

*Object-Oriented Development with ActionScript 2.0*

# Essential ActionScript 2.0



O'REILLY®

*Colin Mook*

# The Model-View-Controller Design Pattern

In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.

—from *Applications Programming in Smalltalk-80(TM)*:

*How to use Model-View-Controller (MVC)*, by Steve Burbeck, available at: <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>

The Model-View-Controller (MVC) design pattern separates user interface code into three distinct classes:

### *Model*

Stores the data and application logic for the interface

### *View*

Renders the interface (usually to the screen)

### *Controller*

Responds to user input by modifying the model

For example, in a toggle button, the model would store the state of the button (on or off), the view would draw the button on screen, and the controller would set the state of the button in the model (to on or off) when the button is clicked. But an interface need not be visual. In some cases, the view might play a sound or, as a non-ActionScript example, the view might make a video game controller vibrate.

MVC originated in the Smalltalk language and has been used widely for years in many different incarnations. Though the basic principles of the pattern are easy to understand, its details are complex enough to foster an enormous amount of debate and contradictory implementations. In this chapter, we'll study a relatively traditional implementation of the pattern, bearing in mind that there is no single "right" way to implement MVC.

The basic principle of MVC is the separation of responsibilities. In an MVC application, the model class concerns itself only with the application's state and logic. It has no interest in how that state is represented to the user or how user input is received. By contrast, the view class concerns itself only with creating the user interface in response to generic updates it receives from the model. It doesn't care about application logic nor about the processing of input; it just makes sure that the interface reflects the current state of the model. Finally, the controller class is occupied solely with translating user input (provided by the view) into updates that it passes to the model. It doesn't care how the input is received or what the model does with those updates.

Separating the code that governs a user interface into the model, view, and controller classes yields the following benefits:

- Allows multiple representations (views) of the same information (model)
- Allows user interfaces (views) to be easily added, removed, or changed, at both compile time and runtime
- Allows response to user input (controller) to be easily changed, at both compile time and runtime
- Promotes reuse (e.g., one view might be used with different models)
- Allows multiple developers to simultaneously update the interface, logic, or input of an application without affecting other source code
- Helps developers focus on a single aspect of the application at a time

Despite those benefits, not all user interfaces are best implemented with MVC. For example, in Chapter 12 we built a simple currency converter application using a single class, *CurrencyConverter*. The conceptual responsibilities of the model, view, and controller were still manifest in that application, but they were encompassed by a single class. The currency converter application was so simple that the cost of implementing formal MVC outweighed the benefits.



Design patterns offer more benefits to larger projects than to smaller ones, but the concepts in a pattern like MVC can still inform the design of simple applications like the currency converter.

The MVC pattern can be applied to a single user interface element (like a button), to a group of user interface elements (like a control panel), or to an entire application. This chapter uses MVC to create a clock that combines three user interface elements: a digital display, an analog display, and a toolbar for starting, stopping, and resetting the clock.

# The General Architecture of MVC

Before we study our specific clock example, let's explore the general structure of the MVC design pattern.

## Communication in MVC

Although the model, view, and controller classes in MVC are intentionally segregated, they must communicate regularly. The model must send notifications of state changes to the view. The view must register the controller to receive user interface events and, possibly, request data from the model. The controller must update the model and, possibly, update the view in response to user input.

To facilitate this communication, each object in MVC must store a reference to the other object(s) with which it interacts. Specifically, the model instance needs a reference to the view instances that render it, while the view and controller each needs a reference to the model and reciprocal references to each other. Figure 18-1 shows how the objects in MVC reference one another. The diamond shape in the figure represents a composition relationship, in which one object stores an instance of another.

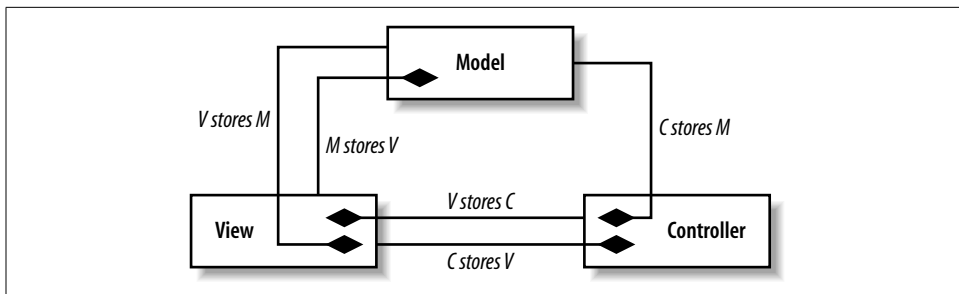


Figure 18-1. Object references in MVC

Communication proceeds in a single direction (as shown in Figure 18-2) through the object references shown in Figure 18-1, as follows:

1. The view receives user input and passes it to the controller.
2. The controller receives user input from the view.
3. The controller modifies the model in response to user input (or, in some cases, the controller modifies the view directly and does not update the model at all).
4. The model changes based on an update from the controller.
5. The model notifies the view of the change.
6. The view updates the user interface, (i.e., presents the data in some way, perhaps by redrawing a visual component or by playing a sound).

Figure 18-2 depicts the MVC communication cycle. The starting point for the cycle is typically the receipt of user input. However, another part of the program could also start the cycle by modifying the model directly (perhaps in response to new data arriving from a server).

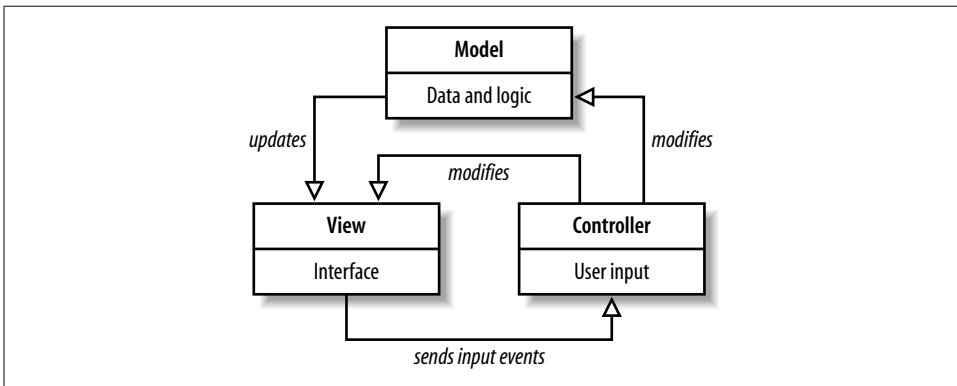


Figure 18-2. The MVC communication cycle

Note that Figure 18-2 shows the simplest case of MVC, with a single model, a single view, and a single controller. However, it's not uncommon for an application to provide two or more views for a single model. For example, in our MVC clock implementation, we'll use one model to manage the time, but we'll have three views—one to represent the clock in digital format, one to represent the clock in analog format, and one to display buttons that start, stop, and reset the clock. You may wonder how these buttons qualify as a view of the clock given that they do not represent the time. Views do not necessarily have to represent the entire model and need not even represent the same information (the way the analog and digital views do). As we'll see later, the buttons in our clock do not indicate the current time; instead, the button states reflect whether the clock is currently running, which is simply another aspect of the model.

While an MVC implementation may contain multiple views per model, every view has exactly one controller instance and vice versa. Each view's controller is dedicated to that view's sole service.



The view and the controller form an indivisible pair. MVC requires that each view has a controller (even if it is just the placeholder value, `null`) and each controller has a view.

Some views do not allow user input. For example, a view might be a simple graph that cannot be edited. When a view accepts no user input, it does not need a controller to translate user input into model updates. Hence, views that do not allow user input have either `null` in place of a controller or a controller that does nothing in response to user input.

## Class Responsibilities in MVC

As we've already learned, responsibilities in an MVC implementation are divided among the model, view, and controller. By the end of this chapter, you should be mumbling the MVC mantra to yourself at the grocery store—"the model manages data and logic, the view creates the interface, and the controller processes user input." Those general responsibilities break down into many specific tasks, covered next.

