

Chapter

13

Software Delivery

CHAPTER AUTHORS

Huang Shihong

Leow Ruishen

Peter Hans Martin Helldahl

Srajna Lath

CONTENTS	
1	Introduction 5
2	Software Portability..... 8
2.1	Java as a 'portable' technology..... 8
2.2	Bytecode Distribution 8
2.3	JAR Archives 9
2.4	Applets..... 10
2.5	Natively Compiled Applications..... 10
2.6	Portability – Concluding Remarks..... 10
3	Installation..... 11
3.1	Installation on Windows 12
3.2	Installation on Linux..... 12
3.3	Installation on Mac OSX Snow Leopard 12
4	Package Management Systems..... 13
4.1	Package Management System versus Installers..... 14
4.2	Package management on Windows..... 15
4.3	Package management on Linux..... 17
4.4	Package management on Mac OS 19
5	Conclusion..... 22

1 INTRODUCTION

As programmers we are greatly concerned with creating new and interesting software but while developing, we often overlook the next steps, making the software deliverable and delivering it. This chapter discusses issues related to packaging the different parts of a software product into a single unit (e.g. an installer) and the mechanisms of delivering that unit to the end-user digitally.

Consider a mail delivery system. We write a letter on pieces of paper, put them in an envelope, add a stamp, address etc and post the letter. The postman manages the letter and if everything is in order the mail letter is delivered to the intended receiver's letter box.

The pages of the letter are like the different elements of our software product and the envelope is these different elements packaged together in order to make the software deliverable. The software equivalent of the postal system that delivers the letter can be called a software delivery system. Just like the letter box receives the letter, a package management system receives software products in our computers. However this mapping is sometimes not ideal, as package management systems also act as distribution systems. The envelope, stamp and the address is what makes the letter deliverable through the system, and these are like the distribution specifications. Similarly, we have to package the software in a certain way to make use of the software delivery system and the package management system. Another part of this system would be configuration which will not be covered within the scope of this book. That would be like your settings at home that you may change based on the instructions in the letter, to stretch the analogy and try to fit the system as much as we can. The table below summarizes the analogy.

Postal System	Software Packaging and Delivery System
Pages	Elements
Envelope	Packages
Postman	Package Management System
Settings	Configuration

Windows

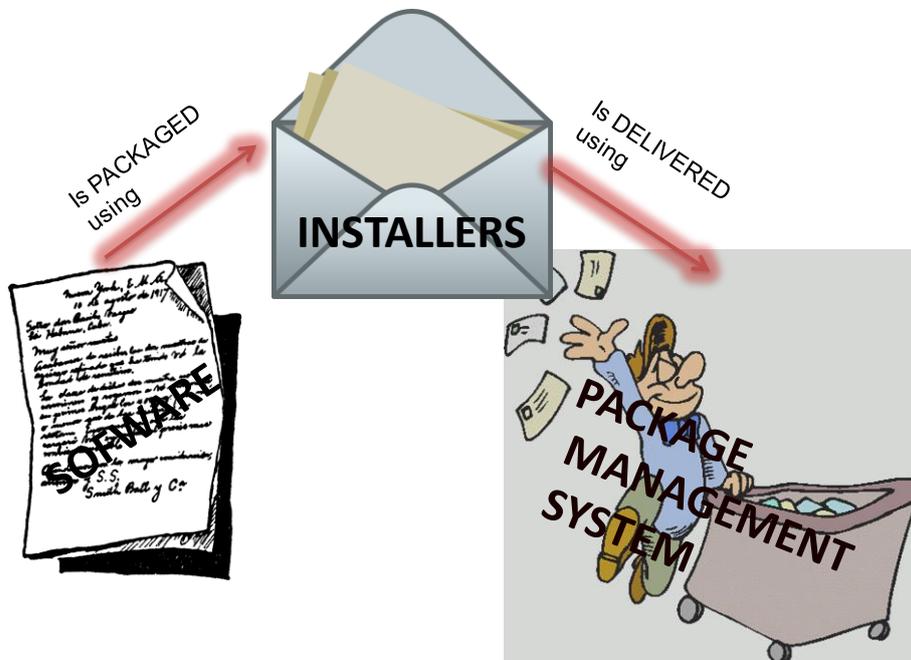
- Elements are dll files, registry edits etc.
- Package is an installer.exe.
 - Although MSI format could be considered as a package, it isn't absolutely correct since it is common practice to use an installer.exe to bootstrap packages such as MSI format. Bootstrapping will be further explained in follow chapter 3.1.1.
- Package Management System is '*Control Panel\Programs\Programs and Features*' which you use to uninstall and repair your programs.

Mac

- Elements are .bundle bundle files, .pkg package files, .plist property list files, and .dist distribution specification files.
- Package is a disk image file.
- Package Management System is Apple's AppStore.
- Delivery system is Apple's AppStore.

Linux

- Elements in a tarball are uncompiled source code.
- Package is the tarball itself.
- Package Management System is for example RPM Package Manager.
- Delivery system is Advance Packaging System, Ubuntu Software Store.



Within the scope of this chapter we discuss the issue of **Software Portability**, writing software that can run on multiple platforms and its trade-offs. We use Java as an example as it covers multiple methods of portability. In the mail delivery metaphor, it is like writing letters that can be easily transported. We also discuss packaging software using **Installers** where a programmer can choose to manage the resources and the user can use the software by installing it. It is like creating an envelope for your software so that the delivery system does not have access to the data inside. Next we discuss **Package Management Systems (PMS)** that are used by the operating systems to manage all the applications, software and resources while at times also being used for distributing. PMS are like the Post Office and *Postmen*. To better explain the role PMS can play as distributing system we use the example of **App Store** and explore the workings of such a PMS.

In the final topic, we will delve deep into the whole process of installation, and discuss **Registries** or local software information databases on different operating systems. Knowledge of Registry, its structure and Hives, is one of the most important things an installer programmer needs to know. Programming an installation is non-trivial as it requires very specific knowledge on where certain files are placed and how certain registry entries are updated. We will discuss that in detail in this topic. Registries are sort of like the *Address Field* in your envelope.

