Making Your Applications Scriptable

One of the more powerful features of the Mac OS is scripting. Scripting enables average users to become power users and tap into the latent potency of their systems without having to use mouse or keyboard. For example, a script that executes when a system boots up could run a mail program, scan messages in the in box for URLs of a certain form, and then open those URLs in Web browsers.

In a scripting language such as AppleScript, users write a series of statements, each of which seems like (or at least close to) a natural sentence in a language such as English. For example,

```
tell application "Finder"
      activate
      set the_version to (get the version) as text
      if the_version is less than "8" then
          beep
          display dialog "This script requires Mac OS 8 which
              is notinstalled on this computer."
              buttons {"Cancel"} default button 1 with icon 0
      end if
end tell
```

But once the script is interpreted and run, these sentences become control structures and commands to one or more applications on the user's system (including operating-system components such as the Finder). The beauty of scripting is that, with appealing simplicity, it extends and integrates what the operating system and each scriptable application have to offer.

Scripting and Mac OS X

Mac OS X includes scripting as a feature that spans the Blue Box/Yellow Box divide. That means you can write an AppleScript script on the Blue Box using the Script Editor application, run it, and have it send commands to a Yellow Box application. A script can thereby take advantage of the strengths of applications on either side of the divide.

You could, for example, have a script that processes a scanned photograph using Adobe PhotoShop on the Blue Box and transfers that photograph to an Enterprise Objects Framework (EOF) application running on the Yellow Box, which stores it in a database used by a WebObjects application. You can also run scripts that get data as well as send it; you might, for instance, have a script that fetches the sender addresses of new e-mail in a MailViewer mailbox and responds to them using Claris Emailer.

Important: In the first Customer Release you are not able to compose AppleScript scripts in the Yellow Box or send AppleScript commands from the Yellow Box to the Blue Box. These features will be added in future releases.

Default Suites

The Yellow Box frameworks includes two suites—exported declarations of scriptability—that application's get "for free": the Core suite and the Text suite. In these suites, the Application Kit exports scriptable APIs for many of the key classes, such as NSApplication, NSDocument, NSWindow, NSColor, and NSTextStorage.

Suites have nonlocalized and localized definitions, which are contained in external files loaded by a scriptable application. The nonlocalized file is called a "suite definition," and the localized one is called a "suite terminology." You can use any text editor to examine the class, command, and terminology definitions contained in these files. Suite definitions are located in the (nonlocalized) Resources directory of an application, framework, or bundle. For example, if you have a framework named MyStuff, you would look for a suite definition (named MyStuff.scriptSuite) at:

```
MyStuff.framework/Resources/
```

Suite terminologies—which map special AppleScript terminology to the class and command descriptions in the suite definition—are located in localized **.lproj** subdirectories of the Resources directory. For example, if the same framework has a suite terminology for the French dialect, you would find it (**MyStuff.scriptTerminology**) in:

```
MyStuff.framework/Resources/French.lproj/
```

For more on suites, see “Creating Suite Definitions and Suite Terminologies.”

Core Suite

The Core suite provides default implementations for most AppleScript commands. It also defines scriptability for document-based applications (that is, applications based on the Application Kit’s document architecture). The suite also supports strings as a primitive type for properties.

Supported AppleScript commands: (all) Get, Count, Exists, Move, Copy, Create, Delete, Set; (NSDocument) Print, Save, Open, Close. Note that there is no default implementation for the Print command.

Table 1. Core Suite Attributes and Relationship

Class	Attributes	Relationships
all	class name, class code	
NSApplication	name, active flag, version	documents, windows
NSDocument	filename (complete path), last component of file name, edited flag	
NSTextStorage	foreground color, font name, font size	characters, text
NSWindow	tier number, title, various binary-state attributes	document
NSColor	color	

Text Suite

The Text suite provides scriptability for the text object (NSText and related classes, particularly NSTextStorage). You can acquire this functionality for your application simply by adding the text object to your graphical user interface; you can add the text object programmatically or by dragging it from Interface Builder’s DataViews palette.

The Text suite does not directly support any AppleScript commands—it inherits the support it offers from the Core suite—but it does specify the following attributes and relationships:

Table 2. Text Suite Attributes and Relationships

Class	Attributes	Relationships
NSTextStorage	font name, font size, foreground color	characters, words, paragraphs, attribute runs, text
NSAttachmentTextStorage	filename	

Other Suites

Many of the standard Yellow Box applications, frameworks, and bundles will be made scriptable over time. The Sketch (formerly Draw) example

project also includes its own suites; this project provides a good source of example code related to scripting.

