Software Delivery Guide

Tools > Files & Software Installation

2006-07-24

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction

Note: This document was previously titled *Software Distribution*.

This document describes the process of packaging and delivering a software product so that it can be installed on a user's computer. The two major methods of delivering software are manual installs and managed installs.

A manual install is the preferred delivery solution because it offers the simplest install experience for small or compact products, such as a single application package. For example, to install an application, a user may drag the application package from a CD onto a folder of their choosing.

For more complex products, managed installs let you define every aspect of the install experience, including making sure the target computer meets specific requirements. Managed installs are generally used with products comprising several components to tailor the installation of each component depending on its kind. A managed install uses installer packages that define an install experience. When users open such packages, the Installer application guides them through the installation process and copies the product files to the appropriate locations on their file system.

Network administrators can use a type of managed install, a remote install, to install a product on several networked computers using Remote Desktop. No user interaction occurs in this type of install.

Software requirements: This document assumes that delivery solutions are created using Mac OS X v10.4 or later and Xcode 2.3 or later. The delivery solutions described in this document can be installed on computers running Mac OS X v10.2 and later.

This document is meant to provide software delivery guidelines to product developers, product packagers, and network administrators.

- **Product developers** create products or product components (such as applications, frameworks, plug-ins, and so on) using development tools such as Xcode.
- **Product packagers** devise a delivery solution for an entire product.
- **Network administrators** manage a group of networked computers and may need to install the same software on several computers remotely.

Organization of This Document

This document contains the following chapters and appendixes:

- "Overview of Software Delivery" (page 9) introduces the major software delivery mechanisms used in Mac OS X: manual installs and managed installs. It also explains remote installs, which network administrators use to install products on several computers on a network.

- "Product Containers" (page 11) introduces general product delivery mechanisms that can be used in manual installs and managed installs. This chapter shows how to create a disk image for a standalone product. This chapter also shows how to create Internet-enabled disk images, which streamline manual installs. This chapter is especially useful to product packagers.
- "Manual Installs" (page 15) describes manual installs and provides an example of a simple product that should be installed manually. Product developers and packagers should read this chapter.
- "Managed Installs" (page 17) lists the major features managed installs provide, and explains when managed installs are appropriate. This chapter describes the three types of installation package and how they are processed by the Installer application. This chapter also describes the managed installation process and user experience. This information is useful to product packagers and network administrators.
- "Packaging Product Components" (page 27) explains how to create an installation package for a product component or a single-component product for a remote install.
- "Defining a Managed Install" (page 33) explains how to create an install experience using a distribution package or a metapackage. Also shows how to create a hybrid metapackage. This information is useful to product packagers and network administrators.
- "Specifying Install Operations" (page 45) explains how to define install operations for a managed install. This information is useful to product packagers and network administrators.
- "Performing Remote Installs" (page 49) provides an overview of remote installs and an example of one. This chapter is targeted to network administrators.
- "Specifying System and Volume Requirements in Pre-Tiger Systems" (page 53) explains how to define installation requirements using executable files. This information is useful to product packagers and network administrators.
- "Prebinding Applications" (page 57) explains when an application may benefit from having its prebinding information updated after a manual install. Product developers and packagers may find this information useful.
- "Preserving Resource Fork Data" (page 59) provides an overview of resource forks and explains how the PackageMaker and Installer applications handle payloads that include files with embedded resource forks instead of separate resource files. Product developers and packagers may find this information useful.

See Also

- *File System Overview* describes the Mac OS X file system and its domains. Knowledge of the Mac OS X directory hierarchy is paramount when packaging multicomponent products.
- *Runtime Configuration Guidelines* explains the keys used in `Info.plist` files to specify some product properties, such as version and identifier. These properties are required when packaging products to create a managed install.
- *Bundle Programming Guide* describes Mac OS X bundles and file packages, which are used extensively in Mac OS X.
- *Apple Remote Desktop Administrator's Guide Version 3.2* explains how to use Remote Desktop to manage a set of networked computers.

Overview of Software Delivery

Mac OS X provides several mechanisms you can use to deliver a software product to your customers. These mechanisms are both flexible and easy to develop. They support the delivery of simple or complex products to novice or expert users. In addition, Mac OS X and Apple Remote Desktop provide facilities for delivering products to several computers on an intranet.

Whether you are a product developer or a network administrator, Mac OS X allows you to create a product delivery solution to satisfy the product-installation needs of your customers.

Installs for Product Developers

The nature and structure of your product determine the install experience you can provide to its users. If your product is a self-contained application (one that doesn't need to install components at different locations in the file system), you can distribute it as a single file or folder. Users then can drag the product from its container or delivery vehicle to a location of their choice in their file systems. This type of installation process is called **manual install**. This is the preferred method of delivering applications to Mac OS X users.

However, there are situations that require more complex products. For example, some applications require the presence of shared resources such as frameworks or fonts, which, in general, reside at `/Library/Frameworks` and `/Library/Fonts`, respectively. Each item that resides at a distinct location on the file system is known as a **component**. To make it easy for users to install a multicomponent product, instead of making users place each component in the appropriate location you develop an appealing install that frees users from that tedious and error-prone task. To that end, the Installer application (`/Applications/Utilities`) uses a simple interface to guide users through the process of installing multicomponent products. This type of installation process is known as a **managed install**.

Note: This document uses the word *component* as a general term to refer to parts of a multipart product. It has no functional relation to items that are normally placed in `Library/Components` directories.

Managed installs provide users with a GUI (graphical user interface) through which they specify a few details to customize an install. The Installer application handles the file-management tasks on the users' behalf.

You use PackageMaker (`/Developer/Applications/Utilities`) to develop a managed install. You can tailor the install experience with Read Me and license agreement files and by specifying installation requirements to ensure the product is installed only on systems that meet specific criteria, such as available memory and disk space.

Listing 1-1 shows an example of a single-component product, called Atom. This is the kind of product users should be able to install manually.

Listing 1-1 Atom—A single-component product

```
Atom_product/
   Atom.app
```

Listing 1-2, on the other hand, showcases a multicomponent product, called Levon, comprised of an application package, a documentation file, and a framework. The application package should be placed in `/Applications`, the documentation file in `/Library/Documentation`, and the framework in `/Library/Frameworks`. (See *File System Overview* for a more complete list file system locations and component types.) For users, having to drag three files to three locations is not an appealing install experience. A managed install is more appropriate for a product of this kind.

Listing 1-2 Levon—A multicomponent product

```
Levon_product/
   Levon.app
   Levon_User_Guide.pdf
   SharedServices.framework
```

Installs for Network Administrators

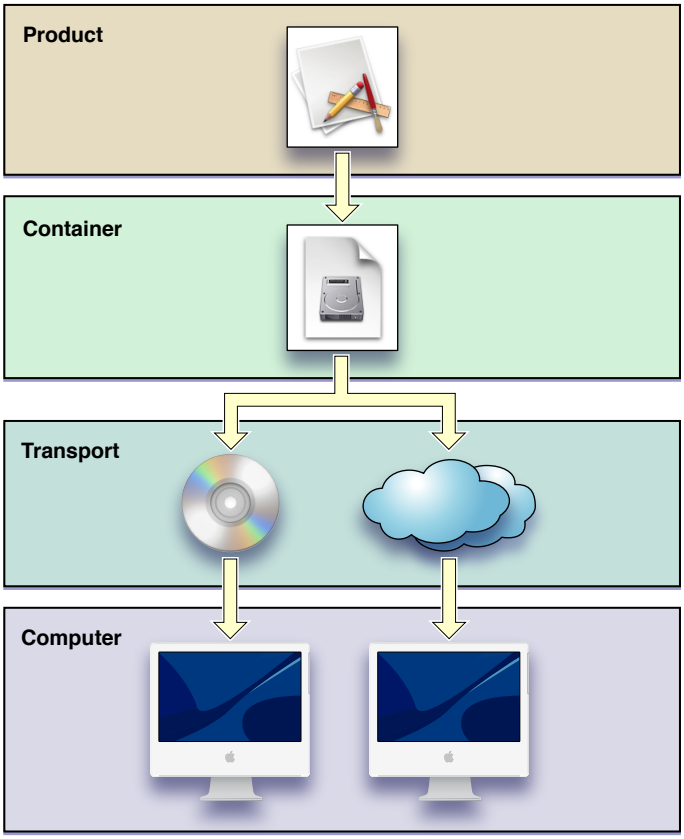
If you're a network administrator, you probably are often faced with the task of distributing the same software to a set of computers. A **remote install** is an installation process that network administrators can use to remotely place products on several networked computers. Remote installs use the same underlying engine that managed installs are based on. But although the Installer application is used to perform the install tasks on the target clients, the users of these computers do not participate in the installation process; that is, they don't interact with Installer. In fact, administrators can install products on computers while users are logged in and actively using their systems. Also, administrators can schedule a Remote Desktop task that performs a remote install, ensuring that unavailable computers receive the product when they become available. (See *Apple Remote Desktop Administrator's Guide Version 3.2* for more information.

The following chapters describe in detail how to develop a delivery solution for your product.

Product Containers

Whether you devise a manual install or a managed install (introduced in "Overview of Software Delivery" (page 9)) for your product, in general you need to place your product into a container. A **container** is a file-based enclosure for a product, which facilitates the product's delivery to a user's computer. To deliver your product to your users, you may use a **transport** (delivery vehicle) such as optical media or the Internet. Figure 2-1 illustrates the preferred delivery solution for products in Mac OS X.

Figure 2-1 Delivering a product using a disk image



ZIP files: If your product can be used in operating systems other than Mac OS X, you may consider using a ZIP archive as the product's container. To create a ZIP product container in the Finder, select the product directory in a Finder window and use the Create Archive command. You may also use the `zip(1)` command-line tool.

Disk images are the preferred container for software products on Mac OS X. They allow you to group a set of files in a compact format that is easily handled by users. These enclosures are easily transported across a network because they appear as a single file. To access the contents of a disk image, a user double-clicks it

in the Finder, which opens a standard Finder window showing the disk image's contents. Compressed disk images allow you to use delivery vehicles—space-limited optical media or bandwidth-strapped networks—more efficiently.

The best container for delivering a product through the Internet is an Internet-enabled disk image. These containers are automatically opened and disposed of. Users need only move the product to a convenient location. See "Manual Installs" (page 15) for more information.