### Responsibilities of the model

The model stores data in properties and provides application-specific methods that set and retrieve that data. The data-management methods are not generic; they are customized per application and must be known to the controller and the view. For example, the model in our clock application defines methods specific to a clock, such as *setTime()* and *stop()*. Controllers in MVC are custom written to manipulate a specific model; for example, a controller in our clock application must know about the *setTime()* and *stop()* methods in order to modify the model.

The model's data can be changed externally by an outside class or internally by its own logic. For example, the time of our clock's model might be reset by the controller in response to user input, or it might be updated because, internally, it detects the passing of a second. (It might be monitoring the operating system's built-in clock or, as in our upcoming example, tracking the passing milliseconds itself.)

The model must also provide a way for views to register and unregister themselves, and it must manage a list of registered views. Whenever the model determines that its state has changed meaningfully, it must notify all registered views.

Finally, the model implements the logic of the MVC triad. For example, the model in our clock application implements a *tick()* method that runs once per second, updating the time. The model might also provide data validation services and other application-specific utilities, such as loading the current time from a server-side application.

### Responsibilities of the view

The view must create the user interface and keep it up-to-date. The view listens for state changes in the model; when the model changes, the view updates the interface to reflect the change. For example, in our clock application, when the time changes, the model notifies the three registered views. In response, the analog clock view positions the hands of a traditional clock, while the digital clock view sets the numbers in its digital display. The third, "buttons" view (*ClockTools*) changes the appearance of the buttons to indicate whether the clock is running or has been stopped.

Each view must forward all input events to its controller. It should not process any inputs itself. For example, our analog clock view might allow the user to set the time by dragging the hands of the clock. When the hands are dragged, the view merely forwards the input information to the controller, and the controller decides what to do.

Depending on the specific implementation, the view might query the model for its state in order to determine what changed when an update is received. The view never changes the model but can retrieve information from it. For example, our clock's model might tell its views that the time changed, and the views, in response, might invoke *getTime()* on the model to determine the new time. Alternatively (and more commonly), the model might send the new time to the views directly in an info object at update time. You should recognize these two options as the push and pull models discussed in Chapter 16 for the Observer pattern. See that chapter for details.

You may be wondering whether the MVC design pattern as presented makes sense architecturally. It may seem odd that, if the user changes something in the view, instead of responding immediately, the view passes the input to the controller, which passes it to the model, which notifies the view of the change, and finally the view requests the details of the change from the model and renders them for the user. Wouldn't it be easier to skip all the detailed communication and just have the view update itself whenever the user makes a change? In some sense that may be easier, and in a simple application it might be appropriate. As an application becomes more complex, however, the MVC pattern offers significant benefits. For example, in the case of our clock implementation, in which there are multiple views, the architecture allows changes in one view to be detected by and reflected in all views. Therefore, you don't have to write code to make the analog clock view notify or update the digital clock view (or vice versa). Furthermore, the logic is centralized in the model, which prevents code duplication and allows us to add or remove views at runtime with minimal effort. For example, you could add a view that displays time in 24-hour (a.k.a. military) format instead of 12-hour (a.m./p.m.) format. Complex applications demand this type of flexibility, and MVC provides the structure to implement it.

### **Responsibilities of the controller**

The controller listens for notifications from the view based on user input and translates that input into changes in the model. In some cases, the controller makes logical decisions about the input before making a corresponding change to the model. For example, our clock application has a Reset button that resets the time to midnight. When the button is clicked, the clock controller translates the input conceptually from "Reset button clicked" to the command "Set model's time to 00:00:00."

In some special cases, the controller might also instruct the view to make changes to the user interface by calling methods on the view. The changes are sent directly to the view only when they are purely cosmetic and have no effect on the model. For example, if a user interface has buttons to alphabetize a list of names in ascending and descending order, the controller may legitimately call the appropriate sort methods on the view when those buttons are clicked. Calling the sort methods is legitimate because it does not change the underlying data stored in the model (i.e., the list of names); only the presentation of that data changes.

## What Creates the MVC Classes?

Through all this talk of the model, view, and controller, we still haven't seen how to make instances of those classes. That's partly because there's no single, definitive way to instantiate the classes in MVC. In our clock example, we'll create a single class, *Clock*, which instantiates the model, its views, and their controllers. Our *Clock* class sets up the MVC classes as follows:

- Create the model
- Create views
- Register views with the model

Notice that the controllers are missing from the preceding list because they are created by their respective views.

Hence, our *Clock* class forms a wrapper around the MVC triad, packaging it into a tidy, self-contained unit. However, that's definitely not the only approach possible. At least one Java implementation suggests that the controller class should create the model, and possibly the view, in its constructor!

In our example code, we'll follow the traditional (Smalltalk) implementation of MVC, in which the model and view(s) are created by some containing class, and then the model and controller are registered for each view. When the controller for a view is not specified, the view class creates a default controller for itself automatically.

## A Generalized MVC Implementation

Now that we've learned the theoretical structure of MVC, let's explore a real-world MVC implementation. To get started, we'll create a basic MVC framework that can be reused across many projects. We'll store our core structure in a package called *mvc*. The classes and interfaces in the *mvc* package are:

*View*

An interface all views must implement

*Controller*

An interface all controllers must implement

*AbstractView*

A generic implementation of the *View* interface

*AbstractController*

A generic implementation of the *Controller* interface

You'll notice that the model class is conspicuously absent from our *mvc* package. That's because the model-view relationship in MVC is essentially the same as the Observer pattern we studied in Chapter 16. Hence, we'll use the core code from our existing Observer implementation rather than rewriting similar code for our MVC

implementation. The model class in our MVC architecture will be a subclass of *Observable*, which resides in the package *util*. The Observer pattern provides the basic services for the model-view relationship—views register with the model using *addObserver()* and *removeObserver()*, and the model sends updates via the standardized method *update()*, implemented by each view. Hence, in order to register with a model, every view class must implement the *util.Observer* interface in addition to implementing the *mvc.View* interface.

By definition, to use the *Observable* class as the basis of our model class, the model class must extend *Observable*. If we want our model class to inherit from a different class, we'd have to use the composition-based implementation of the Observer pattern, from Example 16-7.

## The View Implementation

Example 18-1 shows the *View* interface in our *mvc* package. The *View* interface defines the basic services every *View* class must offer, namely:

- Methods to set and retrieve the controller reference
- Methods to set and retrieve the model reference
- A method that returns the default controller for the view

Notice that references to the controller in Example 18-1 must be instances of a class that implements the *Controller* interface. References to the model must be instances of a class that extends *Observable* (or, in the composition-based version of the Observer pattern, that implements the *Observable* interface).

*Example 18-1. The View interface*

```
import util.*;
import mvc.*;

/**
 * Specifies the minimum services that the view
 * of a Model-View-Controller triad must provide.
 */
interface mvc.View {
    /**
     * Sets the model this view is observing.
     */
    public function setModel (m:Observable):Void;

    /**
     * Returns the model this view is observing.
     */
    public function getModel ():Observable;

    /**
     * Sets the controller for this view.
     */
}
```

Example 18-1. The View interface (continued)

```
public function setController (c:Controller):Void;

/**
 * Returns this view's controller.
 */
public function getController ():Controller;

/**
 * Returns the default controller for this view.
 */
public function defaultController (model:Observable):Controller;
}
```

To make the *View* interface easy to implement, we provide *AbstractView*, a class that implements the basic services defined by the *View* interface (namely, managing interactions with the controller and model). The *AbstractView* class also implements the *Observer* interface. To create a real view class in an MVC application, we need simply extend *AbstractView* and add code to build and update a user interface. The core MVC grunt work in any real view class is taken care of by *AbstractView*.

Example 18-2 shows the code for the *AbstractView* class. Typically, subclasses of *AbstractView* will create a user interface at construction time and modify that user interface from the *update()* method.

