11 Developing applications with Tiles

Co-authored by Cedric Dumoulin and Ted Husted

This chapter covers

- Designing applications with dynamic includes
- Using the Struts and Tiles frameworks together
- Understanding Tiles Definitions and attributes
- Migrating applications to Tiles

A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosophers and divines.

—Ralph Waldo Emerson

11.1 Leveraging layouts

Usability is a prime concern in the design of today's applications—and consistency is a prime ingredient of usability. Users want to stay focused on the task at hand and are easily annoyed by any small inconsistency in an application's interface or screen layout.

Consistency is no small challenge for a dynamic web application; it is commonplace for each page to be coded by hand. Layout tools for static pages are available to most designers today, but few of these are available to applications based on JavaServer Pages.

Worse, the look and feel of an application is usually the last detail to be finalized, and then it will often change between versions—or even arbitrarily as part of a greater website "relaunch." This can create a nightmarish round of last-minute coding and testing—even if only to alter the background color or add a new link to the menu bar.

Of course, consistency is more than a hobgoblin; it's a hallmark of good design. Like any other component, web pages contain many common elements, headers, footers, menus, and so forth. Often, these are cut-and-pasted from one page to the next. But like any component, bugs are found and features are enhanced, leading to another round of cut-and-paste "reuse."

In a web application, page markup is a programming component like any other and should be held to the same standard of reuse.

11.1.1 Layering with dynamic templates

In the first part of this book, we stressed the importance of layering an application to isolate the effects of change. By compartmentalizing an application, we can change one piece without disrupting the other pieces. The same concept can be applied within the presentation layer to separate the look and feel from the actual content.

One approach to separating layout from content is the use of dynamic JSP includes. The JSP specification provides for both static and dynamic includes. The standard JSP action for a dynamic include is `<jsp:include>`.

We make use of dynamic includes by breaking the server page into several fragments, each with its own job to do. A background template can set the default

format and layout, and page fragments can be included at runtime to provide the content. A dynamic include folds the output of the included page into the original page. It acts like a switch that moves processing over to a page fragment and then back to the caller. As shown in figure 11.1, the included template is processed normally, just as if it had been called on its own.

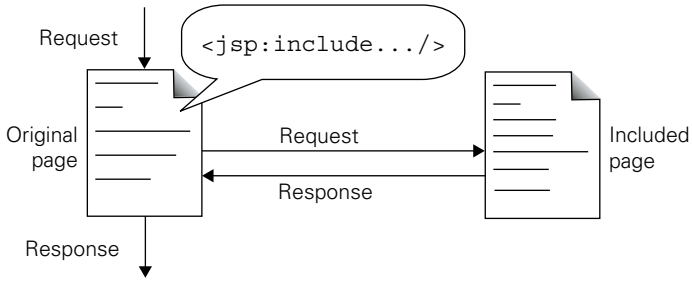

Figure 11.1 The effect of the `<jsp:include>` action on the processing of a request

The Tiles framework, which we explore in this chapter, uses a more advanced form of the JSP include action. In a Tiles application, the background, or layout, template usually defines the position of a header, menu body, content, and footer. Other pages are then included to fill each of these positions. If the header changes, then only that template file need be changed. The change is automatically reflected in all pages that include that template. The effects of changes are minimized—and the hobgoblins appeased.

Standard HTML components, like Cascading Style Sheets (CSSs), also work well with dynamic templates. A style sheet can help keep the templates internally consistent and further minimizes the effects of change.

11.1.2 Template consequences

Every technology comes bundled with compromises. Here are some consequences—pro, con, and mixed—that come with using dynamic templates in your application:

- The JSP include technology is well established and reliable, and tends to scale well in larger applications. The underlying technology for including dynamic templates is part of the core Java Servlet API.
- Most containers are optimized for JSPs and standard features like servlet include.

