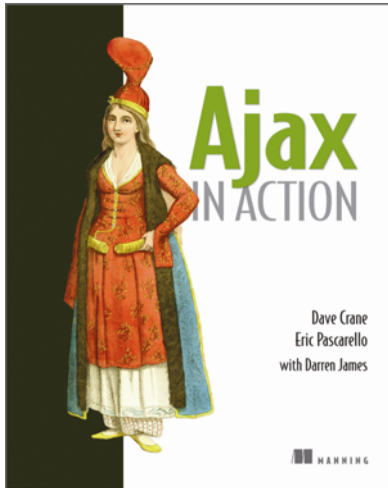


Excerpt

*The following is part two of an excerpt
from chapter 4 of *Ajax in Action*
from Manning Publications*



Under the hood: Managing Events and the Model

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 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.

In this installment, we'll look at the Controller part of MVC as a way of organizing the event handling code in an Ajax client application, and at the introduction of a domain model into the JavaScript tier of code.

4.3 The Controller in an Ajax application

The role of the Controller in MVC is to serve as an intermediary between the Model and the View, decoupling them from one another. In a GUI application such as our Ajax client application, the Controller layer is composed of event handlers. As is often the case with web browsers, techniques have evolved over time, and modern browsers support two different event models. The classic model is relatively simple and is in the process of being superseded by the newer W3C specifications for event handling. At the time of writing, however,

implementations of the new event-handling model vary between browsers and are somewhat problematic. Both event models are discussed here.

4.3.1 *Classic JavaScript event handlers*

The JavaScript implementation in web browsers allows us to define code that will be executed in response to a user event, typically either the mouse or keyboard. In the modern browsers that support Ajax, these event handlers can be assigned to most visual elements. We can use the event handlers to connect our visible user interface, that is, the View, to the business object Model.

The classic event model has been around since the early days of JavaScript, and is relatively simple and straightforward. DOM elements have a small number of predefined properties to which callback functions can be assigned. For example, to attach a function that will be called when the mouse is clicked on an element `myDomElement`, we could write

```
myDomElement.onclick=showAnimatedMonkey
```

`myDomElement` is any DOM element that we have a programmatic handle on. `showAnimatedMonkey` is a function, defined as

```
function showAnimatedMonkey(){
    //some skillfully executed code to display
    //an engaging cartoon character here
}
```

that is, as an ordinary JavaScript function. Note that when we assign the event handler, we pass the Function object, not a call to that object, so it doesn't have parentheses after the function name. This is a common mistake:

```
myDomElement.onclick=showAnimatedMonkey();
```

This looks more natural to programmers unaccustomed to treating functions as first-class objects, but it will not do what we think. The function will be called when we make the assignment, not when the DOM element is clicked. The `onclick` property will be set to whatever is returned by the function. Unless you're doing something extremely clever involving functions that return references to other functions, this is probably not desirable. Here's the right way to do it:

```
myDomElement.onclick=showAnimatedMonkey;
```

This passes a reference to our callback function to the DOM element, telling it that this is the function to invoke when the node is clicked on. DOM elements have many such properties to which event-handler functions can be attached. Common event-handler callbacks for GUI

work are listed in table 4.1. Similar properties can be found elsewhere in web browser JavaScript, too. The `XMLHttpRequest.onreadystatechange` and `window.onload`, which we have encountered already, are also event handler functions that can be assigned by the programmer.

Table 4.1 Common GUI event handler properties in the DOM

Property	Description
<code>onmouseover</code>	Triggered when the mouse first passes into an element's region.
<code>onmouseout</code>	Triggered when the mouse passes out of an element's region.
<code>onmousemove</code>	Triggered whenever the mouse moves while within an element's region (i.e., frequently!).
<code>onclick</code>	Triggered when the mouse is clicked within an element's region.
<code>onkeypress</code>	Triggered when a key is pressed while this element has input focus. Global key handlers can be attached to the document's body.
<code>onfocus</code>	A visible element receives input focus.
<code>onblur</code>	A visible element loses input focus.

There is an unusual feature of the event handler functions worth mentioning here, as it trips people up most frequently when writing object-oriented JavaScript, a feature that we will lean on heavily in developing Ajax clients.

