The following is part one of chapter 4 of Ajax in Action from Manning Publications.

Ajax applications can contain much more client-side code than a standard web application, and hence benefit much more from the order that patterns and refactoring bring. Chapter 4 of Manning's Ajax in Action is the first of three chapters that apply refactoring and patterns to the client-side codebase. You won't see much of the asynchronous requests that give Ajax its name in this chapter, but the style of programming that we're discussing here is a direct consequence of being able to make asynchronous requests.

This chapter is about structuring Ajax applications. It provides examples of how the well-established Model-View-Controller pattern can be used to provide that structure. This first installment of Chapter 4 covers the display aspects of the application, and shows how normal JavaScript code can be refactored into a robust view component.

Chapter 4: The page as an application

In chapters 1 and 2 we covered the basic principles of Ajax, from both a usability and a technology perspective. In chapter 3 we touched on the notion of creating *maintainable* code through refactoring and design patterns. In the examples that we've looked at so far, this may have seemed like overkill, but, as we explore the subject of Ajax programming in more depth, they will prove themselves to be indispensable tools.

In this chapter and the next, we will discuss the details of building a larger, scalable Ajax client, and the architectural principles needed to make it work. This chapter looks at the coding of the client itself, drawing heavily upon the Model-View-Controller (MVC) pattern that we discussed in chapter 3. We'll also encounter the Observer and other smaller patterns along the way. Chapter 5 will look at the relationship between the client and the server.

4.1 A different kind of MVC

In chapter 3, we presented an example of refactoring a simple garment store application to conform to the MVC pattern. This is the context in which most web developers will have come across MVC before, with the Model being the domain model on the server, the View being the generated content sent to the client, and the Controller being a servlet or set of pages defining the workflow of the application.

However, MVC had its origins in desktop application development, and there are several other places in an Ajax application where it can serve us well too. Let's have a look at them now.

4.1.1 Repeating the pattern at different scales

The classic web MVC model describes the entire application in coarse-grained detail. The entire generated data stream is the View. The entire CGI or servlet layer is the Controller, and so on.

In desktop application development, MVC patterns are often applied at a much finer scale, too. Something as simple as a pushbutton widget can use MVC:

- The internal representation of states—pressed, unpressed, inactive, for example—is the Model. An Ajax widget would typically implement this as a JavaScript object.
- The painted-on-screen widget—composed of Document Object Model (DOM) nodes, in the case of an Ajax UI—with modifications for different states, highlights, and tooltips, is the View.
- The internal code for relating the two is the Controller. The event handler code (that is, what happens in the larger application when the user presses the button) is also a Controller, but not the Controller for this View and Model. We'll get to that in a minute.

A pushbutton in isolation will have very little behavior, state, or visible variation, so the payback for using MVC here is relatively small. If we look at a more complicated widget component, such as a tree or a table, however, the overall system is complicated enough to benefit from a clean MVC-based design more thoroughly.

Figure 4.1 illustrates MVC applied to a tree widget. The Model consists of tree nodes, each with a list of child nodes, an open/closed status, and a reference to some business object, representing files and directories in a file explorer, say. The View consists of the icons and lines painted onto the widget canvas. The Controller handles user events, such as opening and closing nodes and displaying pop-up menus, and also triggering graphical update calls for particular nodes, to allow the View to refresh itself incrementally.