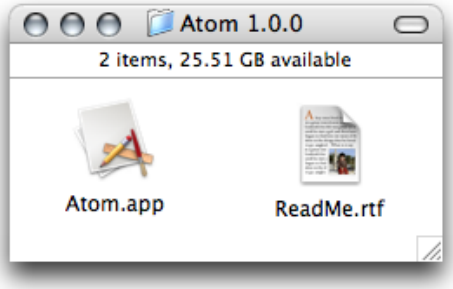
The following sections describe how to place a product on a disk image and how to configure a disk image to provide a streamlined install experience when it's downloaded from a network, such as the Internet or an intranet.

Creating a Disk Image

The Disk Utility application (`/Applications/Utilities`) allows you to create a disk image from a folder, as described in the following steps.

1. Create a directory named after your product, containing the product's files, such as the `Atom 1.0.0` directory shown in Figure 2-2.

   Figure 2-2 A product directory in a Finder window

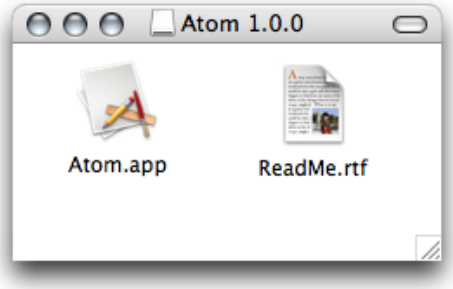
   

2. In Disk Utility, use the New Image from Folder command to create a disk image of the product directory.

   To produce the smallest disk-image file possible, use the compressed disk-image format.

Figure 2-3 shows how a user may see the contents of the disk image after opening it in the Finder.

Figure 2-3 A disk image's contents in a Finder window



Tip: To make it easier for users to place disk-image-delivered applications in the `/Applications` directory, include a symbolic link to the `/Applications` directory as part of the disk image's contents.

Internet-Enabling a Disk Image

Although all disk images can be transported through the Internet, users need to open them, retrieve their contents, and discard them. An Internet-enabled disk image automates this workflow. After a user downloads an Internet-enabled disk image from a network using a web browser, the image is automatically opened, its contents placed at the user's download location, and discarded (in the Trash). This improves the manual install experience by performing a few tedious tasks for the user.

To Internet-enable a disk image, use the `hdiutil(1)` command-line tool, as shown in Listing 2-1.

Listing 2-1 Internet-enabling a disk image

```
hdiutil internet-enable -yes <path_to_disk_image>
```

Note: Only read-only or compressed disk images can be Internet enabled.

Adding a License Agreement to a Disk Image

The Disk Utility application can display a multilingual license agreement before opening a disk image. Disk Utility does not open the image unless the user agrees with the license.

To create a disk image with a license agreement, get the Software License Agreement for UDIFs software development kit (http://developer.apple.com/sdk/index.html). This SDK explains how to add a license agreement to a disk image and includes the resources necessary for this task.

Manual Installs

A manual install is completely controlled by the user. That is, the user gets the product from its container or delivery vehicle and drags it to their file system. This install model works best for simple products, which are made up of one component type. Listing 3-1 shows a typical example of such a product, called Atom. For multicomponent products, a managed install is more appropriate; see"Managed Installs" (page 17) for details.

Listing 3-1 A simple product

```
Atom 1.0.0/
   Atom.app
   ReadMe.rtf
```

The main component of the Atom product is the `Atom.app` file package. Users would normally place this file in `/Applications`, but they are free to place it anywhere. Users may or may not copy the `ReadMe.rtf` file to their file system. This file is not essential for the operation of the Atom product but can include installation instructions for novice users in addition to information about the product.

A manual install is the ideal install experience for Mac OS X users. It is also the easiest way for you to deliver your products. All you need to do is place the product in a container (as described in "Product Containers") and make it available to your customers on a delivery vehicle.

Important: If your product includes an application that has to run on Jaguar-based systems (Mac OS X v10.2), the application should be prebound after the user installs it. See "Prebinding Applications" (page 57) for details.

Managed Installs

As described in "Overview of Software Delivery" (page 9), **managed installs** give you more control over the installation process, which, among other things, allows you to fine-tune the user's install experience. However, when your product is made up of a single component that doesn't need to be placed at privileged locations in the file system, such as `/Applications` or `/Library`, you should provide users with a manual install for your product. Manual installs are faster and easier to perform for novice and expert users alike. See "Manual Installs" (page 15) for details.

Multicomponent products benefit from managed installs because you can specify how each of a product's components is installed. Also, remote installs—which allow you to install products remotely on several computers on a network—are based on managed installs. For more information on remote installs, see "Performing Remote Installs" (page 49).

Managed installs provide these features:

- An automated install experience for multicomponent products
- Support for upgrading your product, which may require replacing only certain components
- Support for custom installs, which allow users to decide what components to install and where to install them

On a more detailed level, managed installs provide fine control over the installation process, including:

- The ability to perform operations before installing, such as:
  - Making sure the target system meets specific criteria
  - Requiring administrative-user authentication before installing components at privileged locations
  - Performing install operations, such as quitting an application to be upgraded or launching daemons (faceless applications)
- Control over details such as whether an install:
  - Allows the user to specify an alternate installation destination
  - Recommends or requires restart, logout, or shutdown after completion
  - Uses the ownership and access permissions of the user installing the product or those specified in the installation package

There are three ways of defining a managed install:

- **Distribution packages** let you define the complete install experience of your product. They also provide you with a great deal of flexibility for defining the install choices users use to customize an install. Distribution packages offer you and the users of your product the best installation solution for Mac OS X–based products. Distribution packages, however, can be installed only on computers running Mac OS X v10.4 and later.

- **Metapackages** provide some of the features distribution packages provide but can be installed on computers running Mac OS X v10.2 and later.
- **Component packages** contain a single product component. They are usually included as part of a distribution package or metapackage but can also be installed individually in computers running Mac OS X v10.2 and later.

The following sections describe the major elements of managed installs and some of their limitations.

Packages

The central part of a managed install is the **installation package**, which contains your product and installation information. The following sections describe installation packages and the various types of packages you may need to create when developing a managed install for a product.

What Is a Package?

An **installation package** (also known as a package) is a file package (a directory that appears in the Finder as a single file) created using the PackageMaker application (`/Developer/Applications/Utilities`). Packages contain a product or product component—the package's **payload**—to be installed on a computer, and install configuration information that determines where and how the product is installed.

Note: Application executables are usually enclosed in a **bundle**: a structured directory hierarchy that contains resources needed by the application, such as images and localized strings. Because these bundles are normally file packages, the term **application package** refers to an application bundle that the Finder displays as a single file. However, in this document the term package refers to an *installation package*, not an application package. For detailed information on Mac OS X bundles, see *Bundle Programming Guide*.

Packages have the extension `.pkg` or `.mpkg`. When a user double-clicks a package in a Finder window, the Installer application opens the package and walks the user through the installation process.

A package can specify details about four aspects of the package itself and its payload:

- **Product information:**

  Title
  Description
  Welcome file
  Read Me file
  License file
  Conclusion file

- **Package properties:**

  Package identifier
  Package version number
  Resource fork processing

- **Installation properties:**

  System requirements
  Volume requirements
  Authentication requirement
  Allowance for choosing an installation volume other than the boot volume
  Installation destination on the installation volume
  Relocation consent (the ability user may have to change the installation destination)
  Revert consent
  Directory-permissions overwrite
  Postinstallation process action

- **Install operations:**

  Preflight
  Preinstall/Preupgrade
  Postinstall/Postupgrade
  Postflight

Component Packages

A complex product, such as the Levon product introduced in "Overview of Software Delivery" (page 9), is made up of distinct components. Except for single-component products, **component packages** are used in conjunction with metapackages or distribution packages (described later in this chapter) to create an install experience. Specifically, component packages:

- Are the basis for the mechanism that allows you to provide users a way to specify which components to install (for example, a user may not want to install a product's tutorial files)
- Let you identify required components (which must be installed) and specify the locations of components to be installed at specific locations on the installation volume
- Allow you to specify system and volume requirements for the component using executable files (see "Specifying System and Volume Requirements in Pre-Tiger Systems" (page 53))

Component packages have the extension `.pkg`. Each component package contains a single product component and specifies product information, package properties, installation properties, and install operations.

Note: During an install of a metapackage (described in "Metapackages" (page 20)), the product information for the contained component packages is not used or shown to the user. The product information for a component package is used only when installing the component package by itself, with no enclosing metapackage. Therefore, product information on a component package is generally used only by someone creating a metapackage or a distribution package containing that component package as they define the product information for the containing package.

Component packages can be installed on their own or as part of the install of a multicomponent product. After the payload of a component package is installed, Installer places a receipt in the `/Library/Receipts` directory of the installation volume. An **installation receipt** is a token that Installer uses to determine whether a package has already been installed on a system. As long as the receipt is present, subsequent installs of packages using the same package filename on the same volume are processed as upgrades.

Metapackages

A **metapackage** is an installation package that contains other installation packages. The enclosed packages can be component packages or metapackages (but not distribution packages).

Metapackages allow you to define a simple install experience for a multicomponent product. When users open a metapackage with the Installer application, they can choose to install only the components they need. Each enclosed package becomes an **install choice**. For example, if a product includes a tutorial-files component that the user performing the install doesn't need, they can choose not to install that component.

Metapackages have the extension `.mpkg`. Table 4-1 shows what aspects of a metapackage and the packages (component packages and other metapackages) it contains are used in the installation process.

Table 4-1 Installation process of a metapackage

	Product information	Package properties	Installation properties	Install operations
Metapackage	☑	☑	☑	☑
Contained packages	☑	☑	☑	☑

Table 4-1 indicates that the containing metapackage specifies package properties, installation properties, and specify install operations. However, the only aspect of the metapackage used in the installation process is its product information. Conversely, for the packages the metapackage contains, all aspects except product information are used in the installation process.

Compatibility: Metapackages can be installed on computers running Mac OS X v10.2 (Jaguar) or later.

Distribution Packages

A **distribution package** is a metapackage that specifies both product and installation information for a product. Distribution packages provide more sophisticated facilities to tailor the installation process. The major features distribution packages provide are:

- Definition of the entire install experience in one place instead of having it spread out through several component packages
- Definition of system and volume requirements using a requirements editor instead of executables
- Install choices can contain more than one component package
- Users can choose an installation destination for each install choice instead of for the entire install
- Installer loads a distribution package for a multicomponent product with many component packages (and presents the user with the install experience it specifies) faster than a metapackage containing the same product

The central part of a distribution package is the **distribution script**. This is a JavaScript-based script file that contains all the information that defines an install experience. When you create a distribution package using PackageMaker, the package's distribution script is created for you.