Let us look at some key characteristics of each operating system to understand better how the portability, installation and delivery works for each of them.



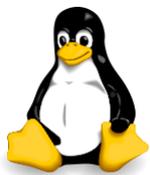
The latest version of Windows is a Windows NT family operating system titled Windows 7. It is programmed in C, C++ and assembly code, and features hybrid kernel. Its design

supports the IA-32, x86-64 and the Itanium platform. You can deploy both 32-bit and 64-bit software on Windows 7. Windows versions have good backward compatibility with older software, and many software written for Windows can have good forward compatibility, unless the software is specifically written for a particular version Windows, which is uncommon. Software developed specifically for windows often use the Windows Presentation Foundation (WPF) or the Windows API. As many computer games make use of Windows' powerful DirectX API, they are also Windows exclusive.



The current version of the Apple Macintosh OS in use today is the Mac OS X version 10.6.6 Snow Leopard, with 10.7 Lion in sight for a 2011 release. Mac OSX is a Unix-based operating system written in C, C++ and Objective-C, with a kernel based on the Mach microkernel.

Apple develops its OS in line with the hardware it produces, and this revision of Mac OSX is developed with the new line of Mac machines featuring Intel based CPUs in mind. Support for Mac OSX is thus limited to IA-32 and x86-64, and IBM PowerPC processor based Macs are no longer supported. On machines with 64-bit processors, Snow Leopard built-in applications will run in 64-bit, and the OS will become a 64-bit OS. Programming applications for Snow Leopard in 64-bit is recommended but not necessary. Most native applications written for Snow Leopard are written in Objective C, and use the Cocoa interface API. The most popular, and Apple supported, IDE used to develop Snow Leopard applications is the XCode IDE. Deploying applications for any Mac OS requires the programmer to be registered as a developer, and an annual fee is applicable. This may partly explain the paid-app trend for third party apps for Mac OS. Programs written for most Mac OS have relatively poor forward compatibility and will usually require updating with each new release of the OS. Mac OSX enjoys limited backward compatibility, some of which is artificially implemented by Apple to encourage users to upgrade their systems.



Linux is not a single operating system. It is a family of Unix-like operating system using the monolithic Linux kernel. Linux is programmed in Assembly Language and C, and has wide-ranging hardware support including mobile devices, and servers. It supports a plethora of platforms including the PowerPC, x86, Itanium, ARM for mobile devices, and even TILE64. As a result, some programmers like to test a software they will make portable, on the Linux platform first. For example,

Adobe's 64-bit version Flash was first previewed on Linux, before a partnership with Microsoft allowed a similar preview version to be quickly available for 64-bit Internet Explorer 9. Linux is packaged for use in a format known as Linux Distribution (distro) for desktop and server use. Some examples of Linux Distros include Debian and its derivatives such as Ubuntu, and Fedora. Linux's extensive support for programming languages means that programmers have a lot of languages and tools to choose from including C++ and Java. However the two main frameworks used to develop Graphic User Interfaces for Linux are GNOME and KDE. Software portability on Linux can be largely dependent on its distro, but in general for each distro, the OS' ability to allow software backward and forward compatibility is strong, and little work needs to be done to keep the software up to date.

The bottom line is that programmers today enjoy significant flexibility when it comes to programming for software, and making software portable because of the streamlining of hardware and OS architectures. The IBM x86 architecture and its widespread adoption helped move programming portability to what it is today. Modern programmers can even write codes that can be run on multiple OS without the need to modify any code, and in the next section, we will discuss the Java programming language, which, thanks to its design, has achieved such a level of portability, with "Write Once, Run Anywhere" being its slogan.

2 SOFTWARE PORTABILITY

Software portability refers to the coding feature that allows programmers to reuse existing code instead of creating new code when moving the software from an environment into another.

Software portability has been described as a desirable attribute for the vast majority of software products (Mooney, 1997). The benefits of software portability are that users can switch platforms or operating systems and still be functional, e.g. software made for an older platform does not have to be changed after each new release. Making your software portable should not be overlooked since it can prove to be your software's success.

To achieve portability you need to think about a number of aspects, both platform and operating system (OS) related and hardware related. This section will primarily cover OS related issues, and will also briefly discuss hardware portability issues, but leave the details for computer engineering texts.

2.1 Java as a 'portable' technology



Java is a programming language developed by James Gosling at Sun Microsystems, now a subsidiary of Oracle Corporation, and released in 1995 as part of Sun's Java platform. Although much of Java's syntax were derived from C and C++, it is architecturally much more similar to NeXT's Objective-C programming language. Java is licensed under GNU GPL. The current Java release is the Standard Edition 6.

In this section, we will cover the four main ways Java distributes its code and applications. These four ways are through its own *bytecode*, through a collection of bytecode known as *Java Archives (JAR)*, embedded on webpages as *Java applets*, and finally, as natively compiled, platform dependent applications. In all these distribution, software installation will not be covered; all four distribution methods do not require the end user to do any installation work.

2.2 Bytecode Distribution

In the previous section, we discussed OS briefly and highlighted some of the key, unique features of each OS that makes portability an issue. We recognize that to remain a portable, a program has to be careful with the tools it uses, sometimes right down to the choice of programming language.

It is particularly important to note the specific APIs that are available for each platform, and whether the software is using them. If the software that is meant to be portable for multiple platform uses platform specific tools, the result could be a non-portable, platform specific software, or a software that is only partially portable, requiring parts of it to be rewritten for a new platform. This traditional model of developing software for multiple platforms is a problem we recognize as "many code for many platforms". Figure 1 on the next page illustrates this phenomenon.

Java provides a mechanism to counter this problem. In Java, programmers can write a single bytecode and expect it to run on Windows, MacOS and Linux. Programmers can technically create write code on a Linux platform and have it run on MacOSX. This portability is what the Java slogan "Write Once, Run Anywhere" refers to.

Java counters this problem through the use of an additional, intermediate layer for platform portability, known as the Java Virtual Machine (JVM). The JVM provides a virtual machine model for software programs and data structures to be executed on. A virtual machine, originally defined by Popek and Goldberg (2005) as an "efficient, isolated duplicate of a real machine", can be seen as a software emulation of a programmable machine. This chapter will not discuss the details of virtualization, which can be found extensively covered in other books on computing

science. However, this model allows Java to have a standardized environment for its bytecode to run, no matter what environment the host machine is in.