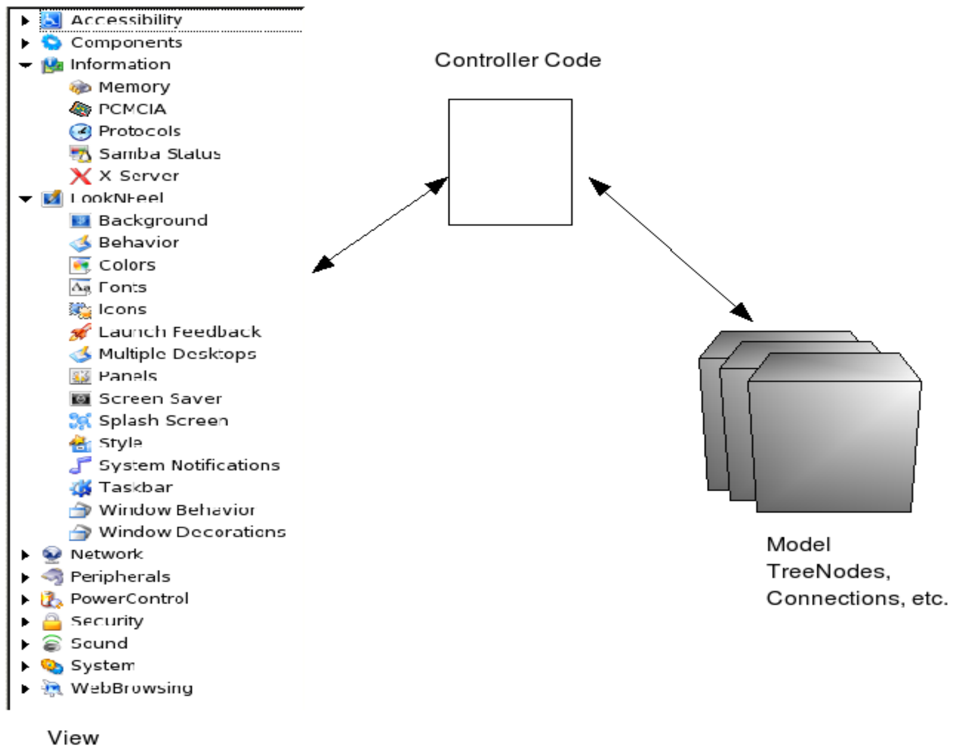


Figure 4.1: Model-View-Controller applied to the internal functioning of a tree widget. The view consists of a series of painted on-screen elements such as nodes, lines, and expand/collapse buttons, composed of DOM elements. Behind the scenes, the tree structure is modeled as a series of JavaScript objects. Controller code mediates between the two, interpreting mouse actions as operations on the model and changes in the model structure as changes to the DOM structure.

We've focused on the very small details of our application. We can also zoom out our perspective, to consider the entire JavaScript application that is delivered to the browser on startup. This, too, can be structured to follow the MVC pattern, and it will benefit from clear separation of concerns if it is.

At this level, the Model consists of the business domain objects, the View is the programmatically manipulated page as a whole, and the Controller is a combination of all the event handlers in the code that link the UI to the domain objects. Figure 4.2 illustrates the MVC operating at this level. In many ways, this is the most significant use of the MVC pattern for our purposes, because it is a significant departure from the conventional wisdom on MVC in web applications. We'll examine the details of this use of the pattern, and what it buys us, in the remainder of this chapter.

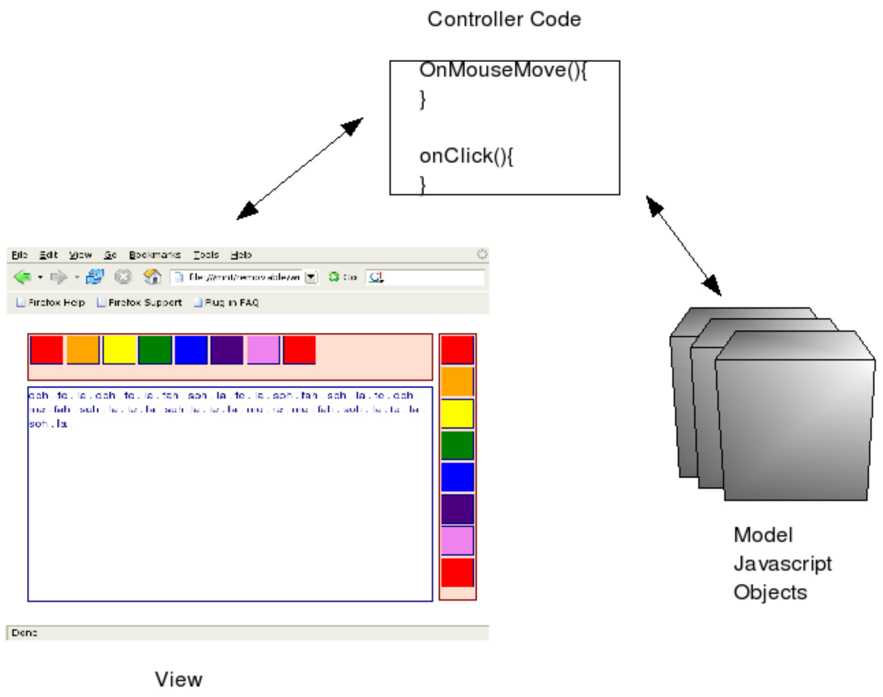


Figure 4.2 Model-View-Controller applied to the Ajax client application as a whole. The Controller at this level is the code that links the user interface to the business objects in the JavaScript.