- The included pages typically output HTML fragments and are not synoptically complete. This can prevent you from maintaining templates with standard HTML editors, which expect markup to be part of a complete, stand-alone page.
- Most sites recompile JSP pages when the source changes. Templates create more pages to be monitored for such changes.
- Templates actually reuse code that would otherwise be duplicated from page to page. This can result in a significantly smaller footprint and conserve server resources.

11.1.3 Using templates

We take a close look at using dynamic templates in this chapter, especially as the Tiles framework implements them. Tiles is a mature product and integrates well with the Struts framework. Tiles templates can even be deployed from a Struts ActionForward, eliminating a good many "red tape" files other template systems require.

Struts and Tiles are a powerful combination. Using dynamic templates to generate presentation pages jibes well with the other programming practices involved in writing a web application. In this chapter, we show how to best combine Tiles with Struts and other assets, like CSS. After introducing Tiles, we provide a refactoring guide to help you migrate an existing product to Tiles.

If you find that Tiles is a good match for your application, be sure to study the example application in chapter 15. See table 11.1 for a glossary of some of the special terms we use in this chapter.

Table 11.1 A glossary of dynamic template terms

Term	Definition
Dynamic element	A portion of a JSP that is recognized by the JSP translator, including an action, directive, expression, JSP tag, or scriptlet.
Template data	A portion of a JSP that is not recognized by the JSP translator and is passed to the response verbatim. Usually markup and visible text.
Template page	A JSP that includes, or is included by, another page.
Template file	A static file or JSP that is included by a template page.
Tile	A synonym for template page.
Layout	A description of where template files, or tiles, should be positioned on a page.

Table 11.1 A glossary of dynamic template terms *(continued)*

Term	Definition
Tiles	A framework that makes templates and layouts easier and more powerful.
Definition	A Tiles feature that allows a layout to be specified as a template page or as a JavaBean. Definitions can also be described by an XML document.

11.1.4 Combining templates, Tiles, and Struts

When HTML tables were first invented, page designers immediately adopted them as a layout mechanism. A borderless table can be used to contain other tables and content and create layouts that were otherwise impossible.

The same idea is often used with dynamic templates. As shown in figure 11.2, a master template is used to provide the layout for the page and position the elements; page fragments are then included to fill in the elements. The page fragments can be included in any type of layout: those that use borderless tables, those that use `<div>` tags, and even very simple layouts that just stack one component over the other.

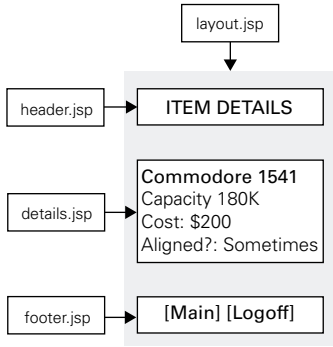

Figure 11.2 A master template provides the layout for a page.

Tiles is a framework that makes using template layouts much easier through use of a simple but effective tag library. It can be used as a drop-in replacement for the Template taglib distributed with Struts 1.0 but provides more functionality. The Tiles package can be used with any JSP application.

1.0 vs 1.1 Tiles is bundled with Struts 1.1. Configuring Tiles for Struts 1.1 is covered in chapter 4. Tiles is also available for Struts 1.0. See the Extensions category on the Struts Resources page for details [ASF, Struts].

Most often, a JSP template system will use one template for the layout and another for the fill-in components. Tiles calls these layout files *Definitions*. Most template systems require that an extra file be used just to specify the layout. To avoid this overhead, Tiles allows the layout Definitions to be stored as a JavaBean. What's more, there is an extension to Struts that allows Definitions to be the target of an ActionForward. This is an exciting feature, which we discuss in section 11.3.3. In section 11.5, we return to Definitions again when we walk through refactoring an existing product into a Tiles-based application using Definitions and Struts.

But, for now, let's start at the beginning and create a new layout from scratch.

11.2 Building a layout template

The first step in building any layout is to identify the component parts. For a classic web page, the parts would be a header, menu, body, and footer. Often a simple sketch can help to bring the layout into focus.

