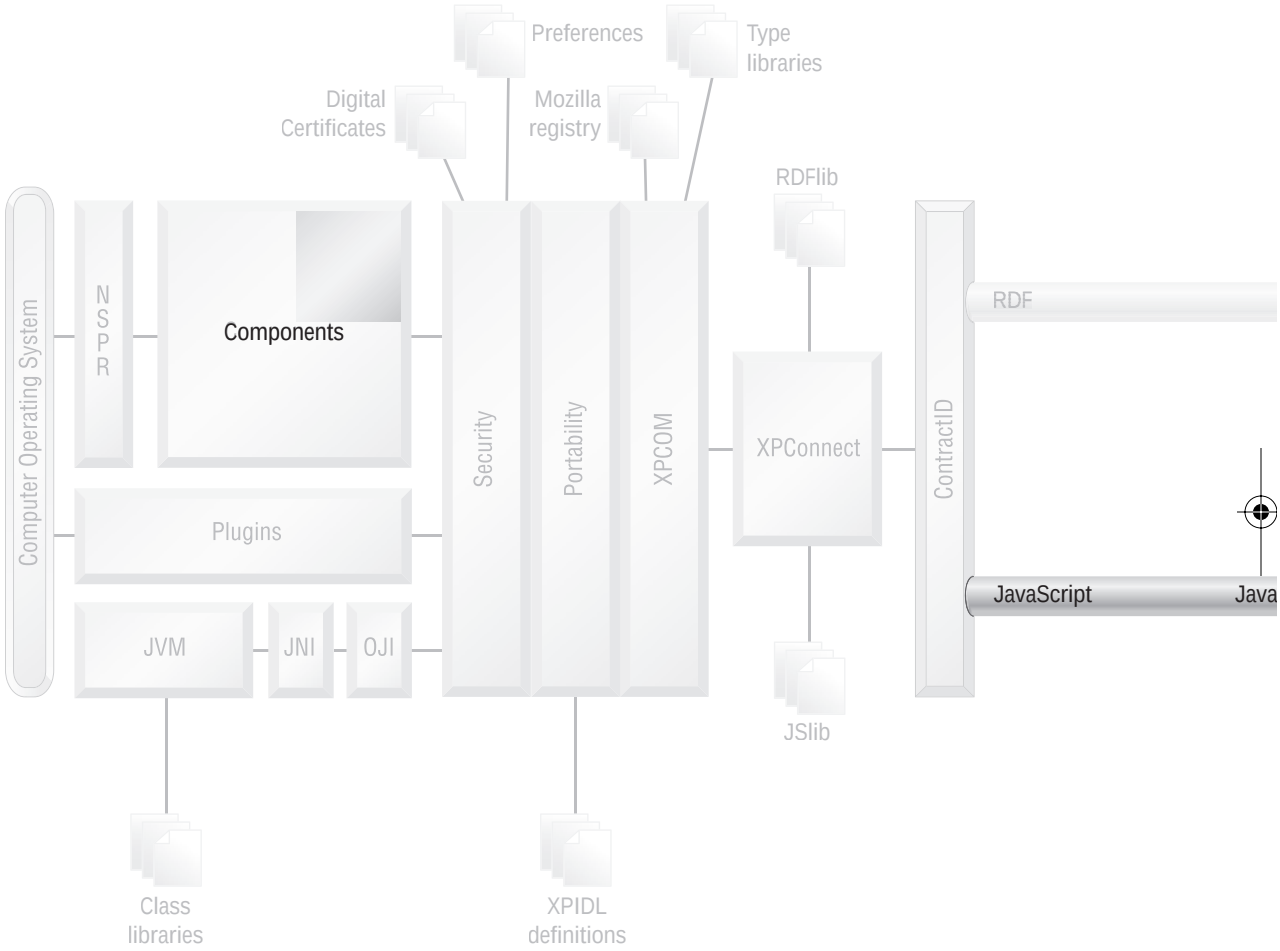
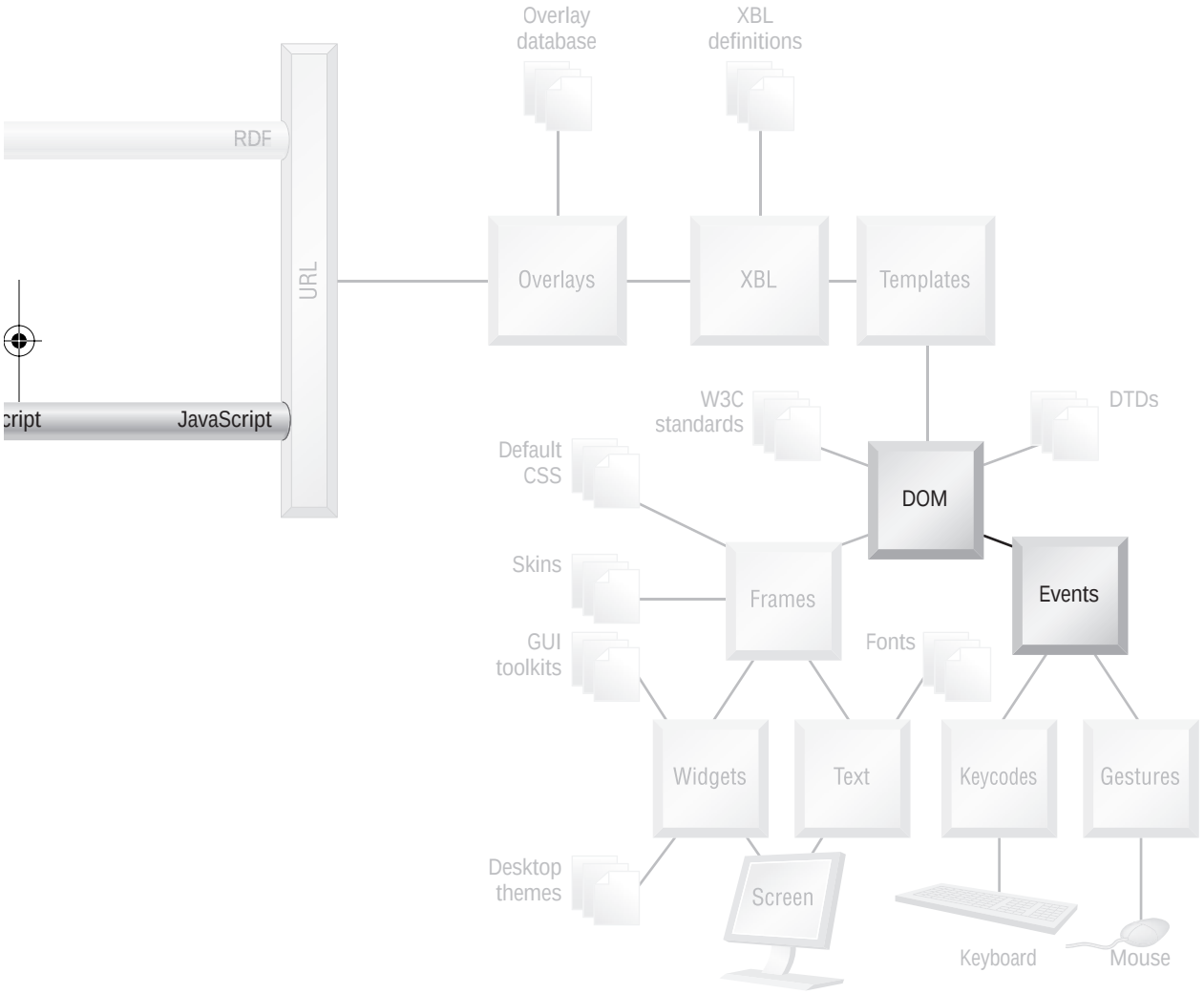

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 9

Commands



This chapter explains the Mozilla command system. The command system is used to separate the functionality of an application from its appearance. It is separate from the DOM Events model, although events and commands can interact a little.