Notice that an instance of the model class must be passed to *AbstractView*'s constructor. Without that reference, the view cannot query the model for its state. An instance of the controller class can also be passed to the view constructor. However, if you don't supply a controller instance, the *AbstractView* class uses *defaultController()* to create the controller automatically the first time it is requested (i.e., the first time *getController()* is invoked). Each *AbstractView* subclass is expected to override *defaultController()*, providing a reference to its own default controller.

Example 18-2. The *AbstractView* class

```
import util.*;
import mvc.*;

/**
 * Provides basic services for the view of a Model-View-Controller triad.
 */
class mvc.AbstractView implements Observer, View {
    private var model:Observable; // A reference to the model.
    private var controller:Controller; // A reference to the controller.

    public function AbstractView (m:Observable, c:Controller) {
        // Set the model.
        setModel(m);

        // If a controller was supplied, use it. Otherwise let the first
        // call to getController() create the default controller.
    }
}
```

Example 18-2. The *AbstractView* class (continued)

```
    if (c !== undefined) {
        setController(c);
    }
}

/**
 * Returns the default controller for this view.
 */
public function defaultController (model:Observable):Controller {
    return null;
}

/**
 * Sets the model this view is observing.
 */
public function setModel (m:Observable):Void {
    model = m;
}

/**
 * Returns the model this view is observing.
 */
public function getModel ():Observable {
    return model;
}

/**
 * Sets the controller for this view.
 */
public function setController (c:Controller):Void {
    controller = c;
    // Tell the controller this object is its view.
    getController().setView(this);
}

/**
 * Returns this view's controller.
 */
public function getController ():Controller {
    // If a controller hasn't been defined yet...
    if (controller === undefined) {
        // ...make one. Note that defaultController() is normally overridden
        // by AbstractView's subclass so that it returns the appropriate
        // controller for the view.
        setController(defaultController(getModel()));
    }
    return controller;
}

/**
 * A do-nothing implementation of the Observer interface's
 * update() method. Subclasses of AbstractView provide a concrete
```

Example 18-2. The *AbstractView* class (continued)

```
    * implementation for this method.
    */
    public function update(o:Observable, infoObj:Object):Void {
    }
}
```

Now that we have our view defined, let's define the controller.

## The Controller Implementation

Example 18-3 shows the *Controller* interface in our *mvc* package. The *Controller* interface defines the basic services every *Controller* class must offer, namely:

- Methods to set and retrieve the view reference
- Methods to set and retrieve the model reference

In the *Controller* interface, references to the view must be instances of any class that implements the *View* interface. References to the model must be instances of any class that extends *Observable*.

Example 18-3. The *Controller* interface

```
import util.*;
import mvc.*;

/**
 * Specifies the minimum services that the controller of
 * a Model-View-Controller triad must provide.
 */
interface mvc.Controller {
    /**
     * Sets the model for this controller.
     */
    public function setModel (m:Observable):Void;

    /**
     * Returns the model for this controller.
     */
    public function getModel ():Observable;

    /**
     * Sets the view this controller is servicing.
     */
    public function setView (v:View):Void;

    /**
     * Returns this controller's view.
     */
    public function getView ():View;
}
```

To make controller classes easy to create, we provide *AbstractController*, a class that implements the basic services defined by the *Controller* interface to manage interactions with the view and model. To create a real controller class in an MVC application, we need to extend *AbstractController* and add input-handling code. The core MVC grunt work in any real controller is taken care of by *AbstractController*.

Example 18-4 shows the code for the *AbstractController* class. Typically, subclasses of *AbstractController* implement input event-handling methods that translate user input received from the view into model modification notices sent to the model.

*Example 18-4. The AbstractController class*

```
import util.*;
import mvc.*;

/**
 * Provides basic services for the controller of
 * a Model-View-Controller triad.
 */
class mvc.AbstractController implements Controller {
    private var model:Observable; // A reference to the model.
    private var view:View;       // A reference to the view.

    /**
     * Constructor
     *
     * @param m The model this controller's view is observing.
     */
    public function AbstractController (m:Observable) {
        // Set the model.
        setModel(m);
    }

    /**
     * Sets the model for this controller.
     */
    public function setModel (m:Observable):Void {
        model = m;
    }

    /**
     * Returns the model for this controller.
     */
    public function getModel ():Observable {
        return model;
    }

    /**
     * Sets the view that this controller is servicing.
     */
    public function setView (v:View):Void {
        view = v;
    }
}
```

Example 18-4. The *AbstractController* class (continued)

```
/**
 * Returns this controller's view.
 */
public function getView ():View {
    return view;
}
}
```

## An MVC Clock

With our core MVC structure in place, we can create a real-world MVC application—a clock with analog and digital displays. The default MVC framework and source files for the clock are available at <http://moock.org/eas2/examples>.

Figure 18-3 shows the graphical user interface for the clock. Notice that in addition to the analog and digital displays, the clock includes buttons to start, stop, and reset the clock's time.

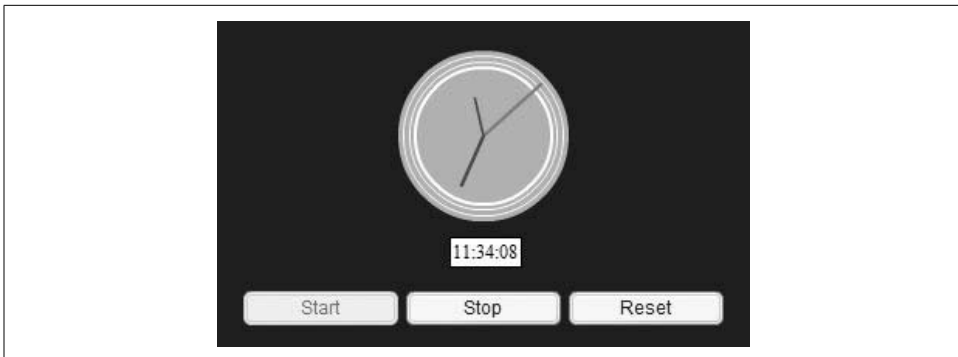


Figure 18-3. The Clock graphical user interface

Our clock application includes seven classes, as follows:

### *Clock*

The main application class, which creates the MVC clock

### *ClockModel*

The model class, which tracks the clock's time

### *ClockUpdate*

An info object class that stores update data sent by *ClockModel* to all views

### *ClockAnalogView*

A view class that presents the analog clock display

### *ClockDigitalView*

A view class that presents the digital clock display

### *ClockTools*

A view class that presents the Start, Stop, and Reset buttons

### *ClockController*

A controller class that handles button input

Figure 18-4 shows how our application's classes integrate into the core MVC structure we built earlier in this chapter.

In Figure 18-4, note that *ClockUpdate* instances are info objects sent by the *ClockModel* to its views (*ClockAnalogView*, *ClockDigitalView*, and *ClockTools*) at update time. The *View* interface specifies operations for registering the model (*ClockModel*) and controller (*ClockController*), while the *Observer* interface specifies the operation used by the model (*ClockModel*) to send updates to its views (again, *ClockAnalogView*, *ClockDigitalView*, and *ClockTools*). The *ClockAnalogView* and *ClockDigitalView* classes use null for their controller while the *ClockTools* class uses a *ClockController* instance for its controller. Finally, Figure 18-4 does not show the *Clock* class, which creates the clock using the *ClockModel* class and its views.

