ActionScript 3.0 Design Patterns

William B. Sanders and
Chandima Cumaranatunge

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

ActionScript 3.0 Design Patterns
by William B. Sanders and Chandima Cumaranatunge

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Steve Weiss
Developmental Editor: Robyn G. Thomas
Production Editor: Philip Dangler
Copyeditor: Sohaila Abdulali
Indexer: John Bickelhaupt
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:

July 20007: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-52846-9
ISBN-13: 978-0-59652846-1
[M]

Adobe Developer Library, a copublishing partnership between O'Reilly Media Inc., and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. These comprehensive resources offer learning solutions to help developers create cutting-edge interactive web applications that can reach virtually anyone on any platform.

With top-quality books and innovative online resources covering the latest tools for rich-Internet application development, the *Adobe Developer Library* delivers expert training straight from the source. Topics include ActionScript, Adobe Flex®, Adobe Flash®, and Adobe Acrobat®.

Get the latest news about books, online resources, and more at *http://adobedeveloperlibrary.com*.

Table of Contents

Part III. Structural Patterns

CHAPTER 12

Model-View-Controller Pattern

According to the standard model billions of years ago some little quantum fluctuation, perhaps a slightly lower density of matter, maybe right where we're sitting right now, caused our galaxy to start collapsing around here.

—Seth Lloyd

We view things not only from different sides, but with different eyes; we have no wish to find them alike.

—Blaise Pascal

The primary symptom of a controller is denial, that is I can't see its symptoms in myself.

—Keith Miller

What Is the Model-View-Controller (MVC) Pattern?

The Model-View-Controller (MVC) is a compound pattern, or multiple patterns working together to create complex applications. The MVC pattern is most commonly used to create interfaces for software applications, and, as the name implies, consists of three elements.

Model
: Contains the application data and logic to manage the state of the application

View
: Presents the user interface and the state of the application onscreen

Controller
: Handles user input to change the state of the application

The power of the MVC pattern can be directly attributed to the separation of the three elements without overlap in each of their responsibilities. Let's look at each element's responsibilities.

Model

The model is responsible for managing the state of the application. The application logic in the model performs two important tasks: it responds to requests for information about the state of the application, and takes action on requests to change the state.

View

A view is the external face of the application. Users interact with the application through the view. An application can contain multiple views that can be both inputs and outputs. For example, in the case of a portable digital music player such as an iPod, the screen is a view. In addition, the buttons that control song playback are views as well. The screen shows the name of the current song, song duration, album art, and so on, that communicate the current state of the device. Views don't necessarily have to be visual. In the case of a digital music player, the sound that comes through the headphones represents a view as well. For example, clicking a button may provide some auditory feedback in the form of the click sound. Changing the volume is reflected in the audio output as well. The auditory feedback corresponds to the state of the application.

Controller

Although the term controller implies an interface that controls an application, in an MVC pattern, the controller does not contain any user interface elements. As pointed out previously, user interface elements that provide input belong to the *view* component. The controller determines how views respond to user input.

For example, our digital music player has *volume up* and *volume down* buttons in a *view*. The sound volume of the device is a state variable. The model will hold this variable with the necessary application logic to change it. If the sound volume range is 0 to 10, the controller determines how much the volume should go up or down with a single click on the volume up and down buttons. The controller can tell the model to raise the volume by 0.5 or 1.0, or any value, programmatically. In this sense, controllers are specific implementations that determine how the application responds to user input.

Although each element in the MVC triad has separate and unique responsibilities, they don't function in isolation. In fact, in order to be an MVC pattern, each element needs to communicate with one or more elements. That is what we'll look at next.

Communication Between the MVC Elements

Each element in the MVC pattern communicates with each other in very specific ways. Communication is necessitated by a sequence of events that are generally

triggered by a user interacting with the application. The sequence of events is represented as follows:

1. User interacts with a user interface element (e.g. clicks on a button in a view).
2. The view sends the click event to the controller to decide how to handle it.
3. The controller changes the model based on how it decides to handle the button click.
4. The model informs the view that the state of the model has now changed.
5. The view reads state information from the model and updates itself.

Figure 12-1 shows the graphical representation of the channels of communication between MVC elements. The arrows' directions show the direction of communication.

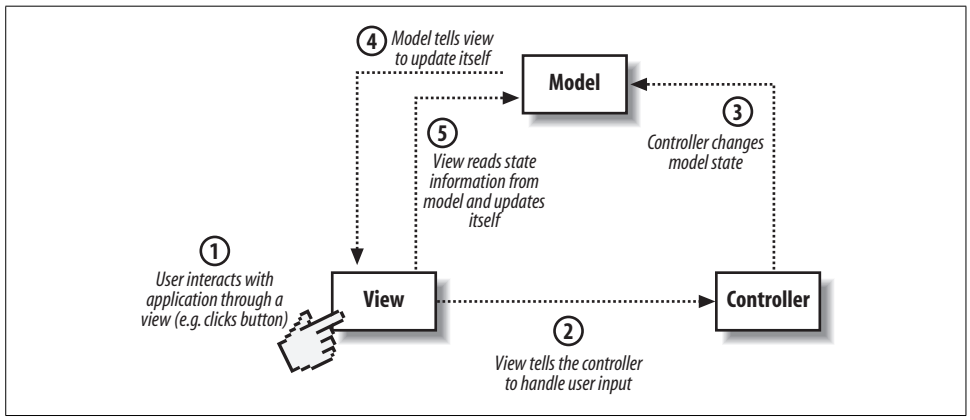

Figure 12-1. Direction of communication between MVC elements

This is a very simple model of how MVC elements communicate. In some cases the controller can directly tell the view to make changes as well. This is the case only when the changes in the view necessitated by user action don't require a change in the model itself, but simply a change in visuals. For example, think about the process whereby a user selects a song to play in our portable digital music player. The user selects songs from a list using buttons that scroll the list. The view would tell the controller that the scroll up or scroll down button has been clicked, but the controller won't inform the model of this. It'll directly tell the view to scroll the list of songs in the appropriate direction. This user action doesn't represent a change in the model. However, when the user actually selects a song from the list to play, the controller will change the model to reflect this change in the currently selected song.

Furthermore, changes in the model are not always initiated by user interaction. The model can update itself based on certain events. For example, think of a stock ticker application. The model would hold the current prices of certain stocks. However, stock prices change, and the model could set a timer to periodically update the stock

prices from a web service. Then, whenever the model updated its stock prices, it would inform the view that its state has changed.

Another feature of the MVC is that each model can have more than one view associated with it. For example, in our portable music player, the volume setting of the device can be viewed on the display screen using a level indicator. In addition, sound level is represented in the level of the sound output from the headphones as well. Both the display and auditory feedback from the headphones represent views of the device state.

Take a look at Figure 12-1 and make note of the arrows' directions. This shows who initiates communication between elements. In order for an MVC element to communicate with another element, it needs to know about and hold a reference to that element.

Think of the model, view, and controller as three separate classes. Let's look at which classes need to have references to which other classes.

Model
: Needs to have a reference to *views*

View
: Needs references to both the *model* and *controller*

Controller
: Needs a reference to the *model*

We started off by saying that the MVC is a compound pattern that consists of several patterns. You may be wondering where the embedded patterns are. It is precisely at this point that they can be introduced. The primary advantage of using the MVC pattern is the loose coupling it allows us to implement between the three elements. It lets us bind multiple views to the same model, and swap out models and controllers without breaking other elements. But some element in the MVC triad needs to hold references to other elements, and there's a whole lot of talking going on between them. How can we call this loose coupling? This is where the observer, strategy, and composite patterns help us out.

Embedded Patterns in the MVC

We pointed out that a model can be associated with several views. In the MVC, the model needs to inform all associated views that a change has taken place. It also needs to do this without knowing specific details about the views, or how many views need to be changed. This is a recurring problem best solved by implementing an *observer pattern* (see Chapter 8).

Each model can have multiple views associated with it. Views can also be complex, with multiple windows or panels that contain other user interface elements. For example, user interface elements such as buttons, text fields, lists, sliders, etc. can be

grouped together in a tabbed panel that in turn will be part of a window with other tabbed panels. Each button or group of buttons can be a view. So can a collection of text fields. It would be very useful to treat a panel or window that contains collections of simple views the same way as we would treat any other view. This is where the *composite pattern* will save us a lot of effort (see Chapter 6). Why would a composite pattern implementation be useful in this context? If views can be nested, as they would be if they were implemented in a composite pattern, the update process would be simpler. Update events would automatically cascade down to child views. Creating complex views would be easier without having to worry about sending update events to each nested view.

Views confine themselves solely to the external representation of the model state. They delegate user interface events to a controller. Therefore, the controller is essentially an algorithm of how to handle user input in a particular view. This delegation encapsulates the implementation of how a particular user interface element behaves in terms of modifying the model. We can easily substitute a different controller for the same view to get different behavior. This is a perfect context to implement a *strategy pattern*.