It is critical that somehow, somewhere, the tasks that an application window performs are separated from their user interfaces. One reason for this separation is that XUL user interfaces are highly changeable, both at design time and at run time. Such interfaces are a demanding constraint on application architecture. A second reason is that many existing software engineering techniques can be applied to application tasks if they can be expressed by themselves. Use cases and functional hierarchy diagrams are examples of such techniques. Separately defining tasks also make reuse more likely. Overall, such a separation strategy is an attractive and flexible design environment.

A professionally built application will have all its commands formally identified at design time. Identifying to-be-implemented commands is part of the design breakdown process and can be used to detect convoluted, duplicate, and messy actions. These design problems can then be addressed before implementation rather than afterward.

Traditional HTML-based applications often consist of a spaghetti pile of JavaScript code. The command system is also an attempt to get away from this unstructured approach.

The NPA diagram that precedes this chapter illustrates the pieces of Mozilla that implement commands. From the diagram, the command system builds on top of the DOM and the DOM Events subsystems, but it also requires access to a number of XPCOM components. The most important components also appear as AOM objects. The XUL language has tags that support the command system, but those tags have no visible appearance; they are similar to the `<key>` tag in that respect. The command system goes further than the `<key>` tag because it has no user interface of any description. The command system is entirely internal to the platform.

9.1 COMMANDS AND MOZILLA

Mozilla's command system is one of the less obvious aspects of the platform, and yet it is powerful and flexible. It allows the application programmer to think of application functionality as a set of messages. Each message is either a command or about a command. Used correctly, the command system acts as an integration and organizational point for application functions. Classic Mozilla's own use of its command system is fairly organized, but it is also buried in an abundance of other code.

The Mozilla command system is designed to support complex applications. Trivial applications do not need a command system; they can get by

with simple event handlers. For complex applications, Mozilla's design goals are to provide a system where

- User interface widgets can share a command, even across source files.
- Commands can have their own state, which can be changed and reported on.
- Commands and widgets can be added or changed independently.
- All kinds of programmers, not just application programmers, are catered to.
- Useful default behavior exists.
- Simple uses are supported with trivial syntax.
- The existing DOM Event system is reused where possible.

In Mozilla, a command is very easy to find. Simple operations such as Save File, Add Bookmark, Select Content, Bold, Scroll One Page, and Undo are all commands.

It is not so easy to pin a command down in terms of code. The most obvious programming signature a command has is a simple string whose name is the command name. Such names can be predefined by the platform, or they can be programmer-defined. Unfortunately, the rest of the command infrastructure is rather spread out. Bits of the command system exist in XUL, JavaScript, XPCOM, existing chrome files, and the platform's internals.

9.1.1 Hello, World

A simple example of the command system at work is shown in Listing 9.1.

Listing 9.1 hello, world implemented as a Mozilla command.

```
<?xml version="1.0"?>
<!DOCTYPE window>
<window
  xmlns= "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <command id="hello" oncommand="alert('hello, world');"/>
  <button label="Say It" command="hello"/>
</window>
```

The `<command>` tag implements a command, in this case a simple event handler that calls `alert()`. The `<button>` tag identifies that command by name. When the button is pressed, a DOM 2 Event with the special type command is generated, and the identified command traps this event and runs. The result is that an `alert()` box is thrown up.

Commands are not always so simple, and they do not always involve events. If a command is implemented in JavaScript, then the effect achieved in Listing 9.2 is the same as that in Listing 9.1.

Listing 9.2 hello, world implemented as a Mozilla JavaScript command.

```
<?xml version="1.0"?>
<!DOCTYPE window>
<window
  xmlns= "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <script>
    var control = {
      supportsCommand  : function (cmd) { return true; },
      doCommand        : function (cmd) { alert(cmd + ", world"); },
      isCommandEnabled : function (cmd) { return true; },
      onEvent          : function (event_name) {}
    };
    window.controllers.appendController(control);

    function execute(cmd) {
      var disp = document.commandDispatcher
      var cont = disp.getControllerForCommand(cmd);
      cont.doCommand(cmd);
    }
  </script>
  <button label="Say It" onclick="execute('hello')"/>
</window>
```

This code makes a custom object named `control` that implements the command; in fact, this object can implement several commands. The command is implemented in the `doCommand()` method. This object needs to be installed into the platform's command infrastructure as well. It is retrieved again when the command to be run is required. Finally, using this approach, the command can be run from any point in the XUL document that calls the `execute()` function. In this case, it is convenient to use an `onclick` handler.

This second example requires and explanation, and that is where this chapter begins. Obviously this second example is more complex than the first, and there are good reasons why using a more complex approach is sometimes better.

9.2 COMMAND CONCEPTS

The Mozilla command system is a command delivery system. Actual commands are implemented by the application programmer. Creating and running commands is straightforward compared to understanding how they are invoked, found, and executed.

This delivery system is very different from a traditional client-server architecture. In such an architecture, servers implement commands that are entirely separate and remote from client GUIs. They often run silently and only report when finished. For example, HTTP GET and POST requests ignore

browser GUIs entirely; they merely return a status code, and possibly a new document.

Mozilla commands are not like client-server commands. Mozilla commands are close to the GUI and are likely to operate on it extensively. An example is the Bold operation in the Classic Composer (or similarly in any word processor). Bold applies a style to the current selection, which is usually on display. The Mozilla command delivery system must allow commands to be executed close to the GUI, not buried in some server. Of course, those commands can use server-like features if they so choose.

This last point means that commands are not "distant" from the application programmer. They are part of the content that the application programmers create and are loaded into the XML document environment like other content.

Mozilla uses a common piece of design called the Command pattern to separate command names from command implementations. Most modern GUI toolkits include technology based on the Command pattern. The next few topics build up a picture of how this pattern works.

9.2.1 The Functor Pattern

The Functor pattern is a well-known piece of design that is the lowest level of a command system. A functor can be implemented as an object. That object represents a single function. Listing 9.3 illustrates a functor and a normal function. Both provide an implementation of a "halve" command that calculates half of a supplied number:

Listing 9.3 Function versus functor object for a single operation.

```
// Plain function
function halve_function(num) { return num / 2.0; }

// Functor
var num = null;

var halve_functor = {
  enabled : true,
  exec : function () { num /= 2.0; return true; }
}

// Examples of use
num = halve_function(23);  // sets num = 11.5

num = 23;
halve_functor.exec();      // sets num = 11.5
```

The functor object not only seems unnecessarily complex compared with the simple function, but it also has a loose global variable to deal with. The functor, however, is far more flexible than a function because it has a stan-

dardized interface. All functor objects have a single `exec()` method that executes the implemented command and reports success or failure. All functor objects therefore look the same to an application programmer.