We've got a handle on a DOM element, and assigned a callback function to the `onclick` property. When the DOM element receives a mouse click, the callback is invoked. However, the function context (that is, the value that variable `this` resolves to—see appendix B for a fuller discussion of JavaScript Function objects) is assigned to the DOM node that received the event. Depending on where and how the function was originally declared, this can be very confusing.

Let's explore the problem with an example. We define a class to represent a button object, which has a reference to a DOM node, a callback handler, and a value that is displayed when the button is clicked. Any instance of the button will respond in the same way to a mouse

click event, and so we define the callback handler as a method of the button class. That's a sufficient spec for starters, so let's look at the code. Here's the constructor for our button:

```
function Button(value, domEl) {
  this.domEl=domEl;
  this.value=value;
  this.domEl.onclick=this.clickHandler;
}
```

We go on to define an event handler as part of the Button class:

```
Button.prototype.clickHandler=function(){
  alert(this.value);
}
```

It looks straightforward enough, but it doesn't do what we want it to. The alert box will generally return a message `undefined`, not the `value` property that we passed to the constructor. Let's see why. The function `clickHandler` gets invoked by the browser when the DOM element is clicked, and it sets the function context to the DOM element, not the Button JavaScript object. So, `this.value` refers to the `value` property of the DOM element, not the Button object. You'd never tell by looking at the declaration of the event-handler function, would you?

We can fix things up by passing a reference to the Button object to the DOM element, that is, by modifying our constructor like this:

```
function Button(value, domEl) {
  this.domEl=domEl;
  this.value=value;
  this.domEl.buttonObj=this;
  this.domEl.onclick=this.clickHandler;
}
```

The DOM element still doesn't have a `value` property, but it has a reference to the Button object, which it can use to get the value. We finish up by altering the event handler like this:

```
Button.prototype.clickHandler=function(){
  var buttonObj=this.buttonObj;
  var value=(buttonObj && buttonObj.value) ?
    buttonObj.value : "unknown value";
  alert(value);
}
```

The DOM node refers to the Button, which refers to its `value` property, and our event handler does what we want it to. We could have attached the `value` directly to the DOM node, but attaching a reference to the entire backing object allows this pattern to work easily with

arbitrarily complex objects. In passing, it's worth noting that we've implemented a mini-MVC pattern here, with the DOM element View fronting a backing object Model.

That's the classic event model, then. The main shortcoming of this event model is that it allows only one event-handler function per element. In the Observer pattern that we presented in chapter 3, we noted that an observable element could have any number of observers attached to it at a given time. When writing a simple script for a web page, this is unlikely to be a serious shortcoming, but as we move toward the more complex Ajax clients, we start to feel the constraint more. We will take a closer look at this in section 4.3.3, but first, let's look at the more recent event model.

4.3.2 *The W3C event model*

The more flexible event model proposed by the W3C is complex. An arbitrary number of listeners can be attached to a DOM element. Further, if an action takes place in a region of the document in which several elements overlap, the event handlers of each are given an opportunity to fire and to veto further calls in the event stack, known as “swallowing” the event. The specification proposes that the event stack be traversed twice in total, first propagating from outermost to innermost (from the document element down) and then bubbling up again from the inside to the outside. In practice, different browsers implement different subsets of this behavior.

In Mozilla-based browsers and Safari, event callbacks are attached using `addEventListener()` and removed by a corresponding `removeEventListener()`. Internet Explorer offers similar functions: `attachEvent()` and `detachEvent()`. Mike Foster's `xEvent` object (part of the `x` library—see the Resources section at the end of this chapter) makes a brave attempt at creating a Façade (see chapter 3) across these implementations in order to provide a rich cross-browser event model.

There is a further cross-browser annoyance here, as the callback handler functions defined by the user are called slightly differently. Under Mozilla browsers, the function is invoked with the DOM element receiving the event as a context object, as for the classic event model. Under Internet Explorer, the function context is always the Window object, making it impossible to work out which DOM element is currently calling the event handler! Even with a layer such as `xEvent` in place, developers need to account for these variations when writing their callback handlers.

The final issue to mention here is that neither implementation provides a satisfactory way of returning a list of all currently attached listeners.