If you think back to the conventional web MVC that we discussed in chapter 3 as well, you'll remember that we have at least three layers of MVC within a typical Ajax application, each performing different roles within the lifecycle of the application and each contributing to clean, well-organized code. Figure 4.3 illustrates how these MVC patterns at different scales are nested within each other in the application architecture.

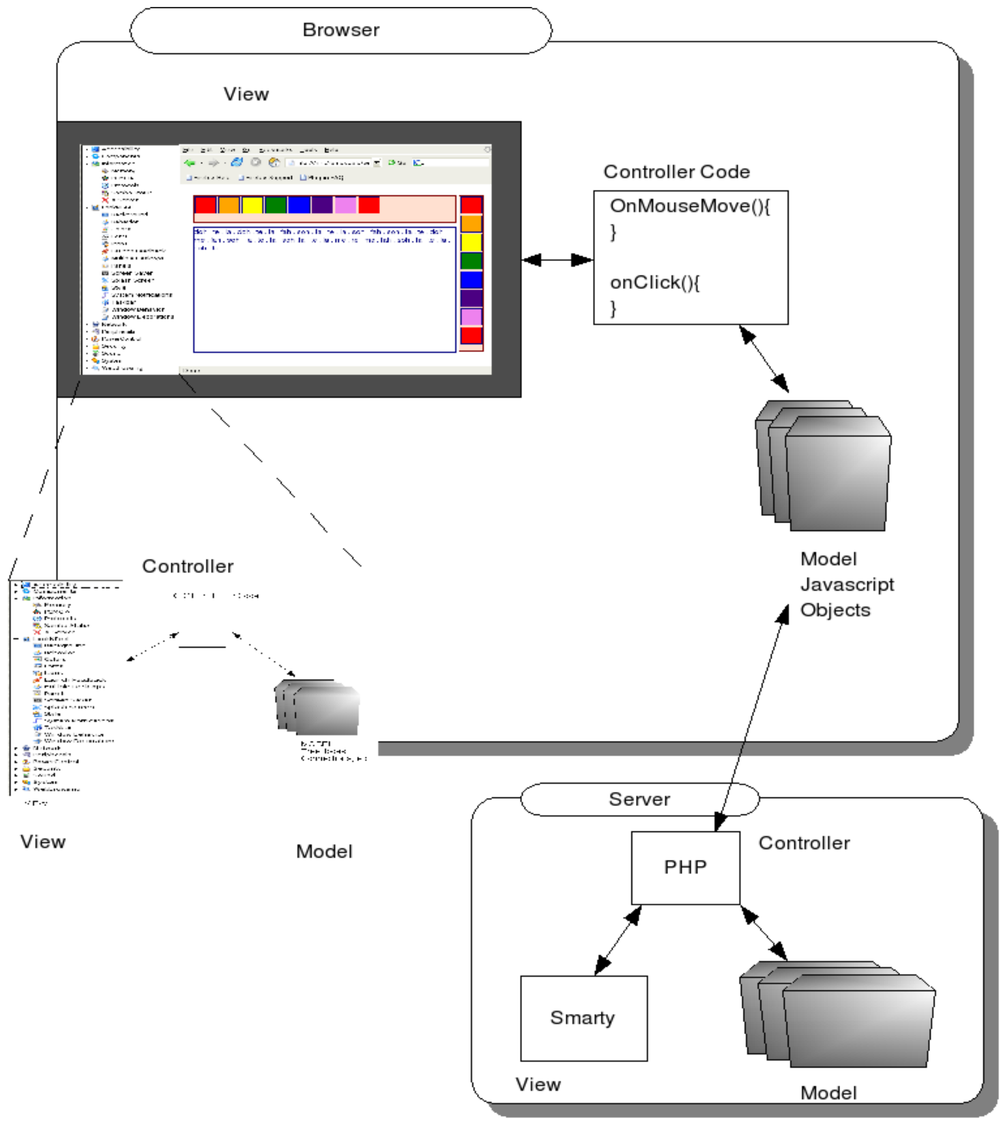


Figure 4.3: Nested MVC architecture, in which the pattern repeats itself at different scales. At the outermost level, we can see the pattern defining the workflow of the application as a whole, with the model residing on the web server. At a smaller scale, the pattern is replicated within the client application and, at a smaller scale than that, within individual widgets in the client application.