Overview of Scripting in Mac OS X

Mac OS X supports scripting using an architecture whose primary goal is ease of scriptability. It relies on a few key concepts, such as metadata registry, key-value coding, and the Model-View-Controller paradigm. As long as you adhere to a few design principles (which are based on these concepts), you can make you applications scriptable with little effort.

This section is an overview and does not attempt to provide a comprehensive discussion of scripting concepts. Its aim is to present enough conceptual background to give you a sense of how scripting works in Mac OS X and thus to prepare you for making your applications scriptable.

Architectural Overview

As depicted in Figure 1, the architecture for AppleScript support in Mac OS X relies on a number of interdependent subsystems.

As in Mac OS 8.*x*, an AppleScript script in the Blue Box is sent to the AppleScript engine for execution. The engine converts the statements in the script into discrete Apple events and sends these to the Apple event manager (which receives Apple events from other sources as well). The manager coalesces sends the events to the destination application using the Program-to-Program Communication toolbox (PPC) as dispatcher.

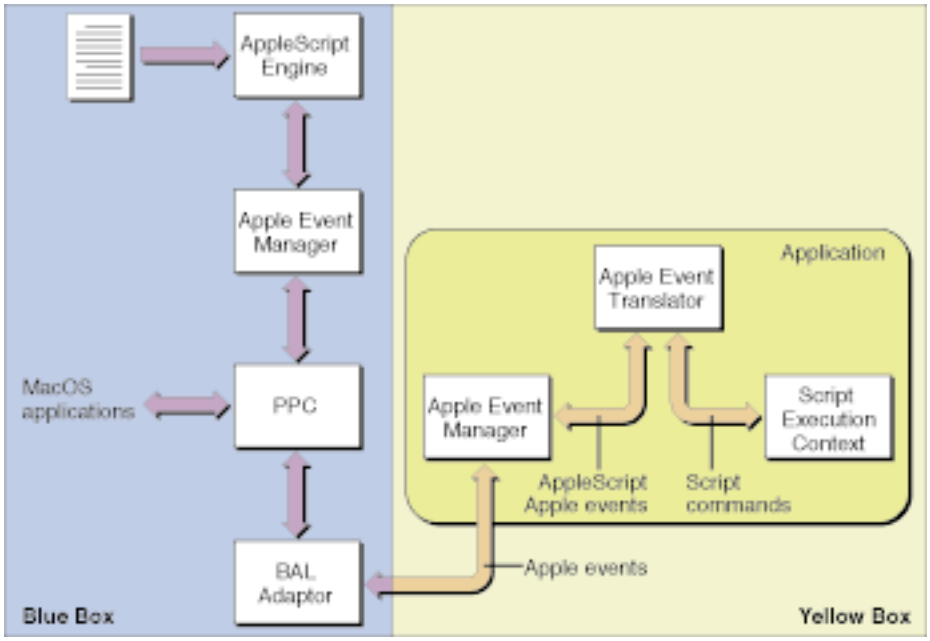

Figure 1. Scripting Subsystems

It is at the PPC level that the architecture begins to diverge from the Mac OS model. The PPC sends Apple events to the applications they are destined for. As in the Mac OS, it sends scripting Apple events to Blue Box (Mac OS) applications. But for Mac OS X, the PPC has been modified to notice if Apple events are meant for Yellow Box applications. If the PPC encounters such an Apple event, it hands it off to the Blue Abstraction Layer (BAL). The BAL packages the Apple event as a distributed object and, through the Distributed Object (DO) mechanism, transports the Apple event to the target Yellow Box application.

The Yellow Box application dynamically links in several frameworks that comprise the subsystems supporting scripting on the Yellow Box side. (Scriptable applications must link against the Scripting and AppKitScripting frameworks.) The first of these subsystems called into play is the Apple event manager for the Yellow Box. This manager receives from the Blue Box a distributed object “wrapping” an Apple event; it converts this object back to an Apple event and dispatches the event appropriately. If the Apple event is related to scripting, the manager sends it to the Apple event translator; the manager knows when to do this because the translator registers with the manager all Apple events it knows to be script commands.

Using the script terminologies loaded by the application, the Apple event translator converts the Apple event into a script command (that is, an NSScriptCommand object). The script command is executed in the script execution context (a single, shared instance of NSScriptExecution context). For details on what happens just before and during command execution, see "How a Command Is Composed and Executed."