To provide a quick example, we will build a layout for a classic web page with a header, a menu, a body, and footer. Our layout sketch is provided in figure 11.3.

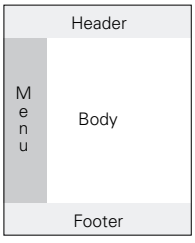

Figure 11.3
The classic master template layout includes a header, body, menu, and footer.

Creating the template page for a layout like the one in figure 11.3 is as easy as it looks:

1. Open a new JSP page.
2. Import the Tiles taglib.
3. Create an HTML table with cells that match the sketch.
4. Use a Tiles JSP tag (`<tiles:insert>`) to name each part of the layout.

As you can see in listing 11.1, wherever we create a cell to include one of our layout components, we place a JSP tag like `<tiles:insert attribute="aName"/>`. This tag says to insert the tile (or template file) identified by the value of the specified attribute. When it is time to use this layout in an application, we pass to it the paths to use for each of these tiles. This allows us to use the same layout over and over again, simply by passing a different path for one or more of the tiles.

Listing 11.1 Markup fragment for our layout page

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<TABLE border="0" width="100%" cellspacing="5">
<TR>
  <TD colspan="2"><tiles:insert attribute="header"/></TD>
</TR>
<TR>
  <TD width="140" valign="top">
    <tiles:insert attribute="menu"/>
  </TD>
  <TD valign="top"  align="left">
    <tiles:insert attribute="body"/>
  </TD>
</TR>
<TR>
  <TD colspan="2">
    <tiles:insert attribute="footer" />
  </TD>
</TR>
</TABLE>
```

In most cases, the body tile will change for each page, but all pages in the same area could share the same header and menu tiles. Meanwhile, all pages on the same site might share a single footer tile. When a new year rolls around and it is time to update the copyright notice to display the new current year on every page, only the one footer tile need be edited.

Our classic layout can be made into a complete, stand-alone template page just by adding the rest of the HTML markup.

In listing 11.2, you'll note that we slipped in a new Tiles tag, `<tiles:getAsString name="title"/>`. This tag says to return an attribute value as a literal string rather than as a pathname or other command. To do this, Tiles calls the object's standard `toString()` method. The result is inserted directly into the page at runtime.

Listing 11.2 Classic layout as a complete template page: myLayout.jsp

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<HTML>
  <HEAD>
    <TITLE><tiles:getAsString name="title"/></TITLE>
  </HEAD>
<BODY>
<TABLE border="0" width="100%" cellspacing="5">
<TR>
  <TD colspan="2"><tiles:insert attribute="header" /></TD>
</TR>
<TR>
  <TD width="140" valign="top">
    <tiles:insert attribute='menu'/>
  </TD>
<TD valign="top" align="left">
    <tiles:insert attribute='body' />
  </TD>
</TR>
<TR>
  <TD colspan="2">
    <tiles:insert attribute="footer" />
  </TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

11.2.1 But what is a tile?

The template features offered by the Tiles framework far surpass what the standard Servlet and JSP includes offer. The framework refers to its templates as *tiles*. This is to help indicate that tiles are more powerful than simple JSP templates. Tiles are building blocks for your presentation layer.

Technically, a tile is a rectangular area in a JSP, sometimes referred to as a *region*. A tile may be assembled from other tiles. Tiles can be built recursively and represented as a tree, as shown in figure 11.4. Each node on the tree is a region. The root node is usually the page. Final nodes, or leaves, contain the page content. Intermediate nodes are usually layouts. The layout nodes are utility tiles that either position a tile within the page or provide background markup for the content tiles.

The tile objects support several important features, including parameters and Definitions.

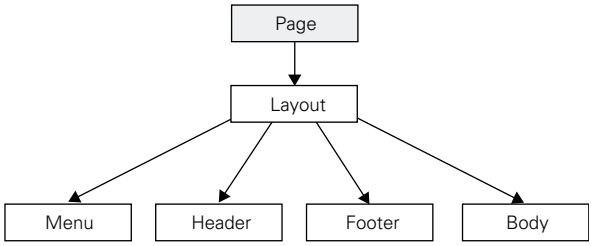

Figure 11.4 Tiles can be represented as a tree: the page is the root, and the layout tile is a branch (intermediate node), which then includes its own leaves (final nodes).

Parameters

A tile can accept variable information at runtime in the form of parameters or attributes. This means that tiles are parameterizable. They can accept variable information and act upon themselves accordingly. The tiles parameters are usually called *attributes* to avoid confusion with request parameters.

Tile attributes are defined when inserting the tile and are visible within the tile only. They aren't visible in subtiles or to a page enclosing the tile. This avoids name conflicts when the same tile is used several times in the same page. Developers can stay focused on making the best use of Tiles without worrying about name contention.

Tile attributes can be strings and other types. See section 11.4 for more about tile attributes.

Definitions

Taken together, the various attributes passed to a tile create a description of the screen. In practice, many of these descriptions are related and tend to build on one another.

Definitions store a set of attributes so that a screen description becomes a discrete object with its own identity. Declare a base screen Definition and then create other Definitions derived from that base. If the particulars of a base screen change, then all the Definitions extended from that base are also changed. This brings the object-oriented principles of inheritance and encapsulation to your dynamic pages. We cover the Tiles Definition in section 11.4.

Definitions are optional. You can also deploy a tile at any time using a simple JSP tag.

11.2.2 Deploying a Tiles template

The layout template we built in 11.2.1 defined *where* to position its tiles but not *what* tiles to use. Those details—and any other particulars—are passed to the layout when it is deployed. The simplest way to do this is to call the layout from another JSP.

Listing 11.3 shows a JavaServer Page being used to pass to a layout what tiles it should use. When hello.jsp is rendered, it will return myLayout.jsp with the content of the specified tiles.

Listing 11.3 Deploying an instance of the classic layout: /pages/hello.jsp

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:insert page="/layouts/myLayout.jsp" flush="true">
  <tiles:put name="title"  value="Hello World" />
  <tiles:put name="header" value="/tiles/header.jsp" />
  <tiles:put name="footer" value="/tiles/footer.jsp" />
  <tiles:put name="menu"   value="/tiles/menu.jsp" />
  <tiles:put name="body"   value="/tiles/helloBody.jsp" />
</tiles:insert>
```