We will look at how each of these patterns is implemented in an MVC by developing a minimalist example.

Minimalist Example of an MVC Pattern

This simple example keeps track of the last key pressed. When a new key is pressed, it changes the model and informs the view to update itself. The view uses the Flash output panel to print the *character code* of the key that's pressed. The character code is the numeric value of that key in the current character set. This example is meant to clarify how the observer, strategy, and composite patterns are integrated within the MVC.

Model as a Concrete Subject in an Observer Pattern

The relationship between the model and view is that of subject and observer (see Chapter 8). The model has to implement the subject interface that's part of the observer pattern. Fortuitously, ActionScript 3.0 has built in classes that do this already, using the ActionScript event model to notify observers of changes.

The EventDispatcher class in ActionScript 3.0

The `EventDispatcher` class implements the `IEventDispatcher` interface. Among other methods, the `IEventDispatcher` interface defines the following methods required of the subject in an observer pattern. (See AS3 documentation for a detailed explanation of all method parameters.)

```
addEventListener(type:String,
    listener:Function,
    useCapture:Boolean = false,
    priority:int = 0,
    useWeakReference:Boolean = false):void

removeEventListener(type:String,
    listener:Function,
    useCapture:Boolean = false):void

dispatchEvent(event:Event):Boolean
```

For the model to serve as a concrete subject in an observer pattern, it needs to implement the `IEventDispatcher` interface. However, the easiest way for a user-defined class to gain event dispatching capabilities is to extend the `EventDispatcher` class.

Observers register listener methods to receive event notifications from `EventDispatcher` objects through the `addEventListener( )` method.

The model

Our model holds the character code of the last key pressed in a property. It needs to implement *setter* and *getter* methods to enable the view and controller to access and modify it. Let's first define an interface for our model (Example 12-1).

Example 12-1. IModel.as

```
package
{
    import flash.events.*;

    public interface IModel extends IEventDispatcher
    {
        function setKey(key:uint):void
        function getKey( ):uint
    }
}
```

The `IModel` interface shown in Example 12-1 extends the `IEventDispatcher` interface and defines two methods to get and set the character code of the last key pressed. Because the `IModel` interface extends `IEventDispatcher`, any class implementing it has to implement all the methods defined in both interfaces. The `Model` class shown in Example 12-2 implements the `IModel` interface.

Example 12-2. Model.as

```
package
{
    import flash.events.*;

    public class Model extends EventDispatcher implements IModel
    {
```

Example 12-2. Model.as (continued)

```
        private var lastKeyPressed:uint = 0;

        public function setKey(key:uint):void
        {
            this.lastKeyPressed = key;
            dispatchEvent(new Event(Event.CHANGE)); // dispatch event
        }

        public function getKey( ):uint
        {
            return lastKeyPressed;
        }
    }
}
```

The `Model` class extends the `EventDispatcher` class that already implements the `IEventDispatcher` interface. Note the `dispatchEvent( )` function call within the `setKey( )` method. This sends a `CHANGE` event to all registered observers when the value of `lastKeyPressed` is changed within the `setKey( )` method.

Controller as a Concrete Strategy in a Strategy Pattern

The relationship between the controller and view is that of strategy and context in a strategy pattern. Each controller will be a concrete strategy implementing a required behavior defined in a strategy interface.

The controller

For our minimalist example, the behavior required of the controller is to handle a key press event. `IKeyboardInputHandler` is the strategy interface (Example 12-3), and defines a single method called `keyPressHandler( )`.

Example 12-3. IKeyboardInputHandler.as

```
package
{
    import flash.events.*;

    public interface IKeyboardInputHandler
    {
        function keyPressHandler(event:KeyboardEvent):void
    }
}
```

The concrete controller is the `Controller` class (Example 12-4) that implements the `IKeyboardInputHandler` interface.

Example 12-4. Controller.as

```
package
{
    import flash.events.*;

    public class Controller implements IKeyboardInputHandler
    {
        private var model:IModel;

        public function Controller(aModel:IModel)
        {
            this.model = aModel;
        }

        public function keyPressHandler(event:KeyboardEvent):void
        {
            model.setKey(event.charCode); // change model
        }
    }
}
```

Note that the controller has a constructor that takes an instance of the model as a parameter. This is necessary as the controller initiates communication with the model as shown in Figure 12-1. Therefore, it needs to hold a reference to the model.

The `keyPressHandler( )` method takes the user interface event (a `KeyboardEvent` in this case) as a parameter and decides how to handle it. In this example, it simply sets the last key pressed in the model to the character code of the key pressed.

View as a Concrete Observer in an Observer Pattern and Context in a Strategy Pattern

The view is arguably the most complex element in the MVC pattern. It plays an integral part in both the observer and strategy pattern implementations that form the basis of its relationship with the model and controller. The `View` class shown in Example 12-5 implements the view for the minimalist example.

Example 12-5. View.as

```
 1 package
 2 {
 3     import flash.events.*;
 4     import flash.display.*;
 5
 6     public class View
 7     {
 8         private var model:IModel;
 9         private var controller:IKeyboardInputHandler;
10
```

Example 12-5. View.as (continued)

```
11          public function View(aModel:IModel, oController:
                            IKeyboardInputHandler,target:Stage)
12          {
13              this.model = aModel;
14              this.controller = oController;
15
16              // register to receive notifications from the model
17              model.addEventListener(Event.CHANGE, this.update);
18
19              // register to receive key press notifications from the stage
20              target.addEventListener(KeyboardEvent.KEY_DOWN,
21
22          }
23
24          private function update(event:Event):void
25          {
26              // get data from model and update view
27              trace(model.getKey( ));
28          }
29
30          private function onKeyPress(event:KeyboardEvent):void
31          {
32              // delegate to the controller (strategy) to handle it
33              controller.keyPressHandler(event);
34          }
35      }
36  }
```

The view needs to hold references to both the model and controller as it initiates communication with them as shown in Figure 12-1. Both the model and controller instances are passed to the view in its constructor. Also, the view in our example needs a reference to the stage to register itself to receive key press events.

In addition to drawing the user interface, the `View` class does a couple of important tasks. It registers with the model to receive update events, and delegates to the controller to handle user input. In our example, the view does not have an external visual representation on the stage, but displays the model state in the output panel. It needs to receive key press events, and registers the method called `onKeyPress( )` to receive `KEY_DOWN` events from the stage (line 20). The second task is to register a listener method called `update( )` to receive a `CHANGE` event from the model (line 16). On notification of a change, the `update( )` method reads the character code for the last key pressed from the model and prints it to the output panel using the trace function.

Building the MVC Triad

We have looked at the individual implementations of the three elements that make up the MVC pattern. However, there has to be a client that initializes each element and builds the MVC model. There's no real building involved – all that needs to be

done is to instantiate the model, view, and controller classes. Example 12-6 shows the Flash document class that instantiates the MVC elements.

Example 12-6. Main.as (document class for minimalist example)

```
package
{
    import flash.display.*;
    import flash.events.*;

    /**
    *    Main Class
    *    @ purpose:        Document class for movie
    */
    public class Main extends Sprite
    {
        public function Main( )
        {
            var model:IModel = new Model( );
            var controller:IKeyboardInputHandler = new Controller(model);
            var view:View = new View(model, controller, this.stage);
        }
    }
}
```

After the model, controller, and view are instantiated, they'll communicate with each other and work. Clicking a key on the keyboard will result in the key code for that key being printed in the output panel.

You have to disable keyboard shortcuts to test for key presses. Otherwise the Flash user interface intercepts certain key press events that match keyboard shortcuts. To disable keyboard shortcuts select Disable Keyboard Shortcuts from the Control menu when your Flash movie is playing.

Note that the model instance is passed to the controller. Similarly, the model and controller instances are passed to the view as well. We can easily substitute different models and controllers on the condition that they implement the `IModel` and `IKeyboardInputHandler` interfaces. Additional view elements can also be added without disruption due to the subject-observer relationship between the model and view. The model doesn't know about views as it's the responsibility of the view to register itself to receive update notifications from the model. This is the beauty of the MVC pattern; the model, view, and controller are separate, loosely coupled elements that allow for flexibility in their use.

Nested Views as Leaves and Nodes of a Composite Pattern

You may remember that the view is arguably the most complex element in the MVC triad because it participates in both the observer and strategy pattern implementa-

tions in the MVC. Our view elements are going to get more complex as they can implement a third pattern, the composite (see Chapter 6 for examples of the composite pattern). Implementing views as elements of a composite pattern only makes sense for complex nested user interfaces that contain multiple views. Nested views bring several advantages to updating the user interface, as updates can cascade down the composite view tree structure. Also, composite views can create and remove child views based on application state and user mode. A good example of a complex user interface is the *Properties inspector* panel in the Flash authoring environment. The *Properties inspector* is context sensitive, and adds or removes user interface elements based on the object selected on the stage.

