Code Loading Programming Topics for Cocoa

2007-08-10

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Dynamically Loading Code

This programming topic describes the available techniques for loading executable code while an application is running.

Limitations

Due to a limitation in the Objective-C runtime system, at the time of writing there is no way to unload Cocoa loadable bundles.

Organization of This Document

To learn about the concepts related to dynamic loading, read the following articles:

- "About Loadable Bundles" (page 9) describes how loadable bundles are structured and when you should use them.
- "Loadable Bundles in Cocoa" (page 13) describes features specific to loadable bundles in Cocoa.
- "CFBundle and NSBundle" (page 15) describes the relationship between the Core Foundation CFBundle opaque type and the Cocoa NSBundle class.
- "Multi-Bundle Applications" (page 17) explains how to organize your application into multiple bundles for increased modularity and extensibility.
- "Plug-in Architectures" (page 21) describes the plug-in concept and how to architect an application around plug-ins.

The following tasks are covered:

- "Loading Bundles" (page 29)
- "Creating Loadable Bundles" (page 37)
- "Building Applications with Multiple Bundles" (page 39)
- "Creating Plug-in Architectures" (page 47)
- "Preventing Name Conflicts" (page 55)

- "Loading Objective-C Libraries From Java" (page 57) describes how to load an Objective-C dynamic library into a Java application.

See Also

It is recommended that you read *Bundle Programming Guide* as a prerequisite; this document provides an overview to bundles, including their purpose, types, structure, and the API for accessing bundle resources. *Resource Programming Guide*, which is a document related to *Dynamically Loading Code*, describes how to access non-code bundle resources, particularly those in nib files. Finally, to learn more about the Core Foundation CFPlugIn API (`CFPlugInRef`), which is a popular architecture for bundle-based plug-ins, see *Plug-ins*; in addition, "Core Foundation CFPlugIn" (page 26) in this document gives a summary of the architecture.

About Loadable Bundles

Loadable bundles are packages of executable code and related resources that can be loaded at runtime. This flexibility allows you to design highly modular, customizable, and extensible applications. After reading this document, you will understand how loadable bundles are structured and when you should use them.

Introduction to Loadable Bundles

Mac OS X uses a directory structure called a **bundle** throughout the system for packaging executable code and associated resources. The bundle directory, in essence, "bundles" a set of resources in a discrete package. Resources include such things as Interface Builder nib files, images, sounds, and localized character strings. Because code and associated resources are in one place in the file system, installation, uninstallation, and other forms of software management are easier.

Bundles come in several varieties: applications, frameworks, and loadable bundles. This document deals specifically with **loadable bundles**, which contain code that can be loaded at runtime, freeing you from a number of limitations of static compilation. Architecting your application around loadable bundles affords you a number of advantages:

- You can delay loading code until it is needed and in some cases unload code when it is no longer needed.
- You can componentize your application into pieces that can be compiled independently, allowing you to more easily divide up development work and test different versions of different components with each other.
- You can make your application extensible by designing a plug-in architecture. This way, you or third-party developers can easily add new features without recompiling the whole application or even having access to its source code.

Unloading is limited to applications that do not use the Cocoa runtime environment. At the time of this writing, the Objective-C runtime does not support unloading Objective-C symbols, so Cocoa bundles, once loaded, cannot be unloaded.

Mac OS X defines the extension `.bundle` to identify loadable bundles. You can also define your own extension (and associated icon) for a particular type of bundle.

Kernel extensions are a type of loadable bundle that the system bundle routines recognize and handle appropriately (although their internal structure is different from other loadable bundles). These bundles have an extension of `.kext`. The Kernel Manager, which claims kernel extensions as a document type, dynamically loads them into the kernel environment. This document does not deal with kernel extensions further. To learn more, see the Darwin Documentation.

For more information about bundles, see *Bundle Programming Guide*. For detailed information about how Mac OS X loads and executes code, see *Mac OS X ABI Mach-O File Format Reference*.

When to Use Loadable Bundles

Fundamentally, loadable bundles let you load code and bind symbols at runtime. In practice, they can help you meet three related needs:

- Delayed ("lazy") linking and loading
- Modularity of code components
- Application extensibility

Lazy Linking and Loading

Large, complex applications perform a variety of tasks. Usually, the user does not need to perform all of the available tasks in a given session of use. This means that chunks of application code, sometimes significant pieces, are loaded into memory but are never used.

You can avoid unnecessary loading of code by partitioning your application into loadable bundles consisting of logically related code. By also including accessor functions or methods for these pieces of code, you can seamlessly reference the objects without directly checking if the code has been loaded or not—these checks take place implicitly in the accessor methods.

Lazy loading can take a hierarchical form, with large application pieces at the top level and more specific features below. Specific features typically take the form of plug-ins, as described in "Application Extensibility" (page 11).

Code Modularity

There are several ways to modularize your application into logically separate code components. The basic way to modularize your code is to split it into multiple source files—all nontrivial applications are built this way. The next level of complexity is a framework, which contains executable code, headers, and possibly other resources in a directory package. With a little extra work, you can also use loadable bundles as a unit of modularity.

Although not every situation calls for modularization via loadable bundles, you gain a number of benefits by putting in the extra work. When the codebase is split into multiple bundles, each bundle can be developed and tested independently of the others to an even greater extent than frameworks. Because the bundle contains dynamically loaded executable code, the other parts of the application can be agnostic of function addresses, object addresses, and even what class a referenced object belongs

to, as long as it knows enough about the class to use it. Different versions of different code components can be tested without recompiling the application—you can combine different components just by dragging icons in the Finder.

Application Extensibility

Most uses of loadable bundles arise from the need for application extensibility. You can define a **plug-in architecture** if you want your application to be extensible, either internally within your organization or by third-party developers. A plug-in architecture defines an interface through which a properly constructed loadable bundle, called a **plug-in**, can add functionality. Examples of plug-ins are screen saver modules, preference panes, Interface Builder palettes, Adobe Photoshop graphics filters, and iTunes music visualizers.

Things to Keep In Mind

Using loadable bundles does not come "for free," although both Cocoa and Core Foundation provide rich API support in this regard. If you are trying to decide whether or not to use loadable bundles, keep in mind the following costs:

- Loadable bundles, unlike application bundles and frameworks, must be explicitly loaded at runtime. If you need code reusability but not dynamic loading, frameworks often are a better choice.
- In Cocoa, loadable bundles cannot be used as a memory management scheme for large portions of code, because Cocoa bundles cannot be unloaded.
- Defining a plug-in architecture requires careful validation of plug-in modules.
- Allowing external plug-in code in your application comes with stability and security issues.

However, if your application needs delayed loading, more dynamic modularity than frameworks can provide, or extensibility, use loadable bundles.

Anatomy of a Loadable Bundle

A bundle directory contains a hierarchy of resources and executable code, each in its appropriate place. A typical bundle directory hierarchy looks like Listing 1 (page 11).

Listing 1 The directory layout of a typical loadable bundle

```
- MyLoadableBundle
    Contents/
        Info.plist
        MacOS/
            MyLoadableBundle
        Resources/
            Lizard.jpg
            MyLoadableBundle.icns
```

```
English.lproj/
    MyLoadableBundle.nib
    InfoPlist.strings
Japanese.lproj/
    MyLoadableBundle.nib
    InfoPlist.strings
```

Every bundle directory contains one item, the `Contents` directory. `Contents` contains two files present in every bundle, `Info.plist` and `PkgInfo`, as well as the `MacOS` directory, which contains executable code, and the `Resources` directory, which contains all non-code resources. More complex bundles may contain additional directories such as `Frameworks`, `PlugIns`, `SharedFrameworks`, and `SharedSupport`—all the features supported by full-fledged application packages.

The `Info.plist` file, or **information property list**, is an XML property list containing key-value pairs of information about the bundle. System routines allow the bundle executable to read these attributes at runtime. You are free to store any application-defined data in the information property list as well. Xcode provides a user interface for editing information property lists and includes all required keys by default.

Information property list keys can be localized by adding corresponding entries to the `InfoPlist.strings` files contained in language directories (such as `Japanese.lproj`) in the `Resources` directory.

The most relevant information property list keys for loadable bundles are:

- `CFBundleExecutable`: the name of the executable, typically the same as the bundle directory without the extension
- `CFBundleIdentifier`: the globally unique identifier for the bundle, in reverse-DNS order, such as `com.apple.screensaver.Flurry`
- `CFBundleName`: the short display name of the bundle, used as a human-readable identifier (should be localized)
- `CFBundleDisplayName`: the display name of the bundle, used to represent the bundle in the Finder unless overridden by the user (should be localized)

For a full description of all the standard `Info.plist` keys, see “Property List Key Reference”.

The `MacOS` directory contains the Mach-O executable code for a loadable bundle. The name of the executable inside this directory is typically the same as the bundle directory without its extension and is the same as the value for the `CFBundleExecutable` key in the information property list. Certain types of bundles may lack executable code. Loadable bundles, applications, and most other types of bundles, however, need it by their nature.