To use the same layout with a different body, we simply substitute the tags

```
<tiles:put name="title"  value="Hello World" />
<tiles:put name="body"   value="/tiles/helloBody.jsp" />
```

with new particulars, like

```
<tiles:put name="title"  value="Hello Again" />
<tiles:put name="body"   value="/tiles/pageTwo.jsp" />
```

This new page would look much like the original hello.jsp, except with a different title (Hello Again) and a different body tile (the contents of pageTwo.jsp).

You can continue to reuse a layout this way, substituting different attribute values as needed. This passing of parameters makes it possible to use a single base template to lay out every page on a site. If the website layout has to be altered, then only the one base template need be changed.

However, to get the full value of this approach, you have to create at least two JSP files for each new page deployed: one file for the new content and then a second file to insert the template and include the new content in the first file. Later in the chapter, we show you how to use Tiles Definitions with Struts to avoid the overhead of a second file. The body-wrap deployment approach covered in the next section is another way to avoid the second file.

Body-wrap deployments

Any tile can be used any number of times in an application. In practice, though, most of the tiles in your application will be *content* tiles that provide the distinct portion of any given page.

Typically, a page's content tile is used only once in the application. The header, footer, and menu tiles may be used over and over again from page to page. But the multiuse tiles are usually window dressing for each page's singular content tile.

When this is the case, you can simply wrap the content tile with the rest of the screen definition. The trick is to just provide the markup as the value of the `put` tag. As shown in listing 11.4, be sure to specify `type="string"` so that Tiles does not mistake it for the path to a page.