Distribution packages differ from metapackages in these areas:

- They can be installed only on computers running Mac OS X v10.4 (Tiger) and later.
- They must contain only component packages, not metapackages or distribution packages.
- Installer ignores installation properties specified in the contained component packages (installation properties are specified by the distribution script).

Table 4-2 shows what aspects of a distribution package and the component packages it contains are used in the installation process.

Table 4-2 Installation process of a distribution package

	Product information	Package properties	Installation properties	Install operations
Distribution package	☑		☑	
Contained packages	☑	☑	☑	☑

Table 4-2 indicates that a distribution package specifies only product information and installation properties, and both aspects are used in the installation process. The contained packages may specify all package aspects, but only their package properties and install operations contribute to the installation process.

Important: The system and volume requirements for a distribution package must reflect all the system and volume requirements of each of the component packages the distribution package contains.

Compatibility: Distribution packages can be installed on computers running Mac OS X v10.4 or later.

System and Volume Requirements

Two of the installation properties you can specify in a package are system requirements and volume requirements. These two properties define criteria the installation host must meet in order for the installation process to proceed.

- **System requirements** specify criteria the computer or operating system must satisfy. There are two types of system requirements: recommended and required. If the host doesn't meet a required system requirement, the installation process is canceled.
- **Volume requirements** specify criteria each of the host's volumes must meet in order to be considered a valid installation volume.

The Installation Process

After the Installer application opens a package, it performs the installation process in several phases:

- **Requirements Check**

  Installer ensures that the installation host meets the system and volume requirements specified by the package.

- **Preinstall**

  Installer runs `preflight` and `preinstall`/`preupgrade` executables. If an executable returns anything other than `0`, the install is cancelled.

- **Install**

  Installer extracts the payload of component packages and copies it to the appropriate destinations.

- **Save Receipt**

  Installer copies the component package file (with its payload stripped) to the `Library/Receipts` directory in the installation volume.

- **Postinstall**

  Installer runs `postinstall`/`postupgrade` and `postflight` executables.

The following sections detail the operations that the Installer application performs in the Requirements Check, Preinstall, and Postinstall phases of the installation process for the three types of installation package files. For details on the purpose of the executable files used to define installation requirements and install

operations (`InstallationCheck`, `VolumeCheck`, `preflight`, `preinstall`, `preupgrade`, `postinstall`, `postupgrade`, and `postflight`), see "Specifying Install Operations" (page 45) and "Specifying System and Volume Requirements in Pre-Tiger Systems" (page 53).

Component Package Installation Process

This is how Installer performs the Requirements Check, Preinstall, and Postinstall phases of a component package's installation process:

- **Requirements Check**

  `InstallationCheck`

  `VolumeCheck` on each available volume

- **Preinstall**

  `preflight`

  `preinstall` or `preupgrade`

- **Postinstall**

  `postinstall` or `postupgrade`

  `postflight`

Metapackage Installation Process

This is how the Installer application performs the Requirements Check, Preinstall, and Postinstall phases of a metapackage's installation process:

- **Requirements Check**

  `InstallationCheck` for each package but not the top metapackage

  `VolumeCheck` on each available volume for each package but not the top metapackage

- **Preinstall**

  `preflight` for top metapackage

  `preflight` for each package

  `preinstall` or `preupgrade` for top metapackage

  `preinstall` or `preupgrade` for each package

- **Postinstall**

  `postinstall` or `postupgrade` for each package

  `postinstall` or `postupgrade` for top metapackage

  `postflight` for top metapackage

  `postflight` for each package

Distribution Package Installation Process

This is how the Installer application performs the Requirements Check, Preinstall, and Postinstall phases of a distribution package's installation process:

- **Requirements Check**

  Installation Check script

  Volume Check script on each available volume

- **Preinstall**

  `preflight` for each package

  `preinstall` or `preupgrade` for each package

- **Postinstall**

  `postinstall` or `postupgrade` for each package

  `postflight` for each package

Note: Any executable-based install operations specified in the distribution package itself (not in the component packages it contains) are ignored by the Installer application when the distribution package is installed in computers running Mac OS X v10.4 and later. On earlier versions of the operating system, the Installer application invokes the executable-based install operations as was done for metapackages (see "Metapackage Installation Process" (page 23)). You can take advantage of this feature when creating a single managed install solution for Tiger and pre-Tiger systems. See "Creating a Hybrid Metapackage" (page 42) for more information.

The User Install Experience

The install experience that the Installer application shows users after they open an installation package has the following phases:

- **System Requirements**

  The first task Installer performs after opening a package is to ensure that the installation host meets the package's installation requirements. Unsatisfied nonfatal system requirements produce a warning in user-driven installs.

- **Authentication**

  When a package requires admin or `root` user authentication, Installer displays the Authentication dialog. Users must enter the user name and password of an administrative user on the system to perform the install.

- **Welcome**

  Installer always displays the Welcome page. To tailor the Welcome page, include a welcome file in the package.

- **Read Me**

If the package includes a Read Me file, Installer displays it in the Read Me page. The user can save or print this file.

- **License**

  If the package includes a license agreement file, Installer displays it in the License page. The user must accept the license in order to continue the installation process.

- **Volume Requirements**

  Installer checks each available volume to determine whether it meets the package's volume requirements. It uses the information gathered in this phase in the Volume Selection phase.

- **Volume Selection**

  Installer displays the Select Destination page if the package contains at least one package.

- **Customization**

  If the package allows both an easy install and a custom install, Installer displays the Installation Type page, which defaults to the easy install. The user can choose to perform the easy install or the custom install.

  Note: When installing a distribution package, Installer allows users to choose individual installation destinations for each relocatable choice. With a metapackage, the Custom Install pane doesn't allow users to choose install destinations for individual choices. With metapackages, Installer allows only one customized installation destination for all relocatable packages.

- **Install**

  When the user clicks Install (and after admin or `root` user authentication, if needed), Installer shows the Install page and performs the install.

- **Conclusion**

  Installer shows the Conclusion page when the installation process ends. In distribution packages, you can include a conclusion file that Installer shows instead of the default conclusion message.

Postinstallation Process Action

After an installation process is complete, the Installer application can recommend or mandate that the user log out of the system or restart or shut down the computer, depending on the most drastic postinstallation process action specified in the installation package. When specifying a package's postinstallation process action, you must take into account the nature of your product and the way it interacts with Mac OS X and other executables in the system.

Limitations of Managed Installs

There are some issues and limitations of managed installs:

- The Installer application does not support uninstalling products.
- Installer runs only on Mac OS X.

- Without proper care when specifying the ownership and access permissions of component files, it is possible to render a system unusable. Make sure you test all installer packages before shipping them to customers.
- Relocation does not work in installation hosts running Mac OS X v10.3.3 or earlier.

Packaging Product Components

After breaking down a product into distinct components (such as applications, frameworks, fonts, documentation, and so on), you create an installation package for each of the product's components. An **installation package** is a file package that contains one component and information about the package and its payload. See "What Is a Package?" (page 18) for more information about packages.

The Installer application can open a package and place its payload on a user's system. Installation packages whose payload is a component of a multicomponent product are known as **component packages**.

To create a component package, you perform the following tasks:

1. Categorize the component.
2. Create the component package project directory.
3. Add the component files to the project directory.
4. Add executables to the project directory if the component requires them.
5. Create the component package.
6. Test the component package.

The following sections describe these tasks in detail.

Categorize the Component

There are several types of product components available in Mac OS X. Some examples are application packages (file packages that contain an application's executable code and resources), frameworks (directories that contain shared executable code and resources), font files, and plug-ins. Categorizing a component helps you determine the appropriate installation destination for it. Table 5-1 shows some component categories and corresponding installation destinations. See *File System Overview* for detailed information.

Table 5-1 Component types and installation destinations

Type	Destination
Application	`/Applications`
Framework	`/Library/Frameworks`
Font	`/Library/Fonts`
Documentation	`/Library/Documentation`

Type	Destination
Command-line tool	`/usr/local/bin`

Table 5-2 shows the categorization of a multicomponent product.

Table 5-2 Categorization of the components of a multicomponent product, called Levon

Component	Component type	Installation destination
`Levon.app`	Application	`/Applications`
`Levon_User_Guide.pdf`	Documentation	`/Library/Documentation`
`SharedServices.framework`	Framework	`/Library/Frameworks`

Create the Component Package Project Directory

The component package project directory stores the component's files, the package project file, and other installation files. For easy identification, you should include the version number of the component in the name of the project directory. For example, Listing 5-1 shows the names of the project directories for the three components of the Levon product.

Listing 5-1 Component-package project directories for a multicomponent product

```
LevonApp-1.0.0/
LevonDoc-1.0/
SharedServicesFwk-1.0/
```

Add the Component Files to the Package Project Directory

Inside the package project directory, create a directory to hold the component files, as shown in the highlighted lines in Listing 5-2.

Listing 5-2 Adding component files to component-package project directories

```
LevonApp-1.0.0/
   component/
      Levon.app
LevonDoc-1.0/
   component/
      Levon_User_Guide.pdf
SharedServicesFwk-1.0/
   component/
      SharedServices.framework
```

Add Executable Files to the Package Project Directory

If your component requires tailored install operations or needs to specify system and volume requirements, add the `extras` directory to the package project directory containing the appropriate executables. Listing 5-3 highlights the `extras` directories in two package project directories.

Listing 5-3 Adding executables to component-package project directories

```
LevonApp-1.0.0/
   component/
      Levon.app
   extras/
      InstallationCheck
      VolumeCheck
      preflight
      postupgrade
LevonDoc-1.0/
   component/
      Levon_User_Guide.pdf
SharedServicesFwk-1.0/
   component/
      SharedServices.framework
   extras/
      InstallationCheck
      VolumeCheck
      preflight
      postinstall
      postupgrade
```

See "Specifying Install Operations" (page 45) and "Specifying System and Volume Requirements in Pre-Tiger Systems" (page 53) for details on install operations and executable-based installation requirements.

Create the Component Package

After you have created the package project directory, you use PackageMaker (`/Developer/Applications/Utilities`) to create the component package project file.

To create a component package in PackageMaker:

1. Create a single package project.
2. Enter the package's title and description.
3. Identify the directory that contains the component's files (for example, the `component` directory in the package project directory).
4. Edit the component directory's ownership and access permissions settings using the File Permissions editor.

The files that the Installer application places on the installation host have the ownership and access permissions specified in the component files in the package. Therefore, you must set up the owner and access permissions of component files appropriately before generating the installation package; otherwise, users may have difficulty manipulating those files after they are installed. In most cases, the owner should be `root` and the group `admin`. Use the permissions recommended in the File Permissions editor.

