CHAPTER 4

# AJAX COMPONENTS

In this chapter, after examining several patterns, we look at how they apply to actually building a user interface. You learn how to encapsulate AJAX functionality into both imperative, as well as declarative, components. The use of declarative components is increasingly important because various new declarative technologies are created, such as Scaling Vector Graphics (SVG), XML Binding Language (XBL), and Macromedia XML (MXML). The encapsulation of user-interface functionality is a critically important aspect of enterprise AJAX development because it not only facilitates code re-use, but it also removes much of the need for addressing the individual quirks of multiple browsers—a critical step toward rapidly developing high-quality, rich AJAX applications.

We can build an application using conventional classes, some aspect-oriented programming, the DOM, and DOM Events. Until now, our code has, for the most part, been cobbled together using our MVC architecture. The next step is to refractor our Customer list application into something more modular and componentized so that we can re-use the code across an application, or even throughout the enterprise.

By the end of this chapter, we will have converted our customer listing AJAX application into a full-fledged declarative AJAX component. We also look at a few of the available client-side AJAX frameworks.

## Imperative Components

Now that you have a clear idea of how to get your JavaScript running when a web page loads, you can look at how to actually use JavaScript, the DOM, and CSS to make an AJAX component. If you have any experience in server-side programming, you are probably familiar with writing code in an imperative manner. Imperative programming is what most developers are familiar with and is a sequence of commands that the computer is to execute in the specified order. We can easily instantiate a component with

JavaScript by creating a new object and, as is often the case, subsequently specify an HTML element through which the View can be rendered—this would be an imperative component implemented through JavaScript.

Imperative coding is much like making a ham-and-cheese sandwich. To end up with a ham-and-cheese sandwich, you need to follow certain steps:

1. Get the bread.
2. Put mayo and mustard on the bread.
3. Put the ham and cheese on the bread.
4. Close the sandwich.
5. Enjoy!

If you try to close the sandwich at a different stage or put the ham and cheese on the bread before the mayo, you might end up with a mess! This equally applies to writing JavaScript or AJAX in an imperative manner.

A good example of an imperative JavaScript component, that some of you might have used, is the popular Google Map component that we look at how to work with through JavaScript. People generally integrate a Google Map with their own application, building a so called mashup, all using imperative JavaScript code. Although it might seem out of place, it can be useful to include public AJAX applications such as Google Maps in an enterprise setting. Google Maps are extremely useful for visualization of geographic data such as shipment tracking, fleet tracking, or locating customers. At any rate, to begin with, as with any JavaScript component, you need to ensure that the JavaScript libraries provided by Google are included in the web page. In the case of Google Maps, the JavaScript code can be included by using a single `<script>` element `<script>` element; Google Maps such as the following:

```
<html>
  <head>
    <script src="http://maps.google.com/maps?
  file=api&v=2&key=#INSERT_KEY_HERE#"
type="text/javascript"></script>
  </head>
  <body>
    <div id="map" style="width: 370px; height: 380px"></div>
  </body>
</html>
```

To use the Google Maps service, as with many other publicly available AJAX components or web-based data sources, you need to register with Google to get an API key that is passed to the Google service as a query-string parameter in the script location. Having loaded the script from the Google server and using at least one of the bootstrapping techniques from the previous section, you might create a Google Map like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:v="urn:schemas-microsoft-com:vml">
  <head>
    <style type="text/css">
    v\:* {behavior:url(#default#VML);}
    </style>
    <script
src="http://maps.google.com/maps?file=api&v=2&key=#INSERT_KEY_
HERE#" type="text/javascript"></script>
    <script type="text/javascript">
var gmap = {};

function gmap.init()
{
  var map = new GMap2(document.getElementById("map"));
  // Center on Vancouver
  map.setCenter(new GLatLng(49.290327, -123.11348), 12);
}

// Attach the init function to window.onload event
entAJAX.attachAfter(window, "onload", gmap, "init");
    </script>
  </head>
  <body>
    <div id="map" style="width: 370px; height: 380px"></div>
  </body>
</html>
```

There is a considerable amount of overhead here, such as the XHTML doctype and the reference to the Vector Markup Language (VML) behavior that is used for Internet Explorer; the important parts are the inclusion of the external Google Maps JavaScript file and the `init()` function that creates a new map and sets the map center to be Vancouver. The map is placed inside of the DOM element that has an id value of "map." When an instance of the GMap2 class has been created, you can access its various properties and methods through the exposed JavaScript API. Here, we show how a GPolyLine object can be added to the map using an array of GLatLng points:

```
var polyline = new GPolyline([
  new GLatLng(49.265788, -123.069877),
  new GLatLng(49.276988, -123.069534),
  new GLatLng(49.276988, -123.099746),
  new GLatLng(49.278108, -123.112106),
  new GLatLng(49.2949043, -123.136825)], "#ff0000", 10);

map.addOverlay(polyline);
```

The result of imperatively creating this Google Map, as shown in Figure 4.1, is an impressive and highly interactive map centered on Vancouver with a route through the city that looks something like this:

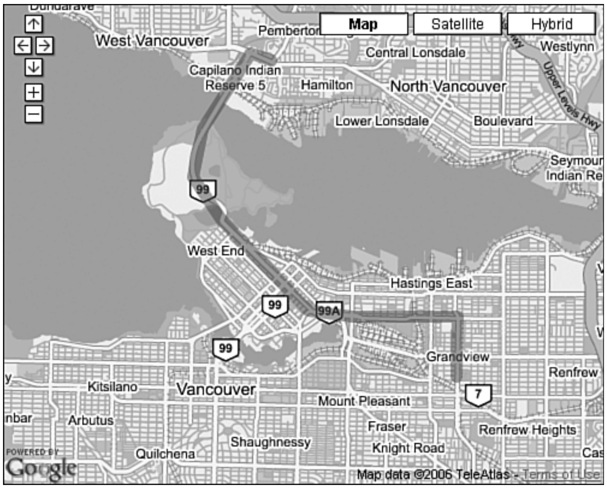

Figure 4.1 Path Displayed on a Google Map Using the Google Map API

The type of JavaScript code required to create a Google Map in a web application is exactly the sort of code you might expect to see in any common user-interface development language. In fact, looking at the code, you might think that it is written in a server-side language. Although today, imperative coding might be the norm; going forward, AJAX development will become increasingly declarative. This is certainly reflected in the fact that companies such as Microsoft and Adobe are pursuing those avenues with XML-Script and Spry, respectively—not to mention the next generation technologies from both of those companies again in WPF/E and Flex, which are both based on XML declarative programming models. Google Maps is a quintessential imperative AJAX component; however, to get a good grasp of declarative programming, let's look at how to convert a Google Map to be a declarative component.

## Declarative Components

Although defining components in an imperative manner can be useful, it is also increasingly common to see components defined using a declarative approach. You probably already know at least one declarative language; HTML and XSLT are two common examples of declarative languages. When using declarative components, the developer doesn't need to worry about how things are achieved behind the scenes but instead only needs to worry about declarative structure; for example, in HTML, the web browser parses and interprets the tags based on some predefined set of rules. The result of this is that when the HTML parser finds text surrounded in `<em>` tags, it presents that text with **emphasis**. Exactly how the text is emphasized by default is left up to the web browser, although that can, of course, be overridden by the developer using CSS. Because the markup or declaration specifies *what* a component does rather than *how* it works is the biggest advantage to declarative programming.

When discussing imperative coding, you learned that making a ham-and-cheese sandwich ended up being a bit of a pain to achieve the right outcome. On the other hand, a ham-and-cheese sandwich created using a declarative approach would go something more like this:

1. Ham and cheese sandwich please.
2. Enjoy!

Rather than having to specify each of the steps involved in making the sandwich, it is more like going to your local café and ordering the sandwich from the waiter. It is certainly fewer steps and probably a lot more convenient to make the sandwich declaratively rather than imperatively; however, there are some drawbacks. The most apparent drawback here is that if you aren't careful, the waiter might bring you a ham-and-cheese sandwich without any mustard!

You might be familiar with declarative programming from any one of the gamut of server-side web application frameworks employing a declarative approach, such as JavaServer Faces, JSP, and ASP.NET. In these languages, you can define a part of the page using a declarative syntax that is then processed on the server and produce standard HTML that is delivered to the client like any other web page.

## Server-Side Declarative Programming

In ASP.NET, you can define a web page with a declarative DataGrid control like this:

```
<asp:DataGrid id="ItemsGrid" BorderColor="black"
  BorderWidth="1" CellPadding="3"
  AutoGenerateColumns="true" runat="server">
</asp:DataGrid>
```

What happens to this declaration is that the .NET framework loads the ASPX web page containing the declaration, and the declaration is processed and replaced with regular HTML by the server, which then gets streamed up to the client as though it were plain HTML page. Of course you can see that there is certainly more to the story than just that simple declaration because there is no mention of what data is to be rendered in the DataGrid. Although these various server-side technologies do provide a nice declarative interface, they still require a little bit of code to hook everything together. Behind the scenes of the ASPX HTML page is a code page that might have some C# code such as this to connect the DataGrid to a database:

```
// Define the DataGrid
protected System.Web.UI.WebControls.DataGrid ItemGrid;
private void Page_Load(object sender, System.EventArgs e)
```

```
{
  ItemGrid.DataSource = myDataSet;
  ItemGrid.DataBind();
}
```

By combing the declarative and imperative approaches, developers get the best of both worlds, enabling them to develop a simple application rather quickly, still having the control to tweak all aspects of the application components.

There are many advantages to taking a declarative approach to building applications. The most obvious advantage of markup is that it is more "designable" than code in that it enables far better tool support. ASP.NET or JavaServer Face components in Visual Studio or JavaStudio Creator are good examples of this where you can drag components into a web page during application design and visually configure without writing code.

The fact that a declaration is just a snippet of XML means that XML Schema can be used to ensure that a declaration adheres to an expected XML structure. Validating against a rigid XML schema makes declarative components much less error prone than the pure JavaScript counterparts. Writing declarations in a web editor such as Eclipse or Visual Studio can also be made easier by using autocomplete features (for example IntelliSense for Visual Studio) that ensure the declaration adheres to the XML schema as the declaration is being written. In fact, at some point, things can become even more simplified because a DataGrid in one declarative framework, like Adobe's MXML language, is little more than an XSLT transformation away from a DataGrid in some other language like XForms—thus, achieving the same functionality across platforms without changing and recompiling a single line of code. Of course, with some effort, this can be said of almost any programming language; however, declarative programming does have the advantage that the order in which statements are declared has no impact on the operation of the component, and declarations are XML-based and, therefore, readily machine readable.

Although a declaration can get a developer most of the way to building a great application, there is always that last little bit that requires more fine control to customize a component in specific ways. In these instances, you can still fall back on the programming language that the declarative framework is build on, be it Java, C#, or JavaScript.

## Declarative Google Map

A declaration is just an abstraction layered over the top of imperative code. Elements in a declaration roughly map to objects and attributes to fields or properties on those objects. Although a declaration does not specify anything about the methods of an object, and it shouldn't, it can express everything about the *state* of an object or, perhaps more familiar to you, the serialized form of an object. In the case of our imperative Google Maps example, we create, set up, and render a Google Map entirely through error prone and uncompiled JavaScript resulting in a map that has a certain zoom level and is centered on some lat/long coordinates. Ideally, we can instead take an XML description of the map containing all the information about the map—zoom level, center coordinates, and so on—and instantiate a map based on that state information stored in the XML. So, rather than defining our Google map with JavaScript, you can use a custom XHTML declaration that describes the state of a serialized map, which gets deserialized (by some code that you can write) resulting in a map as though you had explicitly written the JavaScript code. A Google Map declaration based on the imperative code we wrote previously might look something like this:

```
<g:map id="map" width="370px" height="380px"
smallmapcontrol="true" maptypecontrol="true">
  <g:center zoom="14">
    <g:point lat="49.2853" lng="-123.11348"></g:point>
  </g:center>
  <g:polyline color="#FF0000" size="10">
    <g:point lat="49.265788" lng="-123.069877"></g:point>
    <g:point lat="49.276988" lng="-123.069534"></g:point>
    <g:point lat="49.276988" lng="-123.099746"></g:point>
    <g:point lat="49.278108" lng="-123.112106"></g:point>
    <g:point lat="49.294904" lng="-123.136825"></g:point>
  </g:polyline>
</g:map>
```

The parallels between this declaration and the imperative code are clear—almost every line in the declaration can be identified as one of the JavaScript lines of code. The biggest difference is that, as we have discussed, the declaration specifies only *how* the map should be displayed independent of any programming language and in an industry-standard, easily machine-readable, and valid (according to an XML Schema) format.

The actual code used to convert that to an instance of a Google Map is left up to the declaration processor, which again, can be implemented in any language or platform—in our case, we stick with the web browser and JavaScript. Furthermore, the dependence on order of statements in imperative coding—that you must create the map object before setting properties on it—is masked by the nested structure of the XHTML declaration, making it unnecessary for a developer to understand any dependencies on the order in which code is executed. However, they must understand the XML schema for the declaration. Let's take a closer look at what we have defined here for our map declaration.

First, we defined the root of our declaration using a DOM node with the special name of `<g:map>` `<g:map>` DOM node where the `g` prefix is used to specify a special namespace. This makes the HTML parser recognize those tags that don't belong to the regular HTML specification. When the component is loaded from the declaration, we want it to result in a Google Map created in place of the declaration, and that map will have the specified dimensions, zoom level, and center point. Similarly, it will result in a `polyline` drawn on the map with the given color and start and end points. The only trick is that we need to write the JavaScript code to go from the declaration to an instance of a map!

Because the web browser has no knowledge of our custom XHTML-based declaration, it does not have any built-in code to find and create our component based on the declaration. To go from our component declaration to an instance of an AJAX component, we need to use almost all the technologies that we have learned about so far. To start with, we need to bootstrap using one of the techniques discussed in Chapter 3, "AJAX in the Web Brower,"—the exact same as we would need to do to with an imperative component. Our Google Map sample page now becomes the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:v="urn:schemas-microsoft-com:vml"
    xmlns:g="http://www.enterpriseAJAX.com/gmap">
  <head>
    <link rel="stylesheet" href="gmaps.css"
type="text/css"></link>
    <script
src="http://maps.google.com/maps?file=api&v=2&key=#INSERT_KEY_
HERE#" type="text/javascript"></script>
```

```
    <script type="text/javascript"
src="entajax.toolkit.js"></script>
    <script type="text/javascript" src="gmaps.js"></script>
  </head>
  <body>
    <g:map id="map" width="370px" height="380px"
smallmapcontrol="true" maptypecontrol="true">
      <g:center zoom="14">
        <g:point lat="49.2853" lng="-123.11348"></g:point>
      </g:center>
      <g:polyline color="#FF0000" size="10">
        <g:point lat="49.265788" lng="-123.069877"></g:point>
        <g:point lat="49.276988" lng="-123.069534"></g:point>
        <g:point lat="49.276988" lng="-123.099746"></g:point>
        <g:point lat="49.278108" lng="-123.112106"></g:point>
        <g:point lat="49.294904" lng="-123.136825"></g:point>
      </g:polyline>
    </g:map>
  </body>
</html>
```

There is now no sign of any JavaScript on the web page, but we added two additional external JavaScript files that are responsible for parsing the declaration, moved the special CSS into an external file, and added the declaration itself. This is the sort of page that a designer or someone unfamiliar with JavaScript could write.

The included `entajax.toolkit.js` file contains all the helper classes and functions that we need to make this work on Firefox, Internet Explorer, and Safari. In `gmaps.js` is where all the magic happens. The contents of `gmaps.js` looks like this:

```
entAjax.initComponents = function()
{
  // Iterate over all pre-defined elements
  for (var tag in entAjax.elements)
  {
    // Get the all the <G:*> elements in the DOM
    var components = entAjax.html.getElementsByTagNameNS(tag, g);
    for (var i=0; i<components.length; i++)
    {
      // A custom element is only initialized if it is a root
node
      if (entAjax.isRootNode(components[i]))
      {
        // Call the defined method that handles such as
component
        entAjax.elements[tag].method(components[i]);
      }
    }
  }
}
entAjax.attachAfter(window, "onunload", entAjax,
"initComponents");
```

The `initComponents()` method depends on a few things. First, to facilitate a flexible and extensible approach to building JavaScript components from XHTML, we use a global hash where the keys are the expected HTML element names and the values contain additional metadata about how to deserialize that XHTML element into a JavaScript. This approach is analogous to a more general technique that can deserialize a component based on the XHTML schema. For a small, representative subset of the available parameters that can be used to create a Google Map, the `entAjax.elements` hash might look something like this:

```
entAjax.elements = {

"map":{"method":entAjax.createGMap,"styles":["width","height"]
},

"smallmapcontrol":{"method":entAjax.createSmallMapControl},
    "maptypecontrol":{"method":entAjax.createMapTypeControl},
    "polyline":{"method":entAjax.createPolyline},
    "center":{"method":entAjax.centerMap}};
```

We have defined five keys in the `entAjax.elements` hash that are `map`, `smallmapcontrol`, `maptypecontrol`, `polyline`, and `center`. For each of these keys, which relate to expected DOM node names, we define an object with a `method` field and a possible `styles` field. The method refers to the JavaScript function used to deserialize the DOM node with the specified node name, and the styles is an array that we use to map possible attributes from the `<g:map>` element to CSS style values—in this case, we want to transform `<g:map width="370px" height="380px">` to an HTML element that looks like `<div id= "map-1" style="width:370;height:380px;">`.

We used the `entAjax.getElementsByTagNameNS` function to obtain references to the custom XHTML elements rather than the native DOM `getElementsByTagNameNS` method. The reason for this is that Internet Explorer does not support element selection by namespace, and other browsers such Firefox, Safari, and Opera use it only when the web page is served as XHTML, meaning that it must have content-type `application/xhtml+xml` set in the HTTP header on the server. Internet Explorer has one more quirk in that it completely ignores the element namespace and selects elements based entirely on the local name, such as "map." On the other hand, other browsers accept a fully qualified tag name such as "g:map" when not operating as XHTML. The `entAjax.getElementsByTagNameNS` function effectively hides these browser nuances.