The `Resources` directory contains all resources associated with a bundle, such as sounds, images, Interface Builder nib files, and localized strings. Localized resources are contained within *language*`.lproj` directories. You can create subdirectories in `Resources` to organize types of resources if you like.

For more detailed information about information property lists, see “Information Property Lists”.

Loadable Bundles in Cocoa

Loadable bundles written for the Cocoa runtime environment include a few features specific to Cocoa. Because they are written for Cocoa, they contain code for Objective-C classes. In particular, every Cocoa loadable bundle contains a **principal class**. The code loading mechanism provided by the NSBundle class uses a bundle's principal class as an entry point. Applications loading bundles can ask NSBundle to find the principal class and use the returned `Class` object to create an instance of that class.

NSBundle finds the principal class in one of two ways. First, it looks for the `NSPrincipalClass` key in the bundle's information property list. If the key is present, it uses the class named by the key's value as the bundle's principal class. If the key is not present or the key specifies a class that does not exist, NSBundle uses the first class loaded as the principal class. If the bundle is built with Xcode, the order of classes as viewed in the project determines the order in which they are loaded.

CFBundle and NSBundle

This concept describes the relationship between NSBundle, the Cocoa class for manipulating loadable bundles, and the Core Foundation CFBundle opaque type it is built upon. This material is important background for anyone planning to use loadable bundles in their application.

About Core Foundation CFBundle

The Core Foundation CFBundle opaque type provides Mac OS X's base API for manipulating bundles. The CFBundle programming interface insulate developers from dealing directly with the specifics of things like executable format, resource-locating algorithms, and data extraction from the information property list.

The CFBundle opaque data type makes bundle resources and code available to your application. You initialize a CFBundle object with the location of a bundle and then access the bundle's contents with the various CFBundle routines. The routines include support for

- dynamically loading and unloading executable code
- retrieving the location of localized and nonlocalized resources such as graphics, nib files, and character strings
- accessing localized and nonlocalized information property list values
- retrieving pointers to functions and data in the bundle executable

The dynamic loading routines handle all the details of interacting with the dynamic loaders for the supported executable formats so you don't have to care about them. Loading and unloading code is a matter of initializing a CFBundle object for the desired bundle and using the functions `CFBundleLoadExecutable` and `CFBundleUnloadExecutable`.

In Cocoa applications, you should not use CFBundle routines to load and unload executable code, because CFBundle does not natively support the Objective-C runtime. NSBundle correctly loads Objective-C symbols into the runtime system, but there is no way to unload Cocoa bundles once loaded due to a runtime limitation. You can, however, create a CFBundle object for a Cocoa bundle and use the other CFBundle routines without trouble.

For detailed information about Core Foundation Bundles, including API reference, see the Core Foundation Programming Topic *Bundle Programming Guide*.

About NSBundle

NSBundle is the Cocoa class responsible for bundle management. Most of its methods invoke the corresponding CFBundle routines and are modeled after them. Note, however, that pointers to NSBundle objects cannot be cast to CFBundleRefs, unlike some other Core Foundation types and Foundation equivalents—they are *not* toll-free bridged.

For most of its methods, NSBundle simply calls the appropriate CFBundle routine to do its work, but loading code is different. Because CFBundle does not handle Objective-C symbols, NSBundle has to use a different mechanism for loading code. NSBundle interacts with the Objective-C runtime system to correctly load and register all Cocoa classes and other executable code in the bundle executable file.

At the very least, Cocoa bundles contain a single class, the principal class, which serves as an entry point into a bundle. See "Loadable Bundles in Cocoa" (page 13) for a discussion of the principal class.

Because of a limitation in the Objective-C runtime system, NSBundle cannot unload executable code. Since CFBundle does not know about Objective-C symbols, do not use the CFBundle loading and unloading routines on NSBundle objects. You can, however, create a CFBundle object for a Cocoa bundle and use the other routines—unrelated to loading and unloading—without trouble.

Multi-Bundle Applications

This concept describes how to design an application that uses multiple loadable components. This material is relevant both to developers organizing their application into large loadable components and to developers designing an extensible plug-in architecture.

When to Use Multiple Bundles

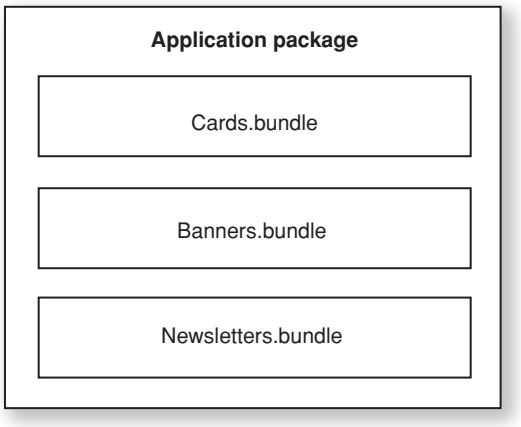
Components are at the heart of the object-oriented programming model. Rather than a monolithic, intertwined mess of code, well-designed applications are built up from components, each of which handles some well-defined unit of functionality. This componentization occurs on many levels, from simple data objects to large self-contained components such as integrated HTML viewers. Even most applications written in "non-object-oriented" languages like C exhibit this sort of componentization.

You can use loadable bundles to separate components into discrete executable packages that can be loaded dynamically at runtime. When designing an application, you may want to use multiple loadable bundles to partition the application into well-defined components that work together but can be developed, compiled, and loaded independently of one another. While you are unlikely to need multiple bundles for simple data objects, they may be appropriate for larger pieces of functionality.

Take as an example an integrated application that produces cards, banners, and newsletters. Although the three components may link against some shared code, in any particular runtime session it is unlikely that more than one of the components will be used. Additionally, each component is independent of the other, and functions essentially as a separate application.

Because of the independent nature of the components and the likelihood that only one of them will be used at any time, such an application is a good candidate for componentization with bundles. Figure 1 (page 18) shows how the example application might be organized with bundles.

Figure 1 Large-scale componentization with loadable bundles



By building each component as its own loadable bundle, you gain several advantages:

- You can develop and test each component independently, and build one without rebuilding the others.
- You can mix and match different versions of the components just by moving file packages around.
- At runtime, your application can load only the bundles that are actually used, instead of loading code that is never used. In non-Cocoa applications, the application can even unload bundles once they're done being used.

Most applications can benefit from this kind of componentization, but you may not need dynamic loading. For shared code used by multiple applications or multiple components of an application, you should use a framework instead.

Additionally, Cocoa applications cannot unload bundles. If you want to be able to load and unload large separately built application components, you can build your application as a number of smaller executable components, and the main application can fork child processes. You can enable communication between the processes with the Cocoa distributed objects architecture. For more information, see *Distributed Objects Programming Topics*.

Another common kind of componentization that applications can benefit from is a plug-in model. Many applications have a number of features that perform similar tasks, like graphics filters in an image processing application or modules in a screen saver. These are best developed as part of a plug-in architecture, so that new modules can be added later with relative ease. Additionally, by publishing the plug-in architecture, you can allow extensibility of your application by third parties. Even if you don't want to publish your plug-in architecture to third parties, using a plug-in model can greatly ease your own internal development. For more information about plug-ins, see "Plug-in Architectures" (page 21).

Organizing a Multi-Bundle Application

To effectively organize an application into multiple bundles, you should keep a few things in mind:

- It's hard to manage complex interdependencies between loadable bundles, so organize your application into largely self-contained, independent units. The easiest way to do this is for each bundle to contain a reference to a single object in another bundle—in Cocoa, typically the principal class—and use it to access any other functionality and data.
- Separate out components that may never be used during a runtime session to prevent code from being loaded and never used.
- If your application contains a number of components that perform variations on the same type of operation, such as graphics filters or export formats, or any component that you want to be replaceable or extensible, consider using a plug-in architecture.

For example, return to the banner/card/newsletter application described in "When to Use Multiple Bundles" (page 17). According to these guidelines, it makes sense to organize the application into four loadable bundles: one for the main application controller, and one each for the three subapplications. Additionally, perhaps the application has a number of different ways of drawing text: in a circle, in the shape of a zebra, etc. Providing a plug-in architecture for text-drawing methods would make the application developer's job easier and allow third parties to add features like drawing text with three-dimensional effects.

Plug-in Architectures

This section describes how to architect an application for extensibility through plug-ins. If you want to make your application modular, customizable, and easily extensible, you should read this section to learn about the different ways to build a plug-in architecture.

About Plug-in Architectures

Plug-in architectures are an attractive solution for developers seeking to build applications that are modular, customizable, and easily extensible. What began as a clever way to allow third parties to add features to an application without access to source code has, for many developers, evolved into a full-blown methodology for application development.