Component and composite views

The first step is to create the component and composite classes for the view. These classes should behave as abstract classes and should be subclassed and not instantiated, as shown in Example 12-7.

Example 12-7. ComponentView.as

```
package
{
    import flash.errors.IllegalOperationError;
    import flash.events.Event;
    import flash.display.Sprite;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class ComponentView extends Sprite {
    {
        protected var model:Object;
        protected var controller:Object;

        public function ComponentView(aModel:Object, aController:Object = null)

        {
            this.model = aModel;
            this.controller = aController;
        }

        public function add(c:ComponentView):void
        {
            throw new IllegalOperationError("add operation not supported");
        }

        public function remove(c:ComponentView):void
        {
            throw new IllegalOperationError("remove operation not supported");

        }

        public function getChild(n:int):ComponentView
        {
            throw new IllegalOperationError("getChild operation not supported");
```

Example 12-7. ComponentView.as (continued)

```
            return null;
        }

        // ABSTRACT Method (must be overridden in a subclass)
        public function update(event:Event = null):void {}
    }
}
```

The `ComponentView` class shown in Example 12-7 defines the abstract interface for component views. This is similar to the classic component class introduced in Chapter 6, with a few key differences. The `ComponentView` class keeps references to the model and view, and includes a constructor. Not all views handle user input, and a component view can be constructed by just passing a model instance. Therefore, the `aController` parameter has a default value of `null` in the constructor. Also note that the `ComponentView` class extends the `Sprite` class. This makes sense as most views draw a user interface on the stage. We can use the properties and methods implemented in the built-in `Sprite` class to draw and add objects to the display list.

The `update( )` method should behave as an abstract method. Leaf views that subclass `ComponentView` must override and implement the `update( )` method to update their user interface. This method is the listener function that intercepts update notifications from the model. For this reason, a parameter of type `Event` is passed to it. This parameter also has a default value of `null`, allowing `update( )` to be called without passing an event parameter. This is useful when initially drawing the user interface in its default state, and our subsequent examples illustrate it.

The `CompositeView` class extends `ComponentView` and overrides the methods that deal with handing child views. In Example 12-8, we will implement only the `add( )` method for simplicity.

Example 12-8. CompositeView.as

```
package {

    import flash.events.Event;

    // ABSTRACT Class (should be subclassed and not instantiated)
    public class CompositeView extends ComponentView
    {
        private var aChildren:Array;

        public function CompositeView(aModel:Object,aController:Object = null)
        {
            super(aModel, aController);
            this.aChildren = new Array( );
        }

        override public function add(c:ComponentView):void
        {
```

Example 12-8. CompositeView.as (continued)

```
            aChildren.push(c);
        }

        override public function update(event:Event = null):void
        {
            for each (var c:ComponentView in aChildren)
            {
                c.update(event);
            }
        }
    }
}
```

Note the overridden `update()` function in the `CompositeView` class shown in Example 12-8. It calls the update method in all its children. Therefore, calling the `update()` function in the root node of the composite view structure will cascade down and traverse the component tree updating all views. Let's subclass `CompositeView` and `ComponentView` classes and create a nested view structure to see how this works.

Creating nested views

To illustrate nested views, we will create a composite view node and two child component views as shown in Figure 12-2.

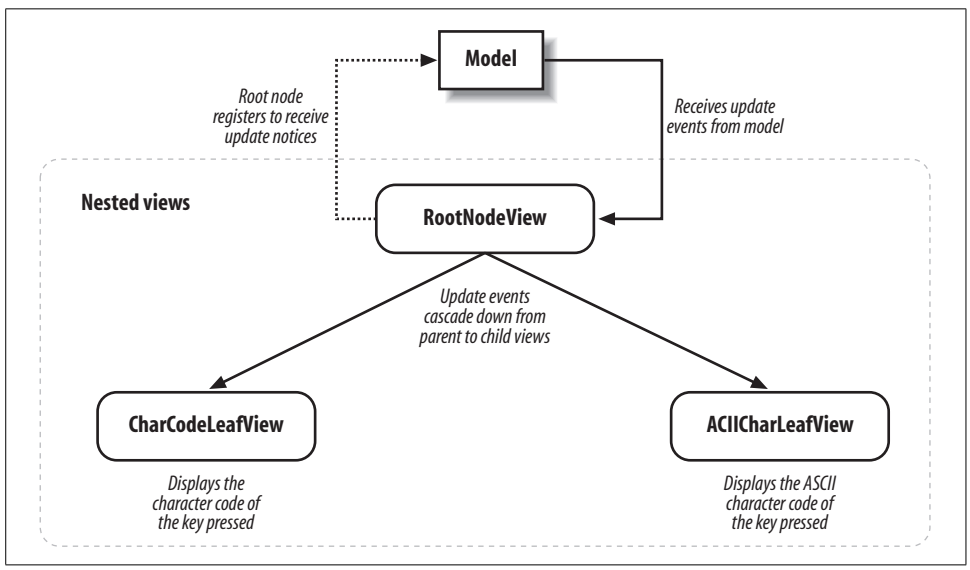

Figure 12-2. Nested view structure in minimalist example

We will first create a composite view called `RootNodeView`. This composite view will receive keyboard input events from the stage. This view will also register to receive

update notices from the model. The RootNodeView composite view will have two child component views called CharCodeLeafView and AsciiCharLeafView. The CharCodeLeafView will trace the *character code* for the last key pressed. Similarly, the AsciiCharLeafView will trace the *ASCII character* corresponding to the character code. Let's create the composite view first, as shown in Example 12-9.

Example 12-9. RootNodeView.as

```
package
{
    import flash.events.*;
    import flash.display.*;

    public class RootNodeView extends CompositeView
    {
        public function RootNodeView (aModel:IModel,

        {
            super(aModel, aController);

            // register to receive key press notifications form the stage
            target.addEventListener(KeyboardEvent.KEY_DOWN,

        }

        private function onKeyPress(event:KeyboardEvent):void
        {
            // delegate to the controller (strategy) to handle it
            (controller as IKeyboardInputHandler).keyPressHandler(event);
        }
    }
}
```

The RootNodeView class shown in Example 12-9 subclasses the CompositeView class (Example 12-8). It does not draw a user interface; it simply listens for key press events and delegates to the controller to handle them. Note the super statement in the constructor. This is required to call the constructor in the superclass. We can now create the two component views (Examples 12-10 and 12-11).

Example 12-10. CharCodeLeafView.as

```
package
{
    import flash.events.Event;

    public class CharCodeLeafView extends ComponentView
    {
        public function CharCodeLeafView(aModel:IModel, aController:Object = null)
        {
            super(aModel, aController);
        }
```

Example 12-10. CharCodeLeafView.as (continued)

```
        override public function update(event:Event = null):void
        {
            // get data from model and update view
            trace(model.getKey( ));
        }
    }
}
```

Example 12-11. AsciiCharLeafView.as

```
package
{
    import flash.events.Event;

    public class AsciiCharLeafView extends ComponentView
    {
        public function AsciiCharLeafView(aModel:IModel,aController:Object = null)
        {
            super(aModel, aController);
        }

        override public function update(event:Event = null):void
        {
            // get data from model and update view
            trace(String.fromCharCode(model.getKey( )));
        }
    }
}
```

Both component view classes `CharCodeLeafView` (Example 12-10) and `AsciiCharLeafView` (Example 12-11) subclass the `ComponentView` class (Example 12-8). Note that they don't receive any input from the user interface. These two views can therefore be instantiated without passing a controller object to the constructor. The controller will default to `null` in this case.

Now all that's left to do is build the MVC triad with the nested view. We'll allow the client to build the nested view and register the root node with the model to receive update events.

Building the Nested View Structure

Building nested view structures is identical to developing composite structures using the composite pattern. We have to visualize the view as a tree (an upside-down tree) and use the `add( )` method in the composite view to add child views.

```
var model:IModel = new Model( );
var controller:IKeyboardInputHandler = new Controller(model);
```

```
// composite view
var rootView:CompositeView = new RootNodeView(model,controller, this.stage);

// add child leaf views
rootView.add(new CharCodeLeafView(model));
rootView.add(new AsciiCharLeafView(model));

// register view to receive notifications from the model
model.addEventListener(Event.CHANGE, rootView.update);
```

Note that only the root view registers with the model to receive update events. Because the root view is a composite view, the event cascades down to each of its child nodes. Now whenever a key is pressed, one child node will trace the character code and the other will trace the ASCII character of the corresponding key. Even though the nested view structure in our minimalist example was simple, this structure can work well for nested views with many components.

In our minimalist example, the client built the nested view structure. However, the build statements for the nested view could have been embedded in the `RootNodeView` class, further encapsulating implementation. Now the root view can dynamically add and remove child views based on application state and user mode. This allows the user interface element of an application to gain some very powerful capabilities.