The flow of commands and Apple events is bi-directional in this architecture. Script commands often define return values; when these commands are executed, values of a specified type are returned to the originating script. To get back to the script, the return value follows the same path as the original command.

Scripting Metadata

A scripting command can involve any object that is scriptable in any object hierarchy. The scripting system needs not only to get and set the data held by any scriptable object but to determine the objects that can be accessed from any particular scriptable object and what commands these objects support. For this reason, every scriptable application, framework, or bundle must publicly declare in an external data source the scriptable objects it exports. At runtime, this information is stored in a globally accessible repository of scripting metadata.

Scripting metadata consists of two general sorts of information: class descriptions and command descriptions. A class description describes the attributes and relationships of a scriptable class; "attribute" and "relationship," which are terms borrowed from database technology, correspond to "property" and "element" in AppleScript. Relationships can be of several kinds: one-to-one, one-to-many, or inverse. A class description also lists the commands a class supports and specifies whether a particular method of the class handles the command or the command's default implementation is used to execute the command. A description of a class can designate a scriptable superclass, and thus inherit the attributes, relationships, and supported commands of that class.

A command description defines the characteristics of an AppleScript command that the application, framework, or bundle specifically supports. This information includes the class of the command, the type of the return value, and the number and types of arguments. Many commands defined in the Core suite have default implementations in subclasses of NSScriptCommand. Descriptions of AppleScript commands are kept separate from specific classes since they are general to all described classes.

The scripting metadata for an application, framework, or bundle is referred to as a “suite.” A file called a “suite definition” contains the language-independent information for a scriptable suite. It describes object attributes and relationships as well as commands using a structured format called a property list. In addition to a suite definition, a suite can have “suite terminologies” for localizing each supported dialect of AppleScript. A suite terminology maps specific AppleScript words and phrases in a particular language to attributes and relationships defined in the suite definition. See “Creating Suite Definitions and Suite Terminologies” for more on this subject.

When a Yellow Box application first needs to, it locates and loads (caches) all suite definitions and suite terminologies, not only its own but those of all imported frameworks and loaded bundles. (If it later loads a bundle dynamically, it loads any suites defined by the bundle.) It also registers class descriptions and command descriptions, making them globally available to the Apple event translator and any other interested client.

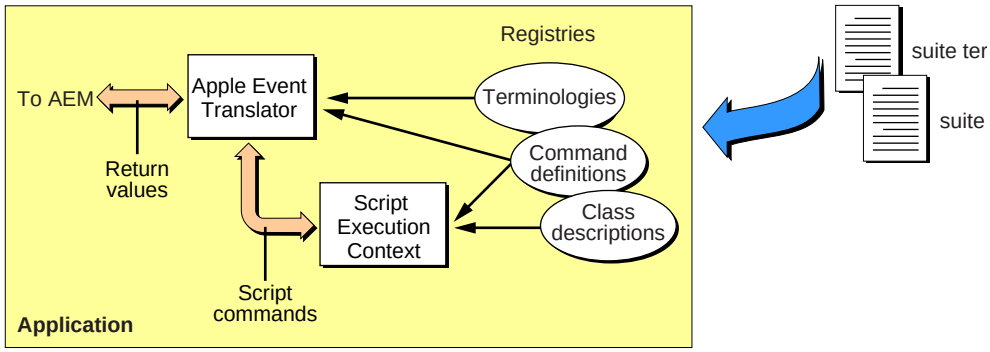

Figure 2. Suites, Registries, and Script Commands

How a Command Is Composed and Executed

When the Apple event translator in the Yellow Box receives an Apple event that corresponds to a script command, it converts the Apple event (and associated data) to a script-command object (NSScriptCommand). Using the suite terminologies for all scriptable objects in the current application, the translator obtains the keys for these objects and uses them to extract data from the class descriptions and command descriptions loaded by that application (including suites of imported frameworks and loaded bundles).

With this data, it composes the script command. A script command has several components:

- The object (or objects) designated to receive the command in the application

- The arguments of the command (if any)
- The definition of the AppleScript command, including command class, Apple event code, number and type of arguments, and return type

Because at this point the receivers and arguments are probably known only as AppleScript reference forms (for example, “word 5 of paragraph 3 of front document”), they are represented as nested “object specifiers” (that is, NSObjectSpecifier objects). The actual objects referenced by the object specifiers cannot be known until the command is executed within the context of the target application.