At this point, I advise you not to use the newer event model. The main shortcoming of the classic model—lack of multiple listeners—can be addressed by the use of design patterns, as we will see next.

4.3.3 *Implementing a flexible event model in JavaScript*

Because of the incompatibilities of the newer W3C event model, the promise of a flexible event listener framework remains just out of reach. We described the Observer pattern in chapter 3, and that seems to fit the bill nicely, allowing us to add and remove observers from the event source in a flexible fashion. Clearly, the W3C felt the same way, as the revised event model implements Observer, but the browser vendors delivered inconsistent and just plain broken implementations. The classic event model falls far short of the Observer pattern, but perhaps we can enhance it a little with some code of our own.

Managing multiple event callbacks

Before going on to implement our own solution, let's come to grips with the problem through a simple example. Listing 4.7 shows a simple web page, in which a large DIV area responds to mouse move events in two ways.

Listing 4.7 mousemat.html

```
<html>
<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<script type='text/javascript'>
var cursor=null;
window.onload=function(){
  var mat=document.getElementById('mousemat');
  mat.onmousemove=mouseObserver;
  cursor=document.getElementById('cursor');
}
function mouseObserver(event){
  var e=event || window.event;
  writeStatus(e)
;
  drawThumbnail(e)
;
}
function writeStatus(e){
  window.status=e.clientX+", "+e.clientY
;
}
function drawThumbnail(e){
  cursor.style.left=((e.clientX/5)-2)+"px";
  cursor.style.top=((e.clientY/5)-2)+"px";
}
</script>
</head>
```

```
<body>
<div class='mousemat' id='mousemat'></div>
<div class='thumbnail' id='thumbnail'>
  <div class='cursor' id='cursor' />
</div>
</body>
</html>
```

First, it updates the browser status bar, in the `writeStatus()` function. Second, it updates a smaller thumbnail image of itself, by repositioning a dot in the thumbnail area, to copy the mouse pointer's movements, in the `drawThumbnail()` function. Figure 4.6 shows the page in action.