So, what does this mean to us when we're working at the codeface? In the following sections, we'll take a more practical look at using MVC to define the structure of our JavaScript application, how it will affect the way we write code, and what the benefits will be. Let's start with a look at the View.

4.2 The View in an Ajax application

From the point of view of the JavaScript application delivered to the browser when the application starts up, the View is the visible page, consisting of the DOM elements that are rendered by HTML markup or through programmatic manipulation (figure 4.2). We've already shown how to manipulate the DOM programmatically in chapter 2.

According to MVC, our View has two main responsibilities. It has to provide a visible interface for the user to trigger events from, that is, to talk to the Controller. It also needs to update itself in response to changes in the Model, usually communicated through the Controller again.

If the application is being developed by a team, the View will probably be the area subject to the most contention. Designers and graphic artists will be involved, as will programmers, particularly as we explore the scope for interactivity in an Ajax interface. Asking designers to write code, or programmers to get involved in the aesthetics of an application, is often a bad idea. Even if you're a one-man band doing both roles, it can be helpful to separate them, in order to focus on one at a time.

We showed in our overview of server MVC how code and presentation could become intertwined, and we separated them out using a template system. What are the options available to us here on the browser?

In chapter 3, we demonstrated how to structure our web pages so that the CSS, HTML, and JavaScript are defined in separate files. In terms of the page itself, this split follows MVC, with the stylesheet being the View and the HTML/DOM being the model (a Document Object Model). From the perspective at which we're looking right now, though, the page rendering is a black box, and the HTML and CSS together should be treated as the View. Keeping them separate is still a good idea, and simply by moving the JavaScript out into a separate file we have started to keep the designers and the programmers off each other's backs. This is just a start, however, as you'll see.

4.2.1 Keeping the logic out of the View

Writing all our JavaScript in a separate file is a good start for enforcing separation of the View, but even with this in place, we can entangle the View with the logic roles (that is, Model and Controller) without having to try too hard. If we write JavaScript event handlers inline, such as

```
<div class='importButton'
onclick='importData("datafeed3.xml", mytextbox.value);'/>
```

then we are hard-coding business logic into the View. What is `datafeed3`? What does the value of `mytextbox` have to do with it? Why does `importData()` take two arguments, and what do they mean? The designer shouldn't need to know these things.

`importData()` is a business logic function. The View and the Model shouldn't talk to one another directly, according to the MVC canon, so one solution is to separate them out with an extra layer of indirection. If we rewrite our DIV tag as

```
<div class='importButton' onclick='importFeedData()'/>
```

and define an event handler thusly

```
function importFeedData(event){
  importData("datafeed3.xml", mytextbox.value);
}
```

then the arguments are encapsulated within the `importFeedData()` function, rather than an anonymous event handler. This allows us to reuse that functionality elsewhere, keeping the concerns separate and the code DRY (at the risk of repeating myself, DRY means "don't repeat yourself").

The controller is still embedded in the HTML, however, which might make it hard to find in a large application. Further, the `onclick` attribute isn't valid XHTML, which might be a requirement of the project that we're working on.

To keep the controller and the view separate, we can attach the event programmatically. Rather than declare an event handler inline, we can specify a marker of some sort that will later be picked up by the code. We have several options for this marker. We can attach a unique ID to the element and specify event handlers on a per-element basis. The HTML would be rewritten as

```
  <div class='importButton' id='dataFeedBtn'>
```

and the following code executed as part of the `window.onload` callback, for example:

```
  var dfBtn=document.getElementById('dataFeedBtn');
  dfBtn.onclick=importFeedData;
```

If we want to perform the same action on multiple event handlers, we need to apply a non-unique marker of some sort. One simple approach is to define an extra CSS class. Let's look at a simple example, in which we bind mouse events to keys on a virtual musical keyboard. In listing 4.1, we define a simple page containing an unstyled document structure.