Once the Apple event translator has composed the script command, it sends the command to the application’s NSScriptExecutionContext object, where it is executed. Command execution is not a simple procedure. It involves several steps:

1. It evaluates the object specifiers in the script command to determine which objects are receivers and arguments (see “How an Object Specifier Is Evaluated” for details).
2. It determines which method to use for executing the command. It first looks in the class description of the receiver to see if it has specified a selector for the command. If it doesn’t, it selects the default implementation for the command.
3. It calls the method indicated by the selector or the method implementing the default behavior for the command. This method has a single argument: the script command.
4. It returns any return value to the Apple event translator, from where it eventually makes its way back to the originating script.

How an Object Specifier Is Evaluated

When the Apple event translator composes a script command, it evaluates an AppleScript statement and converts any reference forms in it to a nested series of NSObjectSpecifier objects (or, simply, object specifiers). It packages these nested object specifiers as the receivers of a command and possibly as the arguments of the command. An application evaluates these object specifiers in its own context to discover the “real” objects so referenced.

A reference form is an expression such as:

```
words whose color is blue of paragraph 3 of front document
```

AppleScript recognizes many types of reference forms; the Yellow Box has subclasses of the abstract NSObjectSpecifier class for most of them.

Table 3. Reference Forms and Object Specifier Classes

Reference Form	Yellow Box Class	Description
Property Every	NSPropertySpecifier	Specifies an attribute (property) or relationship (element) of an object. Example: “color” (Property), “every graphic of the front document” (Every)
Index	NSIndexSpecifier	Specifies an object in a collection. Examples: “word 5”, “front document”
Range	NSRangeSpecifier	Specifies a range of objects in a collection. Example: “words 5 through 10”
Relative	NSRelativeSpecifier	Specifies the position of an object in relation to another object. Example: “before word 5”
Filter Name	NSWhoseSpecifier	Specifies every object in a particular container that matches specified conditions defined by a Boolean expression. Example: “words whose color is blue” or “document named ‘letter to Santa Claus’”
Arbitrary	NSRandomSpecifier	Specifies an arbitrary object in a collection. Example: “any word”
Middle	NSMiddleSpecifier	Specifies the middle object in a collection. Example: “middle word of paragraph 2”
Note: The ID reference form does not yet have a corresponding Yellow Box class.		

Most object specifiers have a reference to their “container specifier”—that is, the parent object specifier that “contains” it in the object hierarchy; the resolution of the container specifier must occur to establish a context in which the current object specifier can be evaluated. An object specifier knows how to evaluate itself in the context of its container specifier. (An object specifier also knows its “child specifier”—that is, the object specifier it is a container for.) An object that has no container specifier is known as the “top-level specifier”; in most cases, the top-level specifier is the application itself.

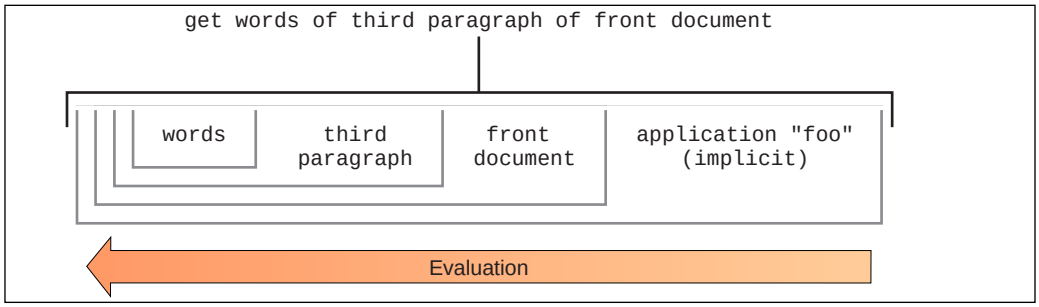

Figure 3. Evaluation of nested object specifiers

Evaluation starts with the top-level specifier and proceeds down the chain of object specifiers, evaluating and resolving each until it determines the identify of the final, nested "child" object. This object is a receiver (or receivers) of the command or one of the command's arguments. Key-value coding is used as the mechanism for evaluation: An object specifier queries its evaluated container (using the `valueForKey:` method or an extension of this method) for the value of the key associated with the object specifier. The key is typically something like "filename" or "windows." The value is the evaluated object, such as an NSString representing a file system path or an NSArray of NSWindow objects. If the object reference has a child reference, the evaluated object is used as the basis for the next query, made on behalf of the child reference. (See "Make Objects Responsive to Key-Value Coding" for an overview of key-value coding.)