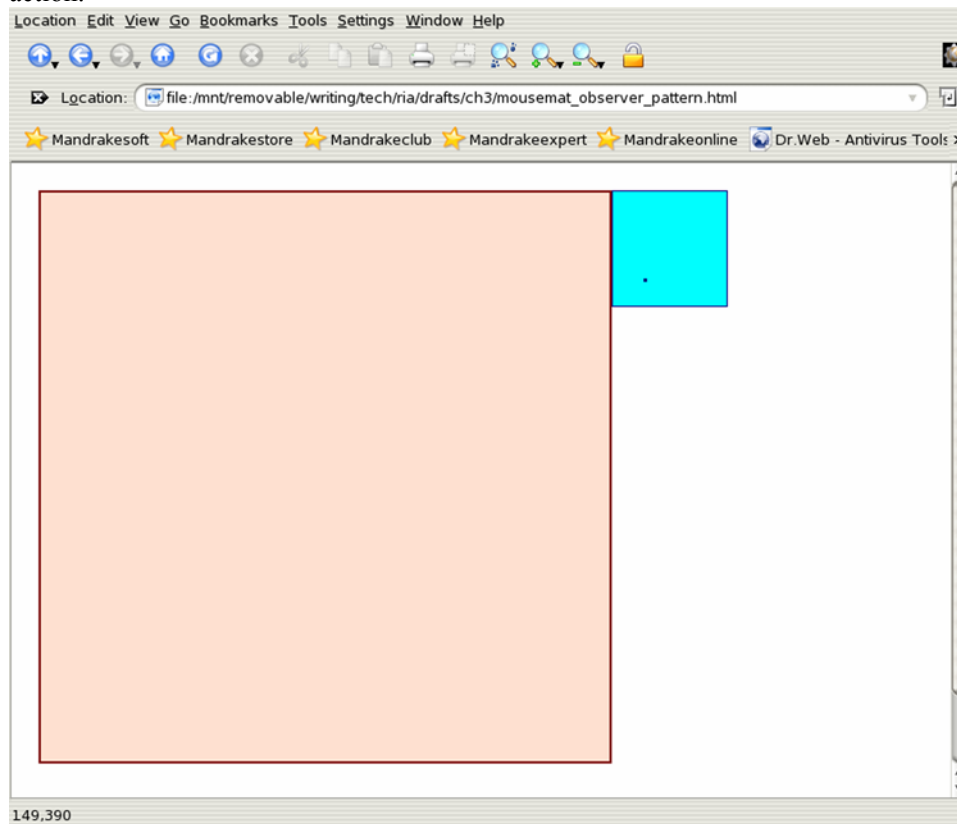


Figure 4.6 The Mousemat program tracks mouse movement events on the main “virtual mousemat” area in two ways: by updating the browser status bar with the mouse coordinates and by moving the dot on the thumbnail view in sync with the mouse pointer.

These two actions are independent of each other, and we would like to be able to swap these and other responses to the mouse movement in and out at will, even while the program is running.

The `mouseObserver()` function is our event listener. (The first line is performing some simple cross-browser magic, by the way. Unlike Mozilla, Opera, or Safari, Internet Explorer doesn't pass any arguments to the callback handler function, but stores the Event object in `window.event`.) In this example, we have hardwired the two activities in the event handler, calling `writeStatus()` and `drawThumbnail()` in turn. The program does exactly what we want it to do, and, because it is a small program, the code for `mouseObserver()` is reasonably clear. Ideally, though, we would like a cleaner way to wire the event listeners together, allowing the approach to scale to more complex or dynamic situations.

Implementing Observer in JavaScript

The proposed solution is to define a generic event router object, which attaches a standard function to the target element as an event callback and maintains a list of listener functions. This would allow us to rewrite our `mousemat` initialization code in this way:

```
window.onload=function(){
  var mat=document.getElementById('mousemat');
  ...
  var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");
  mouseRouter.addListener(writeStatus);
  mouseRouter.addListener(drawThumbnail);
}
```

We define an `EventRouter` object, passing in the DOM element and the type of event that we would like to register as arguments. We then add listener functions to the router object, which also supports a `removeListener()` method that we don't need here. It looks straightforward, but how do we implement it?

First, we write a constructor for the object, which in JavaScript is simply a function. (Appendix B contains a primer on the syntax of JavaScript objects. Take a look if any of the following code looks strange or confusing.)

```
jsEvent.EventRouter=function(el,eventType){
  this.lsnrs=new Array();
  this.el=el;
  el.eventRouter=this;
  el[eventType]=jsEvent.EventRouter.callback;
}
```

We define the array of listener functions, which is initially empty, take a reference to the DOM element, and give it a reference to this object, using the pattern we described in section 3.5.1. We then assign a static method of the `EventRouter` class, simply called `callback`, as the event handler. Remember that in JavaScript, the square bracket and dot notations are equivalent, which means

```
el.onmouseover
```

is the same as

```
el['onmouseover']
```

We use this to our advantage here, passing in the name of a property as an argument. This is similar to reflection in Java or the .NET languages.

Let's have a look at the callback then:

```
jsEvent.EventRouter.callback=function(event){
  var e=event || window.event;
  var router=this.eventRouter;
  router.notify(e)
}
```

Because this is a callback, the function context is the DOM node that fired the event, not the router object. We retrieve the `EventRouter` reference that we had attached to the DOM node, using the backing object pattern that we saw earlier. We then call the `notify()` method of the router, passing the event object in as an argument.

The full code for the Event Router object is shown in listing 4.8.

Listing 4.8 EventRouter.js

```
var jsEvent=new Array();
jsEvent.EventRouter=function(el,eventType){
  this.lsnrs=new Array();
  this.el=el;
  el.eventRouter=this;
  el[eventType]=jsEvent.EventRouter.callback;
}
jsEvent.EventRouter.prototype.addListener=function(lsnr){
  this.lsnrs.append(lsnr,true);
}
jsEvent.EventRouter.prototype.removeListener=function(lsnr){
  this.lsnrs.remove(lsnr);
}
jsEvent.EventRouter.prototype.notify=function(e){
  var lsnrs=this.lsnrs;
```

```

    for(var i=0;i<lsnrs.length;i++){
        var lsnr=lsnrs[i];
        lsnr.call(this,e);
    }
}
jsEvent.EventRouter.callback=function(event){
    var e=event || window.event;
    var router=this.eventRouter;
    router.notify(e)
}
}

```

Note that some of the methods of the array are not standard JavaScript but have been defined by our extended array definition, which is discussed in appendix B. Notably, `addListener()` and `removeListener()` are simple to implement using the `append()` and `remove()` methods. Listener functions are invoked using the `Function.call()` method, whose first argument is the function context, and subsequent arguments (in this case the event) are passed through to the callee.

