Application Architecture Overview

Cocoa > Design Guidelines

2006-08-07

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

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to Application Architecture

This document describes the essential components of a Cocoa application and how they work together. It also discusses features of Cocoa applications and the principles related to their design.

Who Should Read This Document

Every developer who creates Cocoa applications should read this document.

To understand the information in this document you should have a general knowledge of Cocoa programming paradigms, which are described in the *Cocoa Fundamentals Guide*.

Organization of This Document

Three important features of the Cocoa frameworks—the document architecture, scripting, and undo and redo—have a great deal in common conceptually. This document explains their shared conceptual underpinnings. It does not go into great detail about the specifics of the classes implementing these features or how to use them. Instead it concentrates on the recommended structure of an application and how that structure supports these features.

The Cocoa frameworks are `AppKit.framework` and `Foundation.framework`. You can examine them in `/System/Library/Frameworks/`.

This document contains the following articles:

- "Features of a Cocoa Application" (page 9) describes the features provided by the Application Kit that are shared by all Cocoa applications.
- "Document Architecture" (page 11) explains how the Application Kit supports document-based applications. This common type of application enables users to create and edit documents: container objects that manage user data and present it in windows.
- "Scripting" (page 15) explains the concepts you need to understand to make your application scriptable—that is, one that responds to Apple events using the AppleScript system.
- "Undo and Redo" (page 21) describes the support Cocoa provides for implementing undo and redo, and explains how those features work with the document architecture and other application mechanisms.
- "Graceful Application Termination" (page 25) explains how a Cocoa application can quit executing gracefully—that is, ensuring that the user's data is saved and cleaning up after itself.

This document uses Objective-C to describe specific APIs. However, all scripting, document, and undo APIs published in Mac OS X versions through version 10.4 are also available in Java. Special issues related to Java are discussed where appropriate, and if Java isn't mentioned specifically, it is because there is nothing special to say about it.

Features of a Cocoa Application

A Cocoa application has behind it the powerful resources of the Cocoa frameworks, particularly the Application Kit. These software resources—along with Xcode, Interface Builder, and the rest of the Cocoa development environment—make possible the speedy development of robust, full-featured applications.

The simplest Cocoa application, even one without a line of code added to it, includes a wealth of features that you get "for free." In other words, you either do not have to program these features yourself or the programming effort is trivial. You can simply create an application project with Xcode, create a graphical user interface with Interface Builder, and build the application to get the following features:

Window Management and Workspace Integration. In response to user actions, an application takes care of closing, miniaturizing, and resizing its windows. In coordination with the Finder, the application handles its own deactivation and reactivation, hiding and exposing its windows and doing all necessary redraws.

Event Handling. The application creates its event loop and, in coordination with the window system, receives and distributes events to the windows and views in which user actions occurred. Many user actions are handled automatically, but you can implement your own handling of events.

Menu Management. As with an application's windows, an application menu is automatically displayed and removed from the menu bar as the application is started, deactivated, reactivated, and terminated. The application takes care of menu-item tracking and highlighting, submenu display, and accelerator keys. In many instances, an application triggers actions in expected objects when menu items are disabled and it enables or disables items appropriately. Of course, you can customize menu behavior to more sophisticated requirements.

Text and Font Support. When you add the necessary objects to your user interface in Interface Builder, your application automatically gains many capabilities related to text editing:menu selection of font families, sizes, and styles and textual attributes such as alignment, kerning and ligatures; a text object with a ruler, automatic scrolling, and wrapping, built–in support for displaying simple text, RTF, and HTML (and writing simple text and RTF). (Although much of the behavior is free, you still must programmatically provide for saving and reading text files.)

Control Behavior. Applications automatically handle control highlighting, cursor display, pop-up list display, radio-button coordination, and many other aspects of control appearance and behavior. You can customize most of these aspects in Interface Builder. Applications also handle the invocation of actions (methods) in target objects when controls are activated (you set these associations in Interface Builder). You can create your own custom controls.

Cocoa applications are distributed as application bundles. An application bundle is a special type of a bundle: a directory that presents itself to the user in the Finder as a single file. In the case of an application bundle, the file is an executable. Double-clicking it causes the Finder to launch the application. Application bundles have an extension of `.app`.

An application bundle contains the application executable and the resources needed by that executable. For more information about bundles and application packaging, see *Bundle Programming Guide*.

Document Architecture

This article begins by describing the Model-View-Controller (MVC) design pattern because this pattern informs application design that is most supportive of scripting, document-based applications, and undo. It does not fully describe the MVC design pattern in any formal way, because that's not really its purpose, but it does discuss the pattern enough to give some background for the remaining discussion. More information about MVC is presented in the *Cocoa Fundamentals Guide*. In addition, *Cocoa Bindings Programming Topics* explains how you can use Cocoa bindings technology to keep model and view values synchronized.

The document architecture in the Application Kit is based on three classes: `NSDocument`, `NSWindowController`, and `NSDocumentController`. `NSDocument` is the principal class. It represents a single document in your application. Developers must subclass `NSDocument` to give it knowledge of the application's model layer and to implement persistence (loading and saving). `NSWindowController` objects own and control the application's user interface. An `NSDocument` object has one or more `NSWindowController` objects. Developers often subclass `NSWindowController` to add specific knowledge of the view layer that the controller is responsible for managing. `NSDocumentController` is a singleton class. Each document-based application has a single instance of `NSDocumentController` to track and manage all open documents. Developers typically don't need to subclass `NSDocumentController`.

You can find more information about the document architecture in the *Document-Based Applications Overview*.