The Scripting Classes

About two dozen public classes in the Foundation framework support scripting in the Yellow Box. (They are in Foundation instead of the Application Kit in case you want to make server programs—which have no user interface—scriptable.) To make an application scriptable, you rarely have to interact directly with objects of any of these classes. Even rarer is the occasion when you need to create a subclass of a scripting class. But the scripting classes are available if you need to do anything more advanced, such as recording.

The following tables describe the purpose of each class, why you might want to use it, and why you might need to subclass it.

Scripting Commands and Scripting Metadata

The following classes represent scripting commands, the context in which commands are executed, and the scripting metadata associated with an application, framework, or bundle. Objects of these classes are automatically created by the scripting system.

Table 4. Scripting Metadata and Command Classes

Class	Description
NSScriptSuiteRegistry	A shared instance of this class loads and registers the suites associated with an application. Use its methods to get loaded suites, bundles, class descriptions, and command descriptions. Rarely requires subclassing.
NSScriptClassDescription	Represents a description of a scriptable class in a suite definition. Use its methods to get attributes, relationships, supported commands, and related information. Rarely requires subclassing
NSScriptCommandDescription	Represents a definition of a command supported by a suite. Use its methods to get command class and return and argument types. Rarely requires subclassing.

Table 4. Scripting Metadata and Command Classes

Class	Description
NSScriptCommand	Represents an AppleScript command sent to an application. Use its methods to evaluate object references (receivers and arguments) and execute the command. Apple has implemented subclasses for the major AppleScript commands. You may create a subclass of this class if you define a command with a default implementation or which needs some special argument processing.
NSScriptExecutionContext	Represents the context in which an AppleScript command is executed and tracks global state related to that command. It should not be subclassed.

Object Specifiers and Logical Tests

Objects from this group of classes represent particular AppleScript reference forms. Most of these classes are subclassses of NSObjectSpecifier, an abstract class. Objects of these classes—object specifiers—know how to evaluate themselves within the context of another object specifier that contains them. Some of these classes generate objects that represent relative or logical tests performed with object specifiers (particularly NSWhoseSpecifiers).

Table 5. Object Specifier Classes

Class	Description
NSObjectSpecifier	An abstract class for concrete subclasses that represent AppleScript reference forms. An object specifier knows how to evaluate itself (to an actual object) in the context of a container specifier.
NSPropertySpecifier	A subclass of NSObjectSpecifier for object specifiers that represent an attribute or relationship of an object.
NSIndexSpecifier	A subclass of NSObjectSpecifier for object specifiers that specify an object in a collection by index number.
NSRangeSpecifier	A subclass of NSObjectSpecifier for object specifiers that specify a range of objects in a collection by index numbers.
NSRandomSpecifier	A subclass of NSObjectSpecifier for object specifiers that specify an arbitrary object in a collection.
NSMiddleSpecifier	A subclass of NSObjectSpecifier for object specifiers that specify the middle object in a collection.
NSWhoseSpecifier	A subclass of NSObjectSpecifier for object specifiers that specify an object in a collection that matches a specified condition defined by a Boolean expression.
NSPositionalSpecifier	An object of this class represents an insertion point by reference to a point before or after another object, or at the beginning or end of a collection. It contains an object specifier that represents the object referred to for position.
NSWhoseTest	An abstract class for objects that represent Boolean expressions (qualifiers) involving object specifiers (also called “whose” clauses, as in “word whose color is blue”.
NSSpecifierTest	A subclass of NSWhoseTest for objects that represent a comparison between two objects (which can be object references before being evaluated) using a given comparison method.
NSLogicalTest	A subclass of NSWhoseTest for objects that represent the Boolean operations AND, OR, and NOT; used with one or more NSSpecifierTests.

Key-Value Coding and Value Coercion

Table 6. Scripting Utility Classes

Class	Description
NSCoercionHandler	A shared instance of this class coerces object values to objects of another class, using information supplied by classes who register with it. Coercions frequently are required during key-value coding.
NSKeyValueCodingAdditions	Additions to NSObject's implementation of key-value coding that are related to scripting.

Scriptability Guidelines

This section describes what you must do to make your application (or framework or bundle) scriptable and gives you some background rationale for each procedure. For a comprehensive discussion of scripting concepts set alongside discussions of the Yellow Box's document architecture and undo support, see "Application Design for Scripting, Documents, and Undo."

Important: This section describes how you can design your application with scripting in mind. If you have an application that you do not want to redesign according to the principles expressed here, you can still make it scriptable, but to do so will take more effort on your part.