Overwriting directory access permissions: The Installer application assigns access permissions to directories that don't exist on the installation volume but are part of a product's structure. (These directories are specified in the component directory when you created the component package.) When a directory that's part of the component hierarchy already exists on the installation volume, its permissions are modified only if the package specifies that directory permissions must be overwritten. Because modifying the permissions of existing directories in Mac OS X—especially standard directories in the local and system domain—may impair the normal operation of the installation host, you should not generally have packages overwrite directory permissions. However, components that use an internal hierarchy that doesn't include Mac OS X standard directories may need to overwrite the access permissions of their internal directories to ensure a successful install.

To illustrate this situation, analyze the structure of the Nificky component directory, shown here.

```
component/
    Nificky/
        Nificky.app
        ReadMe.rtf
```

The installation destination for the Nificky application component is `/Applications`. Therefore, after the component is installed, the `Applications` directory on the installation volume contains the Nificky directory, which itself contains the `Nificky.app` file package and the `ReadMe.rtf` file. But, if `Applications/Nificky` already exists on the volume and its access permissions do not allow modification, the install fails. Specifying that the installation package for the Nificky application overwrite directory access permissions ensures that the Installer application modifies the permissions on `Applications/Nificky` to allow it to place files within the directory. The access permissions of the `Applications` directory are untouched.

Important: Never include Mac OS X standard directories inside a component's hierarchy. That is, do not reflect any standard part of the Mac OS X file system (in any of the file system domains) inside your component's structure.

5. Set the installation destination for the component, and specify an appropriate authentication level and postinstallation process action, if needed.

Installing into the user's home directory: One way to place files into a user's home directory is to specify `/Users/Shared` as the installation destination and use an install operation to move the product files into the current user's home directory (specified by the `$HOME` environment variable). See "Specifying Install Operations" (page 45) for details.

6. Identify the subdirectory in the project directory that contains executable files that specify installation requirements or install operations, if needed.

7. Set the package identifier and version number.

A package's identifier and version number help the Installer application determine the best way to update a product's files during an install on a system on which the package has already been installed. You must set these two package properties for every package you create.

For applications and frameworks, use the bundle identifier (`CFBundleIdentifier`) specified in their `Info.plist` files as a starting point. For example, if the bundle ID of an application is `com.mycompany.Levon`, the package identifier should be `com.mycompany.Levon.pkg`. Keep in mind that the package identifier must be unique. (For detailed information on the `CFBundleIdentifier` and other `Info.plist` keys, see *Runtime Configuration Guidelines*.)

Note: The `Info.plist` file is located in the `Contents` directory of application and plug-in file packages (use the Finder Show Package Contents command to reveal the contents of file packages). In a framework, the `Info.plist` file is located in the Resources directory.

The package version for application, framework, and plug-in component packages should be the value for the `CFBundleShortVersionString` property of the component they contain. In framework component packages, use the value for the `CFBundleVersion` property. For other component types, choose an appropriate package identifier and package version number.

Important: Installer does not use the identifier and version information specified in the `Info.plist` files of a package's payload.

After you've defined a package, use the PackageMaker Build command to generate the component package (`.pkg`) file. You can also save the package project file in the project directory for future use.

Test the Component Package

Before adding a component package to a metapackage or a distribution package, it's good practice to ensure that the package installs correctly on its own. You accomplish this task by opening the package in the Installer application and performing the installation process, confirming that the component files are installed correctly.

Defining a Managed Install

With a set of component packages, you have the essential ingredients for developing an install experience for the users of your product. There are two mechanisms for creating an install experience: using metapackages and using distribution packages, described in "Overview of Software Delivery" (page 9).

Distribution packages are a major improvement over metapackages because, using distribution packages, you can separate the configuration of a package's payload from the definition of the install experience it provides.

Compatibility: Distribution packages can be installed only on Mac OS X v10.4–based systems. Metapackages can be installed in computers running Mac OS X v10.2 and later.

To support Jaguar and Panther users while at the same time providing Tiger uses the enhanced install experience distribution packages offer, you can create **hybrid metapackages**. These are metapackages into which you copy the distribution script from a corresponding distribution package.

As described in "What Is a Package?" (page 18), an installation process is defined through four aspects of installation packages: product information, package properties, installation properties (which include system and volume requirements), and install operations. Metapackages and distribution packages add an additional aspect: **install choices**. Install choices allow users to customize an install to, for example, prevent the installation of a product's tutorial component.

The following sections describe in detail how to create compelling install experiences using metapackages, distribution packages, and hybrid metapackages.

Creating a Metapackage

To create a metapackage, you perform the following tasks:

1. Create the metapackage project directory.
2. Create the metapackage file.

The following sections describe these tasks in detail.

Create the Metapackage Project Directory

To facilitate the creation of an installation package, you should create a metapackage project directory to store all the files you need in the process. The metapackage project directory contains directories that hold the product's component packages, product information files, executable-based installation requirements, and install-operation files. Listing 6-1 shows a metapackage project directory.

Listing 6-1 A metapackage project directory

```
LevonMeta-1.0.0/
    packages/
        Levon.pkg
        LevonDoc.pkg
        SharedServices.pkg
    product_info/
        Welcome.rtf
        ReadMe.rtf
        License.rtf
        Conclusion.rtf
    extras/
        InstallationCheck
        VolumeCheck
        preflight
        postinstall
        postupgrade
```

Create the Metapackage File

A metapackage project specifies the following items:

- **Title:** The name used to identify this metapackage to the user during installation.
- **Description:** This package's description, displayed by the Installer application in the Customization pane for an enclosing metapackage when the user highlights this metapackage.
- **Packages directory:** The directory that contains the packages this metapackage contains.

  The directory may reside inside or outside the generated metapackage, depending on whether you want to discourage users from installing packages individually. If the directory resides inside the metapackage, you must copy the component packages from the packages directory in the metapackage project directory to the directory you designated as the package holder in the metapackage.
- **Extras directory:** The directory that contains executable files that specify installation requirements and install operations.

To define a metapackage-based installation process, use PackageMaker to:

1. Create a metapackage project.
2. Add product information files to the project by using Installer Interface Editor.
3. In the metapackage project directory, identify the subdirectory that contains executable files specifying installation requirements and install operations.
4. Add install choices by adding the packages in the `packages` subdirectory of the project directory to the metapackage project's packages list.

   Specify the initial selection state of each choice. Choices you identify as required are selected in the Installer Custom Install pane, but the user cannot deselect them.
5. Build the metapackage.

If you choose to place the metapackage's packages inside the metapackage itself (mainly to discourage the individual installation of product components), copy the packages to the appropriate directory in the generated metapackage file. (You can reveal the contents of the metapackage in the Finder using the Show Package Contents command.)

Creating a Distribution Package

Distribution packages provide a more direct, JavaScript-based mechanism for developing an install experience. One of the benefits a JavaScript-based install experience provides is the ability to update install choices dynamically, in response to a user's choice selection. In addition, system and volume requirements are also specified using JavaScript code. Other benefits of using distribution packages are that:

- Users can specify custom installation destinations for relocatable choices individually, rather than as a group.
- Using JavaScript code virtually eliminates the need for the Installer application to display a dialog informing users that it needs to run an external program as part of the installation process.
- You specify the installation requirements for a product in one place instead of across its component packages. This centralization makes it easier to set up and change a product's requirements.

Note: When a user installs a distribution package, the `InstallationCheck` and `VolumeCheck` files in the contained packages are not executed. This is the opposite of what happens when a user installs a metapackage.

To create a distribution package, you perform the following tasks:

1. Create the distribution package project directory.
2. Create the distribution package file.

These tasks are described in the following sections.

Create the Distribution Package Project Directory

The distribution package project directory contains directories that hold the product's component packages and product information files (background image, Read Me file, and so on). Listing 6-2 shows a distribution package project directory.

Listing 6-2 Distribution package project directory

```
LevonDist-1.0.0./
    packages/
        Levon.pkg
        LevonDoc.pkg
        SharedServices.pkg
    product_info/
        Welcome.rtf
        ReadMe.rtf
```

```
License.rtf
Conclusion.rtf
```

Create the Distribution Package File

PackageMaker allows you to specify the product information, installation properties, and install choices that define a product's installation process. Package properties and install operations are specified by each package contained in a distribution package.

As you do with metapackages, you use Installer Interface Editor to specify product information. The Requirements Editor window allows you to specify JavaScript-based installation requirements.

You create install choices using a hierarchical interface. Each install choice can have one or more packages associated with it. To create choice groups, you add choices to an existing choice.

Just as when the Installer application processes a metapackage-based custom install, when the user selects a choice in a custom install, the packages the choice represents are installed. The packages of unselected choices are not installed. However, distribution packages provide additional granularity by allowing you to specify disabled unselected choices (which the user cannot select). You can also define invisible choices (whose existence is not revealed to the user). Three attributes specify the selection and visibility state of an install choice:

- **Selected:** Determines whether the choice is selected.
- **Enabled:** Determines whether the choice's selected state is modifiable by the user.
- **Visible:** Determines whether the choice appears in the Installer Custom Install pane.

You can specify initial values for each of the selection and appearance attributes. But you can also use JavaScript code to compute the initial values before Installer displays the Custom Install pane (see "The User Install Experience" (page 24) for more information). For example, if a component requires the presence of software that is unavailable in the installation host, you can make the install choice for that component invisible. You can also specify dynamic values for the choice attributes. This capability allows you to, for example, select or deselect options in response to the user's actions. You can take advantage of this flexibility in products with interrelated components in which option groups provide no appropriate selection mechanism. For example, a product with a plug-in component that requires a font component could be automatically deselected when the user deselects the option that installs the font package.

These are the tasks you perform to create a distribution package:

1. Create a distribution project.
2. Add product information files to the project.
3. Specify system and volume requirements.

   Important: The system and volume requirements for a distribution package must include all the system and volume requirements of each of the component packages the distribution package contains.

4. Configure install choices.

In PackageMaker, create a distribution project and perform the tasks described in the following sections.