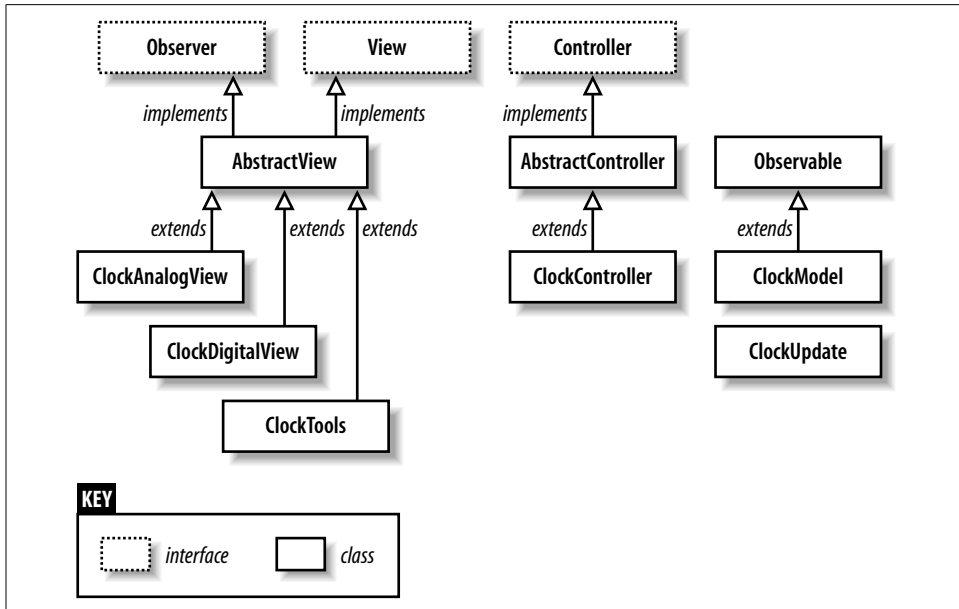


Figure 18-4. Clock application architecture

We've already looked at the supporting classes and interfaces in the *mvc* and *util* packages. The classes specific to the clock application are all stored in the package *mvcclock*. Let's look at them one at a time.

## The ClockModel Class

The *ClockModel* class extends *Observable*. It provides methods for setting the time and runs an internal ticker that updates the time once per second. Whenever the time changes, the *ClockModel* class uses *notifyObservers()* to broadcast the change to all registered views.

Here are the properties defined by the *ClockModel* class. Remember that the model class's properties represent its current state.

*hour*

The current hour, from 0 (midnight) to 23 (11 p.m.)

*minute*

The current minute, from 0 to 59

*second*

The current second, from 0 to 59

*isRunning*

A Boolean indicating whether the clock's internal ticker is running or not

Here are the public methods of the *ClockModel* class. These methods are used by the *ClockController* class and the *Clock* class to manipulate the clock's data:

*setTime()*

Sets the clock's hour, minute, and second, then notifies views if appropriate

*start()*

Starts the clock's internal ticker and notifies the views

*stop()*

Stops the clock's internal ticker and notifies the views

Notice that the *ClockModel* class does not define a *reset()* method, despite the fact that the clock's user interface has a Reset button. Instead, the *ClockController* class defines a *resetClock()* method, used to set the time to midnight. This functionality does not belong in the *ClockModel* class because it is not a logical requirement of the clock's basic operation. That is, the clock provides a means of setting the time, starting it, and stopping it—other classes can and should decide that “setting the time to midnight” constitutes a so-called “reset,” while setting the time to, say, 6 p.m. constitutes a *setToDinnerTime()* operation. These value judgments should be layered on top of the clock's core functionality, not incorporated into it.

Here are the private methods of the *ClockModel* class. These methods are used internally by the *ClockModel* class to update the time and to validate data:

*isValidHour()*

Checks whether a number is a valid hour (i.e., an integer from 0 to 23)

*isValidMinute()*

Checks whether a number is a valid minute (i.e., is an integer from 0 to 59)

*isValidSecond()*

Checks whether a number is a valid second (i.e., is an integer from 0 to 59)

*tick()*

Increments the second property by 1

The range-checking functions are properly implemented as private methods of the model class so that all updates from all controllers use the same data validation. Thus, we minimize duplicate code, standardize the notion of valid clock data, and simplify maintenance.

Note that although the clock class itself determines that, say, 25 is not a legal hour, it is up to the controller classes to format the time into 24-hour format before sending change requests to the model. Each controller must take responsibility for formatting the data according to the requirements of the model. For example, if a view accepts text input, the controller is responsible for converting the text string into a number before sending the time update to the model.

To register views and notify them when the time changes, the *ClockModel* class relies on its superclass's methods: *addObserver()*, *removeObserver()*, and *notifyObservers()*.

Example 18-5 shows the code for the *ClockModel* class. In particular, note the *setTime()* method's use of *setChanged()* to indicate whether an update should be broadcast when the time is set. If *setTime()* is called but the new time specified is the same as the existing time, then no update is sent. An update is sent only if the time has actually changed. (In this case, the time must change by at least one second. Implementing a version that issues updates more frequently, such as every tenth of a second, is left as an exercise for the reader. Naturally, you wouldn't bother unless you were implementing a digital display that displayed the fractional seconds or some other view that relied on the greater resolution.) Updates take the form of a *ClockUpdate* object passed to each view's *update()* method. The *setChanged()* method is inherited from the *Observable* class, discussed in Chapter 16.

*Example 18-5. The ClockModel class*

```
import util.Observable;
import mvcclock.*;

/**
 * Represents the data of a clock (i.e., the model of the
 * Model-View-Controller triad).
 */
class mvcclock.ClockModel extends Observable {
    // The current hour.
    private var hour:Number;
    // The current minute.
    private var minute:Number;
    // The current second.
    private var second:Number;
    // The interval identifier for the interval that
```

Example 18-5. The `ClockModel` class (continued)

```
// calls tick() once per second.
private var tickInterval:Number;
// Indicates whether the clock is running or not.
private var isRunning:Boolean;

/**
 * Constructor
 */
public function ClockModel () {
    // By default, set the clock time to the current system time.
    var now:Date = new Date();
    setTime(now.getHours(), now.getMinutes(), now.getSeconds());
}

/**
 * Starts the clock ticking.
 */
public function start ():Void {
    if (!isRunning) {
        isRunning = true;
        tickInterval = setInterval(this, "tick", 1000);

        var infoObj:ClockUpdate = new ClockUpdate(hour, minute,
                                                    second, isRunning);
        setChanged();
        notifyObservers(infoObj);
    }
}

/**
 * Stops the clock ticking.
 */
public function stop ():Void {
    if (isRunning) {
        isRunning = false;
        clearInterval(tickInterval);

        var infoObj:ClockUpdate = new ClockUpdate(hour, minute,
                                                    second, isRunning);
        setChanged();
        notifyObservers(infoObj);
    }
}

/**
 * Sets the current time (i.e., the hour, minute, and second properties).
 * Notifies observers (views) of any change in time.
 *
 * @param h The new hour.
 * @param m The new minute.
 * @param s The new second.
 */
```

*Example 18-5. The ClockModel class (continued)*

```
public function setTime (h:Number, m:Number, s:Number):Void {
    if (h != null && h != hour && isValidHour(h)) {
        hour = h;
        setChanged();
    }

    if (m != null && m != minute && isValidMinute(m)) {
        minute = m;
        setChanged();
    }

    if (s != null && s != second && isValidSecond(s)) {
        second = s;
        setChanged();
    }

    // If the model has changed, notify views.
    if (hasChanged()) {
        var infoObj:ClockUpdate = new ClockUpdate(hour, minute,
                                                    second, isRunning);

        // Push the changed data to the views.
        notifyObservers(infoObj);
    }
}

/**
 * Checks to see if a number is a valid hour
 * (i.e., an integer in the range 0 to 23).
 *
 * @param h The hour to check.
 */
private function isValidHour (h:Number):Boolean {
    return (Math.floor(h) == h && h >= 0 && h <= 23);
}

/**
 * Checks to see if a number is a valid minute
 * (i.e., an integer in the range 0 to 59).
 *
 * @param m The minute to check.
 */
private function isValidMinute (m:Number):Boolean {
    return (Math.floor(m) == m && m >= 0 && m <= 59);
}

/**
 * Checks to see if a number is a valid second
 * (i.e., an integer in the range 0 to 59).
 *
 * @param s The second to check.
 */
private function isValidSecond (s:Number):Boolean {
    return (Math.floor(s) == s && s >= 0 && s <= 59);
}
```

Example 18-5. The *ClockModel* class (continued)

```
}

/**
 * Makes time pass by adding a second to the current time.
 */
private function tick ():Void {
    // Get the current time.
    var h:Number = hour;
    var m:Number = minute;
    var s:Number = second;

    // Increment the current second, adjusting
    // the minute and hour if necessary.
    s += 1;
    if (s > 59) {
        s = 0;
        m += 1;
        if (m > 59) {
            m = 0;
            h += 1;
            if (h > 23) {
                h = 0;
            }
        }
    }

    // Set the new time.
    setTime(h, m, s);
}
}
```

Notice that our *ClockModel* class defines methods for setting the current time but does not define any methods for retrieving it. That’s because the clock’s time is pushed to all views via a *ClockUpdate* object; views do not need to query the state of the model in our example. However, we could just as sensibly have required views to retrieve the new time from the model. Typically, the push system (in this case, sending the time) is used if pulling information from the model (in this case, retrieving the time) is too processor intensive. That limitation doesn’t apply here, so both the push and pull systems are appropriate.