Make Objects Responsive to Key-Value Coding

Overview

Key-value coding is a set of APIs that establishes a generic and automatic mechanism for setting and getting the properties of objects. It enables you to access object properties indirectly—and consistently—by property name (or "key") rather than through the invocation of a method or directly getting or setting an instance variable.

Key-value coding is based on two primitive instance methods of NSObject: `valueForKey:` and `takeValue:forKey:`. These methods are implemented to look first for the accessor methods corresponding to the key. For example, `[valueForKey:filter]` attempts to use accessor method `filter`; the message `[takeValue:anObj forKey:filter]` attempts to use accessor method `setFilter:` If an accessor method doesn't exist, key-value coding attempts to get or set the value of the instance variable directly.

Keys are of three general types relative to the type of associated value: attribute, one-to-one relationship, and one-to-many relationship. A typical attribute key might be "color," a window's "document" key is a typical one-to-one relationship key, and a common one-to-many relationship is identified by an application's "orderedWindows" key. In AppleScript, "property" refers to the same thing as an attribute and "element" refers to a relationship.

The Yellow Box scripting classes use key-value coding to return and set the attributes and relationships of objects. They also use it to evaluate "object specifiers." AppleScript has the notion of "object hierarchies" that define the structure of the objects of an application. It uses object specifiers (object representations of AppleScript reference forms) to navigate through this hierarchy to a particular object (for example, "word 5 of paragraph 2 of the front

document”). The evaluation of object specifiers is necessary to find the receivers (or targets) of commands and frequently their arguments.

What You Must Do

To let your application take advantage of key-value coding, you should do the following:

- Define the set of keys each scriptable class supports.
- Name your instance variables or (preferably) accessor methods accordingly.
- Implement the accessor methods that get and set the values of these instance variables.
- Specify in the external description of the class the type, Apple event code, and read-only flag for each key.

As an illustration, assume you have a DrawingCanvas object that you want to make scriptable. In a simple scenario, you want scripts to be able to access (and modify, if necessary) the graphical shapes of that object, and you also want scripts to be able to access and set the currently selected shape. To this end, you define two keys, “shapes” and “selectedShape”. These become instance variables of the DrawingCanvas class:

```
@interface DrawingCanvas: NSObject <NSCoding> {
      NSArray *shapes;
      NSBezierPath *selectedShape;
      // ...
```

Next implement accessor methods for these instance variables:

```
- (NSArray *)shapes {
      return shapes;
}

- (void)setShapes:(NSArray *)newShapes {
      [shapes autorelease];
      shapes = [newShapes retain];
}

- (NSBezierPath *)selectedShape {
      return selectedShape;
}

- (void)setSelectedShape:(NSBezierPath *)newShape{
      [selectedShape autorelease];
```

```
        selectedShape = [newShape copy];
}
```

The next step is to specify the keys “shapes” and “selectedShape” in the formal description of the DrawingCanvas class. This class description is in the suite definition you must create for any class with unique attributes, relationships, or commands (see “Create a Suite Definition and Suite Terminologies” for further information). The “selectedShape” key is an one-to-one relationship key; the “shapes” key is a one-to-many relationship. The relevant section of the class-description property list might read as follows (the Apple Event codes, of course, are made up for this example):

```
"DrawingCanvas" = {
         "ToOneRelationships" = {
                "selectedShape" = {
                    "Type" = "NSBezierPath";
                    "AppleEventCode" = "abcd";
                    "ReadOnly" = "NO";
                  };
            };
          "ToManyRelationships" = {
                "shapes" = {
                    "Type" = "NSArray";
                    "AppleEventCode" = "efgh";
                    "ReadOnly" = "NO";
                };
              (and so forth)
```

Concentrate Scriptable Behavior in Model Objects

Overview

The Model-View-Controller (MVC) design paradigm assigns objects in an application to one of the three indicated types, or roles. Model objects encapsulate and manipulate the data used by the application; they typically have no direct connection to the user interface. View objects know how to display and possibly edit data, but typically do not encapsulate any data that is not specific to displaying or editing. Controller objects act as mediators, coordinating the exchange of data between the model and view objects; controllers incorporate most application-specific logic and hence are the least reusable of the three types of objects. Applications that conform to MVC should maintain a distinct separation among objects of different types.