Structuring an application as a well-designed host framework and a set of plug-ins gives you many benefits as an application developer:

- You can implement and incorporate application features very quickly.
- Because plug-ins are separate modules with well-defined interfaces, you can quickly isolate and solve problems.
- You can create custom versions of an application without source code modifications.
- Third parties can develop additional features without any effort on the part of the original application developer.
- Plug-in interfaces can be used to wrap legacy code written in different languages.

End-users also benefit from using applications with a plug-in architecture:

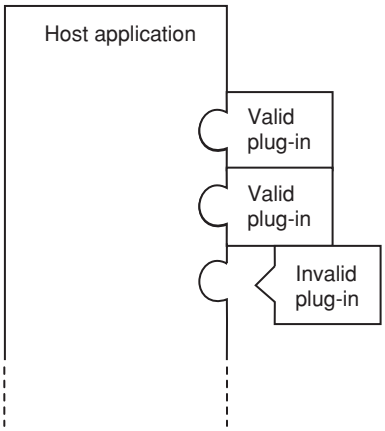
- They can customize feature sets to particular workflows.
- They can disable unwanted features, potentially simplifying the application's user interface, reducing memory footprint, and improving performance.

A **plug-in** is a bundle that adds functionality to an application, called the **host application**, through some well-defined architecture for extensibility. This allows third-party developers to add functionality to an application without having access to the source code. This also allows users to add new features to an application just by installing a new bundle in the appropriate folder. Screen saver modules, preference panes, and Interface Builder palettes, Adobe Photoshop graphics filters, and iTunes music

visualizers are all examples of plug-ins. You use them whenever you want to add multiple instances of a particular type of module that provides a well-defined unit of functionality, such as a new export filter in a graphics program, a new transition style in a video editing program, or other type of feature.

You can think of a host application as a kind of jigsaw puzzle with an infinite number of places to put new pieces. Plug-ins are additional pieces to attach to the puzzle and the plug-in architecture determines the shape of allowable puzzle pieces. If a plug-in has the wrong shape, it can't join the rest of the puzzle. Figure 1 (page 22) illustrates this metaphor: pieces that fit contribute their functionality to the application; pieces that don't fit are left dangling on the sidelines.

Figure 1 Plug-in architecture as an endless jigsaw puzzle



The plug-in architecture typically takes the form either of a list of methods or functions that plug-ins must implement or of a base class that plug-ins must use, but this is not enough. As a host application developer, you must document explicitly not only the form of the plug-in, but also the function, by detailing what type of behavior each method or function must exhibit to behave correctly in the application's plug-in environment.

Plug-in Anatomy and Locations

Plug-ins are usually loadable bundles with executable code that defines the new functionality for the application. Plug-ins may or may not contain resources—in some instances only code is necessary. In rare cases, a plug-in might add resources but no code—such as the `.slideSaver` modules for the Mac OS X screen saver.

Plug-ins are normally installed in one of several standard locations. You can load them from anywhere, but the Mac OS X API layers include support for finding these standard paths.

The standard paths are as follows (where *applicationSupportDirectory* is the directory containing your application's support files):

- `~/Library/Application Support/`*applicationSupportDirectory*`/PlugIns`
- `/Library/Application Support/`*applicationSupportDirectory*`/PlugIns`
- *applicationBundleDirectory*`/Contents/PlugIns`

Plug-in Architecture Design

If you want your application to support plug-ins, you need to define a plug-in architecture so that third-party developers—or developers internal to your organization—have a well-defined interface to implement. The plug-in architecture is up to the developer. The specific architecture that makes sense depends largely on the application. However, most plug-in architectures share the same basic plan.

In object-oriented languages, application developers define the plug-in architecture—the shape of allowed puzzle pieces—by specifying the requirements for a custom class. These requirements typically take the form of an abstract base class or a list of methods (such as an Objective-C protocol). This custom class, known as the **principal class**, is included as part of the plug-in bundle, along with other support code and resources. When it's time to load a plug-in, the host application checks to see if it conforms to the requirements. If the piece fits, then the application asks the class—a factory for instances—to generate an instance. If a plug-in doesn't conform to the architecture, its code is loaded, but the invalid factory never produces an instance.

The C language does not directly support object-oriented concepts, but you can define a plug-in architecture based on entry-point and callback functions for communication with plug-in "objects." Instead of defining the requirements for a plug-in class, the application developer defines a set of functions for plug-ins to implement, and a mechanism for registering callback functions for different types of messages. The application queries a plug-in to see if it implements the necessary entry-point functions. If it does, the application calls those functions to invoke the plug-in's capabilities. Within the entry points, the plug-in can register callback functions to respond to other types of messages.

For more sophisticated behavior in Carbon applications, you can use the Core Foundation CFPlugIn opaque type to define an "object-oriented" architecture that works with both C and C++ plug-ins.

Implementing Plug-in Architectures

You can define an architecture for plug-ins in a number of different ways, each appropriate to different programming environments and plug-in types:

- Define an Objective-C protocol for plug-ins to adopt
- Define an abstract base class for plug-ins to inherit from
- Define C functions for plug-ins to implement and a mechanism for registering callback functions
- Define a plug-in interface with Core Foundation CFPlugIn opaque type

It is easiest to define your plug-in interface in the same language that your application is written in, but this is not necessary. For example, if you are writing an Objective-C Cocoa application, it makes the most sense to use an Objective-C plug-in architecture based on either protocols or base classes. However, you can also provide a C-based architecture for non-Cocoa plug-ins, either with CFPlugIn or with a simple set of predefined callback functions.

If you are writing a truly object-oriented plug-in architecture, you need to decide whether to provide parts of the implementation for plug-ins in a base class or to simply define a set of methods or functions for plug-ins to implement. If plug-ins share a significant amount of functionality, and it would be inconvenient to provide that functionality in the main application code, then using a base class is the

best way to go. The Mac OS X screen saver and preference pane architectures both define abstract base classes that provide some basic functionality for plug-ins. Other plug-ins, such as data processing modules, may not require any base functionality and can simply implement a standard set of methods.

The following sections discuss the plug-in models available to Mac OS X applications, and more details on when you should use them.

Protocols

Objective-C has the notion of an abstract list of methods separate from the class inheritance hierarchy. This lets a developer define a related set of functionality without defining a class or an implementation for the methods. This way, any class can implement the methods, regardless of its place in the inheritance hierarchy. In Objective-C, these lists are called **protocols**. In C++, abstract classes containing only pure virtual functions accomplish essentially the same goal as protocols through a different mechanism.

You can use a protocol to define a plug-in architecture. Plug-in developers then write a class that adopts the protocol and use this class as the plug-in bundle's principal class. At runtime, the host application can check to see if the plug-in conforms to the protocol before using it.

You should use a protocol for your plug-in architecture when at least one of these three conditions is met:

1. Different plug-ins are unlikely to share much code, so a base class would consist of little more than a list of methods.
2. Plug-in developers may want to derive plug-in principal classes from a variety of base classes.
3. The application supports a number of different types of plug-ins, and plug-in developers may want to write a single plug-in that performs the jobs of several different plug-in types.

If plug-ins need to share a core set of code (the first condition), then you might want to define a base class for plug-in principal classes to inherit from. If you need to share code among plug-ins but also want to support different base classes or multiple plug-in types for one plug-in, you should put this code in the application and provide hooks for plug-ins to access it or create a separate class that plug-ins can use as a member.

Cocoa differentiates between **formal protocols** and **informal protocols**. You can use either type of protocol to define a plug-in architecture depending on your needs. When a class adopts a formal protocol, it is in essence "signing an agreement" that it implements all the methods in the protocol. If a class adopts a protocol but fails to implement all its methods, the compiler will give a warning and runtime errors will occur if an unimplemented method is called. Formal protocols are defined with the Objective-C syntax used in Listing 1 (page 24), with method prototypes between `@protocol` and `@end`:

Listing 1 A simple formal protocol

```
@protocol MyStringProcessing

- (NSString *)processString:(NSString *)aString;

@end
```

An informal protocol, on the other hand, is a list of optional methods. Informal protocols are usually implemented as categories on NSObject, so that any Cocoa class can implement its methods, as shown in Listing 2 (page 25). If a class omits an informal protocol method from its implementation, no compile-time warning will be given. However, if a class expects a particular method to be implemented but it is not, a runtime error will occur.

Listing 2 A simple informal protocol

```
@interface NSObject(MyStringProcessing)

- (NSString *)processString:(NSString *)aString;

@end
```

If your host application needs every plug-in to implement a strict set of methods, then you should define the plug-in architecture with a formal protocol. If some or all methods are optional, then use an informal protocol and check at runtime what methods the principal class responds to. If some methods are required and some are not, you can define the architecture as an informal protocol, as long as you clearly document for plug-in authors what methods must be implemented and what methods can be left out.

Abstract Base Classes

Some classes are never meant to be instantiated, but serve as base classes for other classes to inherit from. These **abstract classes** group methods and instance variables that will be used by a number of different subclasses into a common definition. The abstract class is incomplete by itself, but may contain useful code that reduces the implementation burden of its subclasses.