## The *ClockUpdate* Class

The *ClockUpdate* class is a simple data holder used to transfer the state of the *ClockModel* to views when the time changes or the clock is started or stopped. It defines four properties—hour, minute, second, and *isRunning*—which are accessed directly. To avoid distracting from the issues at hand, we won’t implement accessor methods such as *getMinute()* for these properties. The *ClockUpdate* class is simply a data vessel, used like an associative array or a hash, so the direct property access is arguably appropriate.

Example 18-6 shows the code for the *ClockUpdate* class.

*Example 18-6. The ClockUpdate class*

```
/**
 * An info object sent by the ClockModel class to
 * its views when an update occurs. Indicates the
 * time and whether or not the clock is running.
 */
class mvcclock.ClockUpdate {
    public var hour:Number;
    public var minute:Number;
    public var second:Number;
    public var isRunning:Boolean;

    public function ClockUpdate (h:Number, m:Number, s:Number, r:Boolean) {
        hour = h;
        minute = m;
        second = s;
        isRunning = r;
    }
}
```

## The ClockAnalogView and ClockDigitalView Classes

The *ClockAnalogView* and *ClockDigitalView* classes extend *AbstractView*, which implements the *Observer* and *View* interfaces. They are display-only views; that is, the user interface they create has no inputs and therefore needs no controller. Accordingly, neither *ClockAnalogView* nor *ClockDigitalView* overrides the *AbstractView.defaultController()* method. Both classes simply use `null` as their controller, as returned by the *AbstractView.defaultController()* method. If you were to make the clock hands on the analog display draggable, you would implement an appropriate controller class to notify the model of the specified input obtained via the view's GUI. Likewise, if the hours, minutes, and seconds of the digital clock were editable, you would implement an appropriate controller for that view as well.

The *ClockAnalogView* class is responsible for rendering the clock as a traditional circle with moving hands. The *ClockDigitalView* class is responsible for rendering the clock as a numeric display. Both classes have the same basic structure, which includes the following two public methods:

*makeClock()*

Creates the visual display of the clock at construction time

*update()*

Handles updates from the *ClockModel* by setting the appropriate time on screen

In addition to those two methods, the *ClockDigitalView* class defines two private utility methods—*formatTime12()* and *formatTime24()*—for adjusting the format of the time to 12-hour or 24-hour display (the analog view always displays 12-hour

time). Each view is responsible for translating raw information provided by the model into some particular interface representation. Hence, the time-formatting methods belong in the *ClockDigitalView* class, not in a hypothetical controller for that class (whose job would be to process input) nor in the model class (whose job is to manage raw data, not to transform that data for the needs of some particular rendering). That said, formatting time is a particularly common operation that might be required throughout an application. We might, therefore, have alternatively chosen to implement the time-formatting methods in a general utility class, say *DateFormat*. Indeed, date- and time-formatting functionality is built directly into many languages, but is not built into *ActionScript*.

The *ClockDigitalView* class creates its user interface entirely via code in the *makeClock()* method (there are no author-time movie clips). The *ClockAnalogView* class, by contrast, creates its user interface by attaching an instance of a movie clip symbol (*ClockAnalogViewSymbol*) from the Flash document's Library. We won't cover the creation of that symbol in detail here except to say that it contains a circle shape and three movie clips that represent the hands of the clock: *hourHand\_mc*, *minuteHand\_mc*, and *secondHand\_mc*. To see the *ClockAnalogViewSymbol*, download the sample files from <http://moock.org/eas2/examples>. The analog clock is an excellent demonstration of Flash's unique ability to combine hand-drawn graphics with OOP code. Readers of *ActionScript for Flash MX: The Definitive Guide* may recognize some of the code in Example 18-7 from the analog clock developed in Example 13-7 of that book, which is posted at <http://moock.org/asdg/codedepot>. Recipe 10.8 in the *ActionScript Cookbook*, by Joey Lott (O'Reilly), also implements an analog clock created entirely on-the-fly.

Example 18-7 shows the code for the *ClockAnalogView* class, which implements the analog display of the time.

*Example 18-7. The ClockAnalogView class*

```
import util.*;
import mvcclock.*;
import mvc.*;

/**
 * An analog clock view for the ClockModel class. This view has no user
 * inputs, so no controller is required.
 */
class mvcclock.ClockAnalogView extends AbstractView {
    // Contains an instance of ClockAnalogViewSymbol, which
    // depicts the clock on screen.
    private var clock_mc:MovieClip;

    /**
     * Constructor
     */
    public function ClockAnalogView (m:Observable, c:Controller,
```

Example 18-7. The *ClockAnalogView* class (continued)

```
        target:MovieClip, depth:Number,
        x:Number, y:Number) {
    // Invoke superconstructor, which sets up MVC relationships.
    // This view has no user inputs, so no controller is required.
    super(m, c);

    // Create UI.
    makeClock(target, depth, x, y);
}

/**
 * Creates the movie clip instance that will display the
 * time in analog format.
 *
 * @param target The clip in which to create the movie clip.
 * @param depth The depth at which to create the movie clip.
 * @param x The movie clip's horizontal position in target.
 * @param y The movie clip's vertical position in target.
 */
public function makeClock (target:MovieClip, depth:Number,
        x:Number, y:Number):Void {
    clock_mc = target.attachMovie("ClockAnalogViewSymbol",
        "analogClock_mc", depth);
    clock_mc._x = x;
    clock_mc._y = y;
}

/**
 * Updates the state of the on-screen analog clock.
 * Invoked automatically by ClockModel.notifyObservers().
 *
 * @param o The ClockModel object that is broadcasting an update.
 * @param infoObj A ClockUpdate instance describing the changes that
 * have occurred in ClockModel.
 */
public function update (o:Observable, infoObj:Object):Void {
    // Cast the generic infoObj to the ClockUpdate datatype.
    var info:ClockUpdate = ClockUpdate(infoObj);

    // Display the new time.
    var dayPercent:Number = (info.hour>12 ? info.hour-12 : info.hour) / 12;
    var hourPercent:Number = info.minute/60;
    var minutePercent:Number = info.second/60;
    clock_mc.hourHand_mc._rotation = 360 * dayPercent
        + hourPercent * (360 / 12);
    clock_mc.minuteHand_mc._rotation = 360 * hourPercent;
    clock_mc.secondHand_mc._rotation = 360 * minutePercent;

    // Dim the display if the clock isn't running.
    if (info.isRunning) {
        clock_mc._alpha = 100;
    } else {
```

Example 18-7. The *ClockAnalogView* class (continued)

```
        clock_mc._alpha = 50;
    }
}
}
```

Example 18-8 shows the code for the *ClockDigitalView* class, which implements the digital display of the time. In *ClockDigitalView*, for the sake of contrast, we create the clock interface entirely in code. Specifically, *ClockDigitalView.makeClock()* creates a text field in which we display the time. If desired, we could also have made the earlier *ClockAnalogView* entirely in code instead of using the authoring tool approach, in which we create the clock as a movie clip (*ClockAnalogViewSymbol*). Both approaches are valid Flash development practices. You might choose to draw your interface manually in the Flash authoring tool to save time (if the code equivalent would be complex) or to allow the interface to be redesigned by a nonprogrammer.