In addition, the JVM also provides a layer of safety to Java code and Java programming, as it becomes almost impossible to “crash” the host machine if the Java code is running on a virtual machine. The JVM also verifies the bytecode before execution, using three types of checks:

- Branches always lead to valid locations.
- Data is always initialized and references are always type-safe.
- Access to “private” and “package private” data and methods is rigidly controlled.

This helps JVM ensure that the bytecode input can run error-free and the Java maintains its safety. With the JVM, Figure 1 becomes Figure 2, as shown below.

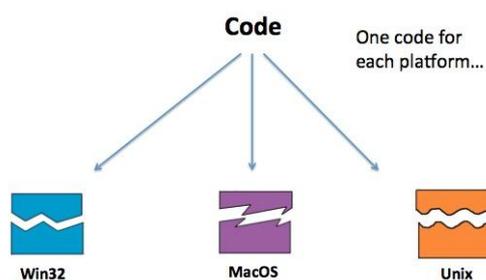


Figure 1 - Many Code, Many Platform

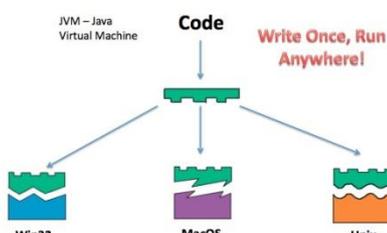


Figure 2 - Java Virtual Machine

Java’s execution environment is known as the Java Runtime Environment (JRE) which also includes the JVM. The JVM runs both Java bytecodes and Java Archives, emulating the JVM instruction set by interpreting it. JRE is available for download at Java’s homepage at <http://www.oracle.com/technetwork/java/javase/downloads/index.html> and all machines and users who want to run Java applications need to have a copy of JRE installed.

2.3 JAR Archives

Java Archives (JAR) aggregate many Java bytecode files into one. You can think of it as a ZIP archive, except it is for Java bytecode files, and also has one important characteristic. That is, JAR files have a mechanism to check if the JAR is a runnable JAR or a non-runnable JAR. A runnable JAR means that, within the package, there is a main class, and a non-runnable JAR simply means that there are no entry points. JAR files maintain this information through the use of a single file within the JAR archive, known as the archive *manifest*. The Manifest file for any JAR archive can always be found in its META-INF folder, and it is always named MANIFEST.MF. The file can be accessed and edited with any text editing tool such as Notepad. This file is unique, and only one of them can exist in a JAR file. The Manifest contains information on the archive relating to its class dependencies as well as version information. We will discuss version control in greater detail in the section on Package Management Systems.



Figure 3 - JAR Archives

JAR archives provide Java with a way to distribute collections of bytecodes and their dependencies without having to distribute them file by file. JAR also maintains the integrity of

these dependencies, making JAR archives ideal as distribution medium for software and programmer libraries to be used in other software.

2.4 Applets

Applets embedded on webpages are one of the original design aims for Java, as Sun predicted that the web and embedded devices will become more popular very quickly. Applets are simply tiny applications written in Java, meant to run on the user's web browser. These programs all require the JRE to run.

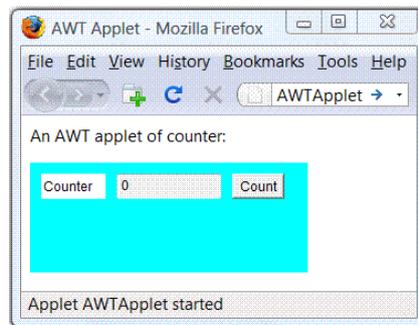


Figure 4 - Simple Java AWT Applet

2.5 Natively Compiled Applications

The advantage of the JVM is that it provides a platform for portability. The disadvantage is that it can cause the software to run quite slowly, especially compared to programs written in C. Although there are currently just-in-time compilers that can help to address the issue, such as the HotSpot (<http://openjdk.java.net/groups/hotspot/>), it can still be helpful, for very demanding programs, to not have to address the problem at all. This is where natively compiled Java applications come in.

Natively Compiled Java applications are Java code compiled using a Native Compiler. The Native Compiler compiles the code into an application that can be accepted, read and run for the given platform. For example, for windows, a ".exe" file might be generated. This application does not require the JRE or the JVM to run, as it is already readily acceptable by the platform. The result is code that runs faster and without the need for a JVM. Some examples of Native Compilers are available on the internet, such as JavaNativeCompiler (<http://jnc.mtsystems.ch/>) and GCJ (<http://gcc.gnu.org/java/>).

Of course, Natively Compiling Java codes are not without drawbacks. First and foremost, portability is sacrificed for performance. In addition there are also complications when the program is designed to be dynamic. For example, if there are classes in the application that loads only at runtime, these classes are now unavailable to the programmer.

For more information on natively compiling Java code, please refer to an Excelsior article by Dmitry Leskov (2009) available here: <http://www.excelsior-usa.com/articles/java-to-exe.html>

2.6 Portability – Concluding Remarks

We see, using the above example, that Java go to great lengths to achieve its level of compatibility. Recall that software portability is concerned with reusing as much of existing code as possible whilst porting the software to different platforms. We have also discussed briefly the major operating systems and how they behave, and what are some characteristics that are unique to them. The problem for us now is how do we use these information.

This section showed you the importance of software portability, so that you as a software developer can decide if portability for your software is an issue important enough to consider portability platforms. In the case of Java, it was important enough for them to develop the JVM

just to make their code more portable. If you are developing games that extensively takes advantage of Windows' DirectX API, and there is no real reason for your software to also be compatible with other OS, then you can choose to develop on Windows, which can make your software run faster.



As a programmer, your choice of development tools can affect your program's portability. Choose the wrong tool, and you will realize that much of your code needs to be rewritten when the software gets moved to a new platform.

This section covered portability without installation. In the next section, we will discuss installation.