NSDocuments Are Model-Controllers

`NSDocument` is a model-controller class. Its main job is to own and manage the model objects that make up a document and to provide a way of saving those objects to a file and reloading them later. Any and all objects that are part of the persistent state of a document should be considered part of that document's model. Sometimes the `NSDocument` object itself has some data that would be considered part of the model. For example, the Sketch example application has a subclass of `NSDocument` named `SKTDrawDocument`; objects of this class might have an array of `SKTGraphic` objects that comprises the model of the document. In addition to the actual `SKTGraphic` objects, the `SKTDrawDocument` object contains some data that should technically be considered part of the model because the order of the graphics within the document's array matters in determining the front-to-back ordering of the `SKTGraphic` objects.

An `NSDocument` object should not contain or require the presence of any objects that are specific to the application's user interface. Although a document can own and manage `NSWindowController` objects—which present the document visually and allow the user to edit it—it should not depend on these objects being there. For example, it might be desirable to have a document open in your application without having it visually displayed. For instance, a script might have opened a document to do some processing on it. If the script does not need the user to become involved in the processing, the script might want the document to be opened, manipulated, saved, and closed again, without it ever appearing onscreen.

NSWindowControllers Are View-Controllers

`NSWindowController` is a view-controller class. Its main job is to own and manage the view objects that are used to display and edit a document. A document that is visible to the user has one or more `NSWindowController` objects to own and manage the visual presentation. Although you can use an `NSWindowController` instance, most often you must subclass `NSWindowController` to add specific knowledge of the interface. An `NSWindowController` object usually gets its interface from a nib file. Subclasses often add outlets and actions for the controls and views within the nib file and the `NSWindowController` object usually acts as the file's owner for the nib.

In very simple cases where there is only one window for a document, you may want your `NSDocument` class to have outlets and actions for the nib. In this case, the `NSDocument` subclass acts as the file's owner for the nib, but it still creates an `NSWindowController` instance to own and manage the objects that are loaded from the nib. If you do choose to adopt this approach when quickly prototyping an application, you should be careful to localize the portions of your code that deal with the user interface, so you can later extract them from the document and put them into a custom window controller as your application becomes more complex.

Type Information and NSDocumentControllers

An `NSDocumentController` object manages documents. It keeps track of all open documents; it knows how to create new documents and how to open existing documents. It knows how to find open documents given either a window whose window controller refers to the document or the path of the file the document was loaded from. Developers typically won't have to worry about what it does. `NSDocumentController` knows how to read and use the metadata that a document-based application provides about the types of documents it can open. `NSDocumentController` can provide information based on that metadata, such as lists of file types supported by an application and which `NSDocument` subclasses are used for them.

All document-based applications declare information about the document types they support in the information property list (`Info.plist`) of the application. Xcode provides an editor for creating and modifying this metadata. See the `NSDocumentController` class specification for details on the `Info.plist` keys required by the document architecture and how to include this metadata in your application project.

The metadata in the information property list declares the types of documents supported by an application. Cocoa defines a set of abstract types; these types are usually the same as the pasteboard type that represents such data. For each abstract type, the `Info.plist` lists specific information such as:

- The file extensions used to identify files of that type
- The HFS four-letter type code for files of that type
- The icon the Finder should use to display files of that type
- The subclass of `NSDocument` used by an application to deal with files of that type

`NSDocumentController` loads all this type information and uses it. When `NSDocumentController` runs an open panel it obtains the list of all file extensions for document types that your application can read; it passes that list to the open panel so that it can list the files that can be opened. When the user actually chooses a file to open, the `NSDocumentController` object uses the metadata to identify the subclass of `NSDocument` to use to create the document and load its data.

Typical Usage Patterns

You can use the document architecture in three general ways. The following discussion starts with the simplest and proceeds to the most complex.

The simplest way to use the document architecture is appropriate for documents that have only one window and are simple enough that there isn't much benefit in splitting the controller layer into a model-controller and a view-controller. In this case, the developer needs only to create a subclass of `NSDocument`. The `NSDocument` subclass provides storage for the model and the ability to load and save document data. It also has any outlets and actions required for the user interface. It overrides `windowNibName` to return the nib file name used for documents of this type. `NSDocument` automatically creates an `NSWindowController` instance to manage that nib file, but the `NSDocument` object itself serves as the nib file's file's owner.

If your document has only one window, but it is complex enough that you'd like to split up some of the logic in the controller layer, you can subclass `NSWindowController` as well as `NSDocument`. In this case, any outlets and actions and other behavior that is specific to the management of the user interface goes into the `NSWindowController` subclass. Your `NSDocument` subclass must override `makeWindowControllers` instead of `windowNibName`. The `makeWindowControllers` method should create an instance of your `NSWindowController` subclass and add it to the list of managed window controllers with `addWindowController:`. The `NSWindowController` should be the file's owner for the nib file because this creates better separation between the view-related logic and the model-related logic. This approach is recommended for all but the most simple cases.

If your document requires multiple windows (or allows multiple windows) on a single document you should subclass `NSWindowController` as well as `NSDocument`. In your `NSDocument` subclass you override `makeWindowControllers` just as in the second procedure described above. However, in this case you might create more than one instance of `NSWindowController`, possibly from different subclasses of `NSWindowController`. Some applications need several different windows to represent one document. Therefore you probably need several different subclasses of `NSWindowController` and you must create one of each in `makeWindowControllers`. Some applications need only one window for a document but want to allow the user to create several copies of the window for a single document (sometimes this is called a multiple-view document) so that the user can have each window scrolled to a different position, or displayed in different ways. In this case, your `makeWindowControllers` may only create one `NSWindowController` instance, but there will be a menu command or similar control that allows the user to create others.

Documents and Scripting

Scripting support is mostly automatic for applications based on the document architecture, for several reasons. First, `NSDocument` and the other classes in the document architecture directly implement the standard document scripting class (as expected by AppleScript) and automatically support many of the scripting commands that apply to documents. Second, because the document architecture is intended to work with application designs that use MVC separation, and because scripting support depends on many of the same design points, applications that use the document architecture are already in better shape to support scripting than other applications that are not designed that way. Finally, the document plays an important role in the scripting API of most applications; `NSDocument` knows how to fill that role and provides a good starting point for allowing scripted access to the model layer of your application.