A host application's plug-ins often share some common functionality, so many plug-in architectures are defined by an abstract class that the principal class of a plug-in inherits from. The base class and related code are typically packaged in a framework. Plug-in developers can then link against the framework in their applications and include the appropriate header.

The screen saver architecture in Mac OS X is a good example of using an abstract base class. All screen savers do essentially the same thing: they draw some kind of animation on the screen. Much of the code needed to draw on the screen is already handled by the NSView class in the Application Kit, so screen savers should not have to reimplement this behavior. Additionally, all screen savers need code to handle animation timing, fading in and out, and other behavior. Accordingly, all Mac OS X screen savers inherit from the ScreenSaverView class, which adds screen saver-specific functionality to the NSView class.

Entry-Point and Callback Functions

If you are writing a Carbon application in C, the simplest way to define a plug-in architecture is to define a set of functions that plug-ins must implement. This is akin to an Objective-C protocol, but there are no classes involved and the list of functions is not defined explicitly in any header file, but only in the documentation for the plug-in architecture. For all but the simplest plug-in situations, you should consider using the Core Foundation CFPlugIn opaque type.

Many plug-in architectures define a single plug-in entry point, which the application uses to send messages to the plug-in. In response to these messages, the plug-in can register callback functions to handle various facets of its operation.

For example, iTunes visual plug-ins receive initialization, clean-up, and idle messages through the main entry point. In the initialization phase, plug-ins register a callback function to handle messages related to visualization. Through the callback, iTunes informs the plug-in when a song is played or paused and when other changes of state occur, asks the plug-in to perform drawing, and sends other visualization-related messages.

Host applications need a way to actually find the entry point function or functions to a plug-in. This is done by defining a standard name for entry point functions, and looking up a function pointer for the function with the appropriate signature at runtime. You can use the Core Foundation CFBundle opaque type to translate between function names and function pointers.

For information about using CFBundle to load plug-ins and look up functions, see the Core Foundation Programming Topic *Bundle Programming Guide*.

Core Foundation CFPlugIn

If you are a Carbon host application developer, you should consider using the Core Foundation CFPlugIn opaque type for all but the simplest plug-in situations. CFPlugIn frees you from having to design, implement, and test a new plug-in model yourself, because CFPlugIn handles all the basic plug-in functionality for you.

Core Foundation's CFPlugIn is compatible with the basics of Microsoft's Component Object Model (COM) architecture. In this model, the host application defines one or more types that each consists of one or more interfaces. A plug-in implements all of the functions in all of the interfaces for each type the plug-in supports, as well as a "factory function" for generating instances of the type. CFPlugIn has an advantage over Cocoa of using Universally Unique Identifiers (UUIDs) to uniquely identify types, interfaces, and factories, which eliminates name and version conflicts.

For information about using CFPlugIn, see the Core Foundation Programming Topic *Plug-ins*.

Security Considerations

Extensibility of any sort is cause for concern when it comes to security. Because plug-ins run their own code in your host application's address space, there are essentially no theoretical limits on what they do with your application's address space.

There are, however, a few things you can do to prevent accidental misuse or abuse of your plug-in architecture:

- Warn your users about installing plug-ins from third parties and the side effects they may cause. What software is installed is really up to them, so be sure they know the implications.
- Limit direct access by plug-ins to your application code and data. Do not provide the plug-in with pointers to your application controller objects, application data, or other information that could be accidentally misused or intentionally abused.

Because there is no easy way to tell well-written, well-intentioned code from badly written or ill-intentioned code, you cannot do much more than warn your users and not directly give out to plug-ins sensitive application data. By implementing a plug-in architecture, you are paving the way to have your application behave in ways you did not expect, both positive and negative.

Loading Bundles

The NSBundle class provides methods for loading Cocoa bundles. This section describes the basics of bundle loading in a Cocoa application. Also covered are loading non-Cocoa bundles from a Cocoa application. This material is relevant for any developer using loadable bundles in their application.

Loading Cocoa Bundles with NSBundle

The NSBundle class provides methods for loading executable code and resources from Cocoa bundles. It handles all the details of loading, including interacting with the Mach-O loader `dyld` and loading Objective-C symbols into the Objective-C runtime.

For information about using NSBundle to load non-code resources, see *Resource Programming Guide*.

Loading Cocoa bundles consists of five basic steps:

1. Locate the bundle.
2. Create an NSBundle object to represent the bundle.
3. Load the bundle's executable code.
4. Query the bundle for its principal class.
5. Instantiate an object of the principal class.

The following sections cover each of these steps in detail.

Locating Bundles

Your application can load bundles from any location, but if they are stored in standard locations you can use functions and methods provided by Cocoa to find them easily.

Loadable bundles that are packaged with your applications are typically included inside the application bundle in `Contents/PlugIns`. To retrieve the plug-in directory for the main application bundle, use NSBundle's `builtInPlugInsPath` method.

This code fragment shows how to use NSBundle to retrieve an application's plug-in directory, which may be named `PlugIns` or `Plug-ins` (the former supersedes the latter):

```
NSBundle *appBundle;
NSString *plugInsPath;

appBundle = [NSBundle mainBundle];
plugInsPath = [appBundle builtInPlugInsPath];
```

Although it is not the standard location, you can gain some convenience by storing loadable bundles in your application bundle's `Resources` directory. Then you can use NSBundle's `pathsForResourcesOfType:inDirectory:` method to find them. This code fragment finds all files and directories with the extension `.bundle` in the application's `Resources/PlugIns` directory:

```
NSBundle *appBundle;
NSArray *bundlePaths;

appBundle = [NSBundle mainBundle];
bundlePaths = [appBundle pathsForResourcesOfType:@"bundle"
               inDirectory:@"PlugIns"];
```

Your application may also support bundles in application support directories within the `Library` directory in multiple domains: user-specific (`~/Library`), system-wide (`/Library`), network (`/Network/Library`). To search for these and other standard directories, use the `NSSearchPathForDirectoriesInDomains` function.

This code fragment creates an array of search paths for your application to find bundles, which you can then search for individual plug-ins:

```
NSString *appSupportSubpath = @"Application Support/KillerApp/PlugIns";
NSArray *librarySearchPaths;
NSEnumerator *searchPathEnum;
NSString *currPath;
NSMutableArray *bundleSearchPaths = [NSMutableArray array];

// Find Library directories in all domains except /System
librarySearchPaths = NSSearchPathForDirectoriesInDomains(
    NSLibraryDirectory, NSAllDomainsMask - NSSystemDomainMask, YES);

// Copy each discovered path into an array after adding
// the Application Support/KillerApp/PlugIns subpath
searchPathEnum = [librarySearchPaths objectEnumerator];
while(currPath = [searchPathEnum nextObject])
{
    [bundleSearchPaths addObject:
        [currPath stringByAppendingPathComponent:appSupportSubpath]];
}
```

Creating an NSBundle Object

To create an NSBundle object for a bundle you want to load, either allocate an object and use the `initWithPath:` initializer or use the convenience creation method `bundleWithPath:`. If an instance already exists for the bundle, both of these methods return the existing instance instead of creating a new one.

This code fragment retrieves the bundle located at `fullPath`:

```
NSString *fullPath; // Assume this exists.
NSBundle *bundle;

bundle = [NSBundle bundleWithPath:fullPath];
```

Loading Code

To load a bundle's executable code, use NSBundle's `load` method. This method returns `YES` if loading was successful or if the code had already been loaded, and `NO` otherwise.

This code fragment loads the code for the bundle at `fullPath`:

```
NSString *fullPath; // Assume this exists.
NSBundle *bundle;

bundle = [NSBundle bundleWithPath:fullPath];
[bundle load];
```

Retrieving the Principal Class

Every Cocoa bundle contains code for a principal class, which typically serves as an application's entry point into a bundle. You retrieve a bundle's principal class with NSBundle's `principalClass` method, which loads the bundle if it is not already loaded. This code fragment retrieves the principal class for the bundle located at `fullPath`:

```
NSString *fullPath; // Assume this exists.
NSBundle *bundle;
Class principalClass;

bundle = [NSBundle bundleWithPath:fullPath];
principalClass = [bundle principalClass];
```

You can also retrieve class objects by name with the `classNamed:` method. This code fragment retrieves the class KillerAppController from the bundle at `fullPath`:

```
NSString *fullPath; // Assume this exists.
NSBundle *bundle;
Class someClass;

bundle = [NSBundle bundleWithPath:fullPath];
someClass = [bundle classNamed:@"KillerAppController"];
```

Instantiating the Principal Class

Once you have retrieved the principal class from a loadable bundle, you typically create an instance of the class to use in your application. (If the class provides all its functionality through class methods, this step is not necessary.) To do this, you use a `Class` variable in the same way you would use any class name.