Example 18-8. The *ClockDigitalView* class

```
import util.*;
import mvcclock.*;
import mvc.*;

/**
 * A digital clock view for ClockModel. This view has no user
 * inputs, so no controller is required.
 */
class mvcclock.ClockDigitalView extends AbstractView {
    // The hour format.
    private var hourFormat:Number = 24;
    // The separator character in the clock display.
    private var separator:String = ":";
    // The text field in which to display the clock, created by makeClock().
    private var clock_txt:TextField;

    /**
     * Constructor
     */
    public function ClockDigitalView (m:Observable, c:Controller,
                                     hf:Number, sep:String, target:MovieClip,
                                     depth:Number, x:Number, y:Number) {
        // Invoke superconstructor, which sets up MVC relationships.
        super(m, c);

        // Make sure the hour format specified is legal. If it is, use it.
        if (hf == 12) {
            hourFormat = 12;
        }

        // If a separator was provided, use it.
        if (sep != undefined) {
            separator = sep;
        }
    }
}
```

Example 18-8. The *ClockDigitalView* class (continued)

```
// Create UI.
makeClock(target, depth, x, y);
}

/**
 * Creates the onscreen text field that will display the
 * time in digital format.
 *
 * @param target The clip in which to create the text field.
 * @param depth The depth at which to create the text field.
 * @param x The text field's horizontal position in target.
 * @param y The text field's vertical position in target.
 */
public function makeClock (target:MovieClip, depth:Number,
                          x:Number, y:Number):Void {
    // Make the text field.
    target.createTextField("clock_txt", depth, x, y, 0, 0);
    // Store a reference to the text field.
    clock_txt = target.clock_txt;
    // Assign text field characteristics.
    clock_txt.autoSize = "left";
    clock_txt.border = true;
    clock_txt.background = true;
}

/**
 * Updates the state of the on-screen digital clock.
 * Invoked automatically by ClockModel.
 *
 * @param o The ClockModel object that is broadcasting an update.
 * @param infoObj A ClockUpdate instance describing the changes that
 *               have occurred in ClockModel.
 */
public function update (o:Observable, infoObj:Object):Void {
    // Cast the generic infoObj to the ClockUpdate datatype.
    var info:ClockUpdate = ClockUpdate(infoObj);

    // Create a string representing the time in the appropriate format.
    var timeString:String = (hourFormat == 12)
        ?
        formatTime12(info.hour, info.minute, info.second)
        :
        formatTime24(info.hour, info.minute, info.second);

    // Display the new time in the clock text field.
    clock_txt.text = timeString;

    // Dim the color of the display if the clock isn't running.
    if (info.isRunning) {
        clock_txt.textColor = 0x000000;
    }
}
```

Example 18-8. The `ClockDigitalView` class (continued)

```
    } else {
        clock_txt.textColor = 0x666666;
    }
}

/**
 * Returns a formatted 24-hour time string.
 *
 * @param h The hour, from 0 to 23.
 * @param m The minute, from 0 to 59.
 * @param s The second, from 0 to 59.
 */
private function formatTime24 (h:Number, m:Number, s:Number):String {
    var timeString:String = "";

    // Format hours...
    if (h < 10) {
        timeString += "0";
    }
    timeString += h + separator;

    // Format minutes...
    if (m < 10) {
        timeString += "0";
    }
    timeString += m + separator;

    // Format seconds...
    if (s < 10) {
        timeString += "0";
    }
    timeString += String(s);

    return timeString;
}

/**
 * Returns a formatted 12-hour time string (not including AM or PM).
 *
 * @param h The hour, from 0 to 23.
 * @param m The minute, from 0 to 59.
 * @param s The second, from 0 to 59.
 */
private function formatTime12 (h:Number, m:Number, s:Number):String {
    var timeString:String = "";

    // Format hours...
    if (h == 0) {
        timeString += "12" + separator;
    } else if (h > 12) {
        timeString += (h - 12) + separator;
    } else {
```

Example 18-8. The *ClockDigitalView* class (continued)

```
        timeString += h + separator;
    }

    // Format minutes...
    if (m < 10) {
        timeString += "0";
    }
    timeString += m + separator;

    // Format seconds...
    if (s < 10) {
        timeString += "0";
    }
    timeString += String(s);

    return timeString;
}
}
```

We've now seen two view classes that do not accept user input and therefore have no controller. Next, we'll consider a view class, *ClockTools*, which accepts user input and shows how to process that input with a controller.

## The *ClockTools* Class

Like the *ClockAnalogView* and *ClockDigitalView* classes, the *ClockTools* class is a view for *ClockModel*, so it extends *AbstractView*. The *ClockTools* class creates the Start, Stop, and Reset buttons below the analog and digital clock display, as shown in Figure 18-3. A *.fla* file that uses *ClockTools* must contain the Flash MX 2004 Button component in its Library and have it exported for ActionScript (see Chapter 12 for information on components).

Unlike *ClockAnalogView* and *ClockDigitalView*, the *ClockTools* view includes a functional controller to process user input. The general structure of *ClockTools* follows the structure of *ClockAnalogView* and *ClockDigitalView*—*ClockTools* has a *makeTools()* method that creates the user interface and an *update()* method that makes changes to that interface based on *ClockModel* updates. However, the *makeTools()* method does more than just render the user interface; it also creates the all-important connection from that interface to the controller (and, indeed, creates the controller itself).

Because *ClockTools* has a functional controller, it also overrides the *AbstractView.defaultController()* method. The *ClockTools.defaultController()* method returns an instance of *ClockController*, the default controller for the *ClockTools* view.

Example 18-9 shows the code for the *ClockTools* class. Read it through, then we'll study the important *makeTools()* and *update()* methods in detail.

Example 18-9. The *ClockTools* class

```
import util.*;
import mvcclock.*;
import mvc.*;
import mx.controls.Button;

/**
 * Creates a user interface that can control a ClockModel.
 */
class mvcclock.ClockTools extends AbstractView {
    private var startBtn:Button;
    private var stopBtn:Button;
    private var resetBtn:Button;

    /**
     * Constructor
     */
    public function ClockTools (m:Observable, c:Controller,
                               target:MovieClip, depth:Number,
                               x:Number, y:Number) {
        // Invoke superconstructor, which sets up MVC relationships.
        super(m, c);

        // Create UI.
        makeTools(target, depth, x, y);
    }

    /**
     * Returns the default controller for this view.
     */
    public function defaultController (model:Observable):Controller {
        return new ClockController(model);
    }

    /**
     * Creates a movie clip instance to hold the Start, Stop,
     * and Reset buttons and also creates those buttons.
     *
     * @param target The clip in which to create the tools_mc clip.
     * @param depth The depth at which to create the tools_mc clip.
     * @param x The tools clip's horizontal position in target.
     * @param y The tools clip's vertical position in target.
     */
    public function makeTools (target:MovieClip, depth:Number,
                               x:Number, y:Number):Void {
        // Create a container movie clip.
        var tools_mc:MovieClip = target.createEmptyMovieClip("tools", depth);
        tools_mc._x = x;
        tools_mc._y = y;

        // Create UI buttons in the container clip.
        startBtn = tools_mc.createClassObject(Button, "start", 0);
        startBtn.label = "Start";
    }
}
```

Example 18-9. The *ClockTools* class (continued)

```
startBtn.enabled = false;
startBtn.addEventListener("click", getController());

stopBtn = tools_mc.createClassObject(Button, "stop", 1);
stopBtn.label = "Stop";
stopBtn.enabled = false;
stopBtn.move(startBtn.width + 5, startBtn.y);
stopBtn.addEventListener("click", getController());

resetBtn = tools_mc.createClassObject(Button, "reset", 2);
resetBtn.label = "Reset";
resetBtn.move(stopBtn.x + stopBtn.width + 5, startBtn.y);
resetBtn.addEventListener("click", getController());
}

/**
 * Updates the state of the user interface.
 * Invoked automatically by ClockModel.
 *
 * @param o The ClockModel object that is broadcasting an update.
 * @param infoObj A ClockUpdate instance describing the changes that
 * have occurred in ClockModel.
 */
public function update (o:Observable, infoObj:Object):Void {
    // Cast the generic infoObj to the ClockUpdate datatype.
    var info:ClockUpdate = ClockUpdate(infoObj);

    // Enable the Start button if the clock is stopped, or
    // enable the Stop button if the clock is running.
    if (info.isRunning) {
        stopBtn.enabled = true;
        startBtn.enabled = false;
    } else {
        stopBtn.enabled = false;
        startBtn.enabled = true;
    }
}
}
```

The *ClockTools.makeTools()* method creates three Button component instances: one that starts the clock, one that stops it, and one that resets it. The *click* event of each Button instance is handled by the *ClockTools* controller, *ClockController*. Let's examine one button, Start, to see how it is wired to the *ClockController*.