If an application is not based on the document architecture, making it scriptable requires that you duplicate work you would otherwise get for free. The TextEdit application project (distributed with Mac OS X) shows how to make a document-based application that is not based on `NSDocument` scriptable. See the Sketch example project for an example of how to implement a scriptable `NSDocument`-based application.

Scripting

This section describes concepts that will help you design a scriptable Cocoa application. You should read this section before reading *Cocoa Scripting Guide*. However, if you are new to AppleScript and scripting, you should start by reading "AppleScript for Mac OS X" and "Glossary of AppleScript Terms".

A scriptable application is one that can respond to Apple events. An Apple event is a type of high-level message used to send commands and data between processes. The AppleScript scripting system lets users automate tasks involving multiple applications by executing scripts, or series of English-like statements. Script statements are converted to Apple events and sent to the specified applications, which can include the Finder and other parts of the Mac OS.

To help you take advantage of the demand for scriptable applications, the Cocoa scripting architecture is designed to minimize the amount of work needed to make your application scriptable.

Scripting and the Model Layer

The scripting support in the Cocoa frameworks is geared towards making it easy for an application to implement scripting through its model objects. AppleScript has never encouraged scripting an application's user interface because much of the time, the most efficient way for a script to do something is not the way a user might do the same thing in the application. If scripting were concerned solely with providing access to the user interface, it would just be glorified journaling.

Scripts that work with model objects are like batch processing. They perform their operations and don't need or want the user's involvement. A scripting system that extracts data from a database, processes it through other applications, then sends it all to a page-layout program to generate the classified ads page for a newspaper, is an example of such batch processing. The goal in batch-processing is not to involve the user, but to go directly to the application's model objects to get the work done.

Sometimes, however, you may want to affect certain aspects of the user interface while scripting. Scripts that work with view objects are like macros. They do a very specific manipulation of an application, usually a relatively small and self-contained one, and their purpose is to automate a small repetitive task for the user. For instance, a script that gets the selected graphic in a page-layout program, places a caption beneath it, and sets up blue-line guides along the outer edges of the resulting group to aid with alignment, would be a script of this type.

Such a script is like a macro in that the user does a little preparation (such as selecting the graphic), invokes the script, then continues on when it is done. For this type of script, your application must make some of its user-interface structure scriptable—for example, you might need to make windows and selections scriptable. Making these user-interface structures scriptable should, however, be in addition to the support you provide for directly scripting model objects.

There are certain programming practices to avoid in designing your application's model layer for scripting. Many simple applications keep state in their view layer (that is, in a user-interface object). For instance, a Preferences panel controller might be implemented so that the state of a Boolean attribute is "stored" in a checkbox in the Preferences panel and is retrieved and set with the `state` and `setState:` methods. However,

keeping state in a view object is generally not a good strategy for data that is part of a document's model because it is antithetical to the MVC pattern. If a script needs to be able to access and modify state, the state value should be separated from the view layer and stored in a model object or, if it doesn't belong in the model layer, in a controller object. Often this separation is necessary or desirable even without scripting as a consideration.

For example, if a Preferences-panel controller stores current preference settings only in the controls of the panel, it cannot answer any questions about the current settings without loading the panel. If other parts of the application need to find out about preferences even if the user has not brought up the Preferences panel (a likely situation), then it would be much better if the preferences controller itself stored the settings. This would allow it to avoid having to load a nib file (a somewhat expensive activity) until it is actually needed.

The same argument holds for primitive behaviors as well. For instance, if you have a Find panel, instead of implementing the logic to actually perform the find in the action method invoked by the Find Next button, you should probably define some API in your document class or in your model objects that is capable of performing the find. The Find Next button's action method would then invoke this API. The advantage of this scheme is that when you want scripts to be able to search documents, you can let the script go through the document or model API instead of requiring use of the Find panel itself.

Scripting and Key-Value Coding

Scripting in Mac OS X relies heavily on key-value coding to provide automatic support for executing AppleScript commands. In key-value coding, each model object defines a set of keys that it supports. A key represents a specific piece of data that the model object has. Some examples of scripting-related keys are "words," "font," "documents," and "color."

The key-value coding API provides a generic and automatic way to query an object for the values of its keys and to set new values for those keys. The primitive methods for key-value coding are `valueForKey:` and `takeValue:forKey:`. `NSObject` has generic implementations of these methods that first look to use standard accessor set and get methods based on the key (such as `color` and `setColor:` for the key named "color"). If the class of the object does not implement accessor methods, key-value coding directly sets or gets the value of the instance variable ("color"). Key-value coding defines many other extended methods that are implemented in terms of the two primitives, but these aren't discussed here because they have little bearing on implementing scripting support.

As you design the objects of your application, you should define the set of keys for your model objects and implement the accessor methods. Then when you define the script suites for your application, you can specify the keys that each scriptable class supports. If you support key-value coding, you will get a great deal of scripting support "for free."

Keys fall into three categories, which have their roots in relational databases. Keys are either attribute keys (for example, "color"), to-one relationship keys (a document's `NSTextStorage` object), or to-many relationship keys (an application's documents). This categorization makes sense in situations other than relational databases, including scripting. In AppleScript parlance, these key types map clearly to properties and elements. Think of AppleScript elements as relationship keys (where no distinction is made between to-one and to-many relationships) and think of AppleScript properties as attribute keys.

So why is key-value coding so important for scripting? In AppleScript, "object hierarchies" define the structure of the model objects in an application. For instance, a drawing application has documents and those documents have graphic objects. The graphic objects in turn have a fill color and line thickness. Most AppleScript commands specify one or more objects within your application by drilling down this object hierarchy from parent container to child element.