This code fragment retrieves the principal class of the bundle at `fullPath` and creates an instance of the principal class:

```
NSString *fullPath; // Assume this exists.
NSBundle *bundle;
Class principalClass;
id instance;

bundle = [NSBundle bundleWithPath:fullPath];
principalClass = [bundle principalClass];
instance = [[principalClass alloc] init];
```

Loading Cocoa Bundles: Example Code

In most applications, the five steps of bundle loading take place during the startup process as it searches for and loads plug-ins. Listing 1 (page 32) shows the implementation for a pair of methods that locate bundles, create NSBundle objects, load their code, and find and instantiate the principal class of each discovered bundle. An explanation follows the listing.

Listing 1 Method implementations for loading bundles from various locations

```
NSString *ext = @"bundle";
NSString *appSupportSubpath = @"Application Support/KillerApp/PlugIns";

// ...

- (void)loadAllBundles
{
    NSMutableArray *instances;                                      // 1
    NSMutableArray *bundlePaths;
    NSEnumerator *pathEnum;
    NSString *currPath;
    NSBundle *currBundle;
    Class currPrincipalClass;
    id currInstance;

    bundlePaths = [NSMutableArray array];
    if(!instances)
    {
        instances = [[NSMutableArray alloc] init];
    }

    [bundlePaths addObjectsFromArray:[self allBundles]];            // 2

    pathEnum = [bundlePaths objectEnumerator];
    while(currPath = [pathEnum nextObject])
    {
        currBundle = [NSBundle bundleWithPath:currPath];            // 3
        if(currBundle)
        {
            currPrincipalClass = [currBundle principalClass];       // 4
            if(currPrincipalClass)
            {
                currInstance = [[currPrincipalClass alloc] init];   // 5
                if(currInstance)
```

```
                {
                    [instances addObject:[currInstance autorelease]];
                }
            }
        }
    }
}

- (NSMutableArray *)allBundles
{
    NSArray *librarySearchPaths;
    NSEnumerator *searchPathEnum;
    NSString *currPath;
    NSMutableArray *bundleSearchPaths = [NSMutableArray array];
    NSMutableArray *allBundles = [NSMutableArray array];

    librarySearchPaths = NSSearchPathForDirectoriesInDomains(
        NSLibraryDirectory, NSAllDomainsMask - NSSystemDomainMask, YES);

    searchPathEnum = [librarySearchPaths objectEnumerator];
    while(currPath = [searchPathEnum nextObject])
    {
        [bundleSearchPaths addObject:
            [currPath stringByAppendingPathComponent:appSupportSubpath]];
    }
    [bundleSearchPaths addObject:
        [[NSBundle mainBundle] builtInPlugInsPath]];

    searchPathEnum = [bundleSearchPaths objectEnumerator];
    while(currPath = [searchPathEnum nextObject])
    {
        NSDirectoryEnumerator *bundleEnum;
        NSString *currBundlePath;
        bundleEnum = [[NSFileManager defaultManager]
            enumeratorAtPath:currPath];
        if(bundleEnum)
        {
            while(currBundlePath = [bundleEnum nextObject])
            {
                if([[currBundlePath pathExtension] isEqualToString:ext])
                {
                 [allBundles addObject:[currPath
                           stringByAppendingPathComponent:currBundlePath]];
                }
            }
        }
    }

    return allBundles;
}
```

Here's how the code works:

1. The `instances` array contains all the objects instantiated from the principal classes of the discovered bundles. This object is shown in the method for clarity, but would typically be an instance variable of a controller class.

2. The `loadAllBundles` method calls the `allBundles` method to retrieve all files ending with the extension `.bundle`. The `allBundles` method just enumerates through all the standard paths for loadable bundles (in the application bundle and in the user, local, and network `Library` directories).

3. For each returned path, an NSBundle object is created. If the file with a `.bundle` extension was not in fact a valid bundle, NSBundle returns `nil` and the rest of the iteration is skipped.

4. This line retrieves the principal class of the current bundle. Calling `principalClass` implicitly loads the code first.

5. Finally, the method instantiates the principal class. As long as `init` does not return `nil`, the new instance is added to the `instances` array. If you are writing an application with a plug-in architecture (as opposed to an application with a few known loadable bundles), you should perform some kind of validation on the plug-ins before creating an instance of the principal class.

Loading Non-Cocoa Bundles with CFBundle

In some instances, you may need to load non-Cocoa bundles from within a Cocoa application. You use the CFBundle routines in Core Foundation to load non-Cocoa bundles: `CFBundleCreate` to create CFBundle objects; `CFBundleLoadExecutable` to load the bundle's executable code; and `CFBundleGetFunctionPointerForName` to look up the address of a loaded routine. See the Core Foundation Programming Topic *Bundle Programming Guide* for more information about these methods and other methods provided by CFBundle.

To integrate the code more cleanly with your Cocoa application, you can write a wrapper class to encapsulate the data and function pointers looked up through CFBundle.

Listing 2 (page 34) shows the interface for a Cocoa wrapper class for a CFBundle and Listing 3 (page 35) shows its implementation. An explanation follows each listing.

Listing 2 Loading and using code from a non-Cocoa bundle

```
#import <CoreFoundation/CoreFoundation.h>

typedef long (*DoSomethingPtr)(long);                                   // 1
typedef void (*DoSomethingElsePtr)(void);

@interface MyBundleWrapper : NSObject
{
    DoSomethingPtr doSomething;                                         // 2
    DoSomethingElsePtr doSomethingElse;

    CFBundleRef cfBundle;                                               // 3
}

- (long)doSomething:(long)arg;                                         // 4
- (void)doSomethingElse;

@end
```

The interface contains four elements:

1. Type definitions for function pointers, one for each function in the bundle
2. Function pointer instance variables
3. A `CFBundleRef` instance variable
4. Objective-C methods to wrap the C functions

Listing 3 Loading and using code from a non-Cocoa bundle

```
#import "MyBundleWrapper.h"

@implementation MyBundleWrapper

- (id)init
{
    NSString *bundlePath;
    NSURL *bundleURL;

    self = [super init];

    bundlePath = [[[NSBundle mainBundle] builtInPlugInsPath]           // 1
                    stringByAppendingPathComponent:@"MyCFBundle.bundle"];
    bundleURL = [NSURL fileURLWithPath:bundlePath];
    cfBundle = CFBundleCreate(kCFAllocatorDefault, (CFURLRef)bundleURL);

    return self;
}

- (void)dealloc
{
    CFRelease(cfBundle);
}

- (long)doSomething:(long)arg
{
    if(!doSomething)                                                   // 2
    {
        doSomething = CFBundleGetFunctionPointerForName(cfBundle,
                                            CFSTR("DoSomething"));
    }
    return doSomething(arg);                                           // 3
}

- (void)doSomethingElse
{
    if(!doSomethingElse)                                               // 2
    {
        doSomethingElse = CFBundleGetFunctionPointerForName(cfBundle,
                                            CFSTR("DoSomethingElse"));
    }
    doSomethingElse();                                                 // 3
}

@end
```

Here's what the implementation does:

1. Initializes the `cfBundle` instance variable with a URL to the bundle in the application's plug-ins directory. The bundle can reside anywhere on disk; the plug-ins directory is just the typical location for built-in loadable bundles.

2. When the method is called, lazily initializes the function pointer associated with the method. The call to `CFBundleGetFunctionPointerForName` implicitly loads the bundle's executable code before looking up the function pointer.

3. Returns the value returned by the loaded function.

Creating Loadable Bundles

This section describes how to create a Cocoa loadable bundle—an application component or plug-in—with Xcode.

Xcode provides graphical tools for creating loadable bundles. Building loadable bundles is very similar to building an application. The process consists of three basic steps:

1. Create a new project from a Xcode template.
2. Set up and edit source files.
3. Modify project settings with information about your bundle.

The following sections describe this process in detail.

Creating the Project

Creating a loadable bundle project is just like creating an application—you just need to pick the appropriate project template. To create a loadable bundle project, perform the following steps:

1. Launch Xcode.
2. Choose New Project… from the File menu.
3. From the template list, select Cocoa Bundle.
4. Click Next.
5. Choose the location for the project and click Finish.

Setting Up Source Files and Resources

A new project built from the Cocoa Bundle template contains one source file, `main.c`, and a reference to the Cocoa framework. In most cases, you can just delete this file and begin adding your own sources. You can add new Cocoa classes and other source files, as well as resources and frameworks, just as with a Cocoa application.

The minimal loadable bundle in Cocoa contains one class in two files—one for the interface (`MyClass.h`) and one for the implementation (`MyClass.m`). One class in the bundle should be set as the principal class, as described in "Modifying Target Settings." If the principal class is not selected, NSBundle will use the first class in the project (as shown in the Xcode window) as the principal class.

If your loadable bundle is a plug-in, the host application developer usually provides an interface for the plug-in architecture in a framework. This framework typically contains a class that all plug-in principal classes inherit from, or a protocol (formal or informal) for the plug-in's principal class to adopt.