Listing 4.1 musical.html

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>

<head>
<link rel='stylesheet' type='text/css' href='musical.css'/>
<script type='text/javascript' src='musical.js'></script>
<script type='text/javascript'>
window.onload=function(){
  assignKeys();
}
</script>
</head>

<body>
<div id='keyboard' class='musicalKeys'>
  <div class='do musicalButton'></div>   | #1
  <div class='re musicalButton'></div>   | #1
  <div class='mi musicalButton'></div>   | #1
  <div class='fa musicalButton'></div>   | #1
  <div class='so musicalButton'></div>   | #1
  <div class='la musicalButton'></div>   | #1
  <div class='ti musicalButton'></div>   | #1
  <div class='do musicalButton'></div>   | #1
</div>

<div id='console' class='console'>
</div>
</body>

</html>
```

(annotation) <#1 Keys on our "keyboard">

We declare the page to conform to XHTML strict definition, just to show that it can be done. The keyboard element is assigned a unique ID, but the keys are not. Note that the keys designated `#1` are each defined as having two styles. `musicalButton` is common to all keys, and a separate style differentiates them by note. These styles are defined separately in the stylesheet (listing 4.2).

Listing 4.2 musical.css

```
.body{
  background-color: white;
}
.musicalKeys{
  background-color: #ffe0d0;
  border: solid maroon 2px;
  width: 536px;
  height: 68px;
  top: 24px;
  left: 24px;
  margin: 4px;
  position: absolute;
  overflow: auto;
}
.musicalButton{
  border: solid navy 1px;
  width: 60px;
  height: 60px;
  position: relative;
  margin: 2px;
  float: left;
}
.do{ background-color: red; }
.re{ background-color: orange; }
.mi{ background-color: yellow; }
.fa{ background-color: green; }
.so{ background-color: blue; }
.la{ background-color: indigo; }
.ti{ background-color: violet; }

div.console{
  font-family: arial, helvetica;
  font-size: 16px;
  color: navy;
  background-color: white;
  border: solid navy 2px;
  width: 536px;
  height: 320px;
  top: 106px;
  left: 24px;
  margin: 4px;
  position: absolute;
  overflow: auto;
}
```

The style `musicalKeys` defines the common properties of each key. The note-specific styles simply define a color for each key. Note that whereas top-level document elements are positioned with explicit pixel precision, we use the float style attribute to lay the keys out in a horizontal line using the browser’s built-in layout engine.

The JavaScript file (listing 4.3) binds the events to these keys programmatically.

Listing 4.3 musical.js

```
function assignKeys(){
  var keyboard=document.getElementById('keyboard');  | #1
  var keys=keyboard.getElementsByTagName("div");     | #2
  if (keys){
    for(var i=0;i<keys.length;i++){
      var key=keys[i];
      var classes=(key.className).split(" ");
      if (classes && classes.length>=2
       && classes[1]=='musicalButton'){
        var note=classes[0];
        key.note=note;                 | #3
        key.onmouseover=playNote;
      }
    }
  }
}