3 INSTALLATION

In the previous section we discussed software portability. In this section we will be talking about software installation. Installation is simply defined as the act of putting a computer program onto a computer system so that it can be executed. In this section we will explore software installation through installers for different platforms, and discuss how they work in greater detail.

An installer is a small program that helps guide you through the setup of the application that you want to install. This program unpacks compressed data included with the installer and writes new information to your hard drive.

Although some software can be executed without an installation procedure most programs require an installer since they need certain resources in order to run. These resources must be properly managed, and the software installer manages this for you in a convenient way.

Some of these resources include:

- Unpacking of files supplied in a compiled format
- Organizing the files in folders
- Providing information about the program to the operating system
- Make changes to the registry
- The program might need additional software to be installed in order to run.
 - These can be defined as requirements in the installer.
 - The installer can search for them, then do an automatic download and installation as necessary.
- Modify registry key entries and environment variables

The installer also allows you to choose which parts of the program you want to install and protects your source files from unwanted modification. If the program cannot be installed

successfully, or if the user cancels the installation, the installer can roll back changes and restore the computer to its original state.

During the installation of computer programs it is sometimes necessary to update the installer itself. This is made possible by a technique called *bootstrapping*. The common way to do this is to use a small executable file which updates the installer and starts the real installation after the update. This small executable is called a *bootstrapper*.

3.1 Installation on Windows

Installation of programs on windows is usually done by double-clicking the installer executable icon and following the instructions.

An example of a windows specified installer is Windows Installer. It is a software component used for the installation, maintenance, and removal of software on modern Windows systems. The installation information is packaged in .msi files. MSI files might install the application straight into the system. However, this process does not verify the system environment. Thus user may experience difficulties using the application which installed by setup.msi, as it may not be compatible with user's operation environment. Common practice today is to wrap up MSI files with an executable bootstrapper such as setup.exe. The bootstrapper will start and check the system environment before actual installation begins. MSI files will then be executed only if the system is capable of running the application.

The Windows Installer contains a number of feature upgrades compared to its predecessor, the Setup API. Key changes include an automatic generation of an uninstallation sequence, as well as a GUI framework. Microsoft encourages third parties to use Windows Installer as the basis for installation frameworks, so that they synchronize correctly with other installers and keep the internal database of installed products consistent. Some features such as rollback and versioning depend on a consistent internal database to work properly.

3.2 Installation on Linux

The two most common methods of installing software on LINUX are RPM and tarballs. RPM stands for RPM Package Manager (originally Redhat Package Manager) and we will be covering this in greater detail in Package Management Systems. A tar ball is an archive of files, similar to a Zip file on windows. These mostly contain programs in source-code. After unpacking you will need to find the README file or INSTALL and read its instructions for installation. Very often, you will have to compile the source-code yourself in order to turn the source-code into an executable binary.

Compiling from source-code can be quite tedious, but there are good reasons for doing so. For example, you may want to customize program features and install paths. Self-compiled programs are usually even more stable and faster than precompiled ones because after the local compilation process they achieve greater compliance with your system settings. Upgrading is easier since you can simply apply a patch and recompile. But because this process can be complicated, especially to regular users who may not be familiar with source code and compiling, the method remains largely unpopular with a majority of computer users.

3.3 Installation on Mac OSX Snow Leopard

Many software installers and updaters are disk image (.dmg) files and basically functions like an actual CD image file. The purpose of the .dmg file is compression since the format can greatly reduce the size of the files. Running .dmg files on Snow Leopard is similar in concept to Windows Installers; users simply need to double click on the .dmg file to begin the process. When this is done, Snow Leopard will "mount" the image, using a metaphor similar to a cd image, and an installer window will usually appear that will take care of the rest of the installation for you. If there are no installer windows, a window will usually appear showing the

contents of the image, and one of them is usually an installation package. Running that package by double clicking it will begin the process.

If there is no installer window and no installer when you mount the disk image, simply install the software by dragging and dropping the file or folder from the disk image to the Applications folder. These types of files already have all the necessary files pre-installed and don't require an installer to install it.

4 PACKAGE MANAGEMENT SYSTEMS

A software package is distribution of files which usually in archive format and contain computer software, application and data to be installed by Package Management System (PMS) or other means which mostly involves a self-sufficient installer. For example, most of Windows packages will be in forms of installers which are able to run on their own. Linux packages are distributed in forms of archives of either uncompiled source code or pre-compiled binary files require PMS for further action. Not known to many, Mac OS packages are merely a bunch of folders, each containing data files for the application to work, disguised as a single file. Such folders are completely different from those directory folders. All packages also contain many other information, like their purpose, description, version and vendor etc. which are called metadata. Such metadata will be stored in local package database maintained by PMS, which will then use these data to perform software update and prevent certain potential problems such as missing software prerequisite and dependency mismatch.

A Package Management System (PMS) is, according to Ian Murdock *Package “the single biggest advancement Linux has brought to the industry”, that it blurs the boundaries between operating system and applications, and that it makes it “easier to push new innovations ... into the marketplace and ... evolve the OS”. (2008) A PMS is usually made up of a series of distinct software tools to consistently manage and automate the process of installing, upgrading, configuring, and removing of software packages for a specific operating system. Many could take PMS as a convenient way of managing software packages. With growing numbers of packages, especial in Unix like system which consists hundreds even thousand package components, a PMS will be essential for both user and the system to operate on the platform.*

Package management systems are charged with the task of organizing and managing all of the packages installed on a system and the system resources. A PMS typically:

- Verifies file checksums to ensure correct and complete packages.
- Verifies digital signatures to authenticate the origin of packages.
- Applies file archivers to manage encapsulated files.
- Upgrades software with latest versions, from a software repository.
- Organizes packages to achieve cross platform similarity and normalization.
- Facilitates easy installation and uninstallation and at times provides front end for locally compiled packages.

Some other features of the PMS that is important for creating deliverable software are:

Resolving Conflicts and Dependencies: Many programs share dependencies, that we would not want to install by default each time a new software requests for it. Before installing the package the PMS checks the resources it uses and takes care of the dependencies if they are not already installed.