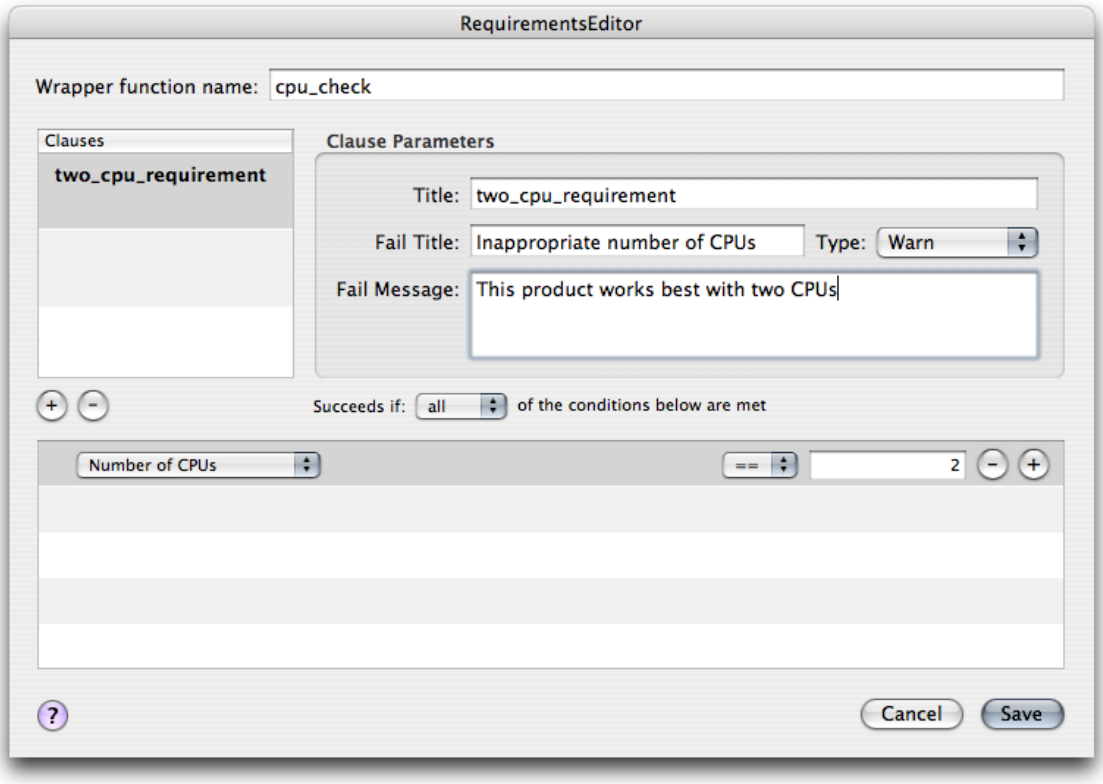
Add Product Information Files

Use Installer Interface Editor to add project information files to the package. These files may include a background image and a Read Me file.

Specify System and Volume Requirements

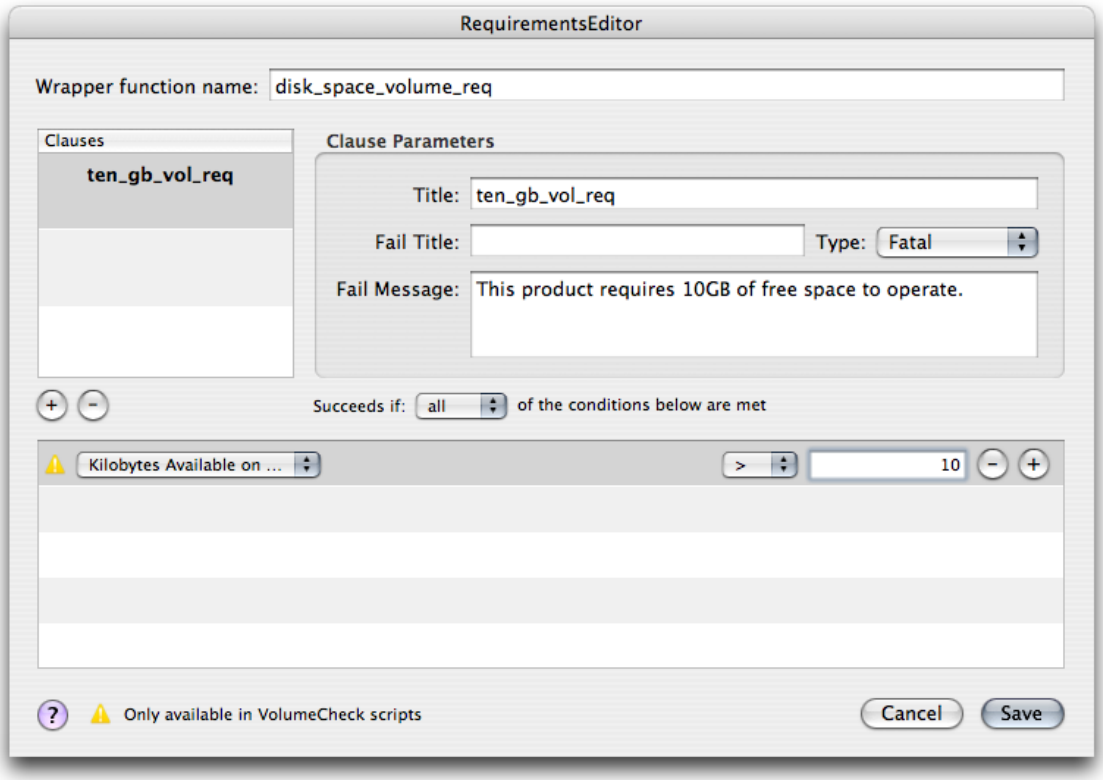
You can specify system and volume requirements using the requirements editor. You first need to add a global function to the project. When editing the function, use the requirements editor to specify an installation requirement. For example, Figure 6-1 shows the definition of a system requirement.

Figure 6-1 Defining a system requirement in a distribution package



The requirements editor provides access to several system and volume properties. For example, Figure 6-2 shows the specification for a volume requirement.

Figure 6-2 Defining a volume requirement in a distribution package



Before this volume requirement can work, however, you need to modify the JavaScript code generated by the requirements editor. Modify the code as indicated in Listing 6-3.

Listing 6-3 JavaScript code for a volume requirement

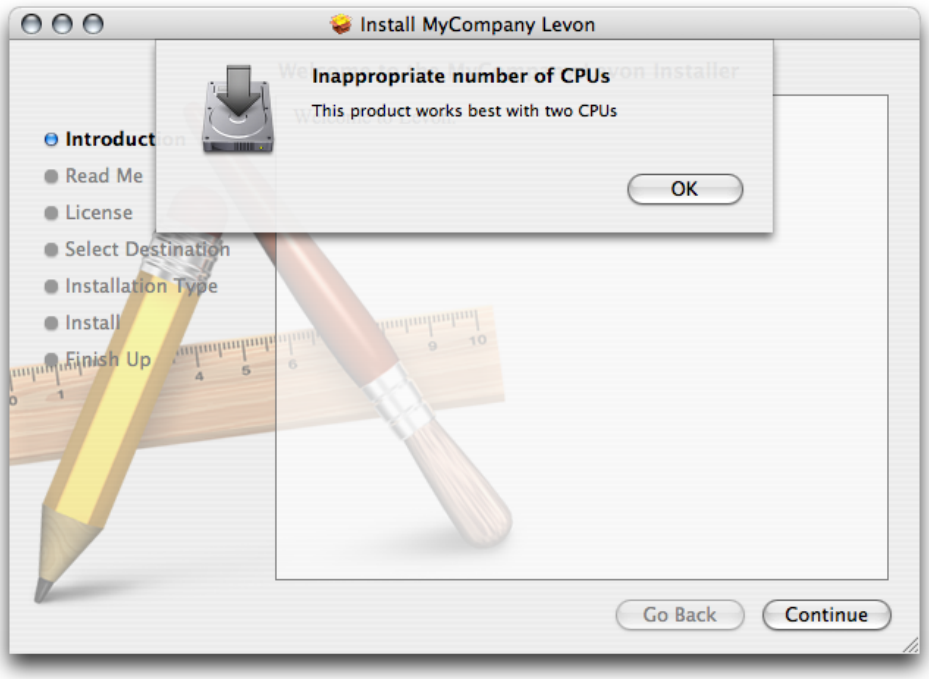
```
/* js:pkmk:start */
function disk_space_volume_req() {
     return  ten() ;
}
/* js:pkmk:end */


/* js:pkmk:start */
function ten() {
    var result = false;
    try {
        result = my.target.kilobytesAvailable > 10;                 // Remove
        result = my.target.availableKilobytes > 10*1024*1024*1024;  // Add
    } catch (e) {}

    if(!result) {
        my.result.type = 'Fatal';
        my.result.title = '';
        my.result.message = 'This product requires 10GB of free space to
operate.';
    }
    return result;
```

```
}
/* js:pkmk:end */
```

After the functions are defined, choose them as the system requirement and volume requirement scripts in the distribution package project.

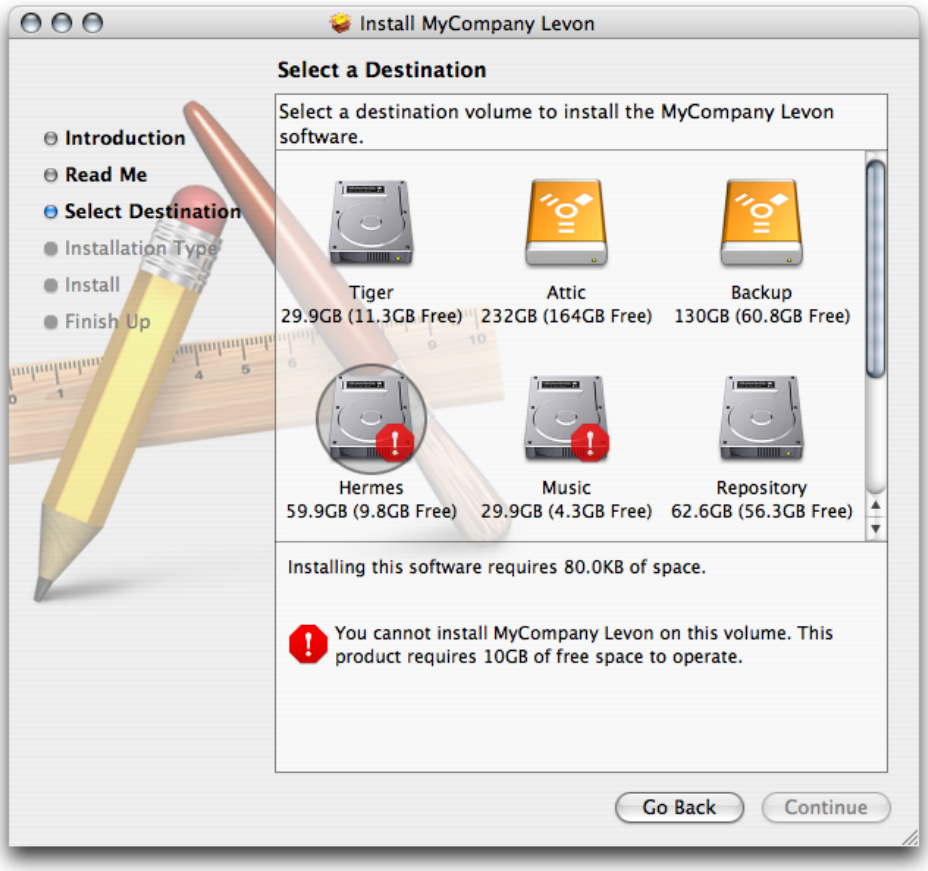
Installer checks system requirements as soon as it opens a distribution package. Figure 6-3 shows what users see when their systems don't meet that requirement.