Key Features of the MVC Pattern

The primary usefulness of the MVC pattern is the flexibility it affords when creating applications that have user interfaces. The pattern separates the model, view, and controller elements and leverages the observer, strategy, and composite patterns to decouple them.

- MVC consists of three elements called the model, view, and controller that separate the responsibilities of an application with a graphical user interface.
- The relationship between models and views is that of a concrete subject and a concrete observer in an observer pattern.
- The relationship between views and controllers is that of a context and concrete strategy in a strategy pattern.
- Multiple views can register with the model.
- Views can be nested using the composite pattern to create complex user interfaces that streamline the update process.

Key OOP Concepts in the MVC Pattern

The key concept in the MVC pattern is *loose coupling*. Loose coupling reduces the dependencies between the separate elements in the MVC pattern. For example, the subject-observer relationship between the model and view enables the model to function independently of how its state is displayed in the user interface. It's the responsibility of the view to register for updates and update the user interface elements. The model can function independently, blissfully unaware of the number and type of views. Similarly, nested views can be added without disruption. Update events will trickle down to all nested views as long as the root node is registered with the model to receive updates. This makes the MVC pattern highly extensible.

In addition, each element in the MVC adheres to the single responsibility principle. Each element has a well-defined role. The model manages state, the view represents state, and the controller handles user input. This allows each element to be swapped out without affecting other elements. If we need a different behavior for a particular user interface element in a view, we simply substitute a different controller for it. This makes the MVC pattern very customizable, allowing reuse.

Example: Weather Maps

The National Oceanic and Atmospheric Administration (NOAA), a division of the U.S. Department of Commerce, runs a *Geostationary Satellite Server* on the Web (*http://www.goes.noaa.gov*). The site publishes satellite images of the United States, including Puerto Rico, Alaska, and Hawaii. We will use these images (they're in the public domain and free to use) to develop a simple weather map application leveraging the MVC pattern. For this example we will use the built-in user interface components provided in Flash CS3 to develop view elements.

For the first iteration of our application shown in Figure 12-3, we'll allow the user to choose the map region (East Coast, West Coast, Puerto Rico, Alaska, and Hawaii) using a combo box (a drop-down list that displays the currently chosen item). The application will then load the latest visible satellite image of the corresponding region.

This example illustrates the use of built-in components in Flash CS3 to implement the user interface elements in each view. It also shows the usefulness of nested views for screen layout and automatic view updates. Let's create the model element of our example application.

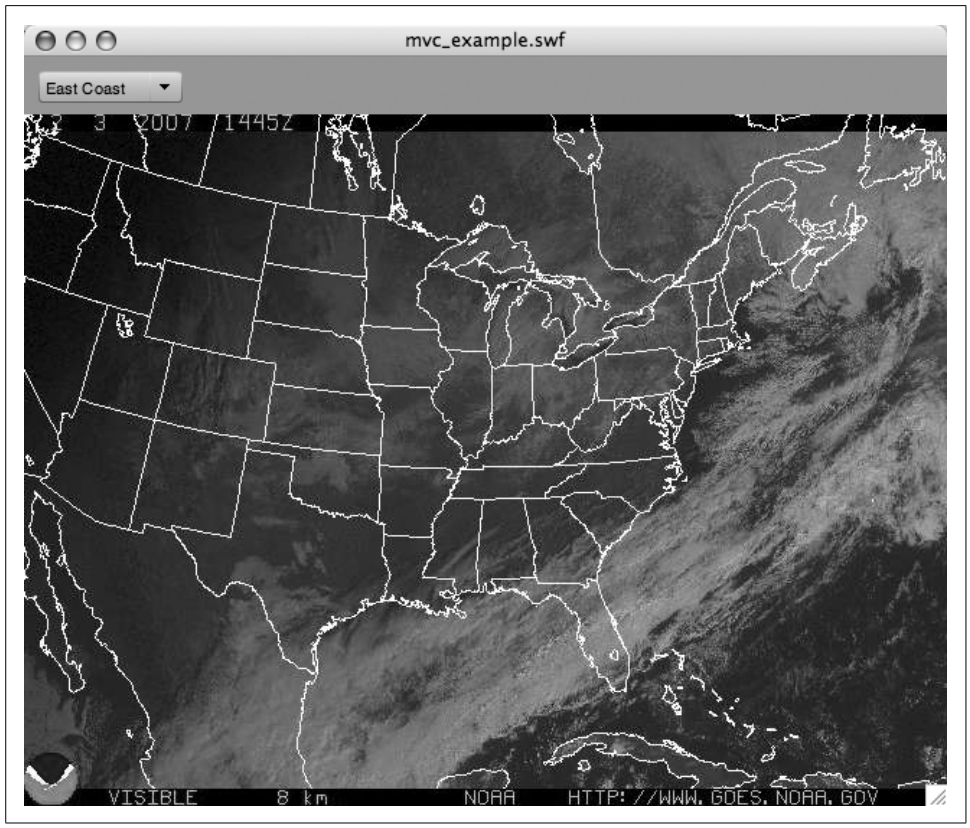

Figure 12-3. Weather map example application showing a visible map of the East Coast

The Model

The model element of the MVC pattern contains the application data and state including the logic to manage data and state. The application state is the region the user has chosen to display. The region can be one of five, corresponding to the regional maps available from the web site. The data is the URLs of the satellite images corresponding to each region.

The application logic should allow users to manage the state of the application. Users should be able to set the current region. The view element that draws the combo box needs to get the list of available regions and the currently selected region. In addition, the view that displays the satellite image needs to access the URL of the currently selected region. Let's develop a model interface (Example 12-12) based on these requirements.

Example 12-12. IModel.as (Model interface for the weather map example)

```
package
{
    import flash.events.*;

    public interface IModel extends IEventDispatcher
    {
        function getRegionList( ):Array
        function getRegion( ):uint
        function setRegion(index:uint):void
        function getMapURL( ):String
    }
}
```

The next step is to implement the `IModel` interface and develop the model (Example 12-13).

Example 12-13. Model.as (Model for the weather map example)

```
package {

    import flash.events.*;

    public class Model extends EventDispatcher implements IModel
    {

        protected var aRegions:Array;
        protected var chosenRegion:uint;

        protected var aImageURLs:Array;

        public function Model( )
        {
            this.aRegions = new Array(
                   "East Coast",
                   "West Coast",
                   "Puerto Rico",
                   "Alaska",
                   "Hawaii");
            this.aImageURLs = new Array(
                   "http://www.goes.noaa.gov/GIFS/ECVS.JPG",
                   "http://www.goes.noaa.gov/GIFS/WCVS.JPG",
                   "http://www.goes.noaa.gov/GIFS/PRVS.JPG",
                   "http://www.goes.noaa.gov/GIFS/ALVS.JPG",
                   "http://www.goes.noaa.gov/GIFS/HAVS.JPG");
            this.chosenRegion = 0;
        }

        public function getRegionList( ):Array
        {
            return aRegions;
        }
```

Example 12-13. Model.as (Model for the weather map example) (continued)

```
        public function getRegion( ):uint
        {
            return this.chosenRegion;
        }

        public function setRegion(index:uint):void
        {
            this.chosenRegion = index;
            this.update( );
        }

        public function getMapURL( ):String
        {
            return this.aImageURLs[chosenRegion];
        }

        protected function update( ):void
        {
            dispatchEvent(new Event(Event.CHANGE)); // dispatch event
        }
    }
}
```

Note the `update( )` method in the `Model` class shown in Example 12-13. It dispatches a `CHANGE` event to registered observers. The `update( )` method is called whenever the application state changes.

As a developer, you may be tempted to store the region list in the view for simplicity. However, this greatly reduces flexibility and reuse. Adding another region, for example, would require changes to both the view and the model. It's always a good practice to adhere to the delineated responsibilities of the three MVC elements. Application data should be accessible only through the model.

The Controller

The combo box view is the only interface element in the application that users can control. The interface for the corresponding controller is shown in Example 12-14.

Example 12-14. ICompInputHandler.as

```
package
{
    public interface ICompInputHandler
    {
        function compChangeHandler(index:uint):void
    }
}
```

The implementation is show in Example 12-15.

Example 12-15. Controller.as (Controller for the weather map example)

```
package
{
    public class Controller implements ICompInputHandler
    {

        private var model:Object;

        public function Controller(aModel:IModel)
        {
            this.model = aModel;
        }

        public function compChangeHandler(index:uint):void {
            (model as IModel).setRegion(index); // update model
        }
    }
}
```

The Views

The weather map application consists of two views. The first is a user interface element that contains a combo box to select a region to display. The second view displays the corresponding regional satellite image.