After getting a list of all the custom tags in the web page, we then use the tag constructor definitions in the `entAjax.elements` hash to find the method that we have written to instantiate that element into the equivalent JavaScript object.

```
entAjax.elements[tag].method(components[i]);
```

We pass one argument to the root tag constructors, which is the declaration element from which the entire component can then be built. Each of the methods in the `entAjax.elements` hash can be thought of as factories according to the `Factory` pattern. In the case of the `<g:map>` XHTML element, the `createGMap` function is called. The `createGMap` function is a custom function used to create an instance of the `GMap2` class as well as set up all the child controls and relevant properties:

```
entAjax.createGMap = function(declaration) {
  var container = document.createElement('div');
  entAjax.dom.insertAdjacentElement("afterEnd", declaration,
container);
  // Move any declaration attributes to the Map style
  parseStyle(entAjax.elements["map"].styles, declaration,
container);
  var gmap = new GMap2(container);
  // Iterate over attributes on DOM node
  forAttributes(declaration, function(attr) {
    container.setAttribute(attr.nodeName, attr.nodeValue);
    if (entAjax.elements[attr.nodeName] != null)
      entAjax.elements[attr.nodeName].method(gmap, attr);
  });
  // Iterate over child DOM nodes
  forChildNodes(declaration, function(elem) {
    entAjax.elements[formatName(elem.nodeName)].method(gmap,
elem);
  });
}
```

For each `<g:map>` element, we create a standard `<div>` element `<div>` elements to which the map will be attached. This will generally be the case that a component needs to be attached to a standard HTML `<div>` element and then create an instance of the `GMap2` class with the newly created `<div>` element as the single constructor argument. Two general operations need to be performed for declaration parsing; first, all attributes on the declaration node must be processed, and second, all child elements of the declaration node need to be processed. Due to the way that the `GMap2` component was designed, we also need to copy some custom style information from the declaration node, such as the width and height, onto the `<div>` container element. Many of these special cases can be generalized in a component framework but are much less elegant when wrapping declarative functionality around a JavaScript component built without declarative functionality in mind.

## Alternative Approaches

Although we used custom XHTML for our declaration, it is also possible to use other techniques for configuring your components. The most popular alternative to configuring components with an XML-based declaration is to use a simple JavaScript object. For our map example, the following would be a likely format for a map configuration:

```
var configuration = {"map":{
  "center":{
    "zoom":10,"point":{"lat":23,"lng":-122}
  },
  "polyline":{
    "color":"#FF0000","size":10,"points":[
      {"lat":49.265788,"lng":-123.069877},
      {"lat":49.276988,"lng":-123.069534}
]}}}
```

This configuration can then be used as the single argument passed to the map factory and would result in a map just the same as the XHTML declarative approach we outlined. Using a JavaScript object such as that is the way that Yahoo's AJAX user-interface components accept configurations.

Another way to configure an AJAX component, although it is currently fairly uncommon, is to use CSS properties. Using CSS to configure AJAX components is particularly effective because CSS can be linked through external files using the HTML `<link>` element, and the technology is familiar to most web designers today. However, CSS does have considerably less expressiveness when compared to either a JavaScript object or an XHTML declaration, and some dynamic CSS functionality is not available in browsers such as Safari. Chapter 2, "AJAX Building Blocks," covered how to dynamically access and manipulate stylesheet rules through the DOM API.

Looking at the Google Map example and seeing how to convert an existing component to a declaration should have been helpful in identifying not only how a declarative approach can make AJAX development easier, but how we can use it to build rich Internet applications. Having gone through this exercise with a Google Map, there might be a few questions in your head now such as how we can deal with events, data binding, or data templating in a declarative manner. We look at of those issues and more in the next section.

## Custom Declarative Component

Now that you have had a chance to consider how a declarative approach might work using a well-known AJAX component as an example, you will build your own custom AJAX declarative component. Let's go through the steps of building an AJAX DataGrid control, which is a useful piece of user interface functionality and is used to iterate over a list of JavaScript objects, such as a list of Product objects, and render each item as a row in a table, applying a common style or formatting to each item. Many server frameworks such as JSF and ASP.NET have DataGrid components that can be attached to a list of objects or a database query and display those objects or records in the user interface. There are also fully client-side alternatives that can connect to the server using AJAX. For now, we keep it simple and look at how to build a declarative AJAX user interface component while using OOP design patterns and applying MVC principles.

The first type of declarative component we look at is exceedingly simple—in fact, so simple that it is entirely based on HTML markup and CSS. In this case, the output of the "component" is a product of explicitly stating all the columns and data for a table of product information. Although this might seem like a strange place to start, HTML is actually the definitive declarative markup. Each element in the declaration has a `class` attribute that connects the HTML to the styling information contained in the associated CSS, and each element has an `id` attribute that is used for both styling and for making the elements uniquely addressable from JavaScript. Markup for an HTML DataGrid might look like this:

```
<table id="myGridList" class="gridlist">
  <thead>
    <tr id="header" class="header-group">
      <td id="header-0" class="header header-0">Product</td>
      <td id="header-1" class="header header-1">Price</td>
    </tr>
  </thead>
  <tbody>
    <tr id="row-0" class="row">
      <td id="cell-0_0" class="column column-0">Acme
Widget</td>
      <td id="cell-0_1" class="column column-1">$19.99</td>
    </tr>
    <tr id="row-1" class="row row-alt">
      <td id="cell-1_0" class="column column-0">Acme Box</td>
      <td id="cell-1_1" class="column column-1">$9.99</td>
    </tr>
    <tr id="row-2" class="row">
      <td id="cell-2_0" class="column column-0">Acme
Anvil</td>
      <td id="cell-2_1" class="column column-1">$14.99</td>
    </tr>
  </tbody>
  <tfoot>
    <tr id="footer" class="footer-group">
      <td id="footer-0" class="footer footer-0">Total</td>
      <td id="footer-1" class="footer footer-1">$43.97</td>
    </tr>
  <tfoot>
</table>
```

We use HTML `<table>` elements (see Figure 4.2) as opposed to the `<div>` elements for accessibility reasons—more on this in Chapter 7, "Web Services and Security." As you can tell from the HTML declaration itself, the result is a basic layout of each product item, column headers, and footers. There is nothing magical about this however, this is an important step in designing a declarative component because it gives you a clear idea of the HTML structure that you want to achieve.

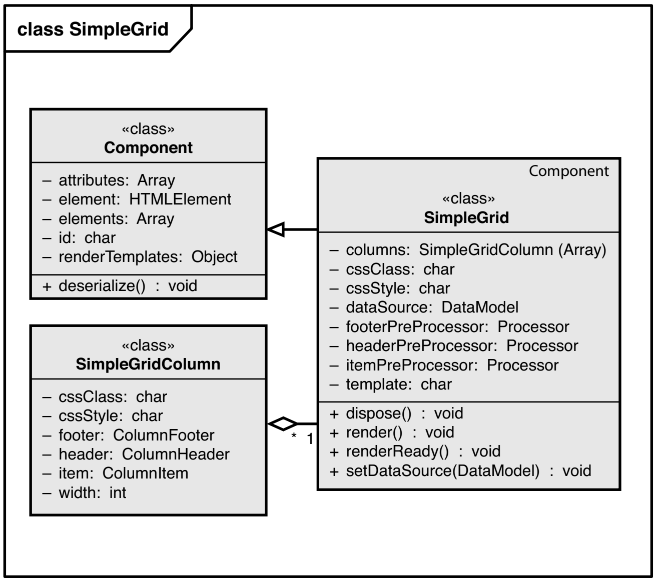

**Figure 4.2 Simple HTML Table Displays a List of Products with Names and Prices**

The CSS classes specified in the HTML provide the styling information to make the header and footer contents bold, make the data background color alternate, and specify the grid dimensions. All the styling information for your component or application needs to be powered by CSS, which is no different for an AJAX application from a traditional post-back HTML application. The contents of the CSS file for the list of previous products looks like this:

```
.theme #myGridList {
  width:200px;
  border:1px solid black;
}
.theme .columnheader-group, .theme .columnfooter-group {
  height:20px;
  font-weight:bold;
  border-bottom:1px solid black;
}
.theme .columnheader, .theme .column, .theme .columnfooter {
  float:left;
  overflow:hidden;
}
.theme .columnheader-0, .theme .column-0, .theme
.columnfooter-0 {
  width:100px;
}
.theme .columnheader-1, .theme .column-1, .theme
.columnfooter-1 {
  width:50px;
}
.theme .row, .theme .column {
  height:18px;
}
.theme .row-alt {
  background-color: #E5E6C6;
}
```

Notice a few things here. First, we used a descendent selector to differentiate the given styles based on the `theme` class; the styles listed will be applied only to elements that have an ancestor element with the `theme` class. Something else that influences design decisions significantly is that in the case of a DataGrid, and most any databound component for that matter, we need to define a custom CSS rule for each of the user-specified columns in the DataGrid and for each of the regions of a column (that is, the header, data, and footer). In the extreme, we could even specify CSS rules for each row of data, as might be the case if we were building a spreadsheet component. This is where dynamic creation of styles can come in handy, and browsers that don't support dynamic style functionality, such as Safari, start to become liabilities to your application performance and ease of development. The alternative to using CSS classes for specifying the column widths is to set the style attribute on the column directly.