Generally, the objects that you make scriptable should be model objects. This principle is in line with how AppleScript is designed, and the Yellow Box accordingly gears its scriptability support to the model layer. The most efficient way for a script to perform a task is not the same thing as the best way for a user

to do the same task. A script should not require the user's involvement, unless it is intended more as macro than as batch processing. In a macro type of script, the user must prepare things for the script, and then invoke it. If you anticipate that your application will be scripted for this purpose, you may move scripting behavior to the appropriate non-model objects in your application. Yet even in this case, ensure that the scriptability of objects such as windows and selections is confined to this purpose.

What You Must Do

You should design your application with MVC in mind and ensure, as much as possible, that the objects you want to make scriptable are the model objects of your application. There are two common violations of model-layer separation that you should guard against:

1. Do not set scriptable state in action methods.

   Action methods are typically owned by a controller object. State that should be scriptable should therefore not be directly set in action methods. For example, instead of this:

   ```
   - (void)shapeSelected:(id)sender {
        /* shape is ivar */
        shape = [[sender selectedItem] representedObject];
   }
   ```

   Move the `shape` instance variable to an appropriate model object and use an accessor method to set it.

   ```
   - (void)shapeSelected:(id)sender {
        NSBezierPath *newShape = [[sender selectedItem]
   representedObject];
        [modelObject setShape:newShape]; //modelObject is ivar
   }
   ```

2. Do not keep scriptable state in user-interface objects.

   For example, suppose your application has an inspector panel with a checkbox in it. Instead of having a controller object "read" the state of this control, store the state in a model object each time the user toggles the state of the control.

Create a Suite Definition and Suite Terminologies

Overview

A suite declares the exported set of scriptable APIs of an application, framework, or loadable bundle as well as the natural-language (AppleScript) interface to those APIs. The declaration is made in two types of files. A "suite definition" contains the language-independent information for a scriptable suite. A "suite terminology" describes the language-dependent interface; one suite terminology file goes in a language-specific resource directory (**.lproj**) if you support a particular AppleScript dialect in that language.

See "Scripting Metadata" for an overview of suite definitions and suite terminologies.

What You Must Do

If your application, framework, or bundle has any objects that you want to be scriptable, you must create a suite definition for it. You also need to create a suite terminology for each AppleScript dialect you support.

Follow the instructions given in the document "Creating Suite Definitions and Suite Terminologies."

Implement Scripting Methods

Overview

Setting aside the methods of the scripting classes (which are overridden only in special circumstances), two kinds of method implementations are commonly required for scripting, accessor methods and command handlers:

Accessor Methods

Key-value coding, which is the mechanism used for getting and setting object properties and for evaluating object references, looks first to use the accessor methods based on a key specified by a scriptable class in a class description. For example, if a class specifies a key of "name," key-value coding looks for accessor method `name` to get the value of the key and for accessor method `setName:` to set the value. If no accessor methods are implemented, it attempts to set or get the value of the instance variable directly.

However, sometimes what an accessor method does in the normal use of an application—getting and setting the value of an instance variable—doesn't work for scripting. The documents managed by a document-based application provide an example. When a script asks for an application's documents,

NSDocumentController's `documents` method could be invoked to return all currently opened documents. But these documents are unordered in the returned NSArray and include both on-screen and off-screen documents. This is not what is required by scripting; AppleScript has the notion of a "front" document and a "last" document and this notion implies an ordering of documents visible on the screen. NSApplication therefore implements a method that, in response to a request for its documents, returns an NSArray that first contains all on-screen windows in their tier order and then all off-screen windows.

Another occasion for implementing an accessor method that does something more than simply set and get values is when it would be impractical to have objects at the level of granularity expected by AppleScript. For example, an AppleScript script can ask for the "characters" of a text document, but it would be quite expensive for an application to represent each character as an object. The Application Kit handles this request in a specially implemented "accessor" method.

Command Handlers

The description of a class (in a suite definition) includes a section for commands supported by the class. In this section you can either indicate that the default implementation (which is based on key-value coding) for a command is sufficient, or you can specify a method that you want to handle the command when it is executed. If the default implementation or handler for a command is insufficient for your purposes, you must specify and implement your own handler.

What You Must Do

Custom Accessor Methods

The procedure is no different than for specifying a class's actual accessor methods, except that the method you implement is used only in scripting.

1. Specify the key of the attribute or relationship, along with supporting data (type, Apple event code, read-only flag), in a property-list format in the class description. (See "Creating Suite Definitions and Suite Terminologies" for procedure and examples.)

2. Define accessor methods with names corresponding to the name of the key: `set`*`KeyName`*`:` and *`keyName`*. Then implement the methods to provide the necessary behavior that supplements or replaces the getting and setting of an instance-variable value.