We will use the built-in `ComboBox` and `UILoader` components in Flash CS3 to implement the user interface elements in each view. Components are movie clips with parameters that allow you to modify their appearance and behavior. They allow developers to build Flash applications with consistent behavior and appearance without creating custom user interface elements such as buttons and sliders. For the weather map application, we need to drag the `ComboBox` and `UILoader` components from the *Components panel* to the *Library panel* in our Flash document. The Components and Library panels can be accessed from the Windows menu in Flash CS3. Note that we don't want to place the component on the stage. We'll add the components to the stage at runtime using ActionScript. The `ComboBox` component uses several additional assets as well. Your library panel will look like Figure 12-4 after the `ComboBox` and `UILoader` components are dragged into it.

We can now develop the combo box and map views for the application. The combo box view will be a composite view containing the map view as one of its children.

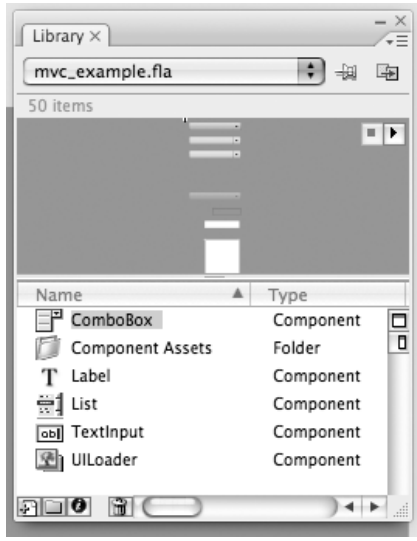

Figure 12-4. Library panel with ComboBox and UILoader components

Combo box view

The `CBView` class (Example 12-16) subclasses `CompositeView` (Example 12-8) and draws the `ComboBox` component.

Example 12-16. CBView.as

```
package
{
    import flash.events.Event;
    import fl.controls.ComboBox;

    public class CBView extends CompositeView
    {

        private var cb:ComboBox;

        public function CBView(aModel:IModel,aController:ICompInputHandler= null)
    {
            super(aModel, aController);

            // get region names from model
            var aRegions:Array = (model as IModel).getRegionList( );

            // draw combo box using region names
            cb = new ComboBox( );
            for (var i:uint = 0; i < aRegions.length; i++)
            {
                cb.addItem( { label: aRegions[i], data:i } );
            }
```

Example 12-16. CBView.as

```
24              update( );
25              addChild(cb);
26
27              // register to recieve changes to combo box
28              cb.addEventListener(Event.CHANGE, this.changeHandler);
29          }
30
31          override public function update(event:Event = null):void
32          {
33              // get data from model and update view
34              cb.selectedIndex = (model as IModel).getRegion( );
35              super.update(event);
36          }
37
38          private function changeHandler(event:Event):void
39          {
40              // delegate to the controller (strategy) to handle
41
42              (controller as ICompInputHandler).compChangeHandler
                   (ComboBox(event.target).selectedItem.data);
43          }
44      }
45  }
```

The `CBView` class (Example 12-16) gets the list of region names from the model (line 16) and adds them to the component. Line 24 calls the `update( )` method without any parameters. The `update( )` method reads the currently selected region from the model and updates the combo box. It then adds the combo box to the display list (line 25) and registers the `changeHandler( )` function to receive change events from the combo box component (line 28). Note that because this is a composite view, the overridden `update( )` function needs to call its superclass (line 35) to ensure that updates trickle down to its children as well.

Map view

The `MapView` class (Example 12-17) subclasses `ComponentView` (Example 12-7) and draws the `UILoader` component.

Example 12-17. MapView.as

```
1  package
2  {
3      import flash.events.Event;
4      import fl.containers.UILoader;
5
6      public class MapView extends ComponentView
7      {
8          private var uiLoader:UILoader;
9
10         public function MapView(aModel:IModel, aController:Object = null)
```

Example 12-17. MapView.as

```
11          {
12              super(aModel, aController);
13
14              uiLoader = new UILoader( );
15              uiLoader.scaleContent = false;
16              update( );
17              addChild(uiLoader);
18          }
19
20          override public function update(event:Event = null):void
21          {
22              // get data from model and update view
23              uiLoader.source = (model as IModel).getMapURL( );
24          }
25      }
26  }
```

The `MapView` class (Example 12-17) loads and displays the satellite image, and doesn't otherwise interact with the user. As in the combo box view, the `update( )` method is called without any parameters from the constructor (line 16) to load the image for the default region. Assigning a URL of an image to the `source` parameter in the `UILoader` component loads the corresponding image. Unlike the combo box, map view is a component view that cannot have any children. Therefore, the overridden `update( )` method does not have to call its superclass method.

Building the MVC Triad

The last task is to instantiate the model and controller, and construct the composite view structure by adding the map view as a child of the combo box view. The following statements should be executed from the document class of the Flash document.

```
var model:IModel = new Model( );
var controller:ICompInputHandler= new Controller(model);

// composite view
var view:CompositeView = new CBView(model, controller);
view.x =  view.y = 10;
addChild(view);

// adding a child view
var map:ComponentView = new MapView(model);
view.add(map);
map.x = 0
map.y = 40;
addChild(map);

// register to view to recieve notifications from the model
model.addEventListener(Event.CHANGE, view.update);
```

Note that only the combo box view that's at the root of the composite view structure has registered with the model to receive update events. These will trickle down from the combo box view to its child map view.

The views extend the `Sprite` class. Therefore, the client has the flexibility to place them on the stage that fits the required application layout.

Setting the Model to Self-Update

Changes in the model are not always initiated by user interaction. You may have noticed a time stamp on the satellite images (in GMT). The time stamp tells us that the satellite images are updated about every 15 minutes. We can easily add a self-update timer to our model to dispatch an event every 15 minutes that tells the map view to reload the updated image. We should add the following statements to the end of the `Model` class constructor in Example 12-13 (make sure to import the `flash.utils.Timer` class first).

```
var updateTimer:Timer = new Timer(1000 * 60 * 15);
updateTimer.addEventListener("timer", timerHandler);
updateTimer.start( );
```

The following listener method should be added to the `Model` class in Example 12-13. The listener method `timerHandler( )` is set to listen for a new `TimerEvent` to be dispatched every 15 minutes. The `timerHandler( )` calls the `update( )` method to dispatch an update event to registered observers.

```
public function timerHandler(event:TimerEvent):void {
    this.update( );
}
```

Extended Example: Infrared Weather Maps

To illustrate how the MVC pattern allows for flexible expansion and reuse of its model, view, and controller elements, we will extend our weather maps example. You may have noticed that there are three types of satellite image maps on the *Geostationary Satellite Server* web site (*http://www.goes.noaa.gov*). What if we want to give the user the option of choosing whether to view a visible or infrared image? What changes would be required to extend our weather maps application to view infrared satellite images?

To begin with, we'll need to add another view such as a radio button group to choose the type of satellite image (visible or infrared as shown in Figure 12-5).

We will need to create a new controller to handle the user input to the new view element. Our model will also need to hold more data, as we need to integrate five additional image URLs, one for each region as an infrared image. The application logic in the model will also need to be updated to handle the new data. Can we add all these new features without changing existing code? Can we leverage the flexibility and reuse of the MVC pattern without breaking anything?

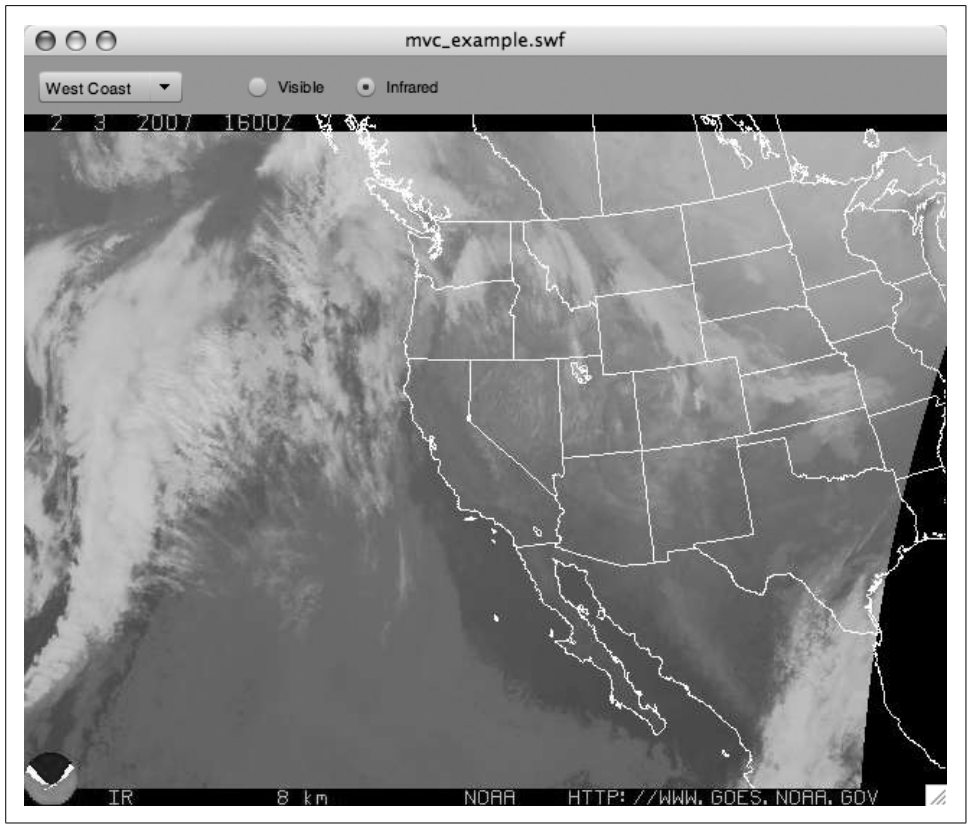

Figure 12-5. Extended example showing an infrared map of the West Coast

Let's look at the changes needed to update the model.