Although you now have a nice-looking DataGrid populated with data, building the component explicitly in HTML is not helpful because the data and structure are all combined making it difficult to change any one aspect of the data or presentation. However, there is a reason we started by looking at a static HTML table. By laying out the HTML, it helps to get a better grasp of how to merge the data, structure, and styling in the most efficient manner both from the point of view of HTML footprint and computational complexity. To ensure that the HTML footprint is small, we used as few HTML elements as possible—only those necessary for laying out the data—and have used both ID and class-based CSS selectors to apply all styling to the data. We have taken advantage of the fact that we can apply a single style to multiple elements; in particular, we use this feature to set the width of the column header, data, and footer all in one CSS rule such as the following:

```
.columnheader-0, .column-0, .columnfooter-0 {width:100px;}
```

## Behavior Component

Another reason that we started by looking at hard-coded HTML to make an AJAX component is that we can make a good start by adding AJAX to our hard-coded HTML using a behavioral approach. This can be a good approach if things such as graceful failure are important because it can easily fall back on the standard HTML markup in cases when users have web browsers that either don't have JavaScript enabled or don't support certain core AJAX functionality such as the XHR object. These are becoming more and more unlikely, particularly in the enterprise where computer software is generally centrally configured and managed. Similarly, behavioral components enable you to incrementally add AJAX features to rendered HTML where the HTML might have come from some existing server-side framework, enhancing the value of your existing web application assets. The Microsoft AJAX Framework (formerly known as Atlas) has a considerable focus on behavioral components. For example, rather than creating an autocomplete component, the Microsoft AJAX Framework has an autocomplete behavior, quintessential AJAX, that can be applied to any existing component such as a standard HTML `<input>` element. In this case, much like a dynamic `<select>` element, it adds the ability to allow users to dynamically choose words from a database as they type in the `<input>` control. For our DataGrid component, adding some grid-like behavior to the standard HTML might entail enabling the end user to edit or sort data in the grid.

Defining behavioral components is often done declaratively by extending the default HTML markup and setting some metadata on a standard HTML element. For the most part, this metadata consists of a CSS class name that generally does not have any actual style information associated with it. To instantiate a behavioral component, the bootstrapper scours the DOM to find elements with recognizable metadata—be it a CSS class name or otherwise. When an element with known metadata is discovered, the standard component instantiation commences. Markup for a sortable HTML table can have an additional class, such as `sortable`, which would look like this:

```
<table id="mySortableTable" class="gridlist sortable"></table>
```

The code to actually attach the sorting behavior to the HTML element uses the popular `getElementsByClassName()` function for which there are several different custom implementations or approaches. Because it is such a popular function, we shortened the name to `$$`. We can use the `$$` function in our bootstrapper along with the `makeSortable()` function to add the sorting behavior to our HTML table.

```
function initSortable()
{
  entAjax.lang.forEach(
    entAjax.html.$$("sortable"),
    entAjax.behaviour.makeSortable
  );
}
```

```
entAjax.behaviour.makeSortable = function(table)
{
  entAjax.forEach(table.rows[0].cells,
    function(item) {
      item.className += " button";
    });
  entAjax.html.attachEvent(table.rows[0], "click",
    entAjax.lang.close(table, sort));
}
```

For a sortable HTML table, we require the `makeSortable()` function to do a few things. Each table that we want to have made sortable needs to have an additional class added to each header cell and an event handler function attached to the click event of the table header. To indicate to the end user that they can click on the column header to sort the table by that column, we add the `button` class that changes the users' mouse cursors to a hand icons; when they click, it causes the global `sort()` function to be executed in the context of the HTML `<table>` element. (You remember from Chapter 2 that running the event handler in the context of the HTML element means the `this` keyword refers to the HTML element that makes writing the sort function a bit easier.)

The sorting of data is something that we should all remember from Computer Science 101. JavaScript is no different from other languages in this regard, and we use the familiar bubble sort algorithm to order our table rows. We can also consider using the JavaScript array sorting functionality; however, it requires a bit more tedious overhead such as copying values between arrays and the like. The `sort()` function is shown here:

```
function sort(evtObj, element)
{
  var aRows = this.rows;
  var nRows = aRows.length;
  var nCol = getCol(evtObj.srcElement);
  var swapped;
  while (true)
  {
    swapped = false;
    for (var i=1; i<nRows-2; i++)
    {
      var sValue1 =
aRows[i].cells[nCol].getAttribute("value");
      var sValue2 =
aRows[i+1].cells[nCol].getAttribute("value");
      if (sValue1 > sValue2)
      {
        a.parentNode.insertBefore(a,
entAjax.dom.getNextSibling(b));
        swapped = true;
      }
      else
      {
        swapped = false || swapped;
      }
    }
    if (!swapped) break;
  }
}
```

Because the `sort()` function is executed in the context of the HTML table element, we can access the collection of table rows using the native table `rows` property and similarly access the collection of cells in each row using the `cells` property. To get the value that is rendered in each cell of the table, rather than using something such as `innerHTML` that returns the rendered value of the cell, we instead get the custom `VALUE` attribute that we created ourselves (this might be an instance where you want to use a custom namespaced attribute) and which contains the raw, unformatted data. This is an important consideration when we deal with things such as prices that might be prepended with a `"$"` character for rendering but sorted as numbers. Having said that, after we dynamically connect our table to a datasource, this will no longer be necessary. Finally, we use some more native DOM manipulation methods such as `element.insertBefore(newNode, refNode)`. The `insertBefore()` method makes sorting the rows quite simple in that we can use that method with DOM nodes that are already rendered—in this case, the table rows—and it actually moves those nodes and re-renders them.

That is all there is to building a small behavioral AJAX component that can be layered on top of an existing web application. The entire idea behind behavioral components is gaining popularity from the world of semantic markup and other technologies such as Microformats. Strictly speaking, a Microformat is not a new technology but instead a set of simple data formatting standards to provide more richly annotated content in web pages. Microformats use the same CSS class extension approach to give general XHTML content more semantic information. Microformats and other burgeoning standards such as the W3C endorsed RDFa are great places to watch to get an idea of where web technologies are heading and finding the best way to create declarative AJAX components.

At any rate, behavioral AJAX using HTML declarations sprinkled with some additional metadata can be a great approach for AJAX development because it can be achieved in an incremental manner, thus avoiding any large up-front investment in training or technology. It can be a great way to test the AJAX waters before a more large scale deployment. Of course, there are still other ways to use your existing architecture when moving toward AJAXifying your applications.

## Declarative Component

The next step beyond a behavioral component that uses HTML markup as the declaration is to create an abstraction of the HTML markup so that you can do more than just add some simple sorting or editing functionality. Using a custom-designed declaration means you can actually generate and output the HTML markup in the web browser populated with data from a client-side datasource—this will be your fully declarative client-side AJAX solution. You need to consider a few aspects when making a custom declarative AJAX component or application. For some insight into these issues, as we have already mentioned, it is always a good idea to look at existing recommendations and specifications put forward by the W3C—no matter how esoteric or generally unrealistic they might sometimes seem. It seems more often than not that just because AJAX seems shiny and new, people tend to forget that most of what they want to do has been figured out in the past in one context or another.

When it comes to creating a declarative AJAX solution, you can look for inspiration in a number of places. From looking at the many good examples of declarative frameworks currently available from private vendors such as Microsoft (XML Application Markup Language) and Adobe (Flex MXML) as well as the W3C (XForms, Scalable Vector Graphics, XML Binding Language), two common themes appear in all of them. These themes are data binding—defining how and where data shows up in a user interface and data templating—defining how the data is formatted in the user interface. We look at some existing solutions and some ideas for custom JavaScript approaches to both of these problems.

### Databinding

A good solution for databinding can be a difficult thing to achieve. By "good," we mean a solution that is flexible and provides multiple levels of indirection so that we can build complex data-bound components. To start with, let's take a quick look at a few of the data-binding solutions that have been prevalent on the web over the past decade.

#### Internet Explorer Databinding

Since version 4.0 came out, Internet Explorer has had client-side databinding functionality baked into the browser. Although it is nothing too advanced, Internet Explorer does provide basic data-binding functionality by supporting two custom HTML attributes—the `DATASRC` and `DATAFLD` attributes—on several different HTML elements. The `DATASRC` attribute specifies a client-side datasource to which the element is bound whereas the `DATAFLD` attribute specifies the specific field in the datasource to which the value of an HTML element is bound. The most common HTML element to bind to a datasource is, as in our behavioral example, the `<table>` element, which is usually found bound to a repeating list of data where the list of data is repeated in `<tr>` elements of the table. A data bound `<table>` element might look like this:

```
<table datasrc="#products">
  <thead>
    <tr><td>Product</td><td>Price</td></tr>
  </thead>
  <tbody>
    <tr>
      <td><span datafld="name"></span></td>
      <td><span datafld="price"></span></td>
    </tr>
  </tbody>
</table>
```

Because the `<td>` element is one that does not support the `datafld` attribute, we use a `<span>` tag that is bound to a field from the datasource. Datasources themselves can be various structures; the most popular of which is likely the XML data island that looks like this:

```
<xml id="products" src="products.xml"></xml>
```

Although this is a useful technology, there is still much to be desired, and it provides little more than a stop gap when it comes to building true RIAs. More recently, W3C-supported technologies such as XForms and the XML binding language (XBL) are excellent examples of thorough approaches to declarative components and databinding in the web browser.