The example functor object also contains some state information about the command. In Listing 9.3, the very common example of an `enabled` state is illustrated. In fact, most command systems, including Mozilla's, explicitly provide support for the `enabled` state. Application code can check this state to see whether the command is available and then react appropriately. A functor can store any kind of state information about a command that seems convenient, not just an enabled state. Other states that might be stored include the command name; a unique id for the command; flags to indicate whether the command is ready, blocked, or optimized; or the language the command is implemented in or its version. Anything is possible.

In this example, the functor object modifies global data (the `num` variable) when the command executes. This keeps the `exec()` method free of parameters so that all functor objects are alike. `num` could be a property of the functor, but that is a bad design choice. It is a bad choice mostly because the functor should be stateless with respect to the command's execution. Although it contains state information about the command (the enabled property), the command's implementation does not use any of that state information when it executes. That state information is used only by those that need the functor object.

The example functor also presents an opportunity. The `exec()` method could be changed to this:

```
exec: function () { return really_halve_it(); }
```

In this alternative, the `really_halve_it()` function does all the actual work for the command and is implemented somewhere else in the application, perhaps in a library.

This last change means that the functor object contains none of the command implementation. Instead, it is the programmer's point of access to the command and its state. It is a proxy object, or a handle, or a façade. It is a representative for the real functionality. This proxy object provides part of the separation required for the Mozilla command system. Application programmers only need to know where the proxy object is, not where the final command implementation is. This is simple abstraction, similar to the design of file descriptors, symbolic links, network mapped drives, and aliases.

In non-Mozilla literature on the Functor pattern, `exec()` is often called `execute()`. In Mozilla, a proxy object (a functor) for a command is called a *command handler*. Application programmers do not use command handlers much because the higher-level concept of a controller is more convenient.

9.2.2 The Command Pattern and Controllers

The command pattern is a well-known piece of design that builds on the functor pattern. It is responsible for separating the GUI side of an application from

a command's implementation. A functor removes the need to know the exact location of a command's implementation. A command pattern removes the need to know whether a command exists.

In reality, application code generally assumes that the commands that it relies on do exist. This is just a matter of forward planning, and peeking "behind the scenes."

To implement the command pattern, create an object that holds a set of functor objects. When the user supplies a command name to that object, execute the matching functor object. That is the command pattern. Listing 9.4 is an example of such an object, called a *controller* in Mozilla. This example implements a simple traffic light.

Listing 9.4 Controller object implementing a stop sign.

```
// functors
var stop = { exec: function () { top.light = "Red"; };
var slow = { exec: function () { top.light = "Amber"; };
var go   = { exec: function () { top.light = "Green"; };

// controller containing functors

var controller = {
  _cmds: { stop:stop, slow:slow, go:go },

  supportsCommand : function (cmd) {
    return (cmd in this._cmds);
  },
  doCommand : function (cmd) {
    return _cmds[cmd].exec();
  },
  isCommandEnabled : function (cmd) {
    return true;
  },
  onEvent : function (event_name) {
    return void;
  }
};

// set the light to green
controller.doCommand("go");
```

The controller object contains an object `_cmds` that maps property names to functors. In a language with full information hiding, like C++, this inner object would be private. The three functions supplied are convenience functions that take a single command name as a string and work on the stored information. `supportsCommand()` reports whether the controller knows about a particular command; `doCommand()` executes the functor for a given command. `onEvent()` is passed any event name that might be relevant to the command.

The final `isCommandEnabled()` method cheats a little. It should check the enabled property of the matching functor, but this controller knows that all three lights on the traffic light always work, so it never checks—it just returns `true`. That is just as well because none of the functors implement the enabled property shown in Listing 9.4. This choice means that the controller and the functors are dependent on each other. They appear to implement three complete functors, but in fact those three functors only implement what the controller really needs.

This last point is important. Because the controller completely hides the command implementations (functors) from the user, those implementations can be done in any way. In Mozilla, it is common for simple controllers written in JavaScript to avoid functors altogether and to implement the command states and command implementations directly. This would not be possible if the controller had a `getFunctor(cmd)` method because then the functor would be exposed to the application programmer, requiring it to be complete. Happily, the controller has no such method, at least in its simplest form, and so implementation shortcuts can be taken. Listing 9.5 shows how Listing 9.4 can be stripped down even further.

Listing 9.5 Controller object with no separate functors.

```
var controller = {
  supportsCommand : function (cmd) {
    return (cmd=="red" || cmd=="amber" || cmd=="green");
  },
  doCommand : function (cmd) {
    if ( cmd == "red"   ) top.light = "Red";
    if ( cmd == "amber" ) top.light = "Amber";
    if ( cmd == "green" ) top.light = "Green";
  },
  isCommandEnabled : function (cmd) {
    return true;
  },
  onEvent : function (event_name) {
    return void;
  }
};

// set the light to green
controller.doCommand("go");
```

In this example, the controller still acts as though three functors are present, but none are actually implemented.

Complex controllers can also benefit from direct implementation. Sometimes commands have coordination problems that are best solved in the controller. An example is implementing Undo in an editor like the Classic Composer. The controller is responsible for executing the individual commands, so it might as well be responsible for maintaining the Undo history

too. If the command to execute is Redo or Undo, then the controller can read the top item from the history and can call the correct command implementation to fill the Redo/Undo request. Any such state machine should first be implemented without a controller and then hidden inside the controller so that the controller's main purpose (controlling) is still clear. Controllers are also good places to hide macro languages, synchronization, and other artifacts that are made out of commands.

9.2.3 Controller Sites

Mozilla does not stop at a single static controller. It is possible to create as many controllers as you want. This is done, for example, in Mozilla's Composer, where there are three controllers, each of which looks after a different subset of the implemented commands.

Any and all controllers created must be placed where the platform can find them. Controllers can be lodged in several places.

- Controllers can be lodged on the window object of a chrome (XUL) window.
- A few XUL tags are suitable sites. Allowed tags are `<button>`, `<checkbox>`, `<radio>`, `<toolbarbutton>`, `<menu>`, `<menuitem>`, and `<key>`.
- The XUL `<command>` tag is like a functor when it directly implements a command using an event handler. If such a tag is created, the platform effectively adds that functor to a permanent controller that it maintains internally.

Use of these sites is discussed later when command syntax is explored.

If more than one controller is lodged at a particular site, then the set of such controllers at that site is called a controller chain. Such a set of controllers is ordered, and when it is consulted, it is searched in order. This means that the first controller in a chain that can fill a request for a command implementation will be the controller that runs that command.

9.2.4 Dispatchers and Dispatching

Controllers hold a set of commands each; controller sites hold zero or more controllers each; and a command dispatcher works with a set of controller sites. Fortunately, it stops there. A command dispatcher has the job of finding and executing a particular command.

Mozilla has one dispatcher designed for HTML documents and one dispatcher designed for XUL documents. The HTML dispatcher is not visible or available to scripts under any circumstances. The XUL dispatcher can be scripted. Each XUL application window has one dispatcher object at its document root, and that object is always present and available.

The dispatcher has a `getControllerForCommand()` method. This method accepts a command name as a string and returns a controller that can execute that command. The dispatcher and the returned controller must be explicitly coded by an application script—they do not do any anything automatically. In the chrome files of Classic Mozilla, calls to the dispatcher are hidden inside a custom-made JavaScript function called `goDoCommand()`. That function is included in every Classic Mozilla window and is used in most.