Figure 6-3 Installer informs users of an unsatisfied system requirement



If any package in the distribution package does not specify that it's a boot-volume-only install, Installer displays the Select Destination page. Just before displaying this page, however, Installer runs the volume requirement script against all the available volumes in the system. Volumes that do not meet the requirement are badged. When the user selects such a volume, Installer displays the failure message in the volume requirement specification, as shown in Figure 6-4. The user can continue only after selecting a volume that meets the requirements.

Figure 6-4 Installer indicates which volumes do not meet requirements



Configure Install Choices

When Installer first displays the Custom Install pane, it uses the attributes that specify the initial state of each option. As the user selects or deselects choices, Installer applies the attributes that specify dynamic state to each choice.

Attribute clauses are JavaScript Boolean expressions. You can use `true`, `false`, or a Boolean expression that may include function invocations. You specify such functions as a global script of the project, similar to the way you specify installation requirements. If you use the requirements editor to generate the JavaScript code for such functions, the type of the requirement must be None.

Important: Although the requirements editor is accessible when you edit attribute clauses, you must not use it. Attribute clauses are not complete JavaScript scripts, which is what the requirements editor generates. Functions for use as part of attribute clauses must be defined as global scripts in the distribution project.

In addition to the visibility attributes, each choice has the following properties:

- **Title**

  Users see this title in the Custom Install pane.

- **Identifier**

You use this identifier to access a choice's attributes.

- **Description**

  Users see this description when they highlight an option in the Custom Install pane.

- **Custom Location**

  The initial installation destination of the choice's packages if you want users to be able to specify a custom installation destination. Leave empty if you don't want users to choose a different installation destination for the choice's packages.

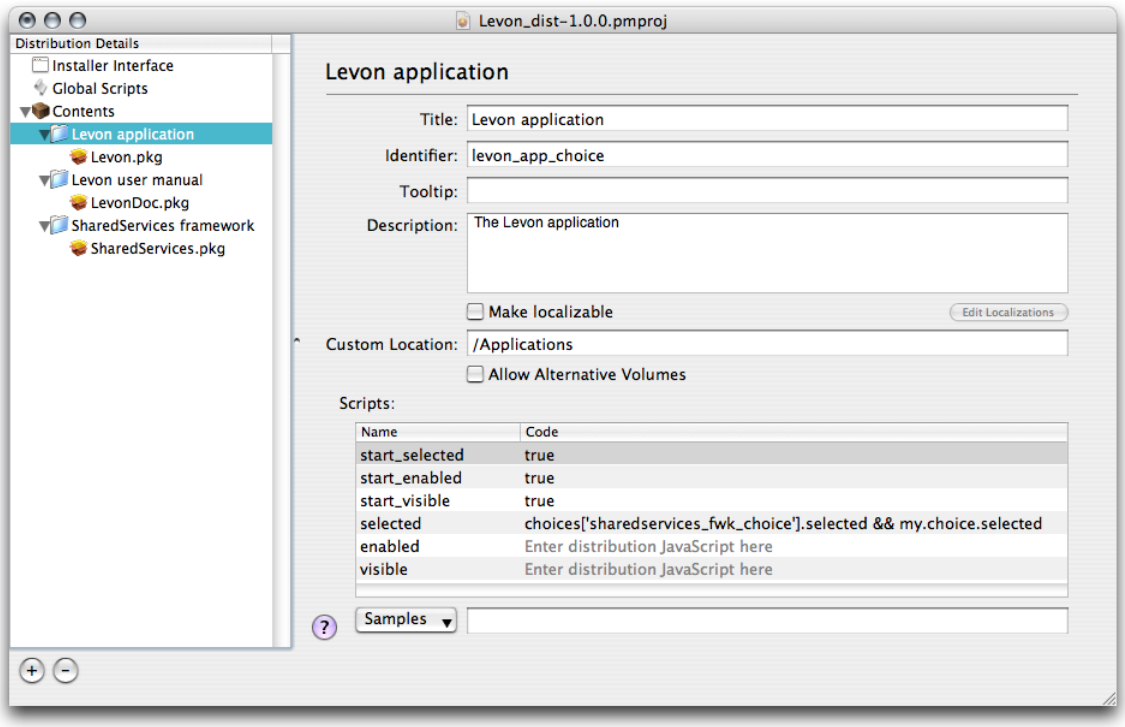
- **Allow Alternate Volumes**

  Specifies whether users can choose a volume other than the boot volume for the install.

After adding an install choice to the distribution project, you associate one or more packages with it. The Installer application installs these packages if the choice is selected in the Custom Install pane when the user clicks Install. Otherwise, these packages are not installed. To associate a package with a choice, drag the package from a Finder window to the choice in the distribution project window, or select the choice in the project and use the Add Package command.

Figure 6-5 shows the definition of the "Levon application" install choice for the Levon product.

Figure 6-5 Defining an install choice for a distribution package



The user can choose to install the choice's package in a location other than `Applications` on a volume other than the boot volume. This choice is also selected the first time the user sees the customization pane. However, its selection state is tied to the selection state of the "SharedServices framework" choice. If the user deselects the framework choice, the application choice becomes deselected.

You use the expression `choices['<choice_identifier>'].<attribute>` to access the attributes of other choices. Therefore, the first part of the clause for selected attribute of the "Levon application" choice evaluates to `true` if the user selects the "SharedServices framework" choice. That is, `choices['sharedservices_fwk_choice'].selected` evaluates to `true`. The second part of the clause, `my.choice.selected`, is needed because Installer—when the user selects or deselects a choice—evaluates the attributes of all install choices except the choice the user changed.

For each of a choice's packages, you can specify its authentication requirement and postinstall action.

After you've configured the install choices, build the distribution package using the PackageMaker Build command.

Creating a Hybrid Metapackage

Although distribution packages are supported only on Mac OS X v10.4 and later, if your product supports Mac OS X v10.2 and v10.3, you can create a single delivery solution that works on Tiger and pre-Tiger systems. Such a solution is called a **hybrid metapackage**.

To create a hybrid metapackage, follow these steps:

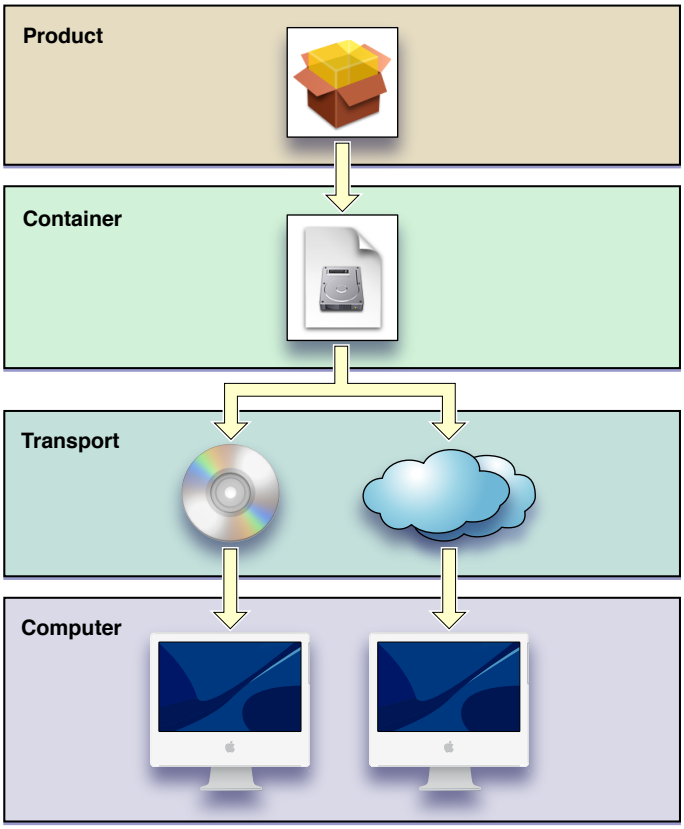
1. Create component packages for your product's components, as described in "Packaging Product Components" (page 27).
2. Create a metapackage for your product, as described in "Creating a Metapackage" (page 33).

   Place the component packages inside the metapackage file at `./Contents/Packages`.
3. Create a distribution package using the same packages you used in step 1, as described in "Creating a Distribution Package" (page 35).
4. Copy the `distribution.dist` file from the distribution package file to the `Contents` directory of the metapackage file.

Placing a Packaged Product in a Container

If you do not have access to the computers of your product's users, you must place your packaged product in a container to facilitate its delivery to your customers, as illustrated in Figure 6-6. "Product Containers" (page 11) provides information on how to place products in containers.

Figure 6-6 Delivering a packaged product using a disk image



Testing the Install Experience

Before shipping a product to users, you should ensure that the install experience is what you expect and that the product's files are installed correctly. You should perform this test in systems that represent the systems your customers use while logged in as a nonadministrative user. In this way, you can make sure that the Installer application asks for authentication when a package requires it and that the installed component files have the appropriate ownership and access-permissions settings.

Specifying Install Operations

In managed installs, install operations allow you to configure the destination environment before the payload is copied to the file system and to perform additional processing afterward. To specify install operations, you use executable files (known as **install operation executables**) that Installer invokes at specific stages during an install, as described in "The Installation Process" (page 22).

Install operation executables must be named according to the install operation you want to define. The files can be binary files or text files containing shell scripts. All install operations are optional. You define only the operations required by a packaged product.

This chapter shows how you use install operation executables to define the install operations the Installer application performs in a managed install.

Overview of Install Operations

After the Installer application finishes checking installation requirements, it performs an install through distinct operations, known as **install operations**. You can define all but one of these operations, which copy payloads to their installation destinations. You should not use install operations to fix install problems, such as incorrect ownership and access permissions. You should use install operations only when other managed-install features, such as installation requirements, are not adequate for the chore you need to perform as part of installing a package or metapackage.