Modifying Target Settings

You need to set two settings in the bundle's information property list for the bundle to be a good citizen:

1. Bundle identifier
2. Principal class

The bundle identifier serves as a unique identifier for all bundles on the system: applications, kernel extensions, all loadable bundles, and other types of bundles. The bundle identifier should be a reverse DNS-style name, such as `com.apple.screensaver.Abstract`.

The principal class serves as the entry point into a Cocoa bundle. It should be named in a globally unique way, as described in "Preventing Name Conflicts" (page 55). If no principal class is selected, NSBundle chooses one for you.

To modify these settings, perform the following steps:

1. Open the project's Targets pane.
2. Select the bundle target listed under the Targets group.
3. Under Info.plist Entries > Simple View, select Basic Information.
4. Enter the desired bundle identifier into the text field labeled "Identifier:".
5. Also under Info.plist Entries > Simple View, select Cocoa-Specific.
6. Enter the name of the principal class into the text field labeled "Principal class:".

Building Applications with Multiple Bundles

There are several different reasons to build an application as multiple loadable bundles and several different ways to do it. This section describes how to use Cocoa to build an application in modular, dynamically loadable components.

Designing Applications with Multiple Bundles

Architecting a Cocoa application around multiple loadable bundles gives you several advantages:

- You can delay loading code until it is needed.
- You can modularize your application into pieces that can be developed and compiled independently.
- You can make your application extensible.

These goals can be met at two hierarchical levels—one for large-scale application organization and the other for small-scale features:

- To enable delayed ("lazy") code loading and modularization, you can use multiple loadable bundles for large-scale application components such as preferences windows and document types.
- To enable extensibility, you can design a plug-in architecture. This way, you and other developers can easily add features without access to the application's source code.

For more information about delayed loading and modularization, see "Lazy Bundle Loading" (page 40) and "Modularizing with Loadable Bundles" (page 41) in this section.

For information about designing and implementing plug-in architectures, see "Plug-in Architectures" (page 21) and "Creating Plug-in Architectures" (page 47).

Lazy Bundle Loading

You may need to access instances of classes in a loadable bundle in potentially many places in your application. If you use a pointer to such objects directly, you need to check to see if the pointer has been initialized with an object before using it. This adds an extra step to using the object every time you use it and introduces the potential for hard-to-find mistakes such as sending messages to `nil` pointers.

You can avoid this complexity by just adding accessor methods for any instances loaded from a bundle. The accessor method performs all the necessary checks for you: if the object is already initialized, it just returns it; otherwise, it loads the bundle, initializes the object, and then returns it. The rest of your application code can then just use the accessor method to use the object and never has to worry if the object has been initialized yet or not.

Listing 1 (page 40) is an example of an accessor method for an object initialized lazily from the principal class of a bundle. An explanation follows the listing.

Listing 1 Accessor method for an object initialized from a loadable bundle's class

```
- (id)bundleObject
{
    if(!_bundleObject)                                                  // 1
    {
        NSString *bundlePath = [[[NSBundle mainBundle] builtInPlugInsPath]
                  stringByAppendingPathComponent:@"MyBundle.bundle"];  // 2
        NSBundle *bundle = [NSBundle bundleWithPath:bundlePath];       // 3

        if(bundle)
        {
            Class principalClass = [bundle principalClass];            // 4

            if(principalClass)
            {
                _bundleObject = [[principalClass alloc] init];         // 5
            }
        }
    }

    return _bundleObject;                                               // 6
}
```

Here's what the code does:

1. Checks to see if the instance variable `_bundleObject` exists. If it already exists, the body of the `if` statement is skipped.
2. Finds the path for the bundle. In this example, the path is hard-coded, although in most applications it is specified in a more sophisticated manner.
3. Retrieves the NSBundle object corresponding to the bundle path. This message implicitly creates an NSBundle object if one does not already exist for the bundle.
4. If the NSBundle object is valid, gets the principal class of the bundle. This message lazily loads the bundle's executable code if it has not yet been loaded.

5. If the principal class exists, the `_bundleObject` instance variable is allocated and initialized.
6. Finally, `_bundleObject` is returned. If the loading process failed, `_bundleObject` maintains a value of `nil`.

Modularizing with Loadable Bundles

There are a number of reasons to modularize your application with loadable bundles. You may want to split up development work so that different application components can be rebuilt without recompiling the whole application. You may want to delay loading different application components. Or you may want to do both.

To accomplish these goals, you will likely want to do one of the following, or both:

- Separate code components into different bundles for easier division of development labor.
- Separate each main window and associated controller code into its own loadable bundle.

The next two sections describe how to perform these tasks.

Bundling Code Components

To create loadable bundles for code components, you can either create a new Xcode project or add a target to an existing one. If your goal is division of development labor, you probably want a separate project for each bundle. If you just want to delay loading code, you can just add a new target to the application project.

In general, the process is as follows:

1. Create the bundle containing the code component.
2. Copy the bundle to the application bundle's "plug-in" directory.
3. Write the application to load and use the code from the bundle.

The following subsections describe the specific steps you need to take to accomplish these tasks.

Creating the Bundle

To create the loadable bundle, you need to first decide what code this bundle will contain, and what class you want to serve as the principal class—the entry point. Once you have figured out what you want the bundle to contain, you can build a bundle project or target.

The process for creating a Xcode project for a loadable bundle is described in "Creating Loadable Bundles" (page 37).

Adding a new target to an existing project follows a similar pattern. Instead of creating a new project, however, perform the following steps to add a new target to an existing project:

1. Open the application project in Xcode.
2. Choose New Target… from the Project menu.
3. Choose Bundle for the target type and click Next.
4. Give the target a name in the Target Name field.
5. Ensure that your project is selected in the Add To Project pop-up menu and click Finish.
6. In the project window, make sure the Files tab is selected.
7. Choose your target from the target pop-up menu.
8. Click the checkboxes next to all the files you want included in this bundle. Make sure you include `Cocoa.framework` in the Frameworks > Linked Frameworks group. Be sure not to include implementations that appear in your application.
9. Modify settings for the bundle target as described in "Modifying Target Settings" (page 38).

Copying Loadable Bundles to the Application Bundle

To copy loadable bundles to the application bundle, you add a copy build phase to the application target, which copies the loadable bundle to the application bundle's "plug-ins" directory.

To set up a copy build phase, go through the following steps:

1. Add the built bundle to your project in the Files tab.
2. Click the Targets tab, and view the application target.
3. Under Build Phases in the target pane, choose the last build phase (usually Frameworks & Libraries). This tells Xcode where to put the next build phase.
4. Choose New Build Phase > New Copy Files Build Phase from the Project menu.
5. In the Copy Files pane, choose Plug-ins from the pop-up menu labeled "Where:".
6. Click the Files tab in the main project pane.
7. Drag the bundle from the list of project files to the box labeled "Files:" in the Copy Files section of the target pane.

Now, when you build your application, the bundle is copied to the `PlugIns` directory of the application bundle, giving you easy access to it from your code. You can add additional bundles to the same copy phase for additional components.

Loading Bundled Code

The generalized process for loading bundles from the application's `PlugIns` directory is the same as described in "Loading Bundles" (page 29).

In applications that use a small number of bundles for large-scale application components, the easiest way to access the code is through a lazy accessor method. "Lazy Bundle Loading" (page 40) describes how to lazily instantiate an object from a bundle.

Bundling Windows and Window Controllers

In many applications, a window is associated with some code to manage the window, typically a subclass of NSWindowController. NSWindowController provides built-in functionality to lazily load the window's nib file. You can go a step further and lazily load the code for the NSWindowController subclass, which in turn may load the nib file.

In this situation, the nib file is packaged in a loadable bundle with the code for its associated window controller class. The application can then load the code for the window controller only when it is first needed and the window controller can load the nib file for the window only when it is needed.

This technique is especially useful for plug-in architectures, where each plug-in may be associated with a window. For example, each filter plug-in in a graphics application might have an associated window that is used to configure settings for the filter. This also applies to static situations as well, where large application components correspond to different windows—for example, a circuit layout window and a graph window in a circuit simulation application.

The process for building a bundle for a window and its associated code consists of these two steps:

1. Create the bundle containing the window and window controller code.
2. Write the application code to load and use the bundled window controller.

The following subsections describe this process in detail.

Creating the Window Controller Bundle

Building a bundle for a window and associated window controller is essentially the same as described in "Loading Bundles" (page 29). In addition, you need to package a nib file in the bundle project containing a window and a window controller class associated with the window.

Once you have a new Cocoa bundle project, create the nib file and add it to your project:

1. Make sure your bundle project is open in Xcode.
2. Launch Interface Builder.
3. Choose New… from the File menu.
4. In the Starting Point window, choose Empty under the Cocoa group.
5. Choose Save As… from the File menu.
6. Navigate to *language*`.lproj/` in your bundle project's directory, where *language* is the language code for the nib file—for example, `English` or `en`.
7. Click Save.
8. In the Add File sheet, select the bundle target under Add to Targets, and click Add.