The dispatcher uses state information about the current focus to work out what controllers it should provide for a given command. This state information consists of the window focus, any XUL `<iframe>` focus, plus any DOM element in the focus ring that is the currently focused member of the ring. In other words, the dispatcher uses all the focus hierarchy information. The dispatcher starts at the most specific item currently in focus (usually a form or menu item) and works its way up the DOM tree to the outermost window. At each focused DOM element it finds, it examines the element's controller chain for controller objects. The dispatcher then checks support for the given command by testing each controller in the chain using that controller's `supportsCommand()` method. The first controller object the dispatcher encounters that implements the dispatched command is returned. If none match, the dispatcher moves further up the focus hierarchy. If the dispatcher reaches the top of the focus hierarchy without finding a suitable controller, it returns `null`.

A XUL window and its contents are not fully initialized with the focus when that window first appears. Even though the window is the current desktop window, it can still be missing the focus. This means that installed controllers are not necessarily available, even if they are installed at the top of the current window. The application programmer must explicitly give the window focus if these window-level controllers are to be made available. The simplest way to do this is with a single line of script:

```
window.focus();
```

It is very important that focus is explicitly set both before and after a command is dispatched. If this is not done, the focus ring can become confused, and problems can occur from that point onward. This constraint has been fixed to a degree with the 1.4 release, but explicit focusing is still recommended practice.

The dispatcher does not search everywhere for a command implementation. If a focused widget has a `command` attribute, the dispatcher will not consider that attribute. The dispatcher only examines controllers.

9.2.5 Change Notifications

One important enhancement to the command system is a change notification system for the commands themselves. This change notification system is also called command updates.

Commands can be created and managed, commands can be found and executed, but what if commands change? Suppose a command's `enabled` state changes from `true` to `false`. How can the pieces of an application that use the command find out that it is no longer available? Change notification is the solution to this problem.

Chapter 6, Events, introduced the observer-broadcaster pattern. Mozilla uses that pattern for command state changes. Instead of one or more XUL tags observing an attribute on another XUL tag, one or more XUL tags observe the proxy functors of particular commands. If a command's state changes, the dispatcher broadcasts change notifications about its functors, and the observing XUL tags catch those notifications. Even when the functor objects are merged in with the ordinary controller code for simplicity, change notifications are still generated. Therefore, imagining that command functors are always there is convenient because it explains neatly the change notifications that are observed.

The dispatcher keeps a list of observer XUL tags in the form of DOM element objects. These are normal DOM objects but are called command updaters when used for this purpose. Methods are provided on the dispatcher for adding and removing such updaters.

A DOM Event is created and inserted into the event system using the `dispatchEvent()` method. A command change notification is inserted into the broadcaster-observer system using the `UpdateCommands()` method of the dispatcher or of the window. This causes all the observers watching the single specified event to receive a `commandupdate` event. This event acts like any event that a XUL tag might receive.

`commandupdate` events work in XUL, but not in HTML. The syntax and use of these events is covered shortly.

9.3 How Commands Are Started

Mozilla commands can be started a number of ways. Here are all the possibilities:

- If a functor or controller object is available, its methods can be called directly from JavaScript. This approach ignores the command system and treats the object like any other object.
- The dispatcher's `getControllerForCommand()` method can be called directly from JavaScript. This method is a first step toward resolving a command name into actionable code. In this case, the dispatcher searches for a suitable command implementation.
- The `doCommand()` method of a focusable XUL tag can be called directly from JavaScript. If this is done, then any `<command>` specified as a value of the `command` attribute will be executed. This method does not use the dispatcher or any controllers.

- If a key with a suitable <key> tag is pressed, or if a focusable XUL tag is clicked or stimulated with a key press, then any <command> specified as a value of the command attribute will be executed. This is the same as the doCommand() example, except that the tag or key first generates a command event. Special processing in the platform is responsible for checking both the event type and any command attribute to see if a command needs to be run.
- If a DOM Event that is a command event is created, then the dispatchEvent() method can be called on any XUL tag. If that tag is a focusable tag, then that method call is the same as the user input generating a command event. It is therefore the same as the previous case.

In any of the previous three cases, the <command> executed might elect to call the dispatcher. This links the <command> tag to a controller.

- Finally, if the UpdateCommands() method of the dispatcher is called, any outstanding change notifications will be sent to all XUL tags observing such changes. This means that a commandupdate event will be sent to all command updaters.

The Classic Mozilla applications include a function called goDoCommand(). This function manipulates the dispatcher. It is the starting point for commands in those applications. It is located in globalOverlay.js in toolkit.jar in the chrome.

9.3.1 A Virtuous Circle

A summary of the Mozilla command system reveals a convenient twist for the application programmer. A command implementation can begin and end in a single XUL document, as the earlier "hello, world" examples illustrate.

- Commands can enter the command delivery system from an ordinary piece of JavaScript. That script might itself be an event handler, perhaps for a more traditional DOM event like the click or select event. In that case, the script must be associated with a XUL tag.
- Commands can leave the command delivery system from an ordinary piece of JavaScript that implements the command. A command functor, or better still a controller, can be written to contain that implementation. That controller can also be attached to a particular XUL tag.
- Finally, a XUL tag can get feedback if a command's state changes, via the command update system.

Altogether, this means that a command can begin and end in XUL (or in XUL and JavaScript). For example, a <toolbarbutton> that initiates a command can also observe the command to see if it should stay enabled. That <toolbarbutton> can also supply the command's implementation with a

controller. The dispatcher and standard event processing are the only pieces of the puzzle supplied by the platform.

9.4 USING COMMANDS VIA XUL

XUL provides the <command> tag for Mozilla commands. The tags <commandset> and <commands> are also used, but they are user-defined tags with no special meaning of their own. A <commandset> tag is used to contain a set of <command> tags and acts like the other XUL container tags, like <keyset>.

XUL also contains a number of XML attributes that can be applied to any tag. They are

```
command events targets commandupdater
```

and two event handlers:

```
oncommand oncommandupdate
```

Outside of XUL, the Mozilla Platform also has many predefined command names, but we'll discuss those in more detail later.

9.4.1 <command> and command=

The <command> tag is used to define a Mozilla command, just as the <key> tag is used to define a Mozilla keypress. It represents the command and can embody the concrete aspects of the command. In terms of a virtuous circle, the <command> tag represents both an identifier to use when invoking a command and an implementation to use when the command executes. The <command> tag has the following special attributes:

```
disabled oncommand
```

The disabled attribute can be set to true, in which case the command does nothing. oncommand is set to JavaScript code that will be used in place of any controllers that might exist. Other attributes, such as disabled, label, accesskey, and checked are sometimes added to <command>. The command update system, discussed next, uses them, but they have no special meaning to the <command> tag.

An example of <command> use is

```
<command id="test-command" oncommand="alert('executed');"/>
```

The id attribute names the command. The oncommand event handler provides an implementation for the command—it is a command handler. If this tag generates a DOM Event named command, then this handler will run. Because the <command> tag has no interactive features (no widget), it is rare that this tag ever generates a command event.

The command attribute allows other tags to be extended by the <command> tag in the same way that the key attribute allows other tags to be extended by a <key> tag. For example,

```
<mytag id="myAppTag" command="test-command"/>
```

If the <mytag> tag generates a command event of the right type, then the alert() specified earlier by the oncommand handler of the <command> tag will execute. Only the following tags can be used in place of "mytag":

```
<button> <checkbox> <radio> <toolbarbutton> <menu> <menuitem> <key>
```

When one of these tags has the focus, user interaction generates a command event automatically. The oncommand handler can be usefully added to these tags, but doing so is poor practice. It is better to record all known commands centrally in <command> tags than to fragment the commands across all the application's widgets.