### *XForms Databinding*

One of the most mature options on the list is XForms.[1] XForms 1.0 became a W3C recommendation in October 2003 and has not moved much beyond that. There are some real advocates of the technology, but it is yet to be championed by mainstream browsers.

In the XForms world, there are Models and controls (or Views). Models define the data and controls that are used to display the data. To bind the View to the Model, a few important declarative attributes need to be understood. First, you have single-node binding attributes. These define a binding between a form control or an action and an instance data node defined by an XPath expression. On an XForms control bound to a single data node, there can be either a `REF` and a `MODEL` attribute or a `BIND` attribute. The `MODEL` and `REF` attributes together specify the ID of the XForms Model that is to be associated with this binding element and the XPath of the data within that Model, respectively. Alternatively, this binding information might be contained in a completely separate declaration that can be referenced using the value of the third attribute of interest that has the name `BIND`.

When you want to bind to a list of data rather than a single value, you can bind to a node-set rather than a single node in the Model. The `NODESET` attribute, much like the `REF` attribute, specifies the XPath to the nodes-set to which the control is bound. Again, either a `MODEL` attribute is required to go along with the `NODESET` attribute or a `BIND` attribute can refer to a separate binding declaration.

Binding declaration elements, rather than just those four attributes, provide a more complete set of options for specifying how the binding to the data is to take place. The `<BIND>` element connects a Model to the user interface with these additional attributes:

**calculate**—Specifies a formula to calculate values for instance data

**constraint**—Enables the user to specify a predicate that must be evaluated for the data to considered valid

[1]http://www.w3.org/TR/xforms

**required**—Specifies if the data required

**readonly**—Specifies if the data can be modified

**type**—Specifies a schema data-type

A final consideration is the evaluation context of the `REF` or `NODESET` XPath expressions. The context for evaluating the XPath expressions for data binding is derived from the ancestor nodes of the bound node. For example, setting `REF="products/product"` on a parent node results in the evaluation context for XPath expressions of descendent nodes to be that same path in the specified `MODEL`. For a select form element, you can use the `<ITEMSET>` element to define a dynamic list of values that are populated from the Model with ID `products`, and the selected products are saved in the Model with ID `order`.

```
<select model="order" ref="my:order">
  <label>Products</label>
  <itemset model="products"
nodeset="/acme:products/acme:product">
    <label ref="acme:name"/>
    <value ref="acme:name"/>
  </itemset>
</select>
```

Because of the evaluation context, the `<LABEL>` and `<VALUE>` `REF` XPath values are evaluated in the context of their direct parent node, which is the root node of the `products` Model.

There are still more examples of declarative programming in the multitude of server or desktop languages that we could investigate such as .NET Web Forms, JavaServer Faces, Flex MXML, XUL, Laszlo, and XAML. What we can say is that most of these technologies are driven by the MVC pattern with extreme care taken to separate the Model and View. Like XForms, most also rely on XML-based data and leverage standards such as XPath and XSLT to achieve the rich functionality that you would expect from an RIA. In particular, some common threads in many of the new languages are the use of XPath in databinding expressions and the inheritance of the XPath execution context within the Model.

### *Templating*

The second important area of building declarative components is templating of data. Templating of data is important if reuse is a priority because it should enable a high degree of customization to the component look and feel. Choosing a robust templating mechanism is a real key to creating flexible and high-performance AJAX applications and components. A few different JavaScript templating libraries are available on the web, the most popular of which is likely the JST library from TrimPath. As with many script-based templating languages (think ASP and PHP, for example), it inevitably turns out to be a mess of interspersed JavaScript code and HTML snippits—actually no different from writing JavaScript by hand. A JST-based template might look something like this:

```
Hello ${customer.first} ${customer.last}.<br/>
<table>
  <tr><td>Name</td><td>Price</td></tr>
  {for p in products}
    <tr>
      <td>${p.name}</td><td>${p.price}</td>
    </tr>
  {forelse}
    <tr><td colspan="2">No products in your cart.</tr>
  {/for}
</table>
```

As mentioned, this “template” looks rather similar to what you might use if you were to generate the HTML by standard string concatenation like this.

```
var s = "";
s += "Hello "+ obj.customer.first+"
"+obj.customer.last+".<br/>";
s += "<table>";
s += "<tr><td>Name</td><td>Price</td></tr>";
for (var i=0; i<obj.products.length)
{
  var p = obj.products[i];
  s += "<tr><td>"+p.name +"</td><td>"+p.price+"</td></tr>";
}
if (obj.products.length == 0)
```

```
  s += "<tr><td colspan="2">No products in your cart.</tr>";
s += "</table>";
$("TemplatePlaceholder").innerHTML = s;
```

Although it might be a template by name, for all intents and purposes both of these approaches are essentially identical, and neither of them provide any of the benefits you should reap from using a templating solution. Namely, there are two primary benefits that you should expect from templating. First and foremost, templating should preferably not expose user interface developers to JavaScript coding and at the very least provide a solution for applying a template to a list of items without requiring an explicit `for` loop. Second, templating should make possible the creation of granular, decoupled templates, which promotes reuse and less error-prone customization. Although there might be a bit of a learning curve, both of these are well achieved by a true templating language such as XSLT, which can be a high-performance and versatile templating solution. XSLT has several advantages when it comes to the realities of implementing some templating solutions, such as good documentation (it is a W3C standard after all), granular templating, template importing capabilities—among many others. An often cited complaint of XSLT is that it is not supported in some browser. However, not only is XSLT supported in the latest versions of Internet Explorer, Firefox, Safari, and Opera, but you can also use the exact XSLT on the server to render data for user agents that do not support the technology.

A basic XSLT stylesheet looks something like this:

```
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <table>
      <xsl:apply-template select="//Product" />
    </table>
  </xsl:template>
  <xsl:template match="Product">
    <tr>
      <td><xsl:value-of select="Name"/></td>
      <td><xsl:value-of select="Price"/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

The values of Name and Price are retrieved based on the XML data that the template is applied to. Any `<xsl:apply-templates select="Product"/>` statements can result in that template being applied. To realize the real power of XSLT, you can do things such as append a predicate to a node selection `<xsl:apply-templates select="Product[Price>10]"/>` and even search entire subtrees of data just by prepending the select statement with `//`. XSLT also chooses the appropriate template to apply based on the specificity of the selector—much like CSS. For example, to apply different styling to products with different prices, you can use the following XSLT:

```
<!— this is the default template that will get applied —>
<xsl:template match="Product">
  <tr>
    <td><xsl:value-of select="Name"/></td>
    <td><xsl:value-of select="Price"/></td>
  </tr>
</xsl:template>

<!— this is a more specific template —>
<xsl:template match="Product[Name='Acme Widget']">
  <tr class="sale-product">
    <td><xsl:value-of select="Name"/></td>
    <td><xsl:value-of select="Price"/></td>
  </tr>
</xsl:template>
```

The above templates render regular product items in a `<tr>` tag for placement in a table and render products where the name is Acme Widget with a CSS class that indicates the item is a sale product.

Extensibility is a key feature of XSLT; given that the word "extensible" is in the name, you should expect as much. By using granular templates at this level, you can add or remove templates to the rendering, and the XSLT processor automatically chooses the most appropriate one. This is a deviation from other templating approaches that would likely depend on an imperative or imperative approach using an explicit `if` statement to check the product name. There can be tradeoffs with execution speed and code size depending on where extensibility is important to your component or application rendering.

It is certainly possible, with a little effort, to replicate the functionality of XSLT in native JavaScript. Again, tradeoffs can be speed and code size; however, you do get the advantage of all their code running in the same JavaScript execution sandbox making customization and some AJAX functionality a lot easier. One instance of this is in an editable DataGrid where rendered cell values can be editing by the end user, and then subsequently the new values can be saved on the server using AJAX—without a page refresh or server post-back. If there is numeric data displayed in the DataGrid, such as the product price in our example, the price needs to be formatted according to a specific number mask to be displayed with the correct currency symbol and number formatting for the location. At first, this seems easy, but there are actually several data interactions that you need to consider. The number mask needs to be applied to the data in several cases, such as the following:

- Initial rendering with all of the data
- After the value is edited
- When a new row is inserted into the DataGrid

Three distinct cases require three different levels of templates to make these sort of interactions as fluid as possible—thus, the motivation for having as granular templates as possible. We can always depend on just the first of those templates, the initial rendering template, which would certainly achieve the goal of redisplaying edited data with the proper formatting or displaying a newly inserted row; however, this would also entail a large performance hit because rerendering all the contents of the DataGrid would make editing data slow and tedious. Instead, we want to have templates for rendering blocks of data that rely on templates for rendering single rows of data that correspondingly rely on cell rendering templates.

## The Declaration

Now that we have looked at the importance of databinding and templating to building our AJAX components and applications, we can look at how to create an abstract declaration for a DataGrid component. To create a declarative DataGrid—or any other component for that matter—it is easiest to start by looking at the end product of the rendered HTML and then refactoring to break out the configurable aspects, as we did when looking at a behavioral component. Here is the first pass at defining a custom declaration for the behavioral DataGrid that we have already looked at. Note that we still use standard HTML markup but that will change.