Next, create the window controller class:

1. Create an NSWindowController subclass.
2. Add any outlets and actions you want to add to the class.
3. Create source files for the class in your Xcode project.
4. Set the Custom Class for the File's Owner proxy to your subclass.

Now you can create the window itself and hook everything up:

1. Add a new window object from the Cocoa-Windows palette and construct it with user interface elements.
2. Connect the `window` outlet of the File's Owner to the newly created window and add any other connections you need to make between the controller and user interface elements.

Finally, write the code for the window controller class in Xcode. The one required method tells NSWindowController what nib file to use:

```
- (NSString *)windowNibName
{
    // Replace MyWindowController with the name of your window
    return @"MyWindowController";
}
```

Writing the Application Code

As with other code components, window controller bundles are best accessed through lazy accessor methods. Listing 2 (page 44) shows the implementation for such a method.

Listing 2 Accessor method for bundled window controller

```
- (NSWindowController *)bundledWindowController
{
    // Assume _bundledWindowController is a private instance variable
    // of type id or NSWindowController *.
    if(!_bundledWindowController)
    {
        NSString *bundlePath = [[[NSBundle mainBundle] builtInPlugInsPath]
                        stringByAppendingPathComponent:@"MyBundle.bundle"];
        NSBundle *windowBundle = [NSBundle bundleWithPath:bundlePath];

        if(windowBundle)
        {
            Class windowControllerClass = [windowBundle principalClass];
            if(windowControllerClass)
            {
                _bundledWindowController = [[windowControllerClass
                                                            alloc] init];
            }
        }
    }
```

```
    return _bundledWindowController;
}
```

The rest of your application code should use the accessor method to refer to the object. For example, this line of code shows the window:

```
[[self bundledWindowController] showWindow:self];
```

Implicit in this example are potentially two lazy loading operations. First, the code for the window controller is loaded when the `principalClass` message is sent. Also, the window's nib file is loaded when the `showWindow:` message is sent.

Creating Plug-in Architectures

Many applications benefit from having a plug-in architecture—a way to extend the application with new features without changing the main application code. This section describes how to create a plug-in architecture for a Cocoa application.

Building the Architecture

The first step in implementing a plug-in architecture is deciding what form you want plug-ins to take. For guidelines on how to decide which mechanism you want to use, read "Plug-in Architecture Design" (page 23).

The Cocoa architecture naturally supports three choices:

- Plug-ins implement a formal protocol
- Plug-ins implement some number of methods from an informal protocol
- Plug-ins inherit from an abstract or concrete base class

In all three cases, you provide a standard interface to plug-in developers, and they write plug-in principal classes to match the interface. The most convenient way to do this is to provide the interface as a framework for plug-in developers to link against.

The following sections describe how to publish the three different types of plug-in interfaces.

Publishing a Formal Protocol Plug-in Interface

To use a plug-in protocol you define, the plug-in developer only needs the header file containing the protocol definition. The easiest way to publish the interface is to simply distribute this header.

A protocol header for a graphics filter plug-in might look something like Listing 1:

Listing 1 Formal protocol for a plug-in architecture

```
/*
   MyGreatImageApp
   Graphics Filter Interface version 0
   MyAppBitmapGraphicsFiltering.h
```

```
*/

#import <Cocoa/Cocoa.h>

@protocol MyGreatImageAppBitmapGraphicsFiltering

// Returns the version of the interface you're implementing.
// Return 0 here or future versions may look for features you don't have!
- (unsigned)interfaceVersion;

// Returns what to display in the Filter menu.
- (NSString *)menuItemString;

// The main worker bee: filters the bitmap and returns a modified version.
- (NSBitmapImageRep *)filteredImageRep:(NSBitmapImageRep *)imageRep;

// Returns the window controller for the settings configuration window.
- (NSWindowController *)configurationWindowController;

@end
```

Notice that this example includes a version method so the application can identify the version of the interface being used. If you are just distributing a header file, this is the best way to ensure that future versions of your application avoid sending messages to old plug-ins that they can't handle. You can also query the plug-in for what messages it responds to, as described in "Validating Plug-ins" (page 51).

Publishing an Informal Protocol Plug-in Interface

Using an informal protocol for a plug-in architecture is somewhat trickier than using a formal protocol. Because the plug-in developer can choose which methods to implement, your application has to check before using a method if the plug-in actually implements it. If some methods are optional and some are required, be sure to make this clear in the documentation for the interface.

You can distribute a single header file to define an informal protocol, just as with a formal protocol. Listing 2 is an alternate implementation of Listing 1 (page 47), using an informal protocol instead of a formal protocol, effectively making some methods optional. Like most informal protocols, this one is implemented as a category on NSObject.

Listing 2 Informal protocol for a plug-in architecture

```
/*
   MyGreatImageApp
   Graphics Filter Interface version 0
   MyAppBitmapGraphicsFiltering.h
*/

#import <Cocoa/Cocoa.h>

@interface NSObject(MyGreatImageAppBitmapGraphicsFiltering)

// REQUIRED
// Returns the version of the interface you're implementing.
// Return 0 here or future versions may look for features you don't have!
- (unsigned)interfaceVersion;
```

```
// OPTIONAL
// Returns what to display in the Filter menu. Defaults to the plug-in
// filename without the extension.
- (NSString *)menuItemString;

// REQUIRED
// The main worker bee: filters the bitmap and returns a modified version.
- (NSBitmapImageRep *)filteredImageRep:(NSBitmapImageRep *)imageRep;

// OPTIONAL
// Returns the window controller for the settings configuration window.
// If this method is not implemented, no Settings option is provided.
- (NSWindowController *)configurationWindowController;

@end
```

Publishing a Base Class Plug-in Interface

If you want to provide shared functionality to all plug-ins, you may want to provide a base class for plug-in principal classes to inherit from. For example, the Mac OS X screen saver interface is provided as a framework containing the ScreenSaverView base class, which handles a number of management details for screen saver plug-ins. To distribute your base class, the best solution is to package it as a framework for plug-in developers to link against.

Listing 3 shows the interface for a hypothetical embedding plug-in architecture, and Listing 4 (page 50) shows its implementation. Instead of a protocol, the interface is provided as a base class that provides some functionality, in this example a fairly simple subclass of NSView that returns version information, maintains a URL containing a data source, and draws a white background. Note that the class contains three reserved pointers. These allow the base class to add new member data without changing the size of the object, thus allowing the host application developer to add features without creating binary incompatibilities.

Listing 3 Base class interface for a plug-in architecture

```
#import <Cocoa/Cocoa.h>

@interface MyAppEmbeddingView : NSView
{
@private
    NSURL *_URL;
    void *_reserved1;
    void *_reserved2;
    void *_reserved3;
}

- (id)initWithFrame:(NSRect)frameRect URL:(NSURL)URL;
- (unsigned)interfaceVersion;
- (NSURL *)URL;
- (void)setURL:(NSURL *)URL;

@end
```

Listing 4 Base class implementation for a plug-in architecture

```
#import "MyAppEmbeddingView.h"

@implementation MyAppEmbeddingView

- (id)initWithFrame:(NSRect)frameRect URL:(NSURL)URL
{
    self = [self initWithFrame:frameRect];
    [self setURL:URL];
    return self
}

- (unsigned)interfaceVersion
{
    return 0;
}

- (void)drawRect:(NSRect)rect
{
    NSEraseRect(rect);
}

- (NSURL *)URL
{
    return _URL;
}

- (void)setURL:(NSURL *)URL
{
    [URL retain];
    [_URL release];
    _URL = URL;
}

@end
```

Once you have finished your base class, you should package the compiled implementation with the header file in a framework for plug-in developers to use. To build a framework, use Xcode's Cocoa Framework project template. Make sure you designate private and public header files as you intend in the Build Phases > Headers section of the Target Settings pane. For more information about building frameworks, see "Creating Frameworks and Libraries" in Xcode Help.

Loading Plug-ins

To use plug-ins, your application needs to go through this process:

1. Find available plug-ins in standard locations
2. Load the executable code for each plug-in
3. Validate conformance of each plug-in to the plug-in interface
4. Instantiate valid plug-ins

The specific details of finding, loading, and instantiating plug-ins are the same as described in "Loading Cocoa Bundles with NSBundle" (page 29). The additional step for plug-in architectures is to validate that each plug-in conforms to the interface you published for plug-in developers to use.

Validation is slightly different depending on whether your plug-in architecture uses a formal protocol, an informal protocol, or a base class.

Validating Plug-ins

For a formal protocol, you query the class to see if it implements the protocol. To be safe, you should also perform a reality check to see that it indeed implements the methods it claims to implement. Listing 5 (page 51) shows the implementation for a method that checks if a plug-in's principal class conforms to the `MyGreatImageAppBitmapFiltering` protocol defined in Listing 1 (page 47), and additionally checks to make sure that it implements all the necessary instance methods.