During software installation and execution there might be conflicts that are needed to be resolved by the PMS. For example software needs a perl interpreter to run but at the time of execution the file the interpreter is not at the default path. To resolve this PMS maintains

registry entries and command name registries so that the file can be accessed using a single standard path which will point to the new location of the file even if the file is shifted. Any such conflict can arise when search paths for C headers are not in the same order as the C libraries.

Conflict also arises when two software are dependent on different versions of the same application. PMS needs to save both the versions of the software that will essentially have the same name in the probably in the same folder, in case of Windows the C:\Programs Files. To solve this, the PMS has different registry entries for both the versions therefore identifying them as different applications.

Maintenance of configuration: As PMS in UNIX are essentially extensions of file archiving utilities, they can usually only overwrite or retain configuration files, rather than applying rules to them. Hence this may create some problem with updating configuration files especially if the new configuration file follows a different format. Problems can be caused if the format of configuration files changes.

For instance, if the old configuration file does not explicitly disable new options that should be disabled. To resolve this, the PMs would sometimes completely remove the old configuration before installing the new one.

4.1 Package Management System versus Installers

PMS often involves in installation process. Thus, confusion arises between installer and PMS. There are few key points where we can draw a clear differentiation between both of them.

Package Management System	Installer
Typically integrates with the operating system	Comes with bundled product
Uses a single installation database	Performs its own installation
Examines and manages all packages on the operating system	Does not work with packages outside its bundled product
Single/Specific package format	Multiple installation formats

Table 4-1

As mentioned in the previous chapter, different OS will require different package and installation method, thus different package management system will be required. Earlier, package management system was typically part of OS, and due to the commercial property of the OS, the functions of package management system were limited to packages having same vendor as itself. To satisfy consumer needs, many third party PMS are developed and many new functions emerged.

OS			
Typical install method	Installer dialogue	Command line	Drag and Drop
Typical package type	Installer	Rpm, tarball	Installer, DMG

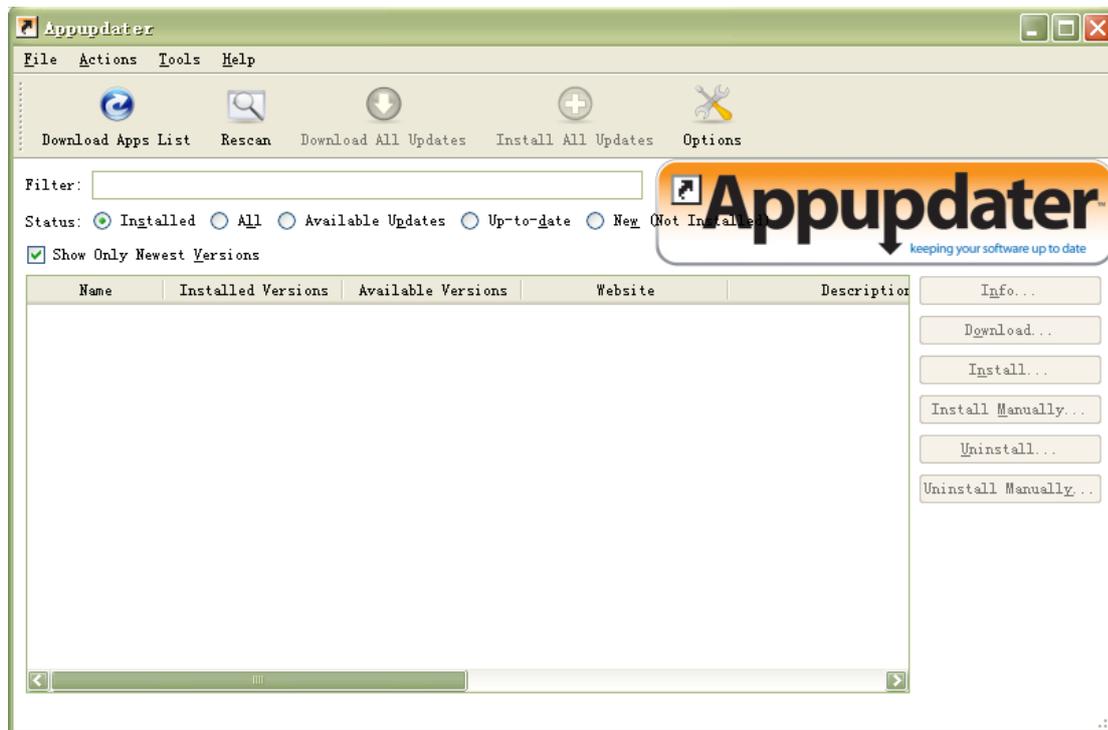
3rd party PMS	available	available	Available
PMS notification	Notify PMS during installation	Notify PMS immediately after installation	Notify PMS only after first run of application

Table 4-2

4.2 Package management on Windows



The Add/Remove Program feature for Windows is a built-in primitive package management system. Though some may not recognize it as one, it does come with all the basic functions and characteristics a PMS should have. As a special applet, the Add or Remove Programs is stored under the name appwiz.cpl in the SYSTEM32 folder. Windows Update handles only packages distributed by Microsoft. The lack of support for updating packages from other vendors had sparked development of many other PMS, most of them are open source programs, like Appupdater written in Python and released under GNU General Public License (GPL).



Appupdater connects to a software repository which hosts a number of packages online. From here, user will be able to view the basic information of a package such as its available version, short description and its license type.

