Application Design for Scripting, Documents, and Undo

Contents:

Three of the more recent features of the Yellow Box frameworks—scripting, the document architecture, and undo support—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 new features.

This document uses Objective-C to describe specific APIs. However, all scripting, document, and undo APIs 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.

This document begins by describing the Model-View-Controller (MVC) 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, since that's not really its purpose, but it does discuss the pattern enough to give some background for the remaining discussion.

Model-View-Controller

The Model-View-Controller pattern is quite old. It has been around at least since the early days of Smalltalk. It is a high-level pattern in that it concerns itself with the global architecture of a program and tries to provide a classification of the different kinds of objects that make up an application.

According to the pattern, there are three types of objects: model objects, view objects, and controller objects. The pattern defines the roles that these types of objects play in the application; as a developer, you design your classes to fall into these three groups.

Model Objects Represent Data and Basic Behaviors

Model objects are the data-bearing objects of your application. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application (whether that persistent state is stored in files or databases or punch cards) should reside in the model objects once the data is loaded into the application.

Ideally, a model object has no connection to the user interface used to present and edit it. For example, if you have a model object that represents a person (say you are writing an address book), you might want to store a birth date. That's a good thing to store in your Person model object. However, storing a date format string or other information on how that date is to be presented is probably better off somewhere else.

In practice, this separation is not always the best thing, and there is some room for flexibility here, but in general a model object should not be concerned with interface and presentation issues. One example where a bit of an exception is reasonable is a drawing application that has model objects that represent the graphics displayed. It makes sense for the graphic objects to know how to draw themselves since the main reason for their existence is to define a visual thing. But even in this case, the graphic objects should not rely on living in a particular view or any view at all, and they should not be in charge of knowing when to draw themselves. They should be asked to draw themselves by the view object that wants to present them.

View Objects Present Information to the User

A view object knows how to display and possibly edit data from the application's model. Ideally, the view should not store the data it is displaying. (This is intended semantically, of course. A view can cache data or do similar tricks for performance reasons). A view object can be in charge of displaying just one part

of a model object, or a whole model object, or even many different model objects. Views come in many different varieties.

A view should ensure it is displaying the model correctly. Consequently, it usually needs to know about changes to the model. Because model objects should not be tied directly to view objects, they need a generic way of indicating that they have changed. Toward this end, they can either post NSNotifications when they are altered or define another general way for passing change notifications to the views, usually through the controller layer.

Controller Objects Tie the Model to the View

A controller object acts as the intermediary between the application's view objects and its model objects. Typically controller objects have logic in them that is specific to an application. Controllers are often in charge of making sure the views have access to the model objects they need to display and often act as the conduit through which views learn about changes to the model.

By confining application-specific code to controller objects, you make model and view objects more general and reusable. Controllers are often the least reusable objects in an application, but that's acceptable. You can't reuse everything and if you keep the non-reusable code in the controller layer, you have a much better chance of reusing the other objects.

The controller layer frequently contains many lines of code. To make this quantity of code more manageable, it is sometimes useful to subdivide the controller layer further into "model-controllers" and "view-controllers". (This split fits well with the document architecture discussed elsewhere in this document.)

- A model-controller is a controller that concerns itself mostly with the model layer. It "owns" the model; its primary responsibilities are to manage the model and communicate with any view-controller objects.
- A view-controller is a controller that concerns itself mostly with the view layer. It "owns" the interface (the views); its primary responsibilities are to manage the interface and communicate with the model-controller.

The section "Document Architecture" provides a little more detail on the distinction between model-controller and view-controller.

Why Is MVC Important?

Apple is adding many new features to the Yellow Box frameworks. Some of these features, such as scripting and undo, are relatively high-level compared to the kind of things that the Application Kit and Foundation frameworks have provided in the past. As more and more higher-level features are added to these frameworks, the assumption will grow that applications using these features are based on high-level designs, such as MVC. A a developer, you need to become more involved in the areas of application design related to these features.

Although developers have always been encouraged to use the MVC pattern, with the advent of new features such as the document architecture, undo support, and scriptability, it is more important than ever for application designers to take the MVC pattern to heart. All of these new high-level features will work best if your application design follows the MVC pattern. It should be almost effortless to use these new features if your application has a good MVC separation, but it will take more effort to use the new features if you don't have a good separation.