Listing 5 Plug-in validation (formal protocol)

```
- (BOOL)plugInClassIsValid:(Class)plugInClass
{
    if([plugInClass
        conformsToProtocol:@protocol(MyGreatImageAppBitmapFiltering)]
    {
        if([plugInClass instancesRespondToSelector:
                        @selector(interfaceVersion)] &&
           [plugInClass instancesRespondToSelector:
                        @selector(menuItemString)] &&
           [plugInClass instancesRespondToSelector:
                        @selector(filteredImageRep)] &&
           [plugInClass instancesRespondToSelector:
                        @selector(configurationWindowController])
        {
            return YES;
        }
    }

    return NO;
}
```

For an informal protocol, you *must* query the class to see which methods it implements. Typically, host application developers define certain methods as required, and others as optional, so a validation method should distinguish between the two properly. Listing 6 (page 51) gives the implementation for an informal version of the plug-in validation method. The method makes sure that the plug-in implements all the required methods from the informal version of `MyGreatImageAppBitmapFiltering` given in Listing 2 (page 48). You can check for optional methods as they are needed elsewhere in the application.

Listing 6 Plug-in validation (informal protocol)

```
- (BOOL)plugInClassIsValid:(Class)plugInClass
{
    if([plugInClass instancesRespondToSelector:
                    @selector(interfaceVersion)] &&
       [plugInClass instancesRespondToSelector:
                    @selector(filteredImageRep)])
```

```
    {
        return YES;
    }

    return NO;
}
```

A plug-in that inherits from a base class is the easiest to validate. You simply query the plug-in's principal class to see if it is a subclass of the base class, as shown in Listing 7 (page 52).

Listing 7 Plug-in validation (base class)

```
- (BOOL)plugInClassIsValid:(Class)plugInClass
{
    if([plugInClass isSubclassOfClass:[MyAppEmbeddingView class]])
    {
        return YES;
    }

    return NO;
}
```

Loading Plug-ins: Example Code

Listing 8 (page 52) is a slightly modified version of Listing 1 (page 32) that validates plug-ins before adding them, indicated by the margin comment `// Validation`. For a full description of how the code works, see the original version.

Listing 8 Implementation for plug-in loading methods

```
NSString *ext = @"plugin";
NSString *appSupportSubpath = @"Application Support/KillerApp/PlugIns";

// ...

- (void)loadAllPlugins
{
    NSMutableArray *instances;
    NSMutableArray *bundlePaths;
    NSEnumerator *pathEnum;
    NSString *currPath;
    NSBundle *currBundle;
    Class currPrincipalClass;
    id currInstance;

    bundlePaths = [NSMutableArray array];
    if(!instances)
    {
        instances = [[NSMutableArray alloc] init];
    }

    [bundlePaths addObjectsFromArray:[self allBundles]];

    pathEnum = [bundlePaths objectEnumerator];
    while(currPath = [pathEnum nextObject])
```

```
    {
        currBundle = [NSBundle bundleWithPath:currPath];
        if(currBundle)
        {
            currPrincipalClass = [currBundle principalClass];
            if(currPrincipalClass &&
               [self plugInClassIsValid:currPrincipalClass])  // Validation
            {
                currInstance = [[currPrincipalClass alloc] init];
                if(currInstance)
                {
                    [instances addObject:[currInstance autorelease]];
                }
            }
        }
    }
}

- (NSMutableArray *)allBundles
{
    NSArray *librarySearchPaths;
    NSEnumerator *searchPathEnum;
    NSString *currPath;
    NSMutableArray *bundleSearchPaths = [NSMutableArray array];
    NSMutableArray *allBundles = [NSMutableArray array];

    librarySearchPaths = NSSearchPathForDirectoriesInDomains(
        NSLibraryDirectory, NSAllDomainsMask - NSSystemDomainMask, YES);

    searchPathEnum = [librarySearchPaths objectEnumerator];
    while(currPath = [searchPathEnum nextObject])
    {
        [bundleSearchPaths addObject:
            [currPath stringByAppendingPathComponent:appSupportSubpath]];
    }
    [bundleSearchPaths addObject:
        [[NSBundle mainBundle] builtInPlugInsPath]];

    searchPathEnum = [bundleSearchPaths objectEnumerator];
    while(currPath = [searchPathEnum nextObject])
    {
        NSDirectoryEnumerator *bundleEnum;
        NSString *currBundlePath;
        bundleEnum = [[NSFileManager defaultManager]
            enumeratorAtPath:currPath];
        if(bundleEnum)
        {
            while(currBundlePath = [bundleEnum nextObject])
            {
                if([[currBundlePath pathExtension] isEqualToString:ext])
                {
                 [allBundles addObject:[currPath
                           stringByAppendingPathComponent:currBundlePath]];
                }
            }
        }
    }
```

```
    return allBundles;
}
```

Preventing Name Conflicts

Because the Objective-C runtime uses a flat name space, you must be careful when developing a plug-in not to choose a name that conflicts with the application code or another plug-in loaded by the application. This section describes how to name your classes and other symbols to avoid this kind of conflict.

The Objective-C runtime provides only a single flat, global name space per process for all exported symbols. This includes all global variables, nonstatic functions, class names, and categories declared for individual classes; protocols have a separate global name space of their own.

Because plug-ins from different vendors must coexist in the same process, you must follow conventions to avoid symbol name collisions. Every exported symbol in a plug-in must be prefixed with an identifier unique to the plug-in. This requirement is not circumvented by unloading each plug-in before loading the next one. Once an Objective-C symbol (class names, protocols and categories) gets loaded, it cannot be unloaded.

Uniqueness Algorithm

Your plug-in should derive its unique prefix from its bundle identifier using the following algorithm:

1. Start with the bundle identifier (`com.apple.preference.sound`)
2. Capitalize the first letter of each period-separated component (`Com.Apple.Preference.Sound`)
3. Remove the periods (`ComApplePreferenceSound`)

Note that this convention depends on the uniqueness of each bundle identifier. To guarantee uniqueness of the bundle identifier, each organization should prefix its identifiers with its reverse-ordered ICANN domain name (for example, `com.apple`).

Each organization should institute its own processes and conventions to avoid bundle identifier collisions among bundles developed within the organization.

To avoid having to use the full, prefixed symbol names in source code, you can create shorthand preprocessor macros. These macros can be defined in a single header file that is imported into every source file. For example:

```
#define SoundPref ComApplePreferenceSoundPref
```

```
#define AlertController ComApplePreferenceSoundAlertController
#define MicrophoneController ComApplePreferenceSoundMicrophoneController
```

These shortcuts are only valid in Objective-C source files that include the header file. References to class names outside of such source files (for example, in the bundle property list and in the main nib file) must specify the full, real name.

Categories

Plug-ins should avoid using Objective-C categories to override methods of classes in public frameworks. If multiple plug-ins attempt to override the same method of the same class, only one override takes effect, leading to unpredictable behavior.

Loading Objective-C Libraries From Java

To load Objective-C dynamic libraries into Java applications, use the NSRuntime class. If the libraries are contained within a bundle, using the NSBundle class may be more appropriate.

NSRuntime's principal method, `loadLibrary`, takes a String argument identifying the dynamic library to load. The string can be either the absolute path to the library or just the library name. If just the library name is given, either with or without the standard prefix `lib` or suffix `.dylib`, NSRuntime searches through a list of directories until it finds the library. For example, to load a library named `libMyCode.dylib` located at `/usr/lib`, all of the following works:

```
NSRuntime.loadLibrary("/usr/lib/libMyCode.dylib");
// Or, if /usr/lib is in the search paths
NSRuntime.loadLibrary("MyCode");
NSRuntime.loadLibrary("libMyCode.dylib");
```

After loading the library, the library is initialized by calling the function *`basename`*`Initialization` where *`basename`* is the library's name with the prefix and suffix stripped off. For example, when creating a library named `libMyCode.dylib`, create a function named `MyCodeInitialization` to initialize the library when it gets loaded. The function takes no arguments.

If the library is not found or if the library lacks an initialization function, the application exits with an `UnsatisfiedLinkError` error.

The NSRuntime class manages a list of directories that are searched when you attempt to load a library without providing its absolute path. Initially, the search path includes `/usr/lib/java` and `/usr/local/lib/java`. You can add paths to the list by using the method `addPathToLibrarySearchPaths`. This method takes a single String argument containing the absolute path of the directory to add to the search list. For example, to add `/usr/lib` to the list, do the following:

```
NSRuntime.addPathToLibraryPaths("/usr/lib");
```

To obtain the current list of search paths, invoke `librarySearchPaths`.

Document Revision History

This table describes the changes to *Code Loading Programming Topics for Cocoa*.

Date	Notes
2007-08-10	Removed out-of-date reference to iMovie plug-ins.
2006-12-05	Added references to the CFPlugIn documentation in the TOC and in the Introduction; changed title from "Dynamic Code Loading."
2003-01-13	Replaced references to Core Foundation “Services” with references to new opaque type documentation.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.