The oncommand handler can also be fired directly from JavaScript:

```
var target = document.getElementById("mytag-id");
target.doCommand();
```

If the oncommand attribute is specified, then the command system's dispatcher is not used, and no controllers are consulted. Only the code in the handler is executed. If the handler code wishes, it can call the dispatcher, which then operates as it always does.

There is one further use of the command attribute. The XBL tag extension system has a <handler> tag, which allows an ordinary DOM Event like click to be translated into a specified command. The command attribute is used to do this in the XBL <handler> tag.

In the XBL case, there is no matching <command> tag for the <handler> tag. The value of the command attribute is sent straight to the dispatcher as the sole argument of getControllerForCommand(). Therefore, command in XBL is different from command in XUL.

9.4.2 commandupdater="true" and oncommandupdate=

If a XUL tag contains this attribute, then it is a command updater and will receive change notifications about commands. These change notifications are received through a broadcaster-observer arrangement. When the platform's XUL parser discovers the commandupdater attribute, it automatically registers the tag holding that attribute as an observer of a broadcaster inside the dispatcher. This broadcaster causes commandupdate events to be placed on all its observers when the UpdateCommands() method of the dispatcher is run. In summary, if a commandupdate event occurs, this tag's oncommandupdate event handler will fire. An example of this code is simply

```
<mytag commandupdater="true" oncommandupdate="update_all()"/>
```

The update_all() function is any custom script that is knowledgeable about the current application. It does whatever work is required to identify the commands that have changed. It also performs any consequential actions. For example, there is such a function in Classic Mozilla. The file globalOverlay.js in toolkit.jar in the chrome contains goUpdateCommand() (and other go… functions). That function performs some standard processing associated with command updates. It sets the disabled attribute of a <command> tag by checking the isCommandEnabled() method of the controller for a command that is passed to it as an argument.

A further example of command update is discussed in "How to Make a Widget Reflect Command State."

A filter can also be applied to a command updater tag:

```
<mytag commandupdater="true" events="focus blur" targets="*"
     oncommandupdate="update_all()"/>
```

The events attribute restricts the events that will cause a commandupdate event. It can be a space- or comma-separated list of events and can include command names. The targets attribute has the same syntax, except the list of names consists of tag ids. It is rarely used and is not recommended. It restricts the events reported on to those whose event target (tag) is in the supplied list of ids. The default for both events and targets is "*", meaning all.

Even if the targets attribute is specified, the commandupdate event only goes to the commandupdater's tag, not to the target tags. We're still waiting to see what this functionality finally evolves into.

A commandupdate event can be synthesized from JavaScript, just as a command event can be. To do this, call the UpdateCommands() method of the dispatcher. In the case of XUL-defined commands, this is not a very flexible or useful arrangement because the simple event handlers used in oncommand have no state. Only when JavaScript is used to construct commands (using controller objects) do commandupdate events start to make sense.

9.4.3 <commandset> and <commands>

The <commandset> tag is an user-defined tag with no special meaning. It is sometimes used to group commands.

Sometimes a XUL application is built from a number of separate definition files. In that case, the <commandset> tag often has an id that allows those separate definitions to be merged, as described in Chapter 12, Overlays and Chrome.

The second use of <commandset> is to be a command updater. Such a tag is well placed to handle command updates because all its direct children are typically <command> tags. When a commandupdate event occurs, the oncommandupdate handler of a <commandset> tag can iterate through its children and get each one to check its status. This strategy is used throughout the Classic Mozilla's chrome.

Finally, the <commands> tag is also a user-defined tag with no special meaning. It is used just to group <commandset> tags. It is also exploited by Mozilla's use of the overlay system. Matters can be arranged using overlays so that all <commandsets> distributed across an application's files end up as children of the <commands> tag.

9.5 USING COMMANDS VIA THE AOM

Direct use of XUL is a convenient way to implement simple commands, but it doesn't allow for imaginative designs. Such designs require scripting, and such scripts can work with several objects.

Every XUL tag in the document hierarchy can be an event target, and so every tag supports the dispatchEvent() method. Which implementation of a command is executed ultimately depends on what DOM element the dispatchEvent() method is called on.

The document object in a XUL document contains a commandDispatcher property. This property is the sole command dispatcher. It contains the UpdateCommands() method and is also used to add command updaters. The command dispatcher also provides an interface that allows the application programmer to move the focus of the document through the widgets in the focus ring.

Every XUL tag in the document hierarchy also has a controllers property. This property is an object that is a collection of controllers. This set of controllers is the set that the dispatcher will look through for a controller that can execute the command dispatched. Because only some tags fully support the command system, and because all commands instigated by the user start with a command event, it is important to choose the correct DOM object for a command. If you choose the wrong object, the dispatcher will look through the wrong set of controllers.

- In XUL, this controllers array is empty for all tags except <textbox>. <textbox> has one controller item because it is constructed from HTML's <input type="textbox">.
- The window.document property is not a tag and does not have a controllers property.
- The window object itself is not a XUL tag, but it has a controllers property, whose array contains a single controller. This single controller is called the focus controller and is responsible for managing window focus and focus ring events.

Individual, application-specific controllers are the objects that application programmers are most interested in. No such controllers exist by default. Even after creation they are not accessible outside the chrome. It is up to the

application programmers to create these objects, which may be built with or without functors.

Command functors managed by controllers can be created directly in JavaScript. Such functors must support the `nsIControllerCommand` interface, which looks like this when implemented in JavaScript:

```
var functor = {
  isCommandEnabled      : function (cmd) { … },
  getCommandStateParams : function (cmd) { },
  doCommandParams       : function (cmd) { },
  doCommand             : function (cmd) { … }
}
```

Normally a functor implements only one command, but in Mozilla a functor may implement many commands at once. This is just an extra feature in case it is efficient to implement several commands with one functor. The single argument of each method of the functor states which command implemented by the functor should be acted on.

In the ordinary case, the `Param`-style methods are never used, and so do nothing. Such a functor is then registered with a command controller object.

If a controller exists, functor registration can easily be done using the interfaces available, for example:

```
controllers.getControllerAt(2).registerCommand(cmd,functor);
```

In practice, only one reusable controller exists, and it is not immediately available in the AOM (it must be created via XPCOM). Therefore, controllers are also made from scratch using JavaScript. Such a controller object must implement the `nsIController` interface. In JavaScript:

```
var controller = {
  supportsCommand  : function (cmd) { … },
  isCommandEnabled : function (cmd) { … },
  doCommand        : function (cmd) { … },
  onEvent          : function (event) { … }
}
```

The functors can then be used, declared, or even implemented inside the body of the controller object, as illustrated earlier. If a more structured approach is desired, the controller can also implement the `nsIController-Context` interface. In that case, the functor objects can be added and removed from the controller at run time, after the controller is initialized with an `nsIControllerCommandTable` object. Such an object can also be created in JavaScript.

After the controller is complete, the next task is to add it to the appropriate tag's DOM element object. This DOM element can only be one of a few tags.

```
aNode.controllers.appendController(controller);
```

The `onEvent()` method only fires if application code calls it explicitly. It does not fire automatically.

The window object's `controllers` collection is the best place to put general-purpose commands.