For instance, some graphics might be identified by the statement `graphics 5 thru 7 of the document 'MyDocument' of application 'MyDraw'`. There must be some way of finding these graphics so they can be acted upon. Key-value coding makes this search entirely automatic. An application has the key "documents," which is a to-many relationship (because the application can open multiple documents). Each document has a "name" key that identifies the file it represents. To find the document named MyDocument the framework can ask for all the documents of the application and check each one's name until it finds the one named MyDocument. Because key-value coding defines a uniform way of asking for the value of a key (`valueForKey:`), all this work can be done automatically with no extra effort from the developer. Similarly, once the key-value coding-driven scripting system finds the document, it obtains the "graphics" key and from it gets elements 5 thru 7.

If you have previous experience with AppleScript, you know that in a Carbon application, the work just described depends on the Object Support Library in the Mac OS. The Cocoa version of the Object Support Library knows how to use key-value coding to evaluate object specifiers. Instead of specifically invoking the library and passing in all sorts of evaluation handlers, the Cocoa developer simply relies on the key-value coding mechanism. Of course, you can be more directly involved in the evaluation if you need to do so for performance reasons or if your scripting model does not match your internal model closely enough for the automatic support to work.

The usefulness of key-value coding does not stop with object-specifier evaluation. Most of the core commands defined by AppleScript have default implementations in Cocoa based on key-value coding. For instance, the Get Data and Set Data commands require no extra code for your objects to support if the classes for these objects define their keys properly and implement the standard accessors. The same holds true of the Move, Clone, Delete, Create, Count, and Exists commands. Most script commands have been generically implemented with key-value coding so most model objects will not have to worry about them at all. If your model class must handle a particular command in a special way, even if the command has a default implementation, it can do so.

Object Specifiers

A script command is an AppleScript expression such as `words whose color is red of the fourth paragraph of the front document of application 'TextEdit'`. Within a Cocoa application, elements of this command are represented by objects of the `NSScriptObjectSpecifier` class, which use key-value coding to evaluate the underlying objects they represent. Concrete subclasses of this abstract class represent the different reference forms supported by AppleScript, such as index references (`word 5`) and filter references (tests or "whose" clauses, such as `words whose color is red`).

NSScriptObjectSpecifiers can be nested, so the example in the preceding paragraph would actually be represented by a chain of three references: one for the words, one for the paragraph, one for the document. (The phrase `application 'TextEdit'` does not need representation because the specifier exists in TextEdit by the time the command is executed.) `NSScriptObjectSpecifier` objects know how to evaluate themselves within their containing specifier. The explicit top-level specifier (`front document` in the example) evaluates itself within a default top-level container, which is usually the application itself.

You should not need to know much about specifiers to make an application scriptable, because Cocoa's built-in scripting support can create and resolve specifiers automatically. However, you will need to know how to work with them if your application has scripting needs that go beyond the built-in support. For example, applications that wish to support recording will need to create object specifiers for recorded actions.

Script Commands

When a scripter executes a script that sends a command (such as `set the height of the first rectangle to 37`) to an application, the application receives an Apple event that encapsulates the script command. Cocoa's built-in scripting support converts Apple events into script command objects based on the `NSScriptCommand` class. An application may receive many consecutive script commands, but each one is separate, distinct and complete.

A script command may be an instance of `NSScriptCommand` itself, but Cocoa also provides several subclasses of `NSScriptCommand`, whose default implementations use key-value coding to handle standard AppleScript commands such as Get Data, Set Data, and others. A subclass might also be needed if a command has arguments that need special processing to be converted to a useful form.

An `NSScriptCommand` object has an object specifier that identifies the receiver (or receivers) of the command and can have another object specifier for any arguments defined by the command. Command arguments can be actual values or object specifiers that identify where to find the actual values within the application's object hierarchy.

A scriptable class declares what commands it supports. For commands that have a default implementation, scriptable classes can choose to use it, or they can choose to implement the behavior required by the command themselves. For commands without default implementations, scriptable objects must implement and specify a method that handles the command.

It may seem odd that script commands are separate from the classes that support them. Although this differs from standard object-oriented style, AppleScript is designed to have a small set of commands that act on a wide set of objects. This provides some advantages—for example, it gives the Cocoa frameworks the ability to support default implementations for commands, if the commands are generic enough to be implemented through key-value coding.

Script Suites

AppleScript groups chunks of scripting information in "suites." A suite consists of a set of class descriptions, a set of command descriptions, and a set of terminologies for each supported AppleScript dialect. On Mac OS X, the suites an application supports are defined in the `'aete'` resource of the application. In the Cocoa frameworks, suites are defined in property lists. You can create and examine property lists with the Property List Editor application, which is distributed with Mac OS X.

Any framework, application, or loadable bundle can declare script suites. The set of suites an application supports is a result of the union of all the suites defined by the application itself, the frameworks it links against, and the bundles it loads dynamically. The Cocoa frameworks declare two suites and thus any scriptable application automatically supports these suites. These suites are the Core suite and the Text suite. Thus, if you expose access to an `NSTextStorage` object through your object hierarchy, that `NSTextStorage` object is fully and automatically scriptable through the standard Text suite. If your application uses the Application Kit's document architecture (discussed below), it automatically supports all the Core suite commands that can be applied to documents.

The property list that describes a suite contains all the information about the classes and commands in that suite that are needed by the scripting frameworks. For classes, this includes all the supported keys (attribute and relationship keys) for the class and their types. It also includes all the commands that the class supports (both from the class's own suite and others). For commands this includes the number and types of the

arguments, whether they are required, and the return data type. The suite definition also includes information needed to map the classes and commands to the appropriate four-letter codes used to structure the data in an Apple event representing a script command.

As a Cocoa application developer, you don't typically have to deal with Apple events directly to support scripting, because Cocoa converts incoming Apple events into script commands. However, you will have to provide the information necessary to map classes, commands, keys, and related information to the codes used in Apple events.