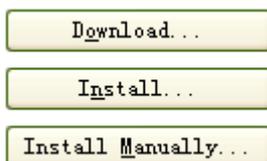
Show Only Newest Versions

	Name	Installed Versions	Available Versions	Website	Description	
1	7-Zip		4.65	http://www.7-zip.org/	7-Zip is a file archiver with a high compression ratio.	GNU
2	AbiWord		2.8.2	http://www.abisource...	AbiWord is a free word processing program similar to ...	GNU
3	Acrobat Reader		9.3.4	http://www.adobe.com/...	Adobe Acrobat Reader	Comm
4	ActivePerl		5.10.0.1004	http://www.activestat...	ActivePerl is the complete, ready-to-install Perl distribu...	Comm
5	Ad-Aware		1.06	http://www.lavasoftus...	Ad-Aware SE Core application	Comm
6	Appupdater		1.4.2	http://www.nabber.org...	Appupdater provides advanced functionality to Windows, simi...	GNU
7	Aria2			http://aria2.sourceforge...	aria2 is a download utility with resuming and segmented downloa...	GNU
8	Aspell		0.50.3	http://aspell.net/win...	GNU Aspell is a Free and Open Source spell checker designed ...	GNU
9	Audacity		1.2.6	http://audacity.sourc...	The Free, Cross-Platform Sound Editor. Audacity is free, ope...	GNU
10	AutoHotkey		1.0.48.03	http://www.autohotkey...	Automation. Hotkeys. Scripting.	GNU
11	AutoIt		3.3.0.0	http://www.autoitscri...	AutoIt is a freeware Windows automation language. It can be...	
12	AVG		9.0	http://free.avg.com/	AVG Free provides you with basic antivirus and antispyware prot...	Comm
13	BitTorrent		6.4.e	http://www.bittorrent...		
14	Blender		2.49.b	http://www.blender.org	Blender is the free open source 3D content creation suite, ava...	
15	CCleaner		2.36	http://www.ccleaner.c...	CCleaner is a freeware system optimization and privacy tool. ...	Comm
16	CDex		1.51	http://cdexos.sourceforge...	CDex is a CD-Ripper, extracting digital audio data from an Aud...	
17	Celestia		1.5.1.0	http://www.shatters.n...	The free space simulation thatlets you explore our unive...	GNU

To know more about a package, the user could click on the “Info” button to take a closer look at the package and information of its previous versions.



Once the choice is made, the user can proceed to “check out” the package. By choosing either installation mode



Upon clicking either of the buttons, the user will be prompted to choose a version to install.



While there are three modes available and self-explanatory, “Download” will download the installer into local hard disk only; “Install Manually” will download the installer and execute it, and let user take over from here; “Install” will automatically complete the installation process with showing a progress bar, the installation will be done in the background.



The update and removing of a package will be carried out in similar manner.

Relying on the Registry as its backbone, Appupdater will identify a package by scanning through registry and installation directory, then automatically detect the current version of all (selected) packages and match them with the latest version of its online software repository to look out for any outdated package.

4.3 Package management on Linux



When a system administrator wants to perform system maintenance or package installation, with a bunch of tarballs, things are going to get complicated. As we mentioned, a PMS is essential especially in Linux where thousands of packages involve in the OS. Manual management, for example, update or remove certain package without breaking the dependencies among packages will be extremely difficult given the size and complexity of them.

Originally designed for Red Hat Linux, RPM is a recursive acronym for RPM Package Management, shipped a number of Linux/UNIX distributions. Making use of the command line, it simplifies the process of installation and removal of packages. Both precompiled binaries and source code (included in SRPM, a front end package for RPM) file can be used by RPM for installation. A simple line of command will complete the installation process.

```
rpm -i apache-1.3.14-3.i386.rpm
```

Admin access will be required for installation and maintenance work, as the user account with administrator privilege and password. Though the process could look easy, problems could arise especially when removing a package. Some packages could rely on each other to run

properly, the relation is called dependency. In a system where thousands of packages are installed, the dependency could get nested. In the case when a package depends on a certain component of another package which gets replaced or gone missing during maintenance, dependency hell arises.

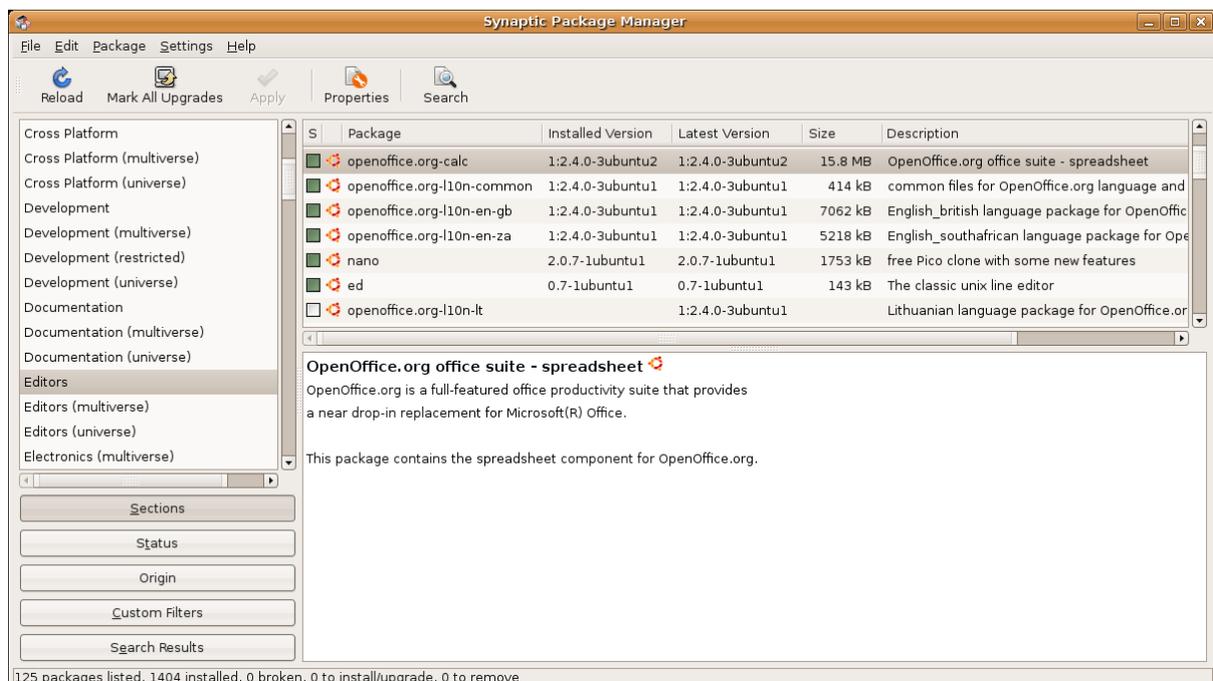
As RPM maintains all the dependencies by having automatic build time evaluation, it will show a warning when user try to remove a package which could break the dependency of other packages. RPM keeps its own installation backend database in typical location such as `/var/lib/rpm` using Berkeley DB. It contains metadata for all installed packages, indexed and replicated for faster query response. Changes such as removal of a package can be reversed. However, it does not do much on resolving dependency hell.