In the case of a pure XUL window, the final implementation step is to initialize the focus so that the dispatcher has at least a focused window to examine when a command is invoked. The simplest way to do this is to focus the whole window, using the window AOM object:

```
window.focus()
```

If this is not done, then you must rely on some other piece of script (or the user) setting the focus before any commands are run.

The Addressbook of the Mozilla Messenger client is a good example of the command system at work. Two controllers are created there, one for each `<tree>` tag in the Addressbook dialog box. The file `abCommon.js` in the chrome file `messenger.jar` is a good starting point for study.

9.5.1 How to Make a Widget Reflect Command State

One of the main goals of the command system is to allow user-visible widgets to change dynamically. For example, a menu offering Cut, Copy, and Paste operations should have the Paste operation grayed out (disabled) if nothing has been cut or copied yet. Any state information responsible for such changes should rest with the command that implements Paste, not with the GUI widget. This is because the Paste action might be offered to the user by several different widgets, such as a menu item and a toolbar button.

The command and command update systems support this design. Listing 9.6 shows a XUL fragment that is half of such a system.

Listing 9.6 Controller object with no separate functors.

```
<updater commandupdater="true" oncommandupdate="update()">
  <command id="paste" oncommand="doPaste()"/>
</updater>
<button label="Paste1" command="paste" observes="paste"/>
<description value="Paste Enabled" observes="paste"/>
```

The `<updater>` tag is a user-defined tag, which could alternately be called `<commandset>`—that tag name is not so important. The `<button>` and `<description>` tags represent two places that reflect the state of the Paste command. Since `<description>` does not have special support for the `command` attribute, it does not act as a user-input widget, it acts as a read-only status indicator.

The `doPaste()` function is the implementation of the Paste operation. This function might or might not require the dispatcher, depending on how it is implemented. The `update()` function is responsible for reflecting the state

of the command into the GUI. It also could be implemented with or without the dispatcher, but if good design is used, it will look for a controller or functor representing the command and extract required state from there.

The second half of this system is JavaScript. Listing 9.7 shows the `update()` function.

Listing 9.7 Controller-based command updater.

```
function update()
{
  var cont, node;
  cont = document.commandDispatcher.getControllerForCommand('paste');
  node = document.getElementById('paste');

  if ( !cont || !node ) return;

  if ( cont.isEnabled(cmd) )
  {
    node.removeAttribute("disabled");
    node.removeAttribute("value");
  }
  else
  {
    node.setAttribute("disabled", true");
    node.setAttribute("value", "Paste Disabled");
  }

  if ( cont.clipboardEmpty() )
  {
    node.setAttribute("value", "Nothing to Paste");
  }
}
```

This function is tied to certain commands: It knows only about the Paste command. It seeks out the implementation for that command and then examines it with `isEnabled()` and `clipboardEmpty()`. `isEnabled()` is a standard method of a controller, but this controller must have extra features because `clipboardEmpty()` is new. This particular command has two states: whether it is enabled, and whether there is anything in the clipboard. If there is nothing in the clipboard, then there is nothing to paste.

Based on the analysis of these states, the `update()` function loads the `<command>` tag with extra XUL attributes. None of these attributes is meaningful for `<command>`. They are put there only so that the widgets observing the `<command>` tag can pick them up. Both the `<button>` and the `<description>` tags pick up attributes `disabled` and `value`, but `disabled` only has special meaning for `<button>`, and `value` only has meaning for `<description>`.

The net result is that the widget tags are updated, and their appearance changes. All this is in response to one call to the `UpdateCommands()` method

of the dispatcher, which starts the change notification/command update process.

Note that there are two broadcaster-observer systems at work in this example. The first is the automatic registration of the `<updater>` tag as a command updater with the dispatcher. The second is the application-specific `observes` attribute put onto the two widget tags. This second system is just a design choice; it is equally possible to update the widget tags directly from the `update()` method, perhaps using ids or, in fact, any other approach.

9.6 COMMANDS AND XPCOM COMPONENTS

Although functors and controllers can be created entirely in JavaScript, XPCOM equivalents also exist. The simple combination of functor, controller, and dispatcher can also be enhanced in a number of ways. These enhancements allow command implementations to be added or removed from controllers dynamically. They also allow extra information to be associated with a given command. None of these enhancements is required for ordinary use of the command system.

If these enhancements are considered, then Mozilla has enough command-related XPCOM objects and interfaces to make your head swim. A naming policy that concatenates and reuses descriptive keywords does not help; instead, it tends to make the different interfaces blend into each other until they all look very similar.

The only way to proceed is to try to characterize each interface until it stands out a little. That is the strategy adopted here. Table 9.1 is based on Mozilla 1.4. Beware that a few of these interfaces are not "frozen" for that version; they may have minor changes in later versions.

Table 9.1 XPCOM components that contribute to Mozilla command infrastructure

Interface name	Useful in scripts?	Existing XPCOM implementations	Purpose
nsICommand		Although it seems a logical name, no such interface exists	
nsICommandHandler		@mozilla.org/embedding/browser/nsCommand-Handler;1	Lowest-level command implementation—not intended for JavaScript use.
nsICommandHandlerInit		@mozilla.org/embedding/browser/nsCommand-Handler;1	Used to initialize nsICommandHandler objects.
nsIController	✓	@mozilla.org/embedcomp/base-command-controller;1	A basic controller, with command table support.

Table 9.1 XPCOM components that contribute to Mozilla command infrastructure (Continued)

Interface name	Useful in scripts?	Existing XPCOM implementations	Purpose
nsIControllerContext	✓	@mozilla.org/embedcomp/base-command-controller;1	Used to initialize a controller.
nsIController-CommandTable	✓	@mozilla.org/embedcomp/controller-command-table;1	A mutable collection of commands.
nsICommandController		@mozilla.org/embedcomp/base-command-controller;1	Poorly named interface that adds parameterized command calls (commands with arguments) to a basic controller.
nsIController-CommandGroup		@mozilla.org/embedcomp/controller-command-group;1	A mutable collection of commands used to give a set of commands group identity.
nsIControllerCommand	✓	None	A basic functor.
nsICommandManager		@mozilla.org/embedcomp/command-manager;1	Exposes intermediate features of a functor object.
nsICommandParams		@mozilla.org/embedcomp/command-manager;1	A data structure used as a parameter list for parameterized command execution.
nsIControllers	✓	None	The controllers collection that is a property of DOM elements.
nsIDOMXULCommand Dispatcher	✓	The dispatcher is not an XPCOM component.	The XUL command dispatcher.
nsPICommandUpdater		Internal interface of no use to applications	
nsISelectionController		Nothing to do with command delivery at all	
nsIEditorController		@mozilla.org/editor/composercontroller;1 @mozilla.org/editor/editor-controller;1	Two controllers used in the Mozilla Composer tool.
nsITransaction nsITransactionList nsITransaction-Manager	✓	@mozilla.org/transaction-manager;1	Three interfaces useful when controllers become complicated. Use these data structures to implement undo/redo, macros, history, auditing, and other advanced controller functionality.

Several interfaces that are used to implement the command-line syntax also exist; they can be used to start platform windows. Those interfaces have nothing to do with the command delivery system.

9.7 EXISTING COMMANDS

So far this discussion has covered XUL, AOM, and XPCOM aspects of command delivery in the Mozilla Platform. Mozilla, however, is a finished application (Classic Mozilla) as well as a platform. Some application commands are provided with the platform. Application programmers sometimes reuse or exploit these platform features. There are several opportunities to do so.