function playNote(event){
  var note=this.note;                     | #4
  var console=document.getElementById('console');
  if (note && console){
    console.innerHTML+=note+" . ";
  }
}
```

(annotation) <#1 Find parent DIV>
(annotation) <#2 Enumerate children>
(annotation) <#3 Add custom attribute>
(annotation) <#4 Retrieve custom attribute>

The `assignKeys()` function is called by `window.onload`. (We could have defined `window.onload` directly in this file, but that limits its portability). We find the keyboard element by its unique ID and then use `getElementsByTagName()` to iterate through all the DIV elements inside it. This requires some knowledge of the page structure, but it allows the designer the freedom to move the keyboard DIV around the page in any way that he wants.

The DOM elements representing the keys return a single string as `className` property. We use the `inbuilt String.split` function to convert it into an array, and check that the element is of class musicalButton. We then read the second styling—which represents the note that this key plays—and attach it to the DOM node as an extra attribute, where it can be picked up again in the event handler. We could have assigned unique IDs to the keys to perform this binding, such as

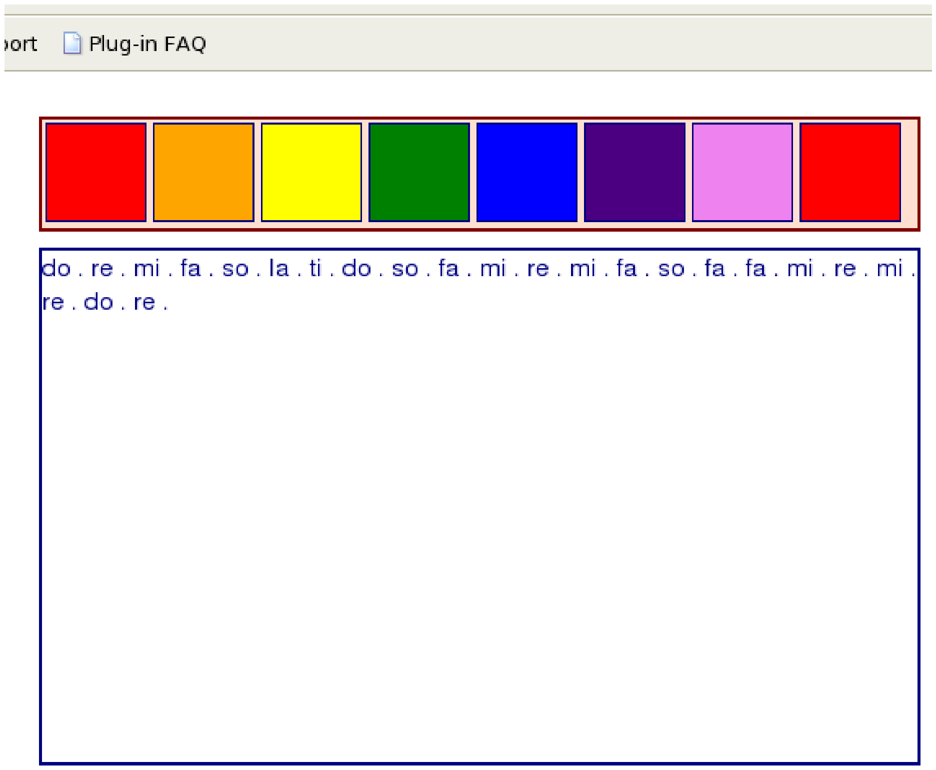
```
  <div id='fa' class='musicalButton'></div>
```

or assigned nonstandard attributes directly, for example

```
  <div note='fa' class='musicalButton'></div>
```

but since we were styling the keys with different colors anyway, using the style name made sense.

Playing music through a web browser is rather tricky, so in this case, we simply write the note out to the "console" underneath the keyboard. `innerHTML` is adequate for this purpose. Figure 4.4 shows our musical keyboard in action. We've achieved good separation of roles here. Provided the designer drops the keyboard and console DIV tags somewhere on the page and includes the stylesheet and JavaScript, the application will work, and the risk of accidentally breaking the event logic is small. Effectively, the HTML page has become a template into which we inject variables and logic.



```
do . re . mi . fa . so . la . ti . do . so . fa . mi . re . mi . fa . so . fa . fa . mi . re . mi .
re . do . re .
```

Figure 4.4 Musical keyboard application running in a browser. The colored areas along the top are mapped to music notes, which are printed out in the lower console area when the mouse moves over them.

This provides us with a good way of keeping logic out of the View. We've worked through this example manually, to demonstrate the details of how it's done. In production, you might like to make use of a couple of third-party libraries that address the same issue.

The Rico framework (http://www.openrico.org/) has a concept of 'Behavior' objects that target specific sections of a DOM tree and add interactivity to them. We looked at the Rico Accordion behavior briefly in section 3.5.2.

A similar separation between HTML markup and interactivity can be achieved by Ben Nolan's 'Behaviour' library (http://ripcord.co.nz/behaviour/) that builds on top of the Scriptaculous library that we looked at, also in section 3.5.2. This library allows event handler code to be assigned to DOM elements based on CSS selector rules (see Chapter 2). In our example above, the assignKeys() function programmatically selects the document element with id 'keyboard', and then gets all DIV elements directly contained by it, using DOM manipulation methods. We can express this using a CSS selector, as:

```
#keyboard div
```

Using CSS, we could style all our keyboard elements using this selector. Using the Behaviour.js library, we can also apply event handlers in the same way as follows:

```
var myrules={
  '#keyboard div' : function(key){
    var classes=(key.className).split(" ");
    if (classes && classes.length>=2
    && classes[1]=='musicalButton'){
      var note=classes[0];
      key.note=note;
      key.onmouseover=playNote;
    }
  }
};