Adding a New Model

Instead of modifying the current weather maps model (Example 12-13), we will extend it to incorporate the new data and additional state information. We need to add another property to indicate the chosen map type, either a visible map or an infrared one. In addition, we need to update the application logic to work with the new data. Example 12-18 shows a new interface for the model that defines the methods required for the new features.

Example 12-18. INewModel.as

```
package
{
    import flash.events.*;

    public interface INewModel extends IModel
```

Example 12-18. INewModel.as (continued)

```
    {
        function getMapTypeList( ):Array
        function getMapType( ):uint
        function setMapType(index:uint):void
    }
}
```

Interface `INewModel` extends the `IModel` interface (Example 12-12) and defines the methods required to get the list of image types as an array, including get and set methods for the currently selected map type (visible or infrared). Example 12-19 shows the `NewModel` class that extends the previous `Model` class to implement the new `INewModel` interface.

Example 12-19. NewModel.as

```
package {

    public class NewModel extends Model implements INewModel {

        protected var aMapTypes:Array;
        protected var chosenMapType:uint;

        protected var aIRImageURLs:Array;

        public function NewModel( ) {
            this.aIRImageURLs = new Array(
                   "http://www.goes.noaa.gov/GIFS/ECIR.JPG",
                   "http://www.goes.noaa.gov/GIFS/WCIR.JPG",
                   "http://www.goes.noaa.gov/GIFS/PRIR.JPG",
                   "http://www.goes.noaa.gov/GIFS/ALIR.JPG",
                   "http://www.goes.noaa.gov/GIFS/HAIR.JPG");
            this.aMapTypes = new Array(
                   "Visible",
                   "Infrared");
            this.chosenMapType = 0;
        }

        public function getMapTypeList( ):Array {
            return aMapTypes;
        }

        public function getMapType( ):uint {
            return this.chosenMapType;
        }

        public function setMapType(index:uint):void {
            this.chosenMapType = index;
            this.update( );
        }
```

Example 12-19. NewModel.as (continued)

```
        override public function getMapURL( ):String {
            switch(chosenMapType) {
                case 1:
                    return this.aIRImageURLs[chosenRegion];
                    break;
                default:
                    return this.aImageURLs[chosenRegion];
                    break;
            }
        }
    }
}
```

The `getMapURL( )` method was overridden, as the returned image URL now depends on the currently chosen map type. Note that we didn't modify any existing code, but extended the existing model class `Model` to implement the new interface requirements.

Adding a New Controller

A new controller (Example 12-20) is necessary to handle the input to the map type selector view. We can implement the same `ICompInputHandler` interface (Example 12-14) for the new controller.

Example 12-20. MapTypeController.as

```
package
{
    public class MapTypeController implements ICompInputHandler {

        private var model:Object;

        public function MapTypeController(oModel:INewModel)
        {
            this.model = oModel;
        }

        public function compChangeHandler(index:uint):void
        {
            (model as INewModel).setMapType(index); // update model
        }
    }
}
```

Adding a New View

We will add a new view (Example 12-21) that consists of two grouped radio buttons that allow the user to select from either visible or infrared map images. The view will use the built-in `RadioButtton` component in Flash CS3. Make sure the radio button component is dragged from the Components panel into the Library panel in the Flash document.

Example 12-21. RBView.as

```
package
{
    import flash.events.*;
    import fl.controls.RadioButton;
    import fl.controls.RadioButtonGroup;

    public class RBView extends ComponentView
    {
        private var rbList:Array = new Array( );
        private var rbGrp:RadioButtonGroup;

        public function RBView(aModel:INewModel,aController:ICompInputHandler)
        {
            super(aModel, aController);
            // get region names from model
            var aMapTypes:Array = model.getMapTypeList( );
            // develop radio buttons using map type names
            rbGrp = new RadioButtonGroup("Map Type");
            for (var i:uint = 0; i < aMapTypes.length; i++)
            {
                var rb:RadioButton = new RadioButton( );
                rb.label = aMapTypes[i];
                rb.value = i;
                rb.group = rbGrp;
                rb.x = i * 75;
                addChild(rb);
                rbList.push(rb);
            }
            update( ); // select default buttton
            // register to recieve changes to radio button group
            rbGrp.addEventListener(MouseEvent.CLICK, this.changeHandler);
        }

        override public function update(event:Event = null):void
        {
            // get data from model and update view
            var index:uint = (model as INewModel).getMapType( );
            rbList[index].selected = true;
            super.update(event);
        }

        private function changeHandler(event:Event):void
        {
            // delegate to the controller (strategy) to handle map type change
            controller.compChangeHandler(event.target.selection.value);
        }
    }
}
```

The `RBView` class (Example 12-21) subclasses `ComponentView` (Example 12-7) and draws the radio buttons. The `RadioButtton` component must be used in a group of at least two `RadioButtton` instances. Only one member of the group can be selected at any

given time. As in previous view implementations, the `update( )` method is called without parameters from the constructor immediately after drawing the components. This ensures that the UI component displays the default selection specified in the model. We can now instantiate the MVC elements and develop the composite view.

Building the MVC Triad

The nested view structure of our extended weather map application is shown in Figure 12-6.

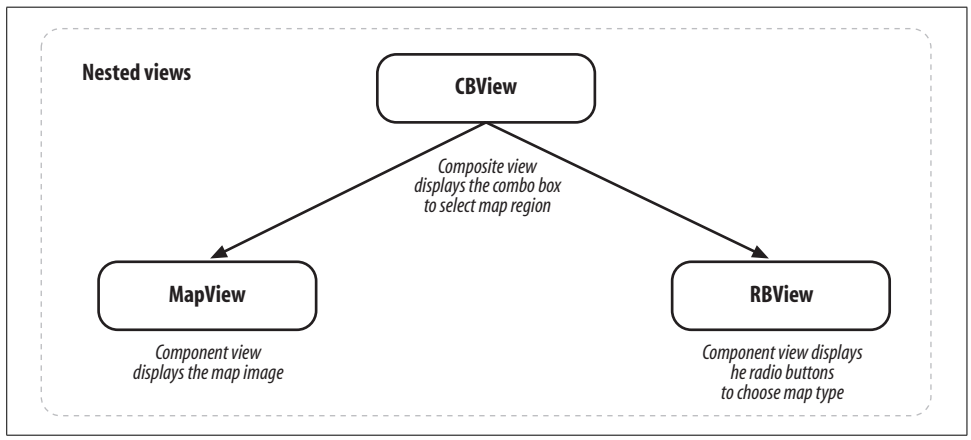

Figure 12-6. Nested view structure for extended weather map example

The following statements should be executed from the document class of the Flash document. Only the root node `CBView` object (see Figure 12-6) will register with the model to receive update events. Update events will trickle down to the child nodes because of the composite pattern implementation.

```
var model:INewModel = new NewModel( ); // new model
var controller:ICompInputHandler = new Controller(model);

// region select combo box view
var view:CompositeView = new CBView(model, controller);
view.x =  view.y = 10;
addChild(view);

// add map view as child
var map:ComponentView = new MapView(model);
view.add(map);
map.x = 0
map.y = 40;
addChild(map);

// controller to handle map type input
var mapTypeController:ICompInputHandler = new MapTypeController(model);
```

```
// add map type select radio button group view as child
var mapTypeView:ComponentView = new RBView(model, mapTypeController);
view.add(mapTypeView);
mapTypeView.x =  150;
mapTypeView.y = 10;
addChild(mapTypeView);

// register root view to recieve notifications from the model
model.addEventListener(Event.CHANGE, view.update);
```

To extend our original weather map application, we added a new view, a new controller and a new model. However, at no point did we modify or change existing code. What the pundits say about the MVC pattern is indeed borne out in this extended example. All the elements in the MVC are loosely coupled, allowing us to add or swap out any element without changing existing elements. In addition, implementing nested views using the composite pattern allows us to reconfigure the screen layout and the view update process from the client without making changes to individual views.