(The Application Kit is mostly a view framework although recently some of the newer classes are beginning to get into the controller layer and there are also a few model objects in there as well. The Foundation framework has mostly model objects along with some controller objects.)

Scripting

Scripting and the Model Layer

AppleScript has always aimed at scripting the model layer of an application. This is a good thing. Much of the time, the most efficient way for a script to do something is not the best way for a user to do the same thing. If scripting were concerned solely with providing access to the user interface of an application, it would just be glorified journaling. Sometimes, however, you do want to affect certain aspects of the user interface while scripting, usually in scripts that are more like macros.

Scripts that work with model objects are like batch processing. They go in and do their thing and do not need or want the user's involvement. A scripting system that extracts data from a database, processes it through other applications and 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 whole idea in these batch-processing cases is not to involve the user. In these cases, you want to go directly to the application's model objects and just get the work done.

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, adds 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. The macro characteristic of these types of scripts is that the user does a little preparation (like selecting the graphic), invokes the script, then continues on when it is done. For this type of script, your application should expose some of its user-interface structure to scripts. You should make things such as windows and selections scriptable to enable this type of scripting. However, the exposure of these user-interface structures should be an addition to the support provided for directly scripting the model objects of your application.

The scriptability support in the Yellow Box frameworks is geared towards making it easy for an application to expose its model objects to scripters. Scriptability is the easiest to implement when the actual model objects of your application are the objects it wants to support for scripting. Keep this fact in mind when you design your application's model layer.

There are certain programming practices to avoid when you design 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. First, 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 on your document class or on 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 having to hack up scripted access to the Find panel itself, which is less desirable.

Scripting and Key-Value Coding

Scripting in Mac OS X relies heavily on key-value coding (KVC) to provide automatic support for executing AppleScript commands. Key-value coding is a very simple concept. 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 keys familiar to those who have used AppleScript 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 KVC 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, KVC directly sets or gets the value of the instance variable ("color"). KVC defines many other extended methods that are implemented in terms of the two primitives as well, but these aren't discussed

here since they have little bearing on how you should implement scripting support.

You should define the set of keys for your model objects and implement the accessor methods. As you design the objects of your application, think about what the keys of these objects should be and write them down somewhere as part of your design. Then when you define the scripting suites for your application, you specify the keys that each scriptable class supports. If you do this, then a great deal of scripting support comes for free.

Keys fall into three categories which have their roots in relational databases (KVC derives from the Enterprise Objects Framework, which primarily provides access to 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 has to be some way of finding these graphics so they can be acted upon. KVC 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 KVC 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 KVC-driven scripting system finds the document, it obtains the “graphics” key and from it gets elements 5 thru 7.

Those familiar with AppleScript probably recognize that the work just described is, on the Mac OS side, accomplished by the Object Support

Library. The Yellow Box version of the Object Support Library knows how to use KVC to evaluate object specifiers. Instead of specifically invoking the library and passing in all sorts of evaluation handlers, the Yellow Box developer simply relies on the KVC 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 the Yellow Box based on KVC. 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 KVC 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

Object specifiers use key-value coding to evaluate the underlying objects they represent. Object specifiers in AppleScript are expressions such as “words whose color is red of the fourth paragraph of the front document of application ‘TextEdit’”. Object specifiers in Yellow Box applications are objects of the NSObjectSpecifier class. 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).

NSObjectSpecifiers are 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 “application ‘TextEdit’” part does not need representation since the specifier exists in TextEdit by the time the command is executed.) NSObjectSpecifiers 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.

Usually you don’t have to worry about specifiers when you make an application scriptable. However, it helps to understand them if you wish to support recordability in your application, which requires you to create them.

Script Commands

A script command is a single AppleScript statement. Script commands are what an application receives when it is being scripted. It may receive many of them consecutively, but each one is separate, distinct and complete. Script commands are represented by NSScriptCommand objects in the Yellow Box. Sometimes script commands are instances of NSScriptCommand subclasses; if the command has a default implementation based on KVC, it has a specific NSScriptCommand subclass that implements that default behavior. Or if a command has arguments that need special processing, a subclass might do the processing required to provide the arguments in a useful form. However, NSScriptCommand objects can be used by themselves, without the need for subclassing.