```
<table id="myGridList" class="grid">
  <thead>
    <tr id="header" class="header-group">
      <td id="header-0" class="header header-0">Product</td>
      <td id="header-1" class="header header-1">Price</td>
    </tr>
  </thead>
  <tbody>
    <tr id="row-template" class="row-template">
      <td id="cell-{$Index}_0" class="column column-
0">{$Name}</td>
      <td id="cell-{$Index}_1" class="column column-
1">${$Price}</td>
    </tr>
  </tbody>
  <tfoot>
    <tr id="footer" class="footer-group">
      <td id="footer-0" class="footer footer-0">Total</td>
      <td id="footer-1" class="footer footer-1">${$Total}</td>
    </tr>
  </tfoot>
</table>
```

There is not that much difference here from our behavioral DataGrid HTML; we have the static header and footer of the DataGrid, as we did previously; however, we have now specified a template for the grid rows to be rendered with rather than having the data for each row explicitly written in HTML. In place of the product name and price values, we have a new syntax that looks something like this `{$FieldName}`. This syntax, which is borrowed from XSLT, is used to indicate where the data from the datasource should be placed, and the string after the `$` character should correspond to a data field in the client side datasource, which could be XML, JSON, or otherwise. Based on what we see in other declarative languages, it would make most sense to use XPath expressions here. After connecting this View to the Model, what we ideally end up with is a rendered grid where the `{$FieldName}` expressions are all replaced with data from the client side datasource. Assuming that the template is applied to a list of data, we also use the `{$Index}` expression to render out the unique numeric index of each item in the collection. In this case, we use this index value to generate a dynamic CSS class name in the HTML that we also create dynamically from JavaScript. You can also be quick to notice that there is a problem here in that the footer of the grid contains the sum of the values in the price column and, therefore, must be calculated dynamically. Also notice that the text that appears at the top of each column, as well as the HTML element styles, and even what HTML elements are used for the structure, are all still statically defined in the HTML which can drastically increase the probability of human error when defining the appearance of the component and dramatically impact usability and user interface skinability. That being said, there are certainly instances where this degree of flexibility—as in the case of the behavioral component—can be advantageous. At any rate, we can get around this problem of having all class names defined explicitly by using an even more abstract representation of our DataGrid.

For example, although we have now defined a row-based template for the data contents of our DataGrid, we can also consider binding the header of the DataGrid to a datasource using a template such as this:

```
<tr id="header-template" class="header-template">
  <td id="header-{$Index}"
      class="header header-{$Index}">{$Label}</td>
</tr>
```

The `<td>` element is repeated for each column defined. In this case, the columns are not bound to the data in our primary Model that contains the product information but instead to another pseudo Model that contains information about the columns such as the column label, width, styles, and other information, such as whether the data in the column is to be summed. This enables us to template the grid header so that each column header can be rendered out to the DOM using this simple HTML template as well. Something similar can be devised for the footer; however, depending on the application scope, things can become complicated quickly.

The nature of a grid is based on columns of data; columns are the basic building block of a grid and contain all the necessary information such as the column header, column footer, and column data. Thinking about the class diagram of a grid, you can quickly realize that rather than trying to fit a rich grid structure to an entirely HTML-based definition, it can be far easier—both for the developer of the component and the application designer using the component—to define a declaration using custom HTML tags. A declaration that make things a bit easier for an application designer might look something like this:

```
<ntb:grid datasource="products" cssclass="myGrid">
  <ntb:columns>
    <ntb:column width="100px">
      <ntb:header value="Name"></ntb:header>
      <ntb:item value="{$Name}"></ntb:item>
      <ntb:footer value="Total" style="font-
weight:bold;"></ntb:footer>
    </ntb:column>
    <ntb:column width="100px">
      <ntb:header value="Price"></ntb:header>
      <ntb:item value="{$Price}" mask="$#.00"></ntb:item>
      <ntb:footer value="{SUM($Price)}"
          style="font-weight:bold;"></ntb:footer>
    </ntb:column>
  </ntb:columns>
</ntb:grid>
```

This looks similar to the explicit HTML; however, it differs in a few important ways. The definition of the grid has been pivoted so that we consider the grid from the point of view of the columns, where each column has a header, data items, and footer, rather than from the point of view of the rows. By doing this simple change, it significantly simplifies the way we approach templating and databinding.

In the case of a DataGrid, we need several different templates. We need to template the DataGrid itself:

```
<table id ="{$id}" class="simple-
grid">{$Header}{$Data}{$Footer}</table>
```

The header group template:

```
<tr id="{$id}-header" class="header-group">{$columns}</tr>
```

The header item template:

```
<td id="{$_parent.id}-header-{$index}" class="header header-
{$index}" style="width:{$columnWidth};">{$header.value}</td>
```

The row item template:

```
<tr id="{$id}-row-{$index}" class="row {$AltRowClass}"
eatype="row">{$columns}</tr>
```

The cell item template:

```
<td id="{$id}-cell-{$RowIndex}_{$index}" class="column column-
{$index}" eatype="cell">{$item.value}</td>
```

The footer group template:

```
<tr id="{$id}-footer" class="footer-group">{$columns}</tr>
```

And finally, the footer item template:

```
<td id="{$id}-footer-{$index}" class="footer footer-
{$index}">{$footer.value}</td>
```

With a DataGrid type control, the templates are rather difficult as some of the templates depend on two sources of data. In particular, the templates are first "bound" to the state of the DataGrid; this ensures that the widths and styles of all the columns are set correctly according to the state of the DataGrid itself. The second binding takes place when the DataGrid binds to an external datasource as specified in the DataGrid state. The data cell item template is the most complicated template because it must contain information provided from both the state of the DataGrid—it needs to have certain formatting applied depending on the data type, for example—as well as information from the externally bound datasource. To ensure that each cell in the DataGrid is uniquely addressable, we generate the `id` attribute of the `<td>` element as `id="{$id}-cell-{$RowIndex}_{$index}"` where `{$id}` comes from the Data Grid state—the unique indentifier of the DataGrid itself—`{$index}` is the index of the column, and `{$RowIndex}` is the index of the row. For all the details about the templating approach, you have to look through the source code provided with the book.

With this array of granular templates, you can render the component quickly and efficiently at various points throughout the component lifetime, such as when the component is rendered, when data is edited, or when data is created or deleted.

## Building the Component

It is easiest to start by building the imperative version of the component and then enabling the use of a declaration to preset any component parameters. This approach is generally a wise one because it ensures a quality API from the point of view of an imperative developer, and it makes the component accessible to those that don't want to use the declaration. Let's look at how to build a declarative component for an application in which we want a list of Customers presented to a user that is populated from a server-side data handler using AJAX.

### Basic Functionality

As a first requirement, we create our DataGrid control in the exact same way as any other instance of a class using the `new` keyword and pass the HTML element as a single constructor argument that refers to the element in which we want our component to be rendered in. However, as with any development effort, whether you use Extreme Programming or the Waterfall approach, we start by doing at least a bit of design up front. A DataGrid control can be represented fairly simply in a UML class diagram, as shown in Figure 4.3.

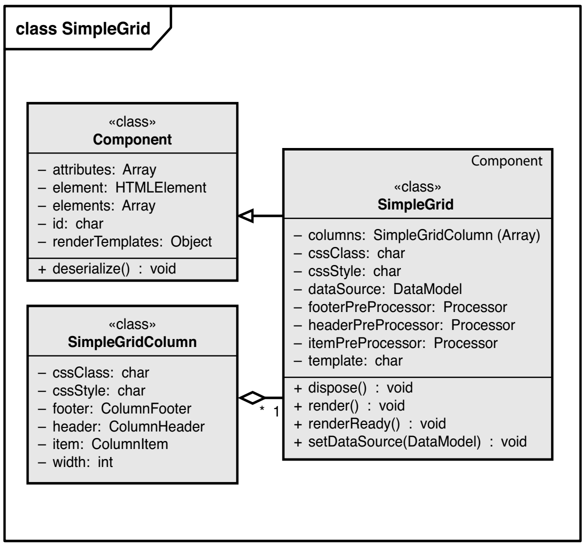

**Figure 4.3 SimpleGrid Class Diagram**

The `SimpleGrid` class consists of a collection of column definitions, header and footer, and collection of rows. Furthermore, there are a number of methods that are inherited from the `Component` class that are used by the declarative framework to instantiate, render, and destroy the component, such as `render()` and `dispose()`. There is a one-to-many relationship between the `SimpleGrid` and the Column class where the Column class contains all the information needed to render a column of data such as the column header, footer, data, width, type, and CSS properties. Similarly, the `SimpleGrid` class inherits from the `Component` class where all the requisite functionality for declaration parsing and templating is defined.

Because we design our component from a UML model, we take advantage of that and actually generate the scaffolding JavaScript code for our component including getters and setters for all the properties, method stubs, and the inheritance chain. So, to start, we get quite a bit for free just from using the tools that we have traditionally used for server or desktop development in C++ or Java.

Our initial `SimpleGrid` constructor and `Component` class look something like this:

```
entAjax.Component = function(element) {
  this.element = element;
  this.id = element.getAttribute("id");
  this.renderTemplates = {};
  this.attributes = ["id","datasource","cssclass","cssstyle"];
  this.elements = [];
}

entAjax.Component.prototype.deserialize = function() {
  for (var i=0; i<this.attributes.length; i++) {
    var attr = this.attributes[i];
    this[attr] = this.element.getAttribute(attr);
  }
}

entAjax.SimpleGrid = function(element) {
  entAjax.SimpleGrid.baseConstructor.call(this, element);
  // Collection of column objects
  this.columns = [];
  // Basic template for the entire component
  this.template = '<table id ="{$id}" class="simple-
grid">{$Header}{$Data}{$Footer}</table>';
```

```
  // Header templates
  this.headerPreProcessor = new entAjax.Processor({
    "root":{"predicate":"true","template":'<tr id="{$id}-
header" class="header-group">{$columns}</tr>'},
    "columns":{"predicate":"true","template":'<td
id="{$_parent.id}-header-{$index}" class="header header-
{$index}" style="width:{$columnWidth};">{$header.value}</td>'}
  });
  // Data row templates
  this.rowPreProcessor = new entAjax.Processor({
    "root":{"predicate":"true","template":'<tr id="{$id}-row-
{$index}" class="row {$AltRowClass}"
eatype="row">{$columns}</tr>'},
    "columns":{"predicate":"true","template":'<td id="{$id}-
cell-{$index}_{$index}" class="column column-{$index}"
eatype="cell">{$item.value}</td>'}
  });
  // Footer templates
  this.footerPreProcessor = new entAjax.Processor({
    "root":{"predicate":"true","template":'<tr id="{$id}-
footer" class="footer-group">{$columns}</tr>'},
    "columns":{"predicate":"true","template":'<td id="{$id}-
footer-{$index}" class="footer footer-
{$index}">{$footer.value}</td>'}
  });
}

entAjax.extend(entAjax.SimpleGrid, entAjax.Component);
```

In the `SimpleGrid` constructor, all we have done is create the three different template processors for the header, footer, and the data with some initial default templates. What happen to these templates is that the information from the DataGrid declaration merges with the initial templates to produce secondary templates. The advantage of doing this is that the declaration might not change during the lifetime of the component, yet the data is likely to change. With that in mind, after merging the declaration information with the templates, we cache the result so that we can reuse those generated templates as the data changes and make the templating process much more efficient.

To instantiate an instance of the SimpleGrid class based on an XHTML declaration, we use the following deserialize method, which also uses the generic deserialization method of the Component base class:

```
entAjax.SimpleGrid.prototype.deserialize = function()
{
  entAjax.SimpleGrid.base.deserialize.call(this);

  // Iterate over the <ea:column> elements
  var columns =
entAjax.html.getElementsByTagNameNS("column","ea",this.element
);
  for (var i=0; i<columns.length; i++)
  {
    // Create a new SimpleGridColumn for each declaration
column
    this.columns.push(new
entAjax.SimpleGridColumn({"element":columns[i],"index":this.co
lumns.length+1}));
  }

  // Cache results of the generated templates based on the
declaration
  this.rowTemplate = this.rowPreProcessor.applyTemplate(this);
  this.headerTemplate =
this.headerPreProcessor.applyTemplate(this);
  this.footerTemplate =
this.footerPreProcessor.applyTemplate(this);
}
```

The deserialization method is responsible for finding elements in the declaration and copying those attributes from the XHTML element to the JavaScript object. In the case of the SimpleGrid class, it copies over attributes from the <ea:grid> XHTML element and then proceeds to search for any <ea:column> elements that are then deserialized into SimpleGridColumn JavaScript objects and added the columns collection of the DataGrid. The SimpleGridColumn objects also deserialize the declaration further to get information about the column header, data, and footer.

At this point, we deserialize the state of the SimpleDataGrid from an XHTML declaration into a JavaScript object with just two lines of JavaScript code:

```
var grid = new entAjax.SimpleGrid($("myGrid"));
grid.deserialize();
```

where `myGrid` is the id of the declaration in the web page. To bring everything together and actually get the component to automatically deserialize, we use the same `initComponents()` function we used when converting the Google Map to be a declarative component. All we need to do is create a factory method that is responsible for creating an instance of the `SimpleGrid` class and put a reference to that method in the global hash table that maps XHTML element names to their respective factory methods:

```
entAjax.GridFactory = {
  "fromDeclaration": function(elem) {
    var grid = new entAjax.SimpleGrid(elem);
    grid.deserialize();
  }
}
entAjax.elements =
{"grid":{"method":entAjax.GridFactory.fromDeclaration}};
```

Now, our DataGrid is still not rendering and it doesn't have any data to render. We can remedy this by adding the render method to the `SimpleGrid` class that looks like this:

```
entAjax.SimpleGrid.prototype.render = function()
{
  this.renderTemplates["root"] =
{"predicate":"true","template":this.template};
  this.renderTemplates["Header"] =
{"predicate":"true","template":this.headerTemplate};
  this.renderTemplates["items"] =
{"predicate":"true","template":this.rowTemplate};
  this.renderTemplates["Footer"] =
{"predicate":"true","template":this.footerTemplate};
```

```
  this.renderTemplates["AltRowClass"] =
{"predicate":"true","template":altRowColor};

  // Create a container for the component and show a loading
indicator
  this.container = document.createElement("div");
  this.element.appendChild(this.container);
  // Create the processor for the cached templates.
  this.gridProcessor = new
entAjax.Processor(this.renderTemplates);
  // Generate the content from the templates and the data
  var html =
this.gridProcessor.applyTemplate(this.dataSource.items);
  this.container.innerHTML = html;
}
```

The render method takes the cached rendering templates that were created in the deserialization method and applies those generated templates to the data, which results in the contents of the DataGrid being generated after which that content is placed into the web page DOM using the `innerHTML` property of the XHTML declaration element. The `dataSource` field of the `SimpleDataGrid` containing the data to be rendered can be populated simply by setting it to some static array of customer data like this:

```
grid.dataSource =
{"items":[{"Name":"Joe","Company":"Acme"},{"Name":"Bob","Compa
ny":"Widgets'r'us"}]};
```

## Connecting to the Server

Rendering static data is hardly that useful in an enterprise application, so let's connect to the server. To retrieve data from the server, we need to go back to our `SimpleDataModel` class and give it some teeth. The first step for creating a remote datasource is retrieving the data from the server—we deal with inserting, updating, and deleting later. The UML diagram for the `RemoteDataModel` class looks like Figure 4.4.

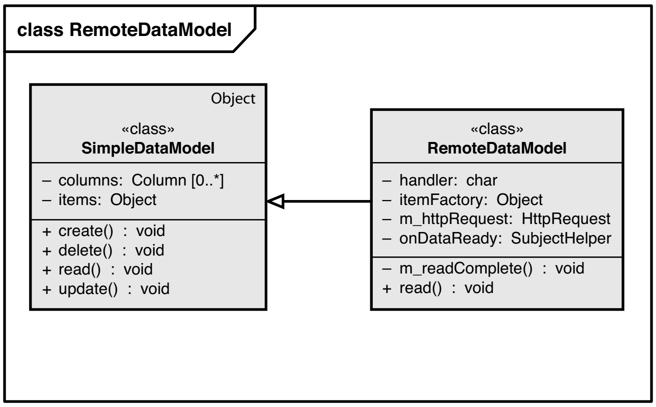

**Figure 4.4 RemoteDataModel Class Diagram**

The important new features of the `RemoteDataModel` compared to the `SimpleDataModel` are the private `m_httpRequest` for retrieving data from the server, the `onDataReady` event for notifying interested parties when the data is ready for consumption, the `m_readComplete` method that handles the asynchronous callback from the XHR object when the data has been retrieved from the server, and finally, the `itemFactory` object that we use to deserialize XML data from the server into JavaScript objects. The code for the `RemoteDataModel` follows:

```
entAjax.RemoteDataModel = function(items)
{
  // Call the base constructor to initialize the event objects
  entAjax.RemoteDataModel.baseConstructor.call(this);
  // onDataReady will fire when data is ready for use
  this.onDataReady = new entAjax.SubjectHelper();
  // The handler is the URL of the data on the server
  this.handler = "";
  // To enable the RemoteDataModel to create any type of
object
  this.itemFactory;
  // Internal XHR object for retrieving data from the server
  this.m_httpRequest = new entAjax.HttpRequest();
}
```

