate process, then the update occurs, and finally the application is re-activated. This set of steps may not be required and, depending on the software, may even be a hindrance.

### *Adapt*

A crucial process to software deployment is the adapt process. During this process the system undergoes a new configuration by modifying the current state of one or more software applications.

Distinguishing an adapt from an update is important. An update modifies a specific software application, whereas, an adapt process modifies any number of existing applications. Normally, an adapt is a corrective measure.

### *Activate*

The activate process is the connection between the dormant and active application. It's the mechanism used to indicate the application's functionality is required.

### *De-activate*

The de-activate process is the connection between the active and the dormant application. Yes, you could say it's the opposite of activate. And you'd be correct. A de-activate request initiates the shutting down process of the application. This process would be required prior to an uninstall and may be necessary as part of an update.

## Summary

In this chapter, we took a walk through the history of software development. The evolution of software design played a large part in the maturing of software deployment. In fact, the evolution of software development was a precursor to the current software-deployment life cycle.

Certainly, software deployment has matured with the proliferation of the Windows operating system. In turn, large corporations are highly motivated to trim costs of distributing software within their organizations. There are also other issues at play. As we'll see in the upcoming chapters, current trends in software design, such as multi-platform support and distributed computing, make understanding the software-deployment life cycle critical to the success of the software industry.

Many of the issues we've discussed have risen from pressure corporations have placed onto software manufacturers to make their software easier to distribute and maintain. We can see this situation by understanding the motivations of both software producers and consumers.

In Chapter 2, we'll take an in-depth look at the questions large-distribution consumers must answer in the process of deploying software. The answers to these questions drive which software applications are selected for internal use and how the deployment process is implemented.