To deal with this problem, we can use Advanced Packaging Tool, APT. This set of tool rely on package repository, somewhere you can retrieve and install software and it is usually hosted over the network, to resolve, or rather, to avoid the dependency hell.

Through APT, a package can be downloaded and installed. When a package is getting installed, APT will examine its dependencies. From there, APT will retrieve all components required to satisfy the dependency requirement for it. Then during the installation process, the particular package will be installed together with its required dependency. Therefore, a package will never fail to start due to missing dependencies.

All information about packages available will be stored in `/etc/apt/sources.list`. This is the location configuration file which tells the system where to locate the desired packages, in this case, the repository. If necessary, certain packages previously containing the dependency will be downgrade for conflict resolution. The repository does not just contain the latest version but all versions available for installation, so that possible upgrade or downgrade can be arranged. To ensure certain preferred package will not be modified, administrator can make use APT pinning feature to ensure no further update will not be performed on it. Such pin can be found in the system directory of `/etc/apt/preferences`.

The command line may not be appealing to certain user, several front ends of APT are made, include graphic user interface, such as Synaptic Package Manager shown below.



4.4 Package management on Mac OS



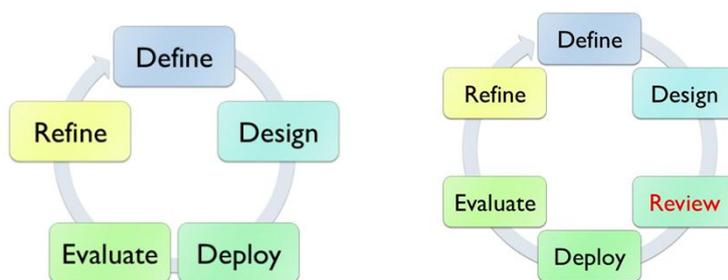
Repository almost solves the dependency issues and Apple's AppStore completely get rids of the dependencies since each app becomes self-contained. No dependency is needed when all apps come with its necessary components.

Throughout this section of the book chapter we will be discussing the Apple Mac App Store as an app store example. Other app stores have similar requirements and will change the development process in similar ways, but we chose to discuss the Mac App Store because it is a popular, fast-growing platform with lots of opportunities for mac developers.

Developing for the App Store is advantageous because it helps you as a software developer mitigate the difficulties of manually distributing and marketing your software product. The App Store is a unified environment where consumers can search for and download applications simply in a process that is defined similarly for all applications.

The user is able to find applications that suit their needs whether or not they know the specific application they want, so an app store helps your program to achieve accessibility.

As a new platform, there are several disadvantages. Unified environment and structure imply a strict guideline to follow. For the Mac App Store, this translates to the App Store guidelines published by Apple, made available to all registered Mac developers. Programming Language and systems support. AppStores can sometimes require your application to be of a particular format or developed in certain languages. The Mac AppStore largely only accepts applications developed for Mac in Objective C using the officially supported IDE X-Code. Developing for an AppStore requires submitting your application to the AppStore's owner for review, much like developing for a third party retailer. This review process can potentially impact your development cycle and must be taken into consideration.



However, there are certain advantages to being well versed in the process behind the development of applications for an AppStore, and the rest of this section will highlight the general process of developing software for the Mac AppStore.

The review process for the Mac AppStore should be factored into the development cycle. The old development cycle, depending on the development group, generally involves the identifying of problems and conceptualization for solution, then developing the application and distributing it to intended audience. When developing for AppStore, you will need to take the review process into account. Typically, the review process for the Mac AppStore will take about one week or so, but in some instances, especially if the app is focused on user-submitted content, the testing will take longer and can take as long as 3 months.

The review process presents a difficulty for developers because the application that is submitted for review is often a finalized release version, and any bugs detected after the release can only be stamped out in the future as patches. If a more updated version of the application is sent for review, the process is restarted. Hence very often developers will find that their development process is stuck whilst their app is being reviewed.

By designing the development cycle around the review process, instead of a fully finalized version, development teams can send an almost final prototype program for review whilst their team focus on stemming out bugs to prepare a first patch ready for final release.

The review process can sometimes be painful for developers, as Apple will contact the development team if they find parts or functions in the program which they need clarifications about. One way to reduce such incidents happening and to reduce the chance of getting the application rejected is to fully and completely document the application. Another, more recommended way, is to follow the Apple App Review Guidelines made available to all registered developers. Whilst we cannot publish the full guidelines here, some of them are as follows:

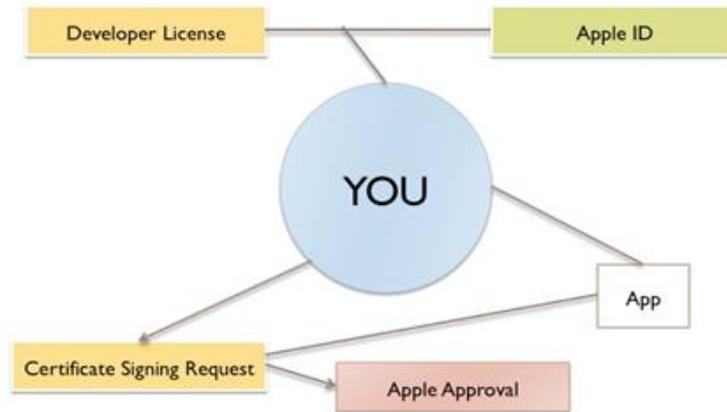
- Apps that crash/exhibit bugs/do not perform as advertised
- Apps that are set to automatically launch without users' permission.
- Apps that use third party material without documented rights.
- Apps that misspell Apple products or services.
- Apps that change the native user element of Mac OSX.
- Apps that suggest Apple endorses the app.
- Apps with metadata that mentions the name of another computer platform.