Table 7-1 lists the install operations in the order Installer performs them.

Table 7-1 Install operations and executables

Install operation	Operation executable	Description
Preflight	`preflight`	Prepares the target system for the install; for example, quitting or stopping specific applications or processes.
Preinstall	`preinstall`	Prepares the target system for a payload for which no receipt is found.
Preupgrade	`preupgrade`	Prepares the target system for a payload for which a receipt is found.
Payload Drop	None	Copies the payload to the installation destination. This operation is not modifiable.
Postinstall	`postinstall`	Cleanup or system setup for an installed payload.
Postupgrade	`postupgrade`	Cleanup or system setup for an upgraded payload.

Install operation	Operation executable	Description
Postflight	`postflight`	Install postprocessing; for example, setting up `cron(8)` jobs or launching the installed application.

The first three install operations, Preflight, Preinstall, and Preupgrade, can stop an install. When one of the corresponding executables returns anything other than `0`, Installer cancels the install.

Important: For install operations to work, operation executables must have their executable bit set. PackageMaker does this automatically when it builds a package. Also, install operation executables must not have a user interface of any kind or affect the installation process by any means other than through return values.

Arguments and Environment Variables for Install Operations

The following list describes the arguments and environment variables available to install operation executables. Note that not all environment variables are available to all executables (see Table 7-2 (page 47)).

`$1`: Full path to the installation package the Installer application is processing. For example:
`/Volumes/Users/michelle/Desktop/Levon.mpkg`

`$2`: Full path to the installation destination. For example:
`/Applications`

`$3`: Installation volume (or mountpoint) to receive the payload. For example:
`/`
`/Volumes/Tools`

`$4`: The root directory for the system:
`/`

`$SCRIPT_NAME`: Filename of the operation executable. For example:
`preflight`

`$PACKAGE_PATH`: Full path to the installation package. Same as `$1`.

`$INSTALLER_TEMP`: Scratch directory used by Installer to place its temporary work files. Install operations may use this area for their temporary work, too, but must not overwrite any Installer files. The Installer application erases this directory at the end of the install. For example:
`/private/tmp/.Levon.pkg.897.install`

`$RECEIPT_PATH`: Full path to a temporary directory containing the operation executable. This is a subdirectory of `$INSTALLER_TEMP`. This location may vary between installs. The executable can use this path to locate other files in the package. For example:
`/private/tmp/.Levon.pkg.897.install/Receipts/Levon.pkg/Contents/Resources`

The four arguments described earlier are available to all install operations. Table 7-2 shows the availability of the environment variables.

Table 7-2 Environment variables in operation executables

Operation executable	Available environment variables
`preflight`	None.
`preinstall`, `preupgrade`	`$SCRIPT_NAME`, `$PACKAGE_PATH`, `$INSTALLER_TEMP`.
`postinstall`, `postupgrade`, `postflight`	`$SCRIPT_NAME`, `$INSTALLER_TEMP`, `$PACKAGE_PATH`, `$RECEIPT_PATH`.

Example: Install Operation Script

Listing 7-1 shows a postflight operation implemented as a shell script that launches an installed application.

Listing 7-1 Sample install operation script

```
#!/bin/sh
echo $SCRIPT_NAME: launching Levon.app
open -b com.mycompany.Levon
exit 0
```

After Installer executes this install operation script, its log shows an entry similar to the one in Listing 7-2.

Listing 7-2 Sample Installer log entry

```
Jun 20 13:30:03 Athene : postflight[2274]: postflight: launching Levon.app
```

Performing Remote Installs

Apple Remote Desktop allows you to install products on multiple client computers from an administrator computer. This type of install is known as a **remote install**. You can perform remote installs immediately or schedule them for later completion.

Remote installs are based on managed installs (described in "Managed Installs" (page 17)). Therefore, the products to be installed on client computers must be packaged as component packages, metapackages, or distribution packages. If the product you want to install remotely is not packaged or if you want to repackage an existing product, you need to create a package for it first. See "Packaging Product Components" (page 27) and "Defining a Managed Install" (page 33) to learn how to create packages.

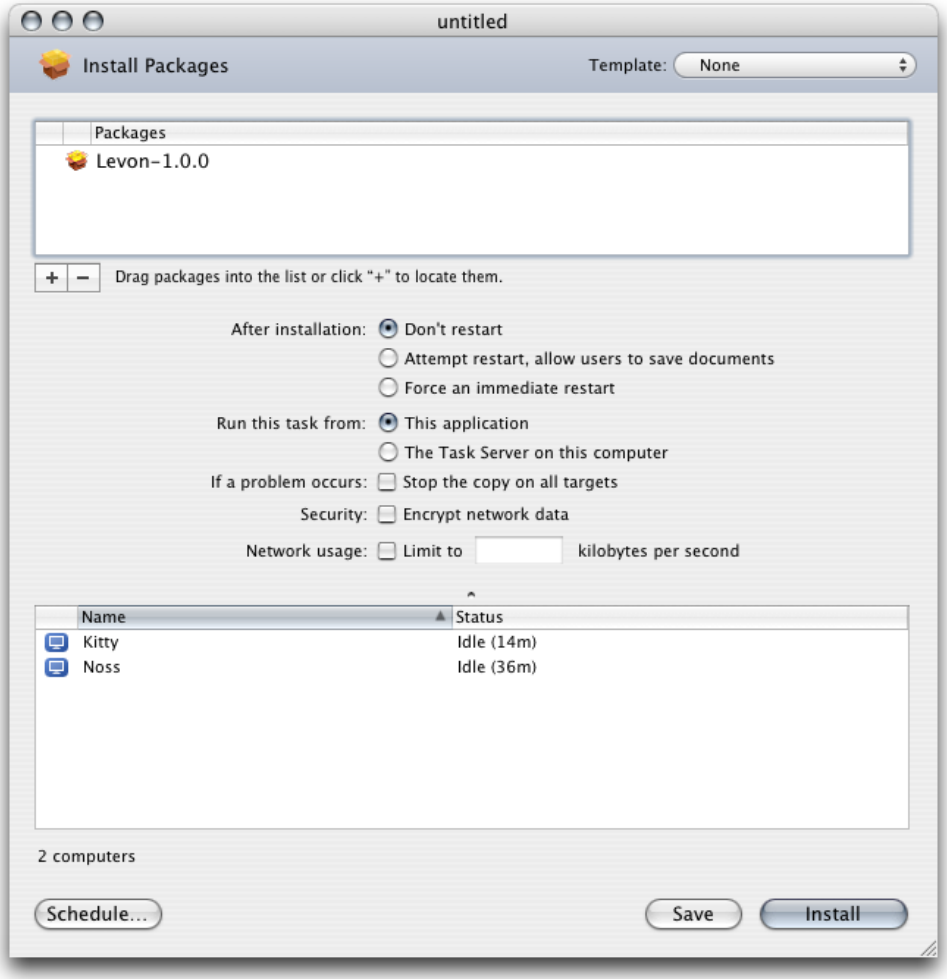
Note: Before installing a package on multiple computers using a remote install, you should install the package on a single computer using the Installer application to familiarize yourself with the package's install experience and to learn whether the package has installation requirements that similarly configured computers do not meet.

Follow these steps to install a package on multiple clients from an administrator computer (see *Apple Remote Desktop Administrator's Guide Version 3.2* for more details):

1. In Remote Desktop, select the computers onto which you want to install the package.
2. Choose Manage > Install Packages.
3. In the Install Packages task window, add the packages you want to install to the package list.

4. Select an appropriate postinstallation process action. Figure 8-1 shows the definition of an Install Packages task.

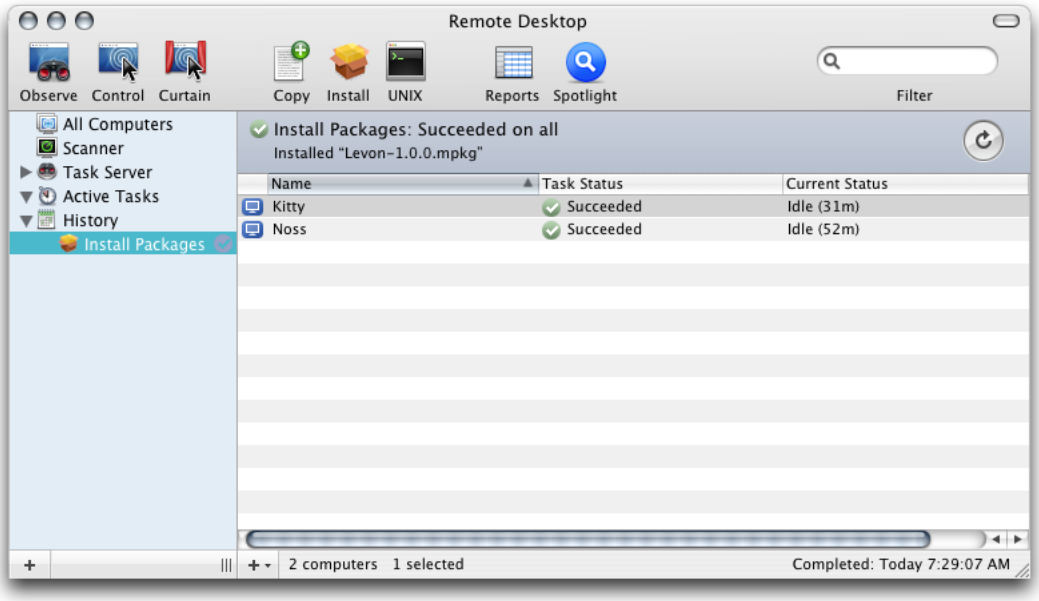
Figure 8-1 Remote Desktop Install Packages task



5. Click Install.

Figure 8-2 shows the result of a successful remote install.

Figure 8-2 Successful Install Packages task in Remote Desktop



When an install fails in any of the clients specified in and Install Packages task, you may need to copy the package to the client computer using a Remote Desktop Copy task and perform the installation by opening the package in Installer on the client.

Specifying System and Volume Requirements in Pre-Tiger Systems

JavaScript-based system and volume requirements are not supported on Pre-Tiger systems. To check installation requirements in computers running Mac OS X v10.2 or v10.3, you must use executable-based installation requirements. These installation requirements rely on two executable files that you create and include in an installation package.

You specify system and volume requirements using executables named `InstallationCheck` and `VolumeCheck`, respectively. These executables can be binary files or text-based scripts.

This chapter shows how to define executable-based installation requirements.