Example: Cars

In this example, we will develop a simple car using the MVC pattern that can be controlled from the keyboard. The emphasis will be on developing custom views using ActionScript as opposed to using built-in components. We will also see how simply changing the controller can turn a car that responds to keyboard input into one that chases another car. Casual game developers will find this example useful for game design. Figure 12-7 shows the final iteration of the example application.

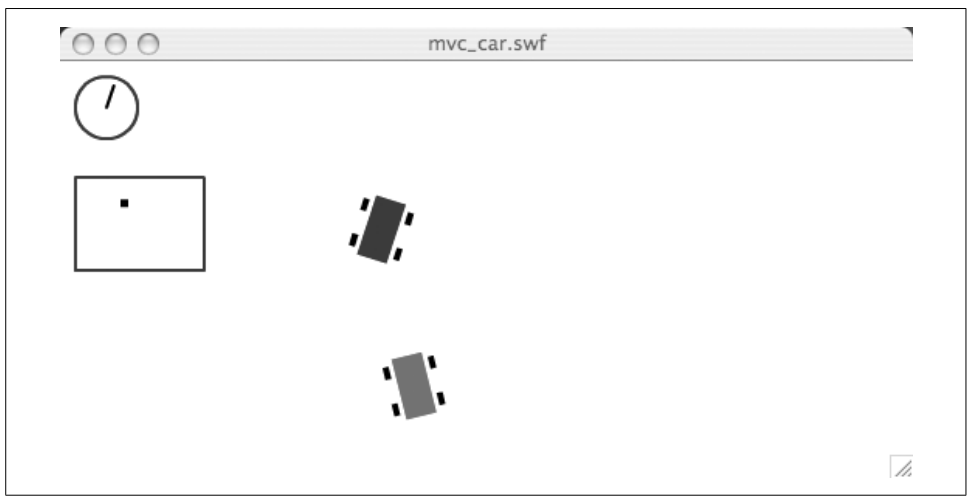

Figure 12-7. Car example showing views and chase car

The Model

Let's first develop the interface for our car model. We should be able to set its location on the stage and steer it by simply changing its rotation angle. We will also declare methods to set its color. Example 12-22 shows the `ICar` interface.

Example 12-22. ICar.as

```
package
{
    import flash.geom.Point;
    import flash.events.IEventDispatcher;

    public interface ICar extends IEventDispatcher {
    {
        function setLoc(pt:Point):void;
        function getLoc( ):Point;
        function setColor(color:uint):void;
        function getColor( ):uint;
        function addToRotationAngle(nAngle:int):void;
        function getRotation( ):int;
    }
}
```

The `CarModel` class shown in Example 12-23 implements the `ICar` interface. The car will move at a constant speed in the forward direction. Note that the model updates the car position using a timer.

Example 12-23. CarModel.as

```
package
{
    import flash.events.*;
    import flash.geom.*;
    import flash.utils.Timer;

    public class CarModel extends EventDispatcher implements ICar
    {
        protected var nSpeed:Number; // holds speed of car in pixels/frame
        protected var nRotation:Number; // car rotation in Degrees
        protected var ptLoc:Point; // current location
        protected var carColor:uint; // car color

        public function CarModel( )
        {
            nSpeed = 3;
            nRotation = 0;
            ptLoc = new Point(0, 0); // default loc is 0,0
            carColor = 0x000000; // default color is black

            // set timer to update car position every 1/20 second
            var carMoveTimer:Timer = new Timer(1000 / 20);
            carMoveTimer.addEventListener("timer", doMoveCar);
```

Example 12-23. CarModel.as (continued)

```
            carMoveTimer.start( );
        }

        public function setLoc(pt:Point):void
        {
            ptLoc = pt;
        }

        public function getLoc( ):Point
        {
            return ptLoc;
        }

        public function setColor(color:uint):void
        {
            this.carColor = color;
        }

        public function getColor( ):uint
        {
            return carColor;
        }

        public function getRotation( ):int
        {
            return nRotation;
        }

        // Add this to car rotation angle (in Degrees)
        public function addToRotationAngle(nAngle:int):void
        {
            nRotation += nAngle;
        }

        // move the car
        private function doMoveCar(event:TimerEvent):void
        {
            var newLocOffset:Point = Point.polar(nSpeed, nRotation * Math.PI / 180);
            ptLoc.x += newLocOffset.x; // move by the x offset
            ptLoc.y += newLocOffset.y; // move by the y offset
            this.update( );
        }

        protected function update( ):void
        {
            dispatchEvent(new Event(Event.CHANGE)); // dispatch event
        }
    }
}
```

The Controller

The controller responds to keyboard input, and updates the model to turn the car by rotating it. Example 12-24 shows the `IKeyboardInputHandler` interface for the controller.

Example 12-24. IKeyboardInputHandler.as

```
package
{
    import flash.events.*;

    public interface IKeyboardInputHandler
    {
        function keyPressHandler(event:KeyboardEvent):void
    }
}
```

The `RHController` class shown in Example 12-25 implements the `IKeyboardInputHandler` interface. It responds to keyboard events and decides how much the car should turn, based on left or right arrow key presses. This is a good example of a controller deciding in what ways and by how much it responds to user input. In this example, each left or right arrow press rotates the car 8 degrees clockwise or counterclockwise.

Example 12-25. RHController.as

```
package
{
    import flash.events.*;
    import flash.ui.*;

    public class RHController implements IKeyboardInputHandler
    {
        private var model:ICar;

        public function RHController(aModel:ICar)
        {
            this.model = aModel;
        }

        public function keyPressHandler(event:KeyboardEvent):void
        {
            switch (event.keyCode)
            {
                case Keyboard.LEFT :
                    model.addToRotationAngle(-8);
                    break;
                case Keyboard.RIGHT :
                    model.addToRotationAngle(8);
                    break;
```

Example 12-25. RHController.as (continued)

```
            }
        }
    }
}
```

The Views

We will implement two nested views: one for keyboard input and another that will draw and update the car on stage. The `KeyboardInputView` class shown in Example 12-26 is a composite view. It registers with the stage to receive key press events, and delegates to the controller to handle them.

Example 12-26. KeyboardInputView.as

```
package
{
    import flash.events.*;
    import flash.display.*;

    public class KeyboardInputView extends CompositeView
    {
        public function KeyboardInputView(aModel:ICar,
                                          aController:IKeyboardInputHandler, target:Stage)
        {
            super(aModel, aController);
            target.addEventListener(KeyboardEvent.KEY_DOWN, onKeyPress);
        }

        protected function onKeyPress(event:KeyboardEvent):void
        {
            (controller as IKeyboardInputHandler).keyPressHandler(event);
        }
    }
}
```

The `CarView` class shown in Example 12-27 is a component view. It draws the car using its assigned color. In the `update( )` method, it reads the current state of the car from the model and sets its location and rotation. The interesting aspect of this view is that its position changes. Views don't necessarily have to be classic user interface elements like buttons, image placeholders, etc. They can be any customized representation of model state.

Example 12-27. CarView.as

```
package {

    import flash.geom.*;
    import flash.events.*;

    public class CarView extends ComponentView {
```

Example 12-27. CarView.as (continued)

```
        public function CarView(aModel:ICar, aController:Object = null) {

            super(aModel, aController);

            // draw car body
            graphics.beginFill(model.getColor( ));
            graphics.drawRect(-20, -10, 40, 20);
            graphics.endFill( );
            // draw tires
            drawTire(-12, -15);
            drawTire(12, -15);
            drawTire(-12, 15);
            drawTire(12, 15);

            // update car
            this.update( );
        }

        private function drawTire(xLoc:int, yLoc:int) {
            graphics.beginFill(0x000000); // black color
            graphics.drawRect(xLoc - 4, yLoc - 2, 8, 4);
            graphics.endFill( );
        }

        override public function update(event:Event = null):void {
            // get data from model and update view
            var ptLoc:Point = (model as ICar).getLoc( );
            this.x = ptLoc.x;
            this.y = ptLoc.y;
            this.rotation = (model as ICar).getRotation( );
        }
    }
}
```

Building the Car

As in our previous examples, building the MVC triad is straightforward. We develop a nested view for the car with the `KeyboardInputView` class object as the root node, with a `CarView` object added as a child. The client code shown in Example 12-28 will instantiate the MVC components and develop the nested view structure.