And the full guidelines will be available to any fully registered Apple developer. Following the Guidelines, although is no guarantee of a rejection-free Review process, can help the make the process smoother for both yourself as the developer, and Apple as the reviewer. The faster the review processes is over, the faster we can get on to do more important and complicated things, so it is beneficial for developers to fully understand and utilize the Guidelines.

Even if the Guidelines are not accessible, Apple Developer portal available at <http://developer.apple.com/> contains documentation and best practices to help developers pass the review even without the Guidelines.

Once you know this, and are ready to submit your program for Apple to review, you can do so as follows.

- The Apple App Submission process can be divided into three parts. The first part is the licensing process. The second part is the preparation of the IDE process. The third part is the App packaging process.
- For the first part, you will need to do two important things. The first thing is to get the Apple developer license. The license is USD\$99 a year and allows you to develop and deploy unlimited apps. The license is tied to your Apple ID, and your Apple ID is tied to your access to everything Apple, so do not lose the password to that ID.
- The second thing to do in the licensing process is to generate the Certificate Signing Request (CSR) for your application. Each application will have its own CSR which apple must approve before you can send the app for review.

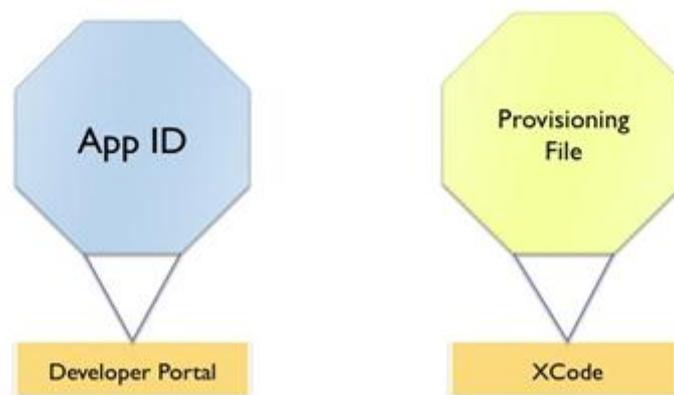


Step 1: Licensing

The second part is the preparation step for the IDE. In this case our IDE will be the default supported IDE XCode. The objective of this step is to first list your application into the Developer Portal, and then configure settings in your XCode IDE in order to package the App for distribution through the right medium

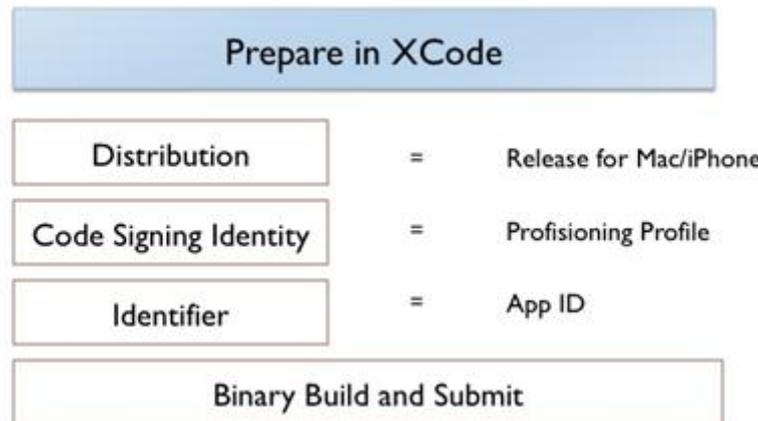
The first step is to create an App ID in the developer portal. This can be done by logging into the Developer Portal with your Apple ID and submitting an ID for the App.

Secondly, in your XCode, you should create a provisioning file. The provisioning file is what XCode will refer to when it wants to package the app for distribution. In this case for the provisioning file we need to set the distribution platform to be AppStore.



Step 2: Prepare your IDE

The last part is to tidy up the packaging process. In XCode ensure that the distribution setup is set to App Store, Ensure that the Code Signing Identity you prepared in step 1 is matched into your provisioning profile, ensure that your identifier is the same as your App ID, then apply binary build and submit the app to iTunes connect at <https://itunesconnect.apple.com/> and the process is completed.



Step 3: Packaging your App

A summary of the submission process is as follows:

Step 1: Licensing

- Obtain Developer License (iPhone/Mac OSX)
- Generate Certificate Signing Request
- Submitting a Certificate Signing Request for Approval
- Downloading/Installing Certificates on your Machine

Step 2: Preparing your IDE

- Create an App ID for your app in the developer portal
- Create a Provisioning File for your Xcode (Select App Store for Distribution Method). This is the provisioning profile you will use to distribute your app.

Step 3: Packaging your app

- Prepare your app for distribution in Xcode
 - Release for iPhone/Mac distribution
 - Match Code Signing Identity with Provisioning Profile
 - Identifier: App ID.
 - BINARY BUILD!

The trend today for software distribution tends towards digital distribution, and the concept of App Store as a centralized distributor will only become more and more valid. Having an understanding of how the AppStore functions will help improve your employability and marketability, giving you an edge in your technical job search.

5 CONCLUSION

In this chapter we have looked at *Software Packaging and Delivery* in context with the three main platforms- Windows, MAC and Linux. We learnt how Java achieves portability through four ways the four ways of distribution: through its own *bytecode*, through a collection of bytecode known as *Java Archives (JAR)*, embedded on webpages as *Java applets*, and finally, as natively compiled, platform dependent applications. We also looked at what is installation and how do the different platforms install applications and softwares; windows uses GUI with

executables to facilitate the installation process, Linux uses the RPM and tarball, MAC has the drag and drop feature for installing softwares and applications.

We have discussed what are Package Management Systems, their importance and necessity. For Windows we discussed in detail AppUpdate, a 3rd party PMS; for Linux we discussed RPM and for MAC we discussed AppStore, how it also works as a digital delivery media and how to develop for it.

To better understand how the installation and Packagemnt Management systems work with the Operating Systems in the background we need to understand how the OS stores its configuration settings and how does it use them. Windows uses a centralized database called *Registry* to store this information and softwares access and edit this database accordingly. MAC uses multiple binary coded text files called *Property Lists* to store the configuration settings.