An NSScriptCommand 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 could seem somewhat odd that script commands are separate from the classes that support them, but this is the nature of AppleScript. AppleScript wants to have a small set of commands that act on a wide set of objects and therefore it defines the commands separately. At least at first, this seems at odds with object-oriented design where you want the behavior and the data rolled up into an a single entity: an object. However, there are some advantages to having them separate, even though the issue is to some degree forced on the frameworks by AppleScript. Having them separate gives the Yellow Box frameworks the ability to support default implementations for commands that are generic enough to be implementable through KVC.

Script Suites

AppleScript groups chunks of scriptability API in "suites." Suites consist of a set of class descriptions, a set of command descriptions, and a set of terminologies for each supported AppleScript dialect. In the Mac OS, the suites an application supports are defined in the "aete" resource of the application. In the Yellow Box frameworks, suites are defined in property lists. Currently there are no special tools for creating property lists; you must

create them by hand. Any bundle-including frameworks and applications-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 Yellow Box frameworks declare two suites and thus any scriptable application automatically supports these suites. These suites are the Core suite (NSCoreSuite) 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. The scripting support uses Apple events as its transport mechanism to provide interscriptability between the Blue and Yellow Box environments. As a Yellow Box application developer, you should not have to deal with an Apple event directly to support scripting, but 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.

Built-in Suites

The Yellow Box frameworks define several of the standard suites, particularly the Core and Text suites. In addition, Yellow Box classes implement scriptability 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.

Coming Features

Recordability—the capability of an application to record user actions in a script—is still under investigation, but is likely to work in a way similar to undo (see “Undo and Redo” for details). The current proposal is that approximately in the same places that you register an action name with the undo manager for a user-initiated action, you also construct a script command which represents the action to be recorded.

The ability to script other applications directly from your application is under investigation. Eventually you should be able to execute script commands or even whole scripts directly from your application that script your application, other applications, or a combination of the two.

Document Architecture

The new 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). NSWindowControllers own and control the application's user interface. An NSDocument has one or more NSWindowControllers. 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.

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 itself has some data that would be considered part of the model. For example, the Sketch example application has a subclass of NSDocument subclass named Document; objects of this class might have an array of Graphic objects that comprises the model of the document. In addition to the actual Graphic objects, the Document object contains some data that should technically be considered part of the model since the order of the graphics within the document's array matters in determining the front-to-back ordering of the Graphics.

An NSDocument 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 NSWindowControllers 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 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 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 to own and manage the objects that are loaded from the nib.

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 that is part of the document or the path of the file a 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 (**CustomInfo.plist**) of the application. Currently, no development tools directly support the creation of this metadata, so you must create it by hand. See the NSDocumentController class specification for details on the **CustomInfo.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. The Yellow Box defines a set of abstract types; these types are usually the same thing as the pasteboard type that represents such data. For each abstract type, the **CustomInfo.plist** lists specific information such as

- The file extensions used to identify files of that type

- The Mac OS four-letter type code for files of that type
- The icon the Workspace 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 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 so 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 to manage that nib file, but the NSDocument 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, but there will be a menu command or similar control that allows the user to create others.

Documents and Scripting

Scripting support is the most automatic for applications based on the new document architecture for several reasons. First, NSDocument and the other classes of 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.

You can make an application that is not based on the document architecture scriptable, but it is not as easy as with an application based on that architecture; you have to duplicate the work you would get for free if the application used the document architecture. The TextEdit example project gives an example of 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.

Undo and Redo

The Yellow Box 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 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; if the user chooses Undo, that invocation (of the method `setColor:` with the parameter being the old color) is invoked. Since 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 has an NSUndoManager. (If you don't want your application supporting Undo, you can use NSDocument's `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 has an NSUndoManager, 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 NSDocument's `updateChangeCount:` method 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, 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 NSUndoManager's `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's `removeAllActionsWithTarget:` method 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 as well as user interface changes. This is just one more reason that your primary undo support should be 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 being used to automate relatively small tasks that are interspersed with direct user manipulation. Especially in these cases, you want

the scripted changes recorded along with the direct user changes 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.