First, we create an empty container movie clip, *tools\_mc*, in which to store all three buttons. The *tools\_mc* clip is attached to *target*, a movie clip instance specified when the *ClockTools* view is instantiated.

```
var tools_mc:MovieClip = target.createEmptyMovieClip("tools", depth);
```

Next, we create the Start button instance inside *tools\_mc*, using *createClassObject()* (which we studied in Chapter 12). We store the Start button instance in the property

startBtn so that we can access it later when it's time to update the interface. The instance name of the button, "start", will be used later in the *ClockController* class to uniquely identify the button:

```
startBtn = tools_mc.createClassObject(Button, "start", 0);
```

Now we add the text on the button (i.e., its *label*) and disable the button. By default, the clock will be running, so the Start button should be disabled at the outset:

```
startBtn.label = "Start";  
startBtn.enabled = false;
```

Finally, we make the crucial connection between the view and the controller by specifying the controller as the listener object for the Start button's *click* event. This implies (indeed, demands) that the controller defines a method named *click()* which will be invoked when the Start button is clicked.

```
startBtn.addEventListener("click", getController());
```

Notice that the *makeTools()* method does not refer to the *ClockController* class directly. It retrieves the controller instance via *getController()*. If a controller already exists, it is returned; if not, an instance of the default controller is created and returned. By specifying and accessing the controller indirectly via *getController()*, we maintain the loose coupling between the view and the controller, giving us the flexibility to change the controller at runtime (or to rewrite a new implementation in the future with little disturbance of existing code). For example, at any point in the program, we could use *setController(SomeOtherController)* to completely replace the current controller, which in turn would change the interface's response to user input. If, on the other hand, we had hardcoded the controller reference, as follows:

```
startBtn.addEventListener("click", new ClockController());
```

we would not be able to change the controller at runtime.

The *ClockTools.update()* method, unlike *ClockAnalogView.update()* and *ClockDigitalView.update()*, does not simply display the current time. Instead, it disables the Start or Stop button depending on whether the clock is running. If the clock is running, only the Stop and Reset buttons are enabled. If the clock is stopped, only the Start and Reset buttons are enabled:

```
if (info.isRunning) {  
    stopBtn.enabled = true;  
    startBtn.enabled = false;  
} else {  
    stopBtn.enabled = false;  
    startBtn.enabled = true;  
}
```

It's a common misconception that a view is always a literal representation of the model's data. While *ClockAnalogView* and *ClockDigitalView* create literal representations of the *ClockModel*, the *ClockTools* view does not. In general, then, a view depicts a user interface whose display depends on the state of the model but isn't

necessarily a direct visualization of the model. Here, the button states depend on whether the clock is running, not on the current time. That is, this view depicts the model's `isRunning` property, which is equally as valid as the other views depicting the hour, minute, and second properties.

Now let's move on to the *ClockController* class, which handles input for *ClockTools*. Again, the *ClockAnalogView* and *ClockDigitalView* views use the default controller created by *AbstractView*, but our *ClockTools* class creates its own controller.

## The ClockController Class

The *ClockController* class extends *AbstractController*. It provides event handling for the user interface created by the *ClockTools* view.

To change the state of the *ClockModel*, the *ClockController* class defines the following methods:

```
startClock()  
stopClock()  
resetClock()  
setTime()  
setHour()  
setMinute()  
setSecond()
```

Notice that many of the preceding methods are wrappers over methods in the *ClockModel* class. Some of them act as direct pass-through methods (e.g., *startClock()* and *stopClock()*), whereas others add some convenience functionality (e.g., the ability to independently set a single aspect of the time, such as the hour, minute, or second).

But the real duty of the *ClockController* class is fulfilled by its *click()* method, which handles events from the buttons created by the *ClockTools* class. Before we look at the full class listing for *ClockController*, let's take a detailed look at the *click()* method:

```
public function click (e:Object):Void {  
    switch (e.target._name) {  
        case "start":  
            startClock();  
            break;  
        case "stop":  
            stopClock();  
            break;  
        case "reset":  
            resetClock();  
            break;  
    }  
}
```

The *click()* method is set up like any component-event-handling method. It accepts a single argument, *e*, which contains details about the event as well as a very important reference back to the component that generated the event. In the case of the *click()* method, the property *e.target* gives us a reference back to the Button component that generated the *click()* event. The property *e.target.\_name* tells us the instance name of that button—either “start”, “stop”, or “reset”. (Recall that we set each button’s instance name when we created it in the *ClockTools.makeTools()* method.) Depending on that instance name, we invoke the appropriate method, either *startClock()*, *stopClock()* or *resetClock()*.

Note that the *click()* handler is implemented in the controller rather than the view. Some alternative MVC variations implement event handlers in the view and then pass input to the controller, but in our implementation, we emphasize the separation between the view’s “interface rendering” role and the controller’s “input processing” role. Notice also that a single view and a single controller implement the event handling for all three buttons. That is, the *click()* handler distinguishes between buttons using the event object, *e*, passed to it. In this case, it would be overkill to implement separate views and controllers for each of the three buttons, but it wouldn’t be insane to do so. MVC can be as nested and granular as the program needs it to be. For example, in a ComboBox component (i.e., a pull-down menu), each individual item in the ComboBox might implement MVC while the entire ComboBox also implements MVC.

Finally, notice that the view is a listener in the model’s list of registered listeners (subscribing to the *update* event) and the controller is a listener in the view’s list of registered listeners (subscribing to the *click* event). In our example, the model doesn’t subscribe to any events. Its methods are invoked manually by *ClockController* and internally by *setInterval()*.

As an example, here’s the event sequence for the Reset button:

1. User clicks on the Reset button, generating a *click* event.
2. *ClockController.click()* receives and interprets the event, eventually calling *ClockModel.setTime()* with zeros for the hour, minute, and second.
3. *ClockModel* receives the command from the controller and changes the time accordingly.
4. The *ClockModel.notifyObservers()* method broadcasts an *update* event (containing a *ClockUpdate* info object) to all registered views.
5. All three views—*ClockAnalogView*, *ClockDigitalView*, and *ClockTools*—receive the *update* event and refresh their user interface elements accordingly.
  - a. *ClockAnalogView* resets both hands to 12 o’clock.
  - b. *ClockDigitalView* resets the digital display to “00:00:00”.
  - c. *ClockTools* makes sure all its buttons are correct (in this case, the Stop and Start button states would depend on whether the clock was running when it was reset).

The sequence would be similar for the Start and Stop buttons, except that the views would update differently according to the *ClockUpdate* object representing the model's state. Notice that none of the views updates its display until notified by the model. But it all happens so quickly that the analog and digital displays appear to reset instantly when the Reset button is clicked.

Now let's look at the *click()* method in the context of the complete *ClockController* class, shown in Example 18-10.