Behaviour.register(myrules);
```

Most of the logic is the same as in our previous example, but the use of CSS selectors offers a concise alternative to programmatically locating DOM elements, particularly if we're adding several behaviors at once.

That keeps the logic out of the view for us, but it's also possible to tangle the View up in the logic, as we shall see.

4.2.2 Keeping the View out of the logic

We've gotten to the point now where the designers can develop the look of the page without having to touch the code. However, as it stands, some of the functionality of the application is still embedded in the HTML, namely, the ordering of the keys. Each key is defined as a separate DIV tag, and the designers could unwittingly delete some of them.

If the ordering of the keys is a business domain function rather than a design issue—and we can argue that it is—then it makes sense to generate some of the DOM for the component programmatically, rather than declare it in the HTML. Further, we may want to have multiple components of the same type on a page. If we don't want the designer to modify the order of the keys on our keyboard, for example, we could simply stipulate that they assign a DIV tag with class `keyboard` and have our initialization code find it and add the keys programmatically. Listing 4.4 shows the modified JavaScript required to do this.

Listing 4.4 musical_dyn_keys.js

```
var notes=new Array("do","re","mi","fa","so","la","ti","do");

function assignKeys(){
  var candidates=document.getElementsByTagName("div");
  if (candidates){
    for(var i=0;i<candidates.length;i++){
      var candidate=candidates[i];
      if (candidate.className.indexOf('musicalKeys')>=0){
        makeKeyboard(candidate);
      }
    }
  }
}
```

```
function makeKeyboard(el){
  for(var i=0;i<notes.length;i++){
    var key=document.createElement("div");
    key.className=notes[i]+" musicalButton";
    key.note=notes[i];
    key.onmouseover=playNote;
    el.appendChild(key);
  }
}

function playNote(event){
  var note=this.note;
  var console=document.getElementById('console');
  if (note && console){
    console.innerHTML+=note+" . ";
  }
}
```

Previously, we had defined our key sequence in the HTML. Now it is defined as a global JavaScript array. The `assignKeys()` method examines all the top-level DIV tags in the document, to see if the `className` matches the value `musicalKeys`. If it does, then it tries to populate that DIV with a working keyboard, using the `makeKeyboard()` function. `makeKeyboard()` simply creates new DOM nodes and then manipulates them in the same way as listing 4.4 did for the declared DOM nodes that it encountered. The `playNote()` callback handler operates exactly as before.
Because we are populating empty DIVs with our keyboard controls, adding a second set of keys is simple, as listing 4.5 illustrates.

Listing 4.5 musical_dyn_keys.html

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>

<head>
<link rel='stylesheet' type='text/css'
  href='musical_dyn_keys.css'/>
<script type='text/javascript'
  src='musical_dyn_keys.js'>
</script>
<script type='text/javascript'>
window.onload=function(){
  assignKeys();
}
</script>
</head>

<body>
<div id='keyboard' class='toplong musicalKeys'></div>

<div id='keyboard' class='sidebar musicalKeys'></div>

<div id='console' class='console'>
```

```
</div>
</body>

</html>
```

Adding a second keyboard is a single-line operation. Because we don’t want them sitting one on top of the other, we move the placement styling out of the `musicalKeys` style class and into separate classes. The stylesheet modifications are shown in listing 4.6.

Listing 4.6 Changes to musical_dyn_keys.css

```
.musicalKeys{
  background-color: #ffe0d0;
  border: solid maroon 2px;
  position: absolute;
  overflow: auto;
  margin: 4px;
}
.toplong{
  width: 536px;
  height: 68px;
  top: 24px;
  left: 24px;
}
.sidebar{
  width: 48px;
  height: 400px;
  top: 24px;
  left: 570px;
}
```

The `musicalKeys` class defines the visual style common to all keyboards. `toplong` and `sidebar` simply define the geometry of each keyboard.
By refactoring our keyboard example in this way, we have made it possible to reuse the code easily. However, the design of the keyboard is partly defined in the JavaScript, in the `makeKeyboard()` function in listing 4.4, and yet, as figure 4.5 shows, one keyboard has a vertical layout and the other a horizontal one. How did we achieve this?