The revised mousemat example is shown in listing 4.9.

Listing 4.9 Revised mousemat.html, using EventRouter

```

<html>
<head>
<link rel='stylesheet' type='text/css' href='mousemat.css' />
<script type='text/javascript' src='extras-array.js'></script>
<script type='text/javascript' src='eventRouter.js'></script>
<script type='text/javascript'>
var cursor=null;
window.onload=function(){
    var mat=document.getElementById('mousemat');
    cursor=document.getElementById('cursor');
    var mouseRouter=new jsEvent.EventRouter(mat,"onmousemove");
    mouseRouter.addListener(writeStatus);
    mouseRouter.addListener(drawThumbnail);
}
function writeStatus(e){
    window.status=e.clientX+", "+e.clientY
}
function drawThumbnail(e){
    cursor.style.left=((e.clientX/5)-2)+"px";
    cursor.style.top=((e.clientY/5)-2)+"px";
}
</script>
</head>
<body>

```

```
<div class='mousemat' id='mousemat'></div>
<div class='thumbnail' id='thumbnail'>
  <div class='cursor' id='cursor' />
</div>
</body>
</html>
```

The inline JavaScript is greatly simplified. All we need to do is create the `EventRouter`, pass in the listener functions, and provide implementations for the listeners. We leave it as an exercise for the reader to include checkboxes to add and remove each listener dynamically.

This rounds out our discussion of the Controller layer in an Ajax application and the role that design patterns—Observer in particular—can play in keeping it clean and easy to work with. In the following section, we'll look at the final part of the MVC pattern, the Model.

4.4 *Models in an Ajax application*

The Model is responsible for representing the business domain of our application, that is, the real-world subject that the application is all about, whether that is a garment store, a musical instrument, or a set of points in space. As we've noted already, the Document Object Model is *not* the model at the scale at which we're looking at the application now. Rather, the model is a collection of code that we have written in JavaScript. Like most design patterns, MVC is heavily based on object-oriented thinking.

JavaScript is not designed as an OO language, although it can be persuaded into something resembling object orientation without too much struggle. It does support the definition of something very similar to object classes through its prototype mechanism, and some developers have gone as far as implementing inheritance systems for JavaScript. We discuss these issues further in appendix B. When implementing MVC in JavaScript so far, we've adapted it to the JavaScript style of coding, for example, passing Function objects directly as event listeners. When it comes to defining the model, however, using JavaScript objects, and as much of an OO approach as we're comfortable with for the language, makes good sense. In the following section, we'll show how that is done.

4.4.1 *Using JavaScript to model the business domain*

When discussing the View, we are very much tied to the DOM. When we talk about the Controller, we are constrained by the browser event models. When writing the Model, however, we are dealing almost purely with JavaScript and have very little to do with browser-specific functionality. Those who have struggled with browser incompatibilities and bugs will recognize this as a comfortable situation in which to be.

Let's look at a simple example. In chapter 3 we discussed our garment store application, from the point of view of generating a data feed from the server. The data described a list of garment types, in terms of a unique ID, a name, and a description, along with price, color, and size information. Let's return to that example now and consider what happens when the data arrives at the client. Over the course of its lifetime, the application will receive many such streams of data and have a need to store data in memory. Think of this as a cache if you like—data stored on the client can be redisplayed very quickly, without needing to go back to the server at the time at which the user requests the data. This benefits the user's workflow, as discussed in chapter 1.

We can define a simple JavaScript object that corresponds to the garment object defined on the server. Listing 4.10 shows a typical example.