Overview of Executable-Based Installation Requirements

During the installation process, the Installer application invokes `InstallationCheck` and `VolumeCheck` when it needs to check system and volume requirements, respectively. See "The Installation Process" (page 22) for details.

Note: For executable-based installation requirements to work, the executables must have their executable bit set. PackageMaker does this automatically when it builds a package.

The `InstallationCheck` executable can use any criteria to allow or disallow an install. It returns a value that tells Installer whether to continue the installation process.

Similarly, `VolumeCheck` can use any criteria to accept or reject a volume as a potential installation volume. The return value of `VolumeCheck` tells Installer whether to allow the user to choose a particular volume as an installation destination.

In addition to stopping the installation process and disallowing volumes to be chosen as installation volumes, the return values of these executables tell Installer which informational message to display to the user. You use strings files to define these informational messages.

Strings Files for InstallationCheck and VolumeCheck Denials

When `InstallationCheck` cancels an install or when `VolumeCheck` disallows a volume to be chosen as the installation volume, the Installer application displays a default message or the message specified by the executable's return value. To specify your own messages, place the files `InstallationCheck.strings` and `VolumeCheck.strings` (for system requirement and volume requirement failure messages, respectively) in a localized directory inside the package's scripts directory. Table A-1 shows the directory names for the languages Installer recognizes.

Table A-1 Localized directories

Dutch.lproj
English.lproj
French.lproj
German.lproj
Italian.lproj
Japanese.lproj
Spanish.lproj
da.lproj
fi.lproj
ko.lproj
no.lproj
pt.lproj
sv.lproj
zh_CN.lproj
zn_TW.lproj

A strings file defines messages in the following format:

```
"<message_ID>" = "<message_string>";
```

`<message_ID>`: Integer between 16 and 31.
`<message_string>`: Informational message string.

Listing A-1 and Listing A-2 show sample `InstallationCheck.strings` and `VolumeCheck.strings` files, respectively.

Listing A-1 A sample `InstallationCheck.strings` file

```
"16" = "Can't install unless you have a Super Drive.";
"17" = "Can’t install unless you have more than 128MB of RAM.";
```

Listing A-2 A sample `VolumeCheck.strings` file

```
"16" = "Can't install on volumes named Panther.";
"17" = "Can’t install on volumes named Jaguar.";
```

InstallationCheck Messages

The `InstallationCheck` executable returns an integer that the Installer application uses to determine whether to proceed with the installation process or which informational message to display as the reason for canceling the install. To allow the install, return `0`. To cancel the process, return an integer that specifies the message ID of the localized message to display. This integer, however, must also have bits 5 and 6 set to `1`.

For example, to instruct Installer to display message 16, return `112`. You can use the Calculator application to determine the correct return value for a specific message ID, or compute it in your executable.

Table A-2 lists the message IDs of the default messages for `InstallationCheck` failures. If you use these messages, you don't need to create an `InstallationCheck.strings` file.

Table A-2 Installer default messages for `InstallationCheck` failures

Message ID	Message string
1	This software cannot be installed on this computer.
2	The software <package_name> cannot be installed on this computer.
3	<package_name> cannot be installed on this computer.
4	An error was encountered while running the InstallationCheck tool for package <package_name>.

VolumeCheck Messages

The `VolumeCheck` executable returns an integer that the Installer application uses to determine whether to allow the user to choose a volume as the installation volume and which informational message to display when the user selects a disallowed volume. Returning `0` allows the user to identify a volume as the installation volume. When a volume doesn't meet the volume requirements, the `VolumeCheck` return value specifies the message ID of the localized message to display when the user selects the volume. This integer, however, must also have bit 5 set to `1`.

For example, to disallow a volume and set the message ID of the appropriate message to `17`, return `49`. You can use the Calculator application to determine the correct return value for a specific message ID, or compute it in your executable.

Table A-3 lists the message IDs of the default messages for volumes that fail `VolumeCheck`. If you use these messages, you don't need to create a `VolumeCheck.strings` file.

Table A-3 Installer default messages for `VolumeCheck` failures

Message ID	Message string
0	You cannot install this software on this disk. You are not allowed to install the software on this disk for an unknown reason.

Message ID	Message string
1	You cannot install this software on this disk. Could not find specified message for index 1.
2	You cannot install this software on this disk. An error was encountered while running the VolumeCheck tool for package <package_name>.

Prebinding Applications

In Mac OS X v10.3.3 and earlier, when a Mach-O–based executable is launched, the dynamic link editor (or dynamic linker) loads the symbols that the executable imports at predetermined addresses in the executable's address space. Prebinding is the process of computing the addresses for the imported symbols, so that the dynamic linker needs to perform less work at launch time. In other words, the launch time of an executable is optimized when these precomputed addresses contain valid data. With outdated prebinding information, an executable takes longer to load. The dynamic linker Mac OS X v10.3.4 and later is implemented in a way that makes prebinding unnecessary. But applications that need to run in earlier versions of Mac OS X may benefit from having their prebinding information up to date.

When you build an application targeted at Mac OS X versions earlier than v10.3.4, the addresses of its imported symbols are computed using the SDK you choose for the project. For example, an application built using the 10.2.8 SDK that is installed on a computer running Mac OS X v10.2.3 would need to have its prebinding information recomputed in order to optimize its launch time. In managed installs, the Installer application automatically performs this update. In manual installs, however, you must perform this task.

The `update_prebinding(1)` command-line tool updates an executable's prebinding information. To optimize the launch time of a manually installed application, users need to run this tool after installing an application. You can provide instructions for how to run this tool in a Read Me file, printed documentation, or other mechanism.

For more information on prebinding, see *Launch Time Performance Guidelines*.

Preserving Resource Fork Data

In Carbon and pre–Mac OS X applications, application resources were stored in the resource fork of an application executable. In Mac OS X applications resources should be put in the data fork of a separate resource file, not the resource fork of the executable. The primary reason for moving application resources out of resource forks is to enable applications to be seamlessly moved around other file systems without loss of their resources; this would include transfer mechanisms such as BSD commands, FTP, email, and Windows and DOS copy commands. Most other computing environments, including the web, recognize single-fork files only and tend to delete the resource fork of HFS and HFS+ files.

When you package software with PackageMaker, you supply a location from which to gather the files to be packaged. If any of these files (such as a Classic application) has a resource fork, PackageMaker supports splitting the file before packaging. Such split files will be reassembled by the Installer application when the package is installed. Although splitting files with resource forks is optional, if you do not split them, the resource fork data will be lost and the file will be unusable.

Because resource fork splitting changes the files that are split, it is recommended that you operate on copies of these files while creating packages.

Compatibility: Support for preserving resource fork data was added to PackageMaker and Installer for Mac OS X v10.2.

Glossary

application package A file package containing the code and other resources that make up a Mac OS X application. Application packages make it easy for users to move applications around their file systems.

bundle A structured directory hierarchy that stores files in a way that facilitates their retrieval. Bundles are used extensively in Mac OS X; in particular most application executables are enclosed in bundles together with the resources the application needs to operate.

container A file-based enclosure for a product that facilitates delivery to its users. Disk images installation packages, and ZIP archives are the most popular product containers.

component A part of a software product that resides at a distinct location on the file system. See also component package.

component package An installation package whose payload is one of the components of a product.

custom install A metapackage or distribution package install that a user performs after modifying the default option selection.

delivery vehicle Transport used by users of a product to obtain the product's files. These include optical media and the Internet.

disk image A file-based enclosure that facilitates the transport of a directory structure on the Internet. Disk images can also be compressed to allow a product's files to be placed on optical media.

distribution package A metapackage that contains a distribution script that specifies the install experience for a product. Distribution packages provide a streamlined packaging experience for developers and an enhanced install experience for users. See also distribution script;metapackage.

distribution script An XML file with the extension `.dist` that contains all the information that defines an install experience in a distribution package. See also distribution package.

easy install A metapackage or distribution package install that a user performs using the default option selection.

file package A directory (often a bundle) that appears as a single file in Finder windows. See also bundle; file package.

hybrid metapackage A metapackage that contains a distribution script. This type of installer package behaves as a distribution package when installed on computers running Mac OS X v10.4 and later. On computers running earlier versions of the operating system, a hybrid metapackage behaves as a regular metapackage. See also metapackage; distribution package.

install choice An option users can select or deselect as part of the installation process to specify whether a product component is to be installed.

install experience The tasks a user needs to perform in order to install a product on their computer.

installation host The computer onto which a package is to be installed.

install operation Installation activity performed by an executable file that is invoked at a specific point during the installation process.

install operation executable An executable file that is invoked by Installer during an install, before or after copying a package's payload to the installation destination.

installation destination The directory in which Installer places a package's payload.

installation package A file package with the `.pkg` or `.mpkg` extension. Installation packages (also known as packages) contain products or product components (known as the package's payload) and installation information used by the Installer application and Remote Desktop to place product files on a file system.

installation property Information in an installation package that specifies an installation requirement or an installation process detail, such as whether relocation is allowed.

installation receipt A token that Installer uses to determine whether a component has already been installed on an installation volume.

installation requirement A condition that the target computer or volume of an installation must meet in order for the install to take place. The two types of installation requirements are system requirements and volume requirements.

installation volume The volume (or mountpoint) onto which an installation package is to be installed.

managed install An Installer-driven installation process. Users open an installer package in the Installer application, which performs all install tasks.

manual install A user-driven installation process. Users drag a product's files to a location of their choosing in their file system.

metapackage Installation package that contains other installation packages, usually component packages. Metapackages are used to deliver multicomponent products to users and to provide them with install choices that allow them to choose which components to install. See also component package.

package properties Installer package data that provides Installer details about the package itself, such as its identifier, version number, and resource fork processing.

payload The product or product components contained in an installation package. See also installation package.

product container A file that contains a packaged or unpackaged product. The two container types are disk image and ZIP archive.

relocation The ability of users to change the installation location of a package before an install.

remote install A network administrator–driven installation process. An administrator uses Apple Remote Desktop to install a package onto a set of client computers.

system requirement A condition that must be met by the computer (and associated operating system) in order for the install to proceed.

volume requirement A condition that must be met by a volume in order qualify as a possible installation volume.

Document Revision History

This table describes the changes to *Software Delivery Guide*.

Date	Notes
2006-07-24	Made major changes to content and added information on distribution packages. Changed title from "Software Distribution."
2003-08-21	Updated to reflect new and revised features in Mac OS X v10.3.
	Added details about installation requirements.
	Added list of supported localization folder names.
	Added additional information on default ownership and access permissions.
2003-05-13	First version.

Document Revision History