The focus controller is a very simple example of reusing existing technology. This controller is always available in a XUL window. The command dispatcher contains `advanceFocus()` and `rewindFocus()` methods that automate Tab-style navigation that the user normally does. These methods effectively send commands to the focus controller. The controller can also be exploited for command updates. A command updater tag can receive change notifications as a result of focus events from the focus controller. Why the controller sends two `commandupdate` events for each navigation step through the focus ring is not clear at this point.

Another kind of reuse is theft. The file `globalOverlay.js` in the chrome file `toolkit.jar` contains handy function for managing commands and controllers. The command architecture used in the Messenger Addressbook and in the three-controller system used in the Composer are good enough to use as guides. Some of the keyboard-command bindings are modular enough that you can reuse the overlay files that hold them.

Beyond that, reuse of existing Mozilla commands is a little harder to achieve. There is one special case where a great deal of reuse is possible.

Classic Mozilla's Composer, Messenger, and Navigator tools implement about a hundred commands. A moment's thought reveals that even trivial user actions must ultimately be some kind of command because even trivial actions make changes to the user interface. This means that everything from the minor (e.g., "Select Current Word") and the ordinary (e.g., "Delete Message") to the substantial (e.g., "Load URL") must have an implementation inside the Web application suite. Many of these commands are implemented efficiently in C++, and many are available directly in JavaScript in the chrome.

If your application is similar to Classic Mozilla's in purpose (perhaps another browser variation), then these commands may be reused. If your goal is to customize the existing Mozilla application deeply, then familiarity with these commands is essential. If your application is an add-on to a browser, it is also important to avoid clashing with the names of existing commands. Unfortunately, these names are not centralized. This is partly because different programmer teams worked on different parts of the Classic suite. Classic Mozilla

does attempt to use the same names for the commands that have the same purpose in several applications.

A readable example of Classic Mozilla commands can be seen in `ComposerCommands.js` in the `comm.jar` chrome file.

It is easy to get excited about the availability of all these commands, but reuse of them is somewhat limited. In the end, the success of a new application depends on its unique value, not on how well it mimics something that has gone before. If your application works the same as an existing application, why did you bother building it? Most new applications should contain at least some new commands. If you manage a little reuse along the way, more power to you.

This last kind of reuse is required in a minor way with the `<textbox>` XUL tag. It is possible to do simple editing operations inside this widget. Such operations are implemented by Mozilla commands that contribute toward Mozilla's finished HTML browser. If all of Mozilla's commands are stripped away from your application, then at least this much must be re-implemented if form elements are to be supported properly.

9.8 Style Options

There are no styles that apply to Mozilla's command system.

9.9 Hands On: Command Design

To date, the NoteTaker example has relied on a simple `action()` function to implement all the tasks that need scripting. NoteTaker is a small project and doesn't require heavy engineering, but for the sake of experience we use this session to wrap those actions up in commands. In order to do this, we need to

1. Design the commands.
2. Implement the commands.
3. Install the command implementations.
4. Invoke commands from wherever they are needed.

The first step is design. The NoteTaker tool has fixed functionality, so there is a fixed and known list of commands. Furthermore, none of the NoteTaker commands can be disabled. Together, these two statements mean that the controllers that handle the commands won't ever change or change state, and so there will be no `commandupdate` events. We therefore have the simple case where the required command system is entirely static after it is set up.

The command names we choose to implement are an issue. By using the command system, we're sharing the command name namespace with everyone

else's code. In the Edit dialog box, this is a lesser issue because there is no other application code present. In the main browser window, however, the NoteTaker toolbar and Tools menu item must share the command system with the rest of the browser application. The command names we choose shouldn't clash with that application.

To prevent clashes, there are two solutions. One solution is to choose carefully command names that have the same spelling and meaning as existing commands. When a "save" command is issued, all interested parties, including NoteTaker, might receive this command. This solution is a very complex integration task. A second solution, chosen here, is to go it alone and make sure that the command names implemented don't clash with any other names. To that end, we'll prefix all our commands with "`notetaker-`". That is a little verbose, but recall that NoteTaker is a unique package name.

In the last chapter, we had four possible arguments to the `action()` function:

```
edit keywords cancel save
```

These tasks could be designed better. `save`, for example, both saves the note and closes the dialog box; those two steps could be separate commands. `edit` and `keywords` are really navigation instructions and don't match the intended purpose of the Edit button on the toolbar, which is to open the Edit dialog box. A better list of commands is

1. Open the dialog box. Used by the Tools menu item and the Edit button on the toolbar.
2. Close the dialog box. Used by the Cancel button and the Save button on the dialog box.
3. Load the current note. Used by the onload handler in the Edit dialog box.
4. Save the current note. Used by the Save button on the toolbar and on the Edit dialog box.
5. Delete the current note. Used by the Delete button on the toolbar.
6. Navigate to the Edit tab. Used by a keyboard shortcut in the Edit dialog box.
7. Navigate to the Keywords tab. Used by a keyboard shortcut in the Edit dialog box.

No doubt we'll find more commands as more chapters pass. So far, our new commands are

```
notetaker-open-dialog notetaker-close-dialog notetaker-load
     notetaker-save notetaker-delete notetaker-nav-edit notetaker-
     nav-keywords
```

Any syntax convention could be used; here we're just following a convention of dash-separators sometimes used by the platform itself. So we're now finished designing the commands.

To implement the commands, we need to use the <command> tag and command= attribute, or we can create a command controller on some DOM object, or we can do both. We choose to use a command controller to start with because it clarifies the discussion earlier, and because it can be used as a debugging aid.

Although we could control everything from one sophisticated controller instance, it's neater and cleaner to use more than one. We'll create one controller that handles all the NoteTaker commands and then use it several times. Just for fun we'll add a second controller implementation; it will pick up any stray commands we haven't planned for.

Our action() function has served us well, and there's no need to remove it. We'll create controller objects that call the action() method when commands arrive. In design pattern language, the command functors will be aggregated into the controller objects, which will delegate the execution of the actual command to the action() function. In short, Listing 9.8 shows the controller we need.

Listing 9.8 NoteTaker command controller.

```
var notetaker = {
  _cmds : { "notetaker-open-dialog":true,
            "notetaker-close-dialog":true,
            "notetaker-load":true,
            "notetaker-save":true,
            "notetaker-delete":true,
            "notetaker-nav-edit":true,
            "notetaker-nav-keywords":true
          },
  supportsCommand  : function (cmd) { return (cmd in this._cmds); },
  isCommandEnabled : function (cmd) { return true; },
  doCommand        : function (cmd) { return action(cmd);},
  onEvent          : function (cmd) {}
};
```

Clearly this object is just a big wrapper around the action() function, one that makes it fit the command infrastructure of the platform. It is particularly simple because (1) no commands can be disabled, and (2) commands cannot be added dynamically. The second just-for-fun controller we'll create looks like Listing 9.9.

Listing 9.9 NoteTaker command detective.

```
var detective = {
  supportsCommand  : function (cmd) { return true; },
  isCommandEnabled : function (cmd) { return true; },
  doCommand        : function (cmd) {
    throw("NoteTaker detective called on command: " + cmd);
    return true;
  }.
  onEvent          : function (cmd) {}
};
```

This controller is a fake; it accepts and pretends to run all commands, and reports what it sees to the JavaScript console. It can be used to detect all command requests. So both controllers are now implemented, except for the little matter of the `action()` function.