3. You can also override or implement extended key-value coding methods for better performance. For example, if it makes sense to perform some special array processing for your class, you could override `valueAtIndex:inPropertyWithKey:`.

Command Handlers

1. Determine the command's return type and the type and key for each command argument. You can usually find this information by looking up the definition of the command in the Core suite definition (**Foundation.framwork/Resources/NSCoreSuite.scriptSuite**).

2. In the Supported Commands section of the class description for the class implementing the command handler, assign the method to the command.

```
"MyClass" = {
            "Superclass" = "NSCoreSuite.NSDocument";
            "Attributes" = {
                // ...
            };
            "ToOneRelationships" = {
               // ...
            };
            "SupportedCommands" = {
                "NSCoreSuite.Save" = "handleSaveCommand:";
            };
            "AppleEventCode" = "docu";
        };
```

   The method must have one and only one argument (and so in Objective-C must end in a colon).

3. Implement the command handler. The signature of this method must be of the form:

```
- (id)methodName:(NSScriptCommand *)command
```

   In the actual implementation code, get the command's arguments, handle the command, and return the expected value. If no return value is expected, return `nil`.

```
- (id)handleSaveCommand:(NSScriptCommand *)command {
    NSDictionary *args = [command evaluatedArguments];
    NSString *file = [args objectForKey:@"File"];
    NSNumber *ftype = [args objectForKey:@”FileType”];

    if (file) {
        /* handle command here */
```

```
        }
        return nil;
    }
```

Define a New Script Command

Overview

Sometimes an application can accomplish something for which none of the default commands suffice: “Fax” for example, or “Collate.” If you want that specialized behavior to be scriptable, you must define and implement a script command for it.

What You Must Do

1. Define the command in the Command Description section of the suite definition; the information you must supply is return type, arguments, and command class (as well as associated Apple event codes).

2. Add terminology information for the command to the suite terminology.

3. For each class that you want to send the command to, specify the command in the Supported Commands section of the appropriate class description.

For the procedures related to steps 1 through 3, see “Creating Suite Definitions and Suite Terminologies.”

4. If you want a default implementation for the command that is based on key-value coding, make a subclass of NSScriptCommand and override `performDefaultImplementation`.

5. If you want a scriptable class to handle the command, define and implement a method for it (see “Command Handlers” for details). Make sure to specify this method in the Supported Commands section for the class implementing the method.

You must complete either step 4 or step 5, and you can complete both steps.

Use the Document Architecture

Overview

The Application Kit provides a few classes that underpin the Yellow Box’s document architecture: NSDocument, NSDocumentController, and NSWindowController. These classes directly implement the standard AppleScript document scripting model. If you use these classes to

implement a document-based application, that application automatically supports scripting by a number of AppleScript commands, including Open, Close, and Save. To acquire this scriptability, you need do nothing more than use the document architecture as it was designed.

Applications that take advantage of the Yellow Box's document architecture put themselves in a better position to support scripting generally. A document in Yellow Box applications (NSDocument) typically owns and manages one or more model objects of the application. It therefore provides a hub for scripted access to the model objects in your application, which are the ones that should be scriptable.

For a discussion of the Application Kit's document architecture, see "Application Design for Scripting, Documents, and Undo."

What You Must Do

Documents are represented by an NSDocument object. To implement a document-based application, you must, first and foremost, create a subclass of NSDocument that gives documents knowledge of the application's model layer and that loads and saves document data. NSWindowController objects manage the user-interface associated with documents. Simple applications can use the default NSWindowController object, but typically you subclass NSWindowController to give window controllers specific knowledge of the user interface they're supposed to manage. In your NSDocument subclass, you then override the `makeWindowControllers` method to create the window controllers used by the document. Finally, there is the NSDocumentController shared instance, which tracks and manages documents and performs common functions of application delegates (handling application termination, for example). You shouldn't need to subclass NSDocumentController.

For a detailed discussion of what you must do to implement a document-based application, see the reference documentation for the NSDocument class.

Access the Text Suite

Overview

The text objects (particularly NSTextStorage) in the Application Kit export scriptable APIs defined in the Text suite. (See "Text Suite" for more information.) Scripts can thus request or select textual elements at different levels of granularity: character, word, paragraph, and entire body of text. Applications automatically acquire this scriptability if they acquire access to a NSTextStorage object through their container hierarchy.

What You Must Do

To acquire text scriptability, drag a text object from Interface Builder's DataViews palette into your application's user interface. Alternatively, your program can programmatically create a text object.