Listing 4.10 Garment.js

```
var garments=new Array();
function Garment(id,title,description,price){
  this.id=id;
  garments[id]=this;
  this.title=title;
  this.description=description;
  this.price=price;
  this.colors=new Object();
  this.sizes=new Object();
}
Garment.prototype.addColor(color){
  this.colors.append(color,true);
}
Garment.prototype.addSize(size){
  this.sizes.append(size,true);
}
```

We define a global array first of all, to hold all our garments. (Yes, global variables are evil. In production, we'd use a namespacing object, but we've omitted that for clarity here.) This is an associative array, keyed by the garment's unique ID, ensuring that we have only one reference to each garment type at a time. In the constructor function, we set all the simple properties, that is, those that aren't arrays. We define the arrays as empty and provide simple adder methods, which uses our enhanced array code (see appendix B) to prevent duplicates.

We don't provide getter or setter methods by default and don't support the full access control—private, protected, and public variables and methods—that a full OO language does. There are ways of providing this feature, which are discussed in appendix B, but my own preference is to keep the Model simple.

When parsing the XML stream, it would be nice to initially build an empty `Garment` object and then populate it field by field. The astute reader may be wondering why we haven't provided a simpler constructor. In fact, we have. JavaScript function arguments are mutable, and any missing values from a call to a function will simply initialize that value to null. So the call

```
var garment=new Garment(123);
```

will be treated as identical to

```
var garment=new Garment(123,null,null,null);
```

We need to pass in the ID, because we use that in the constructor to place the new object in the global list of garments.

4.4.2 *Interacting with the server*

We could parse the XML feed of the type shown in listing 4.10 in order to generate `Garment` objects in the client application. We've already seen this in action in chapter 2, and we'll see a number of variations in chapter 5, so we won't go into all the details here. The XML document contains a mixture of attributes and tag content. We read attribute data using the `attributes` property and `getNamedItem()` function and read the body text of tags using the `firstChild` and `data` properties, for example:

```
garment.description=descrTag.firstChild.data;
```

to parse an XML fragment such as

```
<description>Large tweedy hat looking  
like an unappealing strawberry  
</description>
```

Note that garments are automatically added to our array of all garments as they are created, simply by invoking the constructor. Removing a garment from the array is also relatively straightforward:

```
function unregisterGarment(id){  
    garments[id]=null;  
}
```

This removes the garment type from the global registry, but won't cascade to destroy any instances of `Garment` that we have already created. We can add a simple validation test to the `Garment` object, however:

```
Garment.prototype.isValid=function(){  
    return garments[this.id]!=null;
```

```
}
```

We've now defined a clear path for propagating data all the way from the database to the client, with nice, easy-to-handle objects at each step. Let's recap the steps. First, we generate a server-side object model from the database. In section 3.4.2, we saw how to do this using an Object-Relational Mapping (ORM) tool, which gave us out-of-the-box two-way interactions between object model and database. We can read data into objects, modify it, and save the data.

Second, we used a template system to generate an XML stream from our object model, and third, we parsed this stream in order to create an object model on the JavaScript tier. We must do this parsing by hand for now. We may see ORM-like mapping libraries appearing in the near future.

In an administrative application, of course, we might want to edit our data too, that is, modify the JavaScript model, and then communicate these changes back to the server model. This forces us to confront the issue that we now have two copies of our domain model and that they may get out of sync with each other.

In a classic web application, all the intelligence is located on the server, so our model is located there, in whatever language we're using. In an Ajax application, we want to distribute the intelligence between the client and the server, so that the client code can make some decisions for itself before calling back to the server. If the client makes only very simple decisions, we can code these in an ad hoc way, but then we won't get much of the benefit of an intelligent client, and the system will tend to still be unresponsive in places. If we empower the client to make more important decisions for itself, then it needs to know something about our business domain, at which point it really needs to have a model of the domain.

We can't do away with the domain model on the server, because some resources are available only on the server, such as database connections for persistence, access to legacy systems, and so on. The client-side domain model has to work with the one on the server. So, what does that entail? In chapter 5 we will develop a fuller understanding of the client/server interactions and how to work cleanly with a domain model split across both tiers.

So far we've looked at Model, View, and Controller in isolation. The final topic for this chapter brings the Model and View together again.

This material was excerpted from the book [Ajax in Action](#) from Manning Publications. In the next installment, we will continue with this chapter by discussing a way of reducing overhead in an Ajax application by generating the View layer automatically from the Model.