In addition to the suite definition, which is a language-independent resource, a suite terminology contains dialect-specific terminology information that identifies the actual scripting vocabulary used for the various classes and commands.

Suite definitions and suite terminologies are described in more detail in *Cocoa Scripting Guide*.

Built-in Suites

The Cocoa frameworks define two standard suites, the Core suite and the Text suite. In addition, Cocoa classes implement scripting for these standard suites so that, for instance, the `NSTextStorage` object is completely scriptable using the Text suite and the `NSDocumentController` and `NSDocument` objects support the Core scripting commands that make sense for documents.

Custom Suites

Any application can define its own suites. In these suites they can define new script classes and new script commands.

Undo and Redo

The Cocoa frameworks provide support for implementing undo and redo. `NSUndoManager` objects are responsible for tracking of the actions necessary to undo changes that are made to a document. The basic premise of the undo architecture is that when you are about to do something you first tell the `NSUndoManager` object how to undo it. The main API is invocation based, so if you have a `setColor:` method, it sends a message similar to the following before it actually sets the new color:

```
[[undoManager prepareWithInvocationTarget:self] setColor:oldColor]
```

This message causes the creation of an `NSInvocation` instance; if the user chooses Undo, that invocation (of the method `setColor:` with the parameter being the old color) is invoked. Because undone changes are put on a redo stack, if the user chooses the Redo command, the changes are redone.

Because many discrete changes might be involved in a user-level action, all the undo registrations that happen during a single cycle of the event loop are usually grouped together and are undone all at once. `NSUndoManager` has methods that allow you to control the grouping behavior further if you need to.

Undo and the Document Architecture

If you use the document architecture, some aspects of undo handling happen automatically. By default, each `NSDocument` object has an `NSUndoManager` object. (If you don't want your application supporting Undo, you can use the `NSDocument` method `setHasUndoManager:` to prevent the creation of the undo manager.) You can use the `setUndoManager:` method if you need to use a subclass or if you otherwise need to change the undo manager used by the document.

When an `NSDocument` object has an `NSUndoManager` object, the document automatically keeps its edited state up to date by watching for notifications from the undo manager that tell it when changes are done, undone, or redone. In this case, you should never have to invoke the `NSDocument` method `updateChangeCount:` directly, since it is invoked automatically at the appropriate times.

The important thing to remember about supporting undo in a document-based application is that all changes that affect the persistent state of the document must be undoable. With a multilevel undo architecture, this is very important. If it is possible to make some changes to the document that cannot be undone, then the chain of edits that the `NSUndoManager` keeps for the document can become inconsistent with the document state. For example, imagine that you have a drawing program that is able to undo a resize, but not a delete. If the user selects a graphic and resizes it, the `NSUndoManager` gets an invocation that can undo that resize operation. Now the user deletes that graphic (which is not recorded for undo). If users now try to undo nothing would happen (at the very least) since the graphic that was resized is no longer there and undoing the resize can't have any visual effect. At worst, the application might crash trying to send a message to a freed object. So when you implement undo, remember that everything that causes a change to the document should be undoable.

Undo and the Model Layer

The most important code supporting undo should be in your model layer. Each model object in your application should be able to register undo invocations for all primitive methods that change the object.

It is often useful to structure the APIs of your model object to consist of primitive methods and extended methods. Examples of this sort of separation can be found throughout the Foundation framework (including `NSString`, `NSArray`, and `NSDictionary`) as well as in the Sketch example project. If you have such a separation in your model objects, remember that only the primitives should register for undo since, by definition, the extended methods are implemented in terms of the primitives.

Some situations might require you to temporarily suspend undo registration for certain actions. For example, a Sketch application lets the user resize a graphic by grabbing a resize knob and dragging it. During this dragging, hundreds or thousands of changes may be made to the bounds rectangle of the selected graphic. Changing the bounds of a graphic is a primitive operation and would normally result in an undo registration. While the user is actively resizing, though, it would be better if those thousands of undo registrations did not happen. In these cases, your model object might provide API to temporarily suspend and resume some or all of its undo registration. It is up to you to decide how to handle this. Certainly, it would work if those thousands of undo registrations did happen, but it would be a tremendous waste of memory to have to remember all those intermediate rectangles when you will never have to restore one of those intermediate states.

Undo and the Control and View Layers

Although the most important part of your undo support should be in the model, there are two situations where you need some undo-related code in either your controller or view objects. The first case is when you want the Undo and Redo menu items to have more specific titles. You can use the `NSUndoManager` method `setActionName:` to give a name to the current undo group. The last invocation of `setActionName:` during an event cycle is the effective one. These names should reflect the intent of the user action, not the primitive operation that the action results in. Therefore, it is in your action methods that you should set action names.

It is not absolutely necessary to name an undo group. The menu items just say “Undo” and “Redo” without being specific about what is to be undone or redone. But when you do register a name it can help the user to know what will be undone or redone. It isn’t too hard to sprinkle a few calls to `setActionName:` in your view or controller action messages, so it is recommended that you try to give meaningful action names.

The second case where you might have some undo code in the controller or view layers is when there are some things that change that do not affect the actual state of the document but that still need to be undoable. Undoing selection changes is often such a case. For example, the Sketch application might not consider the selection to be a part of the document. In fact, if the document can have multiple views open on it, you might be able to have different selections in each one. However, you might want changes in the selection to be able to be undone for the user’s convenience and for visual continuity when the user is actually undoing things. In this case, the view that displays the graphics might keep track of the selection. It should register undo invocations as the selection changes.

Controller and view objects can come and go during the lifetime of a document object, and this is a consideration when controller-layer or view-layer events must be undoable. Your model objects typically live for the lifetime of the document and the document also owns the undo manager, so you don’t generally need to worry about what happens when the model goes away. But you may have to worry about what happens when the controller and view objects go away. If your controller or view object registers any undo

invocations, you should make sure that they are cleared from the undo manager when the controller or view is deallocated. You can use the `NSUndoManager` method `removeAllActionsWithTarget:` for this purpose. Once a particular view on your document is closed, there is no point in keeping undo information about things such as selection changes for that view.