Listing 11.4 Deploying a layout using the body-wrap technique

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:insert page="/layouts/myLayout.jsp" flush="true">
  <tiles:put name="title" value="Hello World" />
  <tiles:put name="header" value="/tiles/header.jsp" />
  <tiles:put name="body" type="string">

  <%-- Place the content from /tiles/pageTwo.jsp here --%>

  </tiles:put>
  <tiles:put name="footer" value="/tiles/footer.jsp" />
  <tiles:put name="menu"   value="/tiles/menu.jsp" />
</tiles:insert>
```

This avoids creating an extra tile. A side effect is that it prevents a body tile from being reused on another page. When the body markup does need to be used in more than one place, you would have to refactor the page so that the content is a separate tile again. But for nearly all your pages, the content tile will be used only once.

The body wrap is a very effective approach. The only downside is that the screen definitions are dispersed throughout the site, which can make some global changes more difficult. The Tiles Definitions, described in section 11.3, provide a more centralized approach that works very well with the Struts architecture. But the body-wrap deployment pattern can still be a good choice for smaller applications.

11.2.3 Adding a style sheet

Since tiles are JSP pages, all the usual accouterments are available, including CSSs [W3C, CSS]. Using a style sheet with your tiles is not required but can be helpful.

While full support of CSSs eludes today's browsers, they are still a useful way to define color schemes and some other key attributes. Style sheets help ensure that these niceties remain consistent from tile to tile.

To specify a style sheet, simply insert the usual tag. The `<html:base>` tag can help resolve relative paths to the style sheets and other assets. Better yet, use the `<html:rewrite>` tag to render the path for you, and URL-encode it in the bargain. Listing 11.5 shows how to use both tags.

Listing 11.5 Using <html:base> and <html:rewrite>

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<HTML>
  <HEAD>
    <TITLE><tiles:getAsString name="title"/></TITLE>
    <html:base/>
    <LINK rel="stylesheet" type="text/css"
     ref="<html:rewrite page='/assets/styles/global.css'/>">
  </HEAD>
<BODY>
```

11.2.4 Templates and MVC

Dynamic templates work especially well within a Model-View-Controller architecture. (See chapter 2 for more about MVC.) Used correctly, dynamic templates cleanly separate markup from content. In practice, the portion of a page with the actual content is often tucked away in the center, surrounded by areas devoted to markup and navigation. Content, markup, and navigation map easily to the roles of Model, View, and Controller.

Often, an MVC template system can be created from an existing application just by using standard refactoring techniques. The base markup for the page can be extracted into a master template. The site header and footer are extracted into their own files; the center square with the actual content goes into its own template file as well. The base template includes them back again. Related technologies, like CSS, can be used to define formatting in the base template and have it apply seamlessly to the others that form the final page.

DEFINITION *Refactoring* is the process of improving software by restructuring its source code for clarity and flexibility and by eliminating redundant or unused code.

The Tiles package takes this one step further with its Definition feature. Using Definitions, we can streamline the number of physical templates an application needs and move the implementation details into a central JavaServer Page or, better yet, an XML document.

11.3 Tiles Definitions

In listing 11.3, we saw that even the simplest layout can require a good number of parameters. We also saw that in most cases only one or two of those parameters change from page to page.

What's needed is a way to define all these attributes and properties in a single reusable bundle that could just be overloaded with the parameters that change—which, as it happens, it just what the Tiles Definitions do.

11.3.1 Declaring Definitions

Since they serve similar purposes, writing a Tiles Definition is much like writing a `<tiles:insert>` tag, as we did in 11.2.2. A Definition requires the following information:

- A path to the base template file
- A list of zero or more attributes (name-value couplets) to pass to the template
- An identifier (or name) for the Definition

As you can see, the real difference between a Definition and a `<tiles:insert>` tag is that a Definition can be named. But just by adding *identity* to the feature list, several doors are opened:

- A Definition can be overloaded by passing additional or replacement attributes when it is deployed.
- A Definition can be extended by using one Definition as the base for another.
- A Definition can be reused by storing it in a JSP or loading it from an XML document.
- A Definition can be the target of a Struts ActionForward.