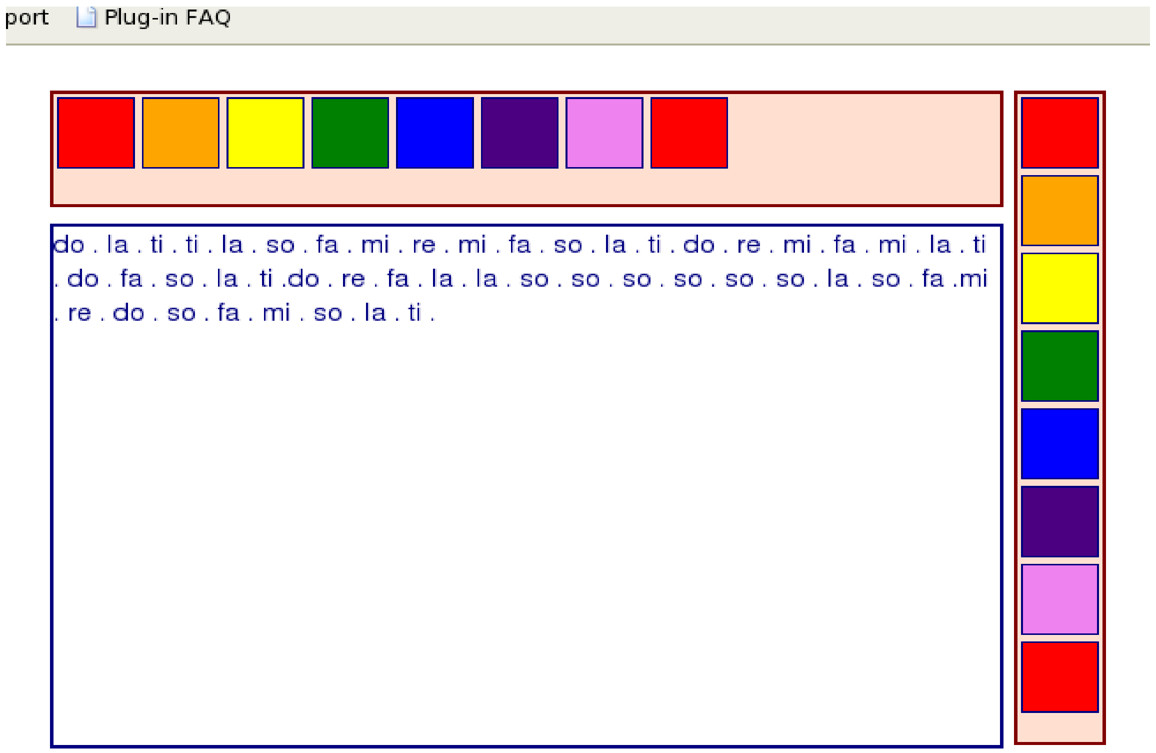

port
Plug-in FAQ
do . la . ti . ti . la . so . fa . mi . re . mi . fa . so . la . ti . do . re . mi . fa . mi . la . ti
. do . fa . so . la . ti .do . re . fa . la . la . so . so . so . so . so . so . la . so . fa .mi
. re . do . so . fa . mi . so . la . ti .

Figure 4.5 Screenshot of revised musical keyboard program allowing the designer to specify multiple keyboards. Using CSS-based styling and the native render engine, we can accommodate both vertical and horizontal layouts without writing explicit layout code in our JavaScript.

`makeKeyboards()` could easily have computed the size of the DIV that it was targeting and placed each button programmatically. In that case, we would need to get quite fussy about deciding whether the DIV was vertical or horizontal and write our own layout code. To a Java GUI programmer familiar with the internals of LayoutManager objects, this may seem all too obvious a route to take. If we took it, our programmers would wrest control of the widget's look from the designers, and trouble would ensue!

As it is, `makeKeyboard()`modifies only the structure of the document. The keys are laid out by the browser's own layout engine, which is controlled by stylesheets, in particular, by the `float` style attribute, in this case. It is important that it is controlled by the designer. Logic and View remain separate, and peace reigns.

The keyboard was a relatively simple widget. In a larger, more complex widget such as a tree table, it may be harder to see how the browser's own render engine can be coerced into doing the layout, and in some cases, programmatic styling is inevitable. However, it's always worth asking this question, in the interests of keeping View and Logic separate. The browser render engine is also a high-performing, fast, and well-tested piece of native code, and it is likely to beat any JavaScript algorithms that we cook up.

That about wraps it up for the View for the moment. In the next section, we'll explore the role of the Controller in MVC and how that relates to JavaScript event handlers in an Ajax application.

In the next installment, we will continue with this chapter by discussing the roles of the Controller and the Model in an Ajax application, when using MVC to define the structure of our JavaScript application.