Undo and Scripting

It is usually desirable to make scripted changes undoable. This is one more reason to put your primary undo support in your model objects. Since scripting is usually directed at the model, if your undo support is in your model primitives, then scripted changes can be undone. Being able to undo scripted changes is actually most important with macro-like scripts, where the script is used to automate relatively small tasks that are interspersed with direct user manipulation. In these cases especially, you want the scripted changes recorded along with the direct user changes, and for the same reason—it is important to have all changes to a document recorded. If an application doesn't do this, a document can easily become inconsistent with the undo stack.

Graceful Application Termination

When a user quits an application (by choosing the Quit command or pressing Command–Q) or when a user logs out, restarts, or shuts down the system, an application should do whatever is necessary to terminate itself gracefully. It should ensure that all data associated with the application and its documents is properly saved, all state (such as user preferences) is stored, and that all necessary clean-up takes place. What graceful termination entails depends on the type of application. For example, an application with multiple documents to save must do a lot more than a simple document-less application that needs only to free allocated resources.

In Cocoa, all raw events requiring application termination result in the invocation of the `NSApplication` delegation method `applicationShouldTerminate:`. If the delegate does not implement this method, the application is terminated regardless of any unsaved documents. Moreover, quitting, logging out, restarting, or shutting down does *not* automatically lead to the invocation of the `NSWindow` delegation method `windowShouldClose:` in any of the application's windows. This method is immediately invoked when users click the close box or choose the Close command. It is typically the place the window's (NSWindow) delegate displays a sheet asking users if they want to save any data associated with the window. To gracefully terminate your application (assuming it has data to save) you must ensure that `windowShouldClose:` is invoked for each of your windows, or that the behavior commonly implemented in this method occurs elsewhere in your application.

An application that gracefully terminates can be one of several kinds:

- multi-document applications based on Cocoa's document architecture
- multi-document applications not based on the document architecture
- single-document applications
- applications that need to save state and do any clean-up tasks not handled by the application itself

The procedure differs for each of these kinds of application. The following discussion focuses primarily on the second type of application— multi-document applications that are not based on the document architecture—because the procedure is most comprehensive. The code examples used to illustrate the procedure come from the Text Edit example application located at `/Developer/Examples/AppKit/TextEdit/`.

Applications Based on the Document Architecture

If your multi-document application uses Cocoa's document architecture—that is, the constellation of `NSDocument`, `NSWindowController`, and `NSDocumentController` objects, along with their delegates—the good news is that you have to do absolutely nothing to effect a graceful termination of the application. This "free" behavior is implemented largely in `NSDocumentController`.

In case you don't use the default `NSDocumentController` object, or want to create a subclass of it, you may need to know more about how the `NSDocumentController` class gracefully terminates execution; here is a summary:

1. In `applicationShouldTerminate:`, if there are multiple unsaved documents, `NSDocumentController` calls a method with an impossibly long name: `reviewUnsavedDocumentsWithAlertTitle:cancellable:delegate:didReviewAllSelector:contextInfo:`. This method displays an alert dialog containing buttons for reviewing unsaved documents, quitting despite unsaved documents, and canceling the impending save operation.

2. If the user chooses to cancel, `NSDocumentController` simply returns `NSTerminateCancel`.

3. If the user chooses to quit without saving or if there are no documents to save, the method identified by the `didReviewAllSelector` selector is invoked with a parameter of `YES`, allowing the specified delegate to do whatever is necessary before terminating.

4. If the user chooses to review unsaved documents, `NSDocumentController` calls `closeAllDocumentsWithDelegate:didCloseAllSelector:contextInfo:`. This method simply displays a sheet, in order, for each of the windows with unsaved document data.

For more information on Cocoa's document architecture, see the programming topic *Document-Based Applications Overview*.

Summary of Document-Saving Procedure

A multi-document application that is not based on Cocoa's document architecture has to do much more of the termination work itself. This work is similar to what `NSDocumentController` does, as described in "Applications Based on the Document Architecture" (page 25). In summary, the steps are the following:

1. The application delegate should implement `applicationShouldTerminate:` to handle any request to quit the application or log out, restart, or shut down the system.

2. In `applicationShouldTerminate:` the delegate should get an array of the application's windows and determine if any associated documents have unsaved data.

3. If there are unsaved documents, the delegate displays an alert dialog asking the user if he or she wants to save the documents before quitting, discard any changes (and quit), or cancel the operation.

   Of course, if there are no unsaved documents, the delegate should return `NSTerminateNow`, which tells the application object to proceed with termination (closing all windows, and so on).

4. If users want to review changes and save document data, the application delegate should, in `applicationShouldTerminate:`, initiate the window-save procedure and return `NSTerminateLater`. Otherwise, it should return `NSTerminateNow` or `NSTerminateCancel`, as appropriate.

5. In a window-save routine, each window with unsaved data should display a sheet asking users if they wish to save the document, close the window without saving, or cancel the operation. Each window should display its sheet in an orderly sequence, not all at once, and should respond appropriately to the user's choice.

   Because the goals are the same (saving document data), the code used for this purpose can be the same code that is executed when the user closes the window (typically invoked by the window's delegate in `windowShouldClose:`).

6. After all document data has been saved (or when users choose "close without saving"), send `replyToApplicationShouldTerminate:` to the application object (`NSApp`) with an argument of `YES`.

If the user is logging out, or is restarting or shutting down the system, You need to send `replyToApplicationShouldTerminate:` within two minutes after returning `NSTerminateLater` in `applicationShouldTerminate:` or the procedure will time out.