*Example 18-10. The ClockController class*

```
import mvcclock.ClockModel;
import mvc.*;
import util.*;

/**
 * Makes changes to ClockModel's data based on user input.
 * Provides general services that any view might find handy.
 */
class mvcclock.ClockController extends AbstractController {

    /**
     * Constructor
     *
     * @param cm The model to modify.
     */
    public function ClockController (cm:Observable) {
        super(cm);
    }

    /**
     * Starts the clock ticking.
     */
    public function startClock ():Void {
        ClockModel(getModel()).start();
    }

    /**
     * Stops the clock ticking.
     */
    public function stopClock ():Void {
        ClockModel(getModel()).stop();
    }

    /**
     * Resets the clock's time to 12 midnight (0 hours).
     */
    public function resetClock ():Void {
        setTime(0, 0, 0);
    }

    /**
     * Changes the clock's time.
     */
}
```

Example 18-10. The *ClockController* class (continued)

```
*
* @param h The new hour, from 0 to 23.
* @param m The new minute, from 0 to 59.
* @param s The new second, from 0 to 59.
*/
public function setTime (h:Number, m:Number, s:Number):Void {
    ClockModel(getModel()).setTime(h, m, s);
}

// As these next three methods show, the controller can provide
// convenience methods to change data in the model.

/**
 * Sets just the clock's hour.
 *
 * @param h The new hour.
 */
public function setHour (h:Number):Void {
    ClockModel(getModel()).setTime(h, null, null);
}

/**
 * Sets just the clock's minute.
 *
 * @param m The new minute.
 */
public function setMinute (m:Number):Void {
    ClockModel(getModel()).setTime(null, m, null);
}

/**
 * Sets just the clock's second.
 *
 * @param s The new second.
 */
public function setSecond (s:Number):Void {
    ClockModel(getModel()).setTime(null, null, s);
}

/**
 * Handles events from the Start, Stop, and Reset buttons
 * of the ClockTools view.
 */
public function click (e:Object):Void {
    switch (e.target._name) {
        case "start":
            startClock();
            break;
        case "stop":
            stopClock();
            break;
        case "reset":
```

Example 18-10. The *ClockController* class (continued)

```
        resetClock();
        break;
    }
}
```

The *ClockController* class is fairly generic. It could, in theory, be extended to provide event handling for other clock-related classes as well. For example, if the *ClockAnalogView* class allowed its hands to be dragged, methods could be added to *ClockController* to handle the clock-hand drag events. In that case, each view would have its own instance of *ClockController*. Alternatively, separate controller classes could be created to handle the drag events. Or, the *ClockTools* class and the *ClockAnalogView* class could use different controllers that inherit from a single *ClockController* superclass, which would define a set of methods common to both classes.

## Putting It All Together: The Clock Class

The pieces of our MVC clock are now complete. All that's left to do is assemble them into a functional application. The final assembly of our application occurs in the *Clock* class, our primary class that performs the following tasks:

- Creates the *ClockModel* instance
- Creates the clock views (*ClockAnalogView*, *ClockDigitalView*, *ClockTools*)
- Registers the clock views to receive updates from the *ClockModel*
- Optionally sets the clock's time
- Starts the clock ticking

All the preceding tasks occur in the *Clock* class's constructor function. However, because the clock needs to run as a standalone application, the *Clock* class also defines a *main()* method that starts the clock application. For information on creating and using a *main()* method as an application entry point, see Chapter 11.

Example 18-11 shows the code for the *Clock* class.

Example 18-11. The *Clock* class

```
import mvcclock.*

/**
 * An example Model-View-Controller (MVC) clock application.
 */
class mvcclock.Clock {
    // The clock data (i.e., the model).
    private var clock_model:ClockModel;

    // Two different displays of the clock's data (i.e., views).
    private var clock_analogview:ClockAnalogView;
```

*Example 18-11. The Clock class (continued)*

```
private var clock_digitalview:ClockDigitalView;

// A toolbar for controlling the clock (also a view).
private var clock_tools:ClockTools;

/**
 * Clock constructor
 *
 * @param target The movie clip to which the digital and
 *              analog views will be attached.
 * @param h The initial hour, 0 to 23, at which to set the clock.
 * @param m The initial minute, 0 to 59, at which to set the clock.
 * @param s The initial second, 0 to 59, at which to set the clock.
 */
public function Clock (target:MovieClip, h:Number, m:Number, s:Number) {
    // Create the data model.
    clock_model = new ClockModel();

    // Create the digital clock view, which uses a default controller.
    clock_digitalview = new ClockDigitalView(clock_model, undefined,
                                             24, ":", target, 0, 253, 265);
    clock_model.addObserver(clock_digitalview);

    // Create the analog clock view, which uses a default controller.
    clock_analogview = new ClockAnalogView(clock_model, undefined,
                                             target, 1, 275, 200);
    clock_model.addObserver(clock_analogview);

    // Create the clock tools view, which creates its own controller.
    clock_tools = new ClockTools(clock_model, undefined, target,
                                  2, 120, 300);
    clock_model.addObserver(clock_tools);

    // Set the time.
    clock_model.setTime(h, m, s);

    // Start the clock ticking.
    clock_model.start();
}

/**
 * System entry point. Starts the clock application running.
 */
public static function main (target:MovieClip, h:Number,
                             m:Number, s:Number) {
    var clock:Clock = new Clock(target, h, m, s);
}
}
```

To start our clock application in motion, we invoke `Clock.main()` on a frame of a `.fla` file (presumably the frame following any preloader). For example:

```
// Import the package containing the Clock class.
import mvcclock.Clock;
// Attach the clock to someClip. The time defaults to the system time.
Clock.main(someClip);
```

As mentioned earlier, the `.fla` file's Library must contain a movie clip symbol named `ClockAnalogViewSymbol`, and it must contain the Button UI component. (To add the Button component to the Library, drag an instance of it from the Components panel to the Stage, and then delete the instance from the Stage.)

In the preceding example, if we had chosen to set the time of the clock to 3:40:25 a.m., we'd have used:

```
Clock.main(someClip, 3, 40, 25);
```

In either case, a clip referenced by the variable or instance name "someClip" must exist on the timeline.

To create a clock set to 3:40:25 p.m. and attach it to the current timeline, we can use:

```
Clock.main(this, 15, 40, 25);
```

Alternatively, we could create an instance of the `Clock` class directly (skipping the `main()` method) like this:

```
var c:Clock = new Clock(this, 15, 40, 25);
```

We'd need to use that approach when creating a clock as part of a larger application. The `main()` method approach simply lets us use the clock as a standalone application.

## Further Exploration

Because our clock is an MVC application, it is incredibly flexible. New interfaces, input responses, and functionality can easily be added to the clock. If you're keen to experiment with MVC in ActionScript, try the following exercises:

- Create a view that can display a different time zone.
- Create a new view that displays a creative representation of the hours, minutes, and seconds of the clock—perhaps use shapes on screen to represent the time: circles for the hours, triangles for the minutes, and squares for the seconds.
- Create a new view that makes a ticking sound every second and, at the top of every hour, gongs to indicate the current time.
- Add components to the `ClockTools` view that let the user set the current time.

- Change the *ClockModel* so that it updates by polling the computer's system time instead of by counting milliseconds with *setInterval()*. Hint: you'll need the *Date()* class.
- Change the clock so it can display hundredths of a second, like a stopwatch.
- Let the user set the time by dragging the analog hands or editing the digital display.

If you'd like to continue reading about the MVC design pattern, see the following online articles:

*Steve Burbeck's canonical Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*

<http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>

*John Hunt's You've got the model-view-controller (excellent article with complete example source in Java)*

<http://www.jaydeetechnology.co.uk/planetjava/tutorials/swing/Model-View-Controller.PDF>

*Richard Baldwin's Implementing The Model-View-Controller Paradigm using Observer and Observable*

<http://www.dickbaldwin.com/java/Java200.htm>

*Sun's Java BluePrints: Model-View-Controller*

<http://java.sun.com/blueprints/patterns/MVC-detailed.html>

In the next (and final) chapter, we'll continue our coverage of interclass update mechanisms by studying the delegation event model.