```
// Inherit from SimpleDataModel
entAjax.extend(entAjax.RemoteDataModel,
entAjax.SimpleDataModel);

entAjax.RemoteDataModel.prototype.read = function()
{
  // Request the data from the server and call m_readComplete
  // when the data is ready
  this.m_httpRequest.completeCallback = entAjax.close(this,
this.m_readComplete);
  this.m_httpRequest.handler = this.handler;
  this.m_httpRequest.get();
}

entAjax.RemoteDataModel.prototype.m_readComplete =
function(serverResponseArgs)
{
  this.items = [];
  // This should be encapsulated but is ok for now
  var data =
serverResponseArgs.response.documentElement.childNodes;
  // Loop though each XML element returned and deserialize it
  for (var i=0; i<data.length; i++)
  {
    this.items.push(this.itemFactory.fromXml(data[i]));
  }
  // Let everyone know that the data is ready
  this.onDataReady.notify(this, serverResponseArgs);
}
```

Consider a few important points about the `RemoteDataModel` class. First, we request the data from the server asynchronously—at the URL specified by the `handler` field—so the `read()` method no longer directly returns data; the `read()` method no longer blocks the JavaScript thread until data is ready to be returned and instead sends the request for data to the server and immediately returns with no return value. The data is actually returned from the new method we added called `m_readComplete()`, which is executed when the data has finally been returned from the server. Just like the callback function that we use on the plain XHR object to be notified when an asynchronous request has been completed, we now need to apply that same pattern to our custom JavaScript classes that rely on asynchronous server requests. Thus, we have introduced the onDataReady event to which handlers can be subscribed and, hence, notified when the data is indeed ready.

The second important point about the RemoteDataModel class is that rather than returning JSON from our web service on the server, we return plain XML, which this is another aspect that can be factored out to create a RemoteXMLDataModel and RemoteJSONDataModel. A problem arises here because our DataGrid component relies on JavaScript-based templating and, therefore, expects an array of JavaScript objects as the items field on a datasource object. To achieve this, we made the itemFactory field on the RemoteDataModel that is used to enable the user to specify a factory object that will provide a fromXml method that will return a JavaScript object based on the information in an XML element returned from the server. In this case, we want to create Customer objects, and we set the itemFactory field of the RemoteDataModel to an instance of the CustomerFactory class:

```
entAjax.CustomerFactory = function(){}
entAjax.CustomerFactory.prototype.fromXml = function(element)
{
  return new entAjax.Customer(element);
}
```

Now we have a choice to make as to how we actually instantiate the Customer class, and we have decided to depend on the class itself to do the deserialization from an XML element. The alternative would be to create an instance of the Customer class and then set all the relevant fields from the outside. To achieve this deserialization, we created a basic Serializable class as follows:

```
entAjax.Serializable.prototype.deserialize = function() {
  for (var item in this) {
    if (typeof this[item] == "string") {
      var attr = this.element.getAttribute(item);
      if (attr != null)
      {
        this[item] = attr;
      }
    }
  }
}
```

This loops through the attributes on the XML element from which the object is to be deserialized and copies each attribute name and value pair onto the JavaScript object. The `Customer` class looks like this:

```
entAjax.Customer = function(element)
{
  this.CustomerID="";
  this.CustomerName="";
  this.ContactName="";
  this.ContactEmail="";
  this.ContactTitle="";
  this.PhoneNumber="";
  this.Address="";
  this.Country="";
  this.RegionID="";
  entAjax.Customer.baseConstructor.call(this, element);
}
entAjax.extend(entAjax.Customer, entAjax.Serializable);
```

## Closing the Loop

Now that we have a `RemoteDataModel` that our DataGrid can connect to, we need to actually connect them. To achieve this, we can update the `GridFactory fromDeclaration()` method so that it also creates an instance of the `RemoteDataModel` class and specifies the appropriate factory for the `RemoteDataModel itemFactory`—in this case, the `Customer Factory` because our DataGrid is rendering `Customer` objects.

```
entAjax.GridFactory = {
  "fromDeclaration": function(elem) {
    var grid = new entAjax.SimpleGrid(elem);
    grid.deserialize();

    var rdm = new entAjax.RemoteDataModel();
    // need to get this from the datagrid...
    rdm.itemFactory = new entAjax.CustomerFactory();

    grid.setDataSource(rdm);
  }
}
```

The `setDataSource()` method of the DataGrid will do a few things, such as ensure that the supplied datasource actually inherits from the `DataModel` class, sets the `handler` field on the remote datasource to the URL of the server side data handler specified on the DataGrid declaration, and subscribes a new method of the `SimpleDataGrid` called `m_render Ready()` to the `onDataReady` event of the datasource.

```
entAjax.SimpleGrid.prototype.setDataSource =
function(dataSource) {
  if (dataSource instanceof entAjax.DataModel) {
    this.dataSource = dataSource;
    this.dataSource.handler = this.handler;
    this.dataSource.onDataReady.subscribe(this.m_renderReady,
this);
  }
}
```

Due to the asynchronous nature of the data retrieval now, the DataGrid `render()` method needs to be reconsidered. The render() method will no longer actually do any rendering but instead simply call the read() method on the datasource. The datasource will then asynchronously request the data from the server and notify all subscribers to the onDataReady event—one of those subscribers just so happens to be the m_renderReady event of the DataGrid, and that is where the actual rendering code gets moved to.

```
entAjax.SimpleGrid.prototype.m_renderReady = function()
{
  var html =
this.gridProcessor.applyTemplate(this.dataSource.items);
  // Remove any activity indicators that were displayed
  this.loadingComplete();
  // Set the contents of the component to the generated HTML
  this.container.innerHTML = html;
}
```

The final piece of the puzzle is adding a call to the DataGrid's `render()` method into the `GridFactory` such as this:

```
entAjax.GridFactory = {
  "fromDeclaration": function(elem) {
    var grid = new entAjax.SimpleGrid(elem);
    grid.deserialize();
    var rdm = new entAjax.RemoteDataModel();
    rdm.itemFactory = new entAjax.CustomerFactory();
    grid.setDataSource(rdm);
    grid.render();
  }
}
```

Now we have a fully operational DataGrid that is requesting data from the server and rendering it in the web browser! The full web page is shown here:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html xmlns:ea="http://www.enterpriseajax.com">
  <head>
    <title>Component Grid</title>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
    <link rel="stylesheet" href="simplestyle.css"
type="text/css">
    <script language="javascript" type="text/javascript"
src="entajax.toolkit.js"></script>
    <script language="javascript" type="text/javascript"
src="RemoteDataModel.js"></script>
    <script language="javascript" type="text/javascript"
src="SimpleDataGrid.js"></script>
  </head>
  <body>
    <ea:grid id="myGrid" handler="data.xml"
cssclass="CustomerGrid">
      <ea:columns>
        <ea:column width="100">
          <ea:header value="Name"
cssclass="myHeaderCSS"></ea:header>
          <ea:item value="{$ContactName)}"
cssclass="myRowCSS"></ea:item>
        </ea:column>
        <ea:column width="100">
          <ea:header value="Company"></ea:header>
          <ea:item value="{$CompanyName}"></ea:item>
        </ea:column>
      </ea:columns>
    </ea:grid>
  </body>
</html>
```

What you will likely notice is that in developing the component the way we did, we can also instantiate the component purely from JavaScript as if there is no declaration at all:

```
var grid = new entAjax.SimpleGrid(elem);
// Setup all the columns through JavaScript
grid.columns.push(new entAjax.SimpleDataGridColumn());
grid.columns[0].header = new
entAjax.SimpleDataGridColumnHeader();
grid.columns[0].header.value = "ContactName";
grid.columns[0].item = new entAjax.SimpleDataGridColumnItem();
grid.columns[0].item.value = "{$ContactName}";
// Create and attach the datasource
var rdm = new entAjax.RemoteDataModel();
rdm.itemFactory = new entAjax.CustomerFactory();
grid.setDataSource(rdm);
// Render the component
grid.render();
```

## Summary

This chapter explained that there is a lot involved in not only developing an AJAX application, but also in having it interact with a user's web browser. Through the course of this chapter, we looked at some of the differences between imperative and declarative approaches to developing AJAX applications and looked at a simple example of making the Google Map component declarative. We also looked at some of the important variations on declarative programming, most notably behavioral. Behavioral AJAX can be a great tool for taking pre-existing HTML markup and adding a little AJAX on top of it to make your application a little more interactive and usable. Using many of the JavaScript techniques, we went through the entire process of developing a declarative DataGrid component from the ground up. In future chapters, we take a closer look at some of the nuances around various aspects of the DataGrid component in the context of a larger application.

## Resources

XForms, http://www.w3.org/MarkUp/Forms/XSLT, http://www.w3.org/TR/xslt
JSONT, http://goessner.net/articles/jsont/
Internet Explorer databinding, http://msdn.microsoft.com/workshop/author/databind/data_binding_node_entry.asp
Google Maps, http://www.google.com/apis/maps/