Let's look at the two ways of specifying a Tiles Definition: with a JSP or via an XML document.

11.3.2 JSP declarations

A quick and easy way to get started with Definitions is to declare them with a JSP. This does not reduce the number of template files your application needs but does allow for reuse through overloading. You can declare a base Definition, and then in other Definitions specify how the new Definition differs from its predecessor. This requires the use of a stub file to deploy the Definition at runtime.

In section 11.3.3, we will look at placing these same Definitions in an XML document and deploying them directly from a Struts ActionForward. Since the underlying process is the same, let's discuss the now-familiar JSP-type declaration first.

The process for using Tiles Definitions with JavaServer Pages includes:

- Declaring a Definition with a JSP
- Deploying a JSP-declared Definition
- Overloading a Definition
- Reusing Definitions with JSPs

Let's discuss each in turn.

Declaring a Definition with a JSP

Listing 11.6 specifies the same particulars as listing 11.4 but uses a Definition.

Listing 11.6 A simple Definition

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:definition id="definitionName" page="/layouts/myLayout.jsp">
  <tiles:put name="title"  value="Hello World" />
  <tiles:put name="header" value="/tiles/header.jsp" />
  <tiles:put name="footer" value="/tiles/footer.jsp" />
  <tiles:put name="menu"   value="/tiles/menu.jsp" />
  <tiles:put name="body"   value="/tiles/helloBody.jsp" />
</tiles:definition>
```

The Definition in listing 11.6 would be saved as a bean in the JSP context scope, using `id` as the attribute key. Like many Struts tags, the `<tiles:definition>` tag supports a `scope` property for specifying a certain context (application, session, request, page). The default is page context. This Definition would be available to the rest of this JSP only.

Deploying a JSP-declared Definition

To put a Definition to use, you can use the `<tiles:insert>` tag, specifying the bean name (Definition ID) and also the scope if needed, as shown in listing 11.7.

Listing 11.7 Deploying a Definition

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:definition id="definitionName" page="/layouts/myLayout.jsp">
  <tiles:put name="title"  value="Hello World" />
  <tiles:put name="header" value="/tiles/header.jsp" />
  <tiles:put name="footer" value="/tiles/footer.jsp" />
  <tiles:put name="menu"   value="/tiles/menu.jsp" />
  <tiles:put name="body"   value="/tiles/helloBody.jsp" />
</tiles:definition>
<tiles:insert beanName="definitionName" flush="true/>
```

At startup, Tiles uses the XML element to create a Definition object (or bean) with the `id` `definitionName`. At runtime, the `<tiles:insert>` tag refers to the Definition `id` through its `beanName` property.

Overloading a Definition

Once we have declared a Definition, we can refer to it by name and overload some or all of its attributes. To overload an attribute, just specify it again with a new value. To create a new attribute, include a new attribute name and its value. This allows you to create a base Definition and then just specify the changes needed to create a new page, as shown in listing 11.8.

Listing 11.8 Overloading a Definition

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:definition id="definitionName" page="/layouts/myLayout.jsp">
  <tiles:put name="title"  value="Hello World" />
  <tiles:put name="header" value="/tiles/header.jsp" />
  <tiles:put name="footer" value="/tiles/footer.jsp" />
  <tiles:put name="menu"   value="/tiles/menu.jsp" />
  <tiles:put name="body"   value="/tiles/helloBody.jsp" />
</tiles:definition>

<tiles:insert beanName="definitionName" flush="true" >
  <tiles:put name="title"  value="New PageTitle" />
  <tiles:put name="body"   value="/tiles/anotherBody.jsp" />
  <tiles:put name="extra"  value="/extra.jsp" />
</tiles:insert>
```

Here, after declaring the