Example 12-28. Main.as (document class for car example)

```
package {

    import flash.display.*;
    import flash.events.*;
    import flash.geom.*;

    /**
    *    Main Class
```

Example 12-28. Main.as (document class for car example) (continued)

```
 9      *    @ purpose:        Document class for movie
10      */
11      public class Main extends Sprite {
12
13          public function Main( ) {
14
15              var carModel:ICar = new CarModel( );
16              carModel.setLoc(new Point(200,200));
17              carModel.setColor(0x0000FF); // blue
18
19              var carController:IKeyboardInputHandler = new RHController(carModel);
20
21              // keyboard input view (composite)
22              var kbInputView:CompositeView = new KeyboardInputView
23
24
25              // car view (component)
26              var car:ComponentView = new CarView(carModel);
27              kbInputView.add(car); // add car view to keyboard input
28
29              addChild(car);
30
31              // register keyboard input view to recieve
32
33              carModel.addEventListener(Event.CHANGE, kbInputView.update);
34
35          }
36      }
37  }
```

The car should now move on the stage and turn based on left- and right-arrow key presses. Note that there are no boundary checks for the car, and it can keep going right off the stage. We will now add some custom views to the car that show its direction and location on the stage.

Custom Views

To illustrate the ease by which custom views can be created and added to an MVC model, we will create two additional views. The first will be a circular gauge with a hand showing the current direction of motion for the car. The other view will show the position of the car relative to stage boundaries, somewhat like the display screen of a global positioning system (GPS).

Direction Gauge View

Example 12-29 shows the DirectionGaugeView class. This is a component view that shows the direction of motion of the car using a circular gauge with a hand like a clock. The gauge hand is a sprite whose rotation is set to the same value as the car rotation.

Example 12-29. DirectionGaugeView.as

```
package
{
    import flash.geom.*;
    import flash.events.*;
    import flash.display.*;

    public class DirectionGaugeView extends ComponentView
    {

        private var guageHand:Sprite;

        public function DirectionGaugeView(aModel:Object, aController:Object = null)
        {
            super(aModel, aController);

            // draw circle for guage
            graphics.lineStyle(2, (model as ICar).getColor( ));
            graphics.drawCircle(10, 10, 20);

            // draw guage hand as sprite
            guageHand = new Sprite( );
            guageHand.graphics.lineStyle(2, 0x000000);
            guageHand.graphics.moveTo(0,0);
            guageHand.graphics.lineTo(15,0);
            guageHand.x = guageHand.y = 10;
            this.addChild(guageHand);
        }

        override public function update(event:Event = null):void
        {
            this.guageHand.rotation = (model as ICar).getRotation( );
        }
    }
}
```

GPS View

The GPSView class shown in Example 12-30 draws a rectangle in proportion to the stage, and displays the location of the car relative to stage boundaries.

Example 12-30. GPSView.as

```
package
{
    import flash.geom.*;
    import flash.events.*;
    import flash.display.*;

    public class GPSView extends ComponentView
    {
        private var carPos:Sprite;
        private static const SF:Number = 0.15; // scale factor

        public function GPSView(aModel:ICar, target:Stage)
        {
            super(aModel);

            // draw rectangle in proportion to stage
            graphics.lineStyle(2, (model as ICar).getColor( ));
            graphics.drawRect(0, 0, target.stageWidth * SF, target.stageHeight * SF);

            // draw guage hand as sprite
            carPos = new Sprite( );
            carPos.graphics.beginFill(0x000000); // black color
            carPos.graphics.drawRect(-2, -2, 5, 5);
            carPos.graphics.endFill( );
            this.addChild(carPos);
        }

        override public function update(event:Event = null):void
        {
            var pt:Point = (model as ICar).getLoc( );
            this.carPos.x = pt.x * SF;
            this.carPos.y = pt.y * SF;
        }
    }
}
```

Adding the Custom Views

Adding the custom views to the nested view structure is easy, without worrying about registering for update events. We can simply add the two composite views as child nodes to the root node of the view structure. The following code inserted at line 23 to Example 12-28 will accomplish this.

```
// direction gaugue view (component)
var dash:ComponentView = new DirectionGaugeView(carModel);
kbInputView.add(dash); // add car view to keyboard input
                       // view as child component
dash.x = dash.y = 20;
addChild(dash);

// GPS view (component)
var gps:ComponentView = new GPSView(carModel, this.stage);
```

```
    kbInputView.add(gps); // add gps view to keyboard input
                          // view as child component
    gps.x = 10;
    gps.y = 75;
    addChild(gps);
```

Adding a Chase Car

How difficult would it be to add a chase car that chases the car controlled by the keyboard? The chase car will need to automatically steer itself to catch up with the user controlled car. We will use a simple chase algorithm for the chase car. If the lead car is to the left of the chaser, the chase car will turn slightly to the left. If the lead car is to the right, then the chase car will turn to the right.

The only additional element we need is a new controller. Example 12-31 shows the `IChaseHandler` interface that defines a chase algorithm based on timer events. It also declares a method to set the chase target, which is of type `ICar`.

Example 12-31. IChaseHandler.as

```
package
{
    import flash.events.*;

    public interface IChaseHandler
    {
        function chaseHandler(event:TimerEvent):void
        function setChaseTarget(car:ICar):void
    }
}
```

The `ChaseController` class shown in Example 12-32 implements the `IChaseHandler` interface. It sets a timer to call the `chaseHander( )` listener method every 1/20 of a second to tell the model how much to change the rotation angle of the chase car. It's a good idea to set the timer to the same frequency as the movie frame rate. Another issue of note is the turn radius of the chase controller. It is half that of the user controlled car (see Example 12-25). The chase car can only turn 4 degrees at a time compared to 8 degrees for the user-controlled car. Therefore the user-controlled car should be able to evade the chaser by making some tight turns.

Example 12-32. ChaseController.as

```
package
{
    import flash.events.*;
    import flash.geom.*;
    import flash.utils.Timer;

    public class ChaseController implements IChaseHandler
    {
```

Example 12-32. ChaseController.as (continued)

```
        private var model:ICar;
        private var target:ICar;

        public function ChaseController(aModel:ICar)
        {
            this.model = aModel;

            // set timer to call chase controller every 1/20 second
            var timer:Timer = new Timer(1000 / 20);
            timer.addEventListener("timer", chaseHandler);
            timer.start( );
        }

        public function chaseHandler(event:TimerEvent):void
        {
            var myLoc:Point = model.getLoc( );
            var targetLoc:Point = target.getLoc( );
            var myRotationAngle:Number = model.getRotation( );
            var angleToTarget:Number = Math.atan2(targetLoc.y - myLoc.y,
                                       targetLoc.x - myLoc.x) * 180 / Math.PI;

            if ((myRotationAngle % 360) < angleToTarget)
            {
                model.addToRotationAngle(4);
            } else {
                model.addToRotationAngle(-4);
            }
        }

        public function setChaseTarget(car:ICar):void
        {
            target = car;
        }
    }
}
```

The controller sets the chase algorithm; the utility of using the strategy pattern here should be clear. We can simply substitute another controller with a more sophisticated chase algorithm without requiring any changes to the other elements of the MVC pattern. The following code will add the chase car to the example.

```
// ** chase car **
var chaseCarModel:ICar = new CarModel( );
chaseCarModel.setLoc(new Point(100,100));
chaseCarModel.setColor(0xFF0000); // red

var chaseCarController:IChaseHandler = new ChaseController(chaseCarModel);
chaseCarController.setChaseTarget(ICar(carModel));

// chase car view (component)
var chaseCarView:ComponentView = new

addChild(chaseCarView);
```

```
// register chase car to receive notifications from the model
chaseCarModel.addEventListener(Event.CHANGE, chaseCarView.update);
```

Note that the chase car consists of a new MVC triad separate from the lead car. It has its own model, views, and controllers. We reuse the same `CarModel` and `CarView` classes, but change the behavior of the chase car by swapping out the controller. Combining new or subclassed model, view, and controller components makes applications designed using the MVC pattern infinitely extensible.

Summary

The Model-View-Controller (MVC) pattern is commonly used to create software applications that contain user interfaces. The power of the MVC pattern can be attributed to the separation of responsibilities among the three elements that make up the pattern. The *Model* contains the application data and logic to manage the state of the application. The *View* presents the user interface and the state of the application onscreen. The *Controller* handles user input to change the state of the application.

The MVC pattern can integrate the observer, strategy, and composite patterns to manage the dependencies both within and between its elements. The relationship between the model and view is that of *concrete subject* and *concrete observer* in an observer pattern. The relationship between the view and controller is that of *context* and *concrete strategy* in a strategy pattern. Views in an MVC pattern can have multiple nested views. The relationship between nested views can be in the form of *components* and *composite* nodes in a composite pattern.

Most importantly, the MVC pattern provides a clear framework for design. The separation of responsibilities among the model, view, and controller elements allows easy substitution of elements without disruptions to the overall application. This lets us easily expand applications based on the MVC pattern to meet changing requirements.

Other resources from O'Reilly

Related titles

Essential ActionScript 3.0
Dynamic HTML: The Definitive Reference
Ajax on Java
Ajax on Rails
Learning JavaScript
Programming Atlas
Head Rush Ajax
Rails Cookbook

oreilly.com

oreilly.com is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.