7. When the application object gets the go-ahead for termination, it closes any open window (among other things). This results in the invocation of the `NSWindow` delegation method `windowWillClose:` in which the delegate can perform any necessary tasks (clean-up, for example) related to the window.

8. Just before the application ceases execution, the application delegate method `applicationWillTerminate:` is invoked; here the delegate can perform any tasks related to the application itself, such as writing out application preferences.

The following section, "An Example: Text Edit," illustrates the procedure outlined above and points out details of implementation.

An Example: Text Edit

When you install the Developer package for Mac OS X, the Text Edit application, which is included in a standard user installation of Mac OS X, is also included as an example Cocoa Application project (`/Developer/Examples/AppKit/TextEdit/`). Even though Text Edit is a multi-document application it does not (currently, at least) make use of the document architecture. Given this, how it handles graceful application termination is instructive.

Listing 1 shows how Text Edit's application delegate—which is its application controller object (`Controller.m`)—implements `applicationShouldTerminate:`.

Listing 1 Implementing applicationShouldTerminate:

```
- (NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication *)app
 {
    NSArray *windows = [app windows];
    unsigned count = [windows count];
    unsigned needsSaving = 0;

    // Determine if there are any unsaved documents...
    while (count--) {
        NSWindow *window = [windows objectAtIndex:count];
        Document *document = [Document documentForWindow:window];
        if (document && [document isDocumentEdited]) needsSaving++;
    }
    if (needsSaving > 0) {
        int choice = NSAlertDefaultReturn;  // Meaning, review changes
    if (needsSaving > 1) { // If we only have 1 unsaved document,
                           // we skip the "review changes?" panel
            NSString *title = [NSString stringWithFormat:
                NSLocalizedString(@"You have %d documents with unsaved
                changes. Do you want to review these changes before
                quitting?", @"Title of alert panel which comes up when user
                chooses Quit and there are multiple unsaved documents."),
                needsSaving];
        choice = NSRunAlertPanel(title,
            NSLocalizedString(@"If you don't review your documents, all
```

```
                changes will be lost.", @"Warning in the alert panel which
                comes up when user chooses Quit and there are unsaved
                documents."),
            NSLocalizedString(@"Review Changes...", @"Choice (on a button)
                given to user which allows him/her to review all unsaved
                documents if he/she quits the application without saving
                them all first."),
            NSLocalizedString(@"Discard Changes", @"Choice (on a button)
                given to user which allows him/her to quit the application
                even though there are unsaved documents."),
            NSLocalizedString(@"Cancel", @"Button choice allowing user to
                cancel."));
          if (choice == NSAlertOtherReturn) return NSTerminateCancel; /* Cancel
 */
        }
        if (choice == NSAlertDefaultReturn) { /* Review unsaved; Quit Anyway
falls through */
            [Document reviewChangesAndQuitEnumeration:YES];
            return NSTerminateLater;
        }
    }
    return NSTerminateNow;
}
```

In this method, the delegate obtains the application object's array of windows and queries the document associated with each for its edited (or "dirty") status. If there are no dirty documents, it returns `NSTerminateNow`. If there are multiple dirty documents, it displays a dialog asking the user if he or she wants to review changes, discard changes, or cancel the operation. Based on the user's response, the delegate returns an appropriate constant: `NSTerminateLater`, `NSTerminateNow`, or `NSTerminateCancel`. If the user wants to review the windows and their documents, or if there is only one window with a dirty document, the delegate sends the `reviewChangesAndQuitEnumeration:` message to the Document class before returning the `NSTerminateLater` constant.

The `reviewChangesAndQuitEnumeration:` cycles through the applications windows and, for each window with unsaved document data, it calls `askToSave:` to have the window's "do you want to save?" sheet displayed. What's important is that it does this in a controlled sequence (instead of having all windows with their alert sheets displayed at the same time). Listing 2 illustrates how Text Edit (in its Document class) implements this class method.

Listing 2 Recursively reviewing document changes

```
+ (void)reviewChangesAndQuitEnumeration:(BOOL)cont {
    if (cont) {
        NSArray *windows = [NSApp windows];
        unsigned count = [windows count];
        while (count--) {
            NSWindow *window = [windows objectAtIndex:count];
            Document *document = [Document documentForWindow:window];
            if (document) {
                if ([document isDocumentEdited]) {
                    [document
askToSave:@selector(reviewChangesAndQuitEnumeration:)];
                    return;
                }
            }
        }
    }
```

```
    [NSApp replyToApplicationShouldTerminate:cont];
}
```

Text Edit accomplishes the orderly sequencing of window alert sheets by having `reviewChangesAndQuitEnumeration:` invoked recursively (as will be shown). The flag passed into the method (*`cont`*), if `NO`, signals that the user has canceled the termination; if `YES`, the method processes the next unsaved document. When there are no more documents to review, or if *`cont`* is `NO`, `replyToApplicationShouldTerminate:` is sent to the application object with the appropriate flag.

The value passed into `askToSave:` is a selector, in this case identifying the `reviewChangesAndQuitEnumeration:` method. As shown in Listing 3, Text Edit simply implements `askToSave:` to make the current document window visible and key and call the `NSBeginAlertSheet` function, which displays the alert sheet asking if the user wants to save the document before closing the window. Note that it passes the selector into this function as the context-information parameter.