Finally, these controllers must be installed on some DOM object. We choose to install these commands on the DOM object for the `<window>` tag (the window AOM object). We can install on one of two windows: the main browser window (for the toolbar and Tools menu item) and the Edit dialog box. We'll install both the notetaker and the detective controllers in both cases, for a total of four controllers installed. Both windows will have support for all seven commands, but each window will use only a selection of that support. Here is a simple function that installs controllers for one window.

```
function install_controllers()
{
  window.controllers.appendController(notetaker);
  window.controllers.appendController(detective);
}
```

Since the `detective` controller is added second, it will only pick up what the `notetaker` controller misses. These two controllers, the `install_controllers()` function, and a call to that function can all be put into one JavaScript file. If we include it with a `<script src="controllers.js">` tag in all relevant windows, then each window will get separate but identical controller objects.

Invoking the commands is also easy. A simple function holds the required logic:

```
function execute(cmd)
{
  window.focus();
  try {
    var disp = window.document.commandDispatcher;
    var ctrl = disp.getControllerForCommand(cmd);
    ctrl.doCommand(cmd);
  }
catch (e) {
    throw (" Command Unknown: " + cmd + ". Error: " + e);
  }
  window.focus();
}
```

This `execute()` function sets a command on its way; it will throw an error if the dispatcher can't find a suitable controller. The `window.focus()` calls that precede and follow the command dispatch are critical; they ensure that the dispatcher has something to work with, and that the window returns to a sensible state when the dispatcher is finished.

In the browser window, we could instead use an existing function: `goDoCommand()`. That function is provided in `globalOverlay.js` in `toolkit.jar` in the chrome. It is almost the same as `execute()` but allows an

unknown command to be submitted. In our case, we want an error rather than silence if an unknown command appears. The `globalOverlay.js` file contains many short functions used by Mozilla Classic.

This `execute()` function is put everywhere that a command might occur. We replace all calls to the `action()` function with calls to `execute()` in the dialog box. That means the `<key>` tags change like so:

```
<key key="e" oncommand="action('edit')"/>
<key key="e" oncommand="execute('notetaker-nav-edit')"/>
```

Previously we didn't implement anything for the buttons at the base of the dialog box. Now we can at least call the right command. We can call `execute()` from an `onclick` handler on each button, or we can send the button events to a `<command>` tag. Let's do the latter. The Cancel button goes from

```
<button label="Cancel" accesskey="C"/>
```

to

```
<button label="Cancel" accesskey="C" command="dialog.cmd.cancel"/>
```

The `command=` attribute requires a matching `<command>` tag, which is

```
<command id="dialog.cmd.cancel"
         oncommand="execute('notetaker-close-dialog')"/>
```

Let's review the way this works: The user generates events that include a "command" DOM 2 Event by clicking the button. The `<command>` tag is the listener for that event, which is a result of the `command` attribute on the `<button>` tag. When that event occurs, the `<command>` tag takes over, and its `oncommand` handler then fires. That is the end of the event's influence. When the `oncommand` handler fires, it runs `execute()`, which sends a named command to the controller that the dispatcher finds for it. The controller runs the command using the `action()` function and then finishes. After that, the `command` event (and other events like click) begins its bubbling phase. Since there are no bubbling phase handlers installed, no further event processing occurs.

Using `<command>` tags for both the Save and Cancel buttons yields this XUL:

```
<commandset>
  <command id="dialog.cmd.save"
     oncommand="execute('notetaker-save');execute('notetaker-close-
     dialog');"/>
  <command id="dialog.cmd.cancel"
     oncommand="execute('notetaker-close-dialog');"/>
</commandset>
```

The Save button causes two commands to be run. We could collapse these two commands into one, but it's worth making the point that any number of commands can be run at a given opportunity. By using `<command>` and `<key>` tags, we've managed to confine all the event handlers for the dialog to the top

of the XUL document. We can do exactly the same thing for the toolbar buttons, except three <command> tags are required instead of two.

Everything is in place now except for an updated action() function. We can write two versions of this function—one for each window that gains a notetaker controller. That saves the function from having to work out what window it is being run in. We'll just include a different .js file in the toolbar XUL and in the Edit dialog box XUL.

For the Edit dialog box, all we need to do is update the action() logic to use the new command names. For the toolbar, we're not sure what to do yet, so the action() function just reads:

```
function action(cmd) {};
```

That ends our experimentation with commands for now.

9.10 DEBUG CORNER: CATCHING UNEXPECTED COMMANDS

When the command system and the focus system of the Mozilla Platform interact, some very subtle problems can occur. It is very important that the focus is correctly set up when a XUL window first appears. This means a call to window.focus(), or window.content.focus(). Such a call is less necessary in recent versions, but it remains good defensive programming.

If this is not done, and an application that has or acquires focus problems is run, then it is possible for both the Mozilla Platform and the desktop environment to become confused. Just shutting down the platform is not always enough. On Microsoft Windows at least, sometimes it is also necessary to reboot in order to clean out problems that result from poorly organized focus.

In the end, commands are just strings, and a string can contain anything. When working with commands, it is important to remember that those strings form a simple namespace that should be managed properly. Do not be tempted to send randomly named commands from random places in the code, just because they're easy to dispatch.

There are numerous examples of code in the Mozilla Platform and Mozilla applications where lazy checking of data values is done. A very common example is XML attributes that can be set to boolean values. The platform typically checks for "true", and assumes that anything else must be false. This is a quick way to do development, but it makes debugging harder because many invalid values are accepted or ignored by the systems that read them. This is especially true for commands, which include a layer of complexity (the dispatcher) between the command invocation and the command implementation.

When developing an application, command strings should always be strictly evaluated, and there should be no default or unmatched case. Such a case is a bottomless pit for accidental typos and other small mistakes, which would otherwise yield a useful error. When commands are scheduled from

timed scripts and other asynchronous tasks, it is even more important that they are properly recognized. You should always know all the commands that are being dispatched to your code.

This strict evaluation is not implemented in the Mozilla Classic because of that application's desire to be open to enhancements. For applications that are point solutions, there is no reason to leave this strict evaluation out. Only in rare cases (like extensible text adventures such as MUDs [Multi-User Dungeons—text games], IRC, and online games) should the command set be left open.

The "detective" controller, illustrated in "Hands On" in this chapter, is one way of ensuring that nothing slips past you. This controller should never receive a command; if it does, something is wrong, and it needs to be fixed.

Note that unexpected commands and unexpected events are different issues. It is not necessary to block or catch unexpected events. Such events can be produced by the global observer service discussed in Chapter 6, Events.

9.11 SUMMARY

Mozilla provides a command specification and delivery system. This system encourages functional analysis of applications. Such functional analysis allows the application programmer to separate application tasks from the visual appearance of the application. That separation encourages a clean design. It protects implemented commands from most changes in the fragile and fashion-conscious GUI of the application. The command system can automatically sort out where a command is implemented, perform that command, and pass on advice about command changes.

The command system is an example of Mozilla-specific functionality that is entirely nonvisual. It is also separate to the W3C standards–based DOM Event system. The mechanics of the command system support the visual appearance of an application, but it contributes nothing to that appearance directly.

The command system is also flexible. For custom applications it provides a small framework on which application-specific commands can be hung. For minor enhancements to existing Mozilla applications, the command system provides a large collection of ready-to-use commands.

Like the focus ring of the last chapter, each instance of the command system is limited to a single platform window. The next chapter explores in general terms the use of more than one window at a time.