Listing 3 Displaying and handling a sheet for saving a document

```
- (void)askToSave:(SEL)callback {
    [[self window] makeKeyAndOrderFront:nil];
    NSBeginAlertSheet(NSLocalizedString(@"Do you want to save changes
        to this document before closing?", @"Title in the alert panel when
        the user tries to close a window containing an unsaved document."),
        NSLocalizedString(@"Save",
        @"Button choice which allows the user to save the document."),
        NSLocalizedString(@"Don't Save",
        @"Button choice which allows the user to abort the save of a
        document which is being closed."),
        NSLocalizedString(@"Cancel",
        @"Button choice allowing user to cancel."),
         [self window], self,
        @selector(willEndCloseSheet:returnCode:contextInfo:),
        @selector(didEndCloseSheet:returnCode:contextInfo:),
        (void *)callback,
        NSLocalizedString(@"If you don't save, your changes will be lost.",
        @"Subtitle in the alert panel when the user tries to close a
        window containing an unsaved document."));
}
```

The Cocoa API for sheets specifies two callback methods that are potentially invoked in the modal delegate as a result of the `NSBeginAlertSheet` call. The first, called the did-end method, is invoked after the user clicks a button in the alert sheet but before the sheet is dismissed; the second, called the did-dismiss method, is invoked after the sheet is dismissed. The function parameters identifying them are selectors. The methods must conform to a certain signature. (See the programming topic *Sheet Programming Topics for Cocoa* for further information.)

The `askToSave:` implementation makes use of both callback methods. For the *`contextInfo`* parameter of `NSBeginAlertSheet` it passes the selector passed it, which in this case identifies the class method `reviewChangesAndQuitEnumeration:`. Then, for the modal delegate (`self`), it implements (as shown in Listing 4) the did-end callback method `willEndCloseSheet:returnCode:contextInfo:` and the did-dismiss method `didEndCloseSheet:returnCode:contextInfo:`.

Listing 4 Callback methods for NSBeginAlertSheet

```
- (void)willEndCloseSheet:(NSWindow *)sheet returnCode:(int)returnCode
contextInfo:(void *)contextInfo {
    if (returnCode == NSAlertAlternateReturn) {     /* "Don't Save" */
```

```
        [[self window] close];
        if (contextInfo) ((void (*)(id, SEL, BOOL))objc_msgSend)([self class],
 (SEL)contextInfo, YES);         // Send callback (YES means continue save
    }
}

- (void)didEndCloseSheet:(NSWindow *)sheet returnCode:(int)returnCode
contextInfo:(void *)contextInfo {
    if (returnCode == NSAlertDefaultReturn) {       /* "Save" */
        [self saveDocument:NO rememberName:YES shouldClose:YES
whenDone:(SEL)contextInfo];
    } else if (returnCode == NSAlertOtherReturn) {  /* "Cancel" */
        if (contextInfo) ((void (*)(id, SEL, BOOL))objc_msgSend)([self class],
 (SEL)contextInfo, NO);          // Send callback indicating save cancel
    }
}
```

Text Edit implements the `willEndCloseSheet:returnCode:contextInfo:` method for the case where the user want to close the window regardless of unsaved data. In this case, it wants the preferred user experience of the window closing before the sheet slides back up "under" the title bar. Note what this callback method does after it closes the window. Using the Objective-C runtime function `objc_msgSend`, `willEndCloseSheet:returnCode:contextInfo:` sends the message identified by the passed-in selector, `reviewChangesAndQuitEnumeration:`, to the Document class with a parameter of `YES`, thus causing the display of the next window's alert sheet.

The `didEndCloseSheet:returnCode:contextInfo:` handles the remaining button-identifying constants potentially sent by the `NSBeginAlertSheet` function. If the user clicks the button to save the window's document, it invokes a method that not only saves the document (displaying the save browser, if necessary), but afterwards closes the window and calls `reviewChangesAndQuitEnumeration:` with a parameter of `YES`. (This detail is not shown.) If the user wants to cancel the termination operation, `didEndCloseSheet:returnCode:contextInfo:` uses the `objc_msgSend` function to send `reviewChangesAndQuitEnumeration:` to the Document class, this time with a parameter of `NO`.

Text Edit ties in the relevant portions of its application-termination code with the code that is invoked when users explicitly close a window. The delegate for a document window implements `windowShouldClose:` in a way that results in the invocation of `askToSave:` if the document has unsaved data. This time `NULL` is passed instead of a selector, so `reviewChangesAndQuitEnumeration:` is invoked only once. Listing 5 illustrates how Text Edit does this.

Listing 5 Handling the explicit closing of a window

```
- (BOOL)windowShouldClose:(id)sender {
    return [self canCloseDocument];
}

- (BOOL)canCloseDocument {
    if (isDocumentEdited) {
        [self askToSave:NULL];
        return NO;
    }
    return YES;
}
```

Cleaning Up

In terminating your application gracefully, there is really not much you typically need to do after saving document data. The objects that comprise an application generally take care of freeing used objects and allocated resources. There are a couple exceptions to this. One is to make sure that objects such as windows with external references have those references removed. One such case is a delegate. In Text Edit's `windowWillClose:` method (which is invoked right after `windowShouldClose:`), the delegate of the window removes itself as a reference on the window (Listing 6).

Listing 6 Removing a reference in windowWillClose:

```
- (void)windowWillClose:(NSNotification *)notification {
    NSWindow *window = [self window];
    [window setDelegate:nil];
    [self release];
}
```

Another case where final tidying up might be necessary is when you need to save persistent data. User preferences are one such case, and `applicationWillTerminate:` is an ideal place to save them. Listing 7 illustrates how Text Edit makes use of `applicationWillTerminate:`.

Listing 7 Saving user preferences in applicationWillTerminate:

```
- (void)applicationWillTerminate:(NSNotification *)notification {
    [Preferences saveDefaults];
}
```

Document Revision History

This table describes the changes to *Application Architecture Overview*.

Date	Notes
2006-08-07	Added reference to "Cocoa Bindings Programming Topics" in MVC discussion.
2006-04-04	Fixed links to related documents and articles. Rewrote introduction and made minor editorial revisions throughout.
2005-10-04	Removed empty articles. Changed the title from "Application Architecture."
2004-05-27	Updated reference from obsolete *System Overview* to *Bundles*.
2003-05-12	Removed a mention of a non-existent feature in "Features of a Cocoa Application" (page 9).
2002-11-12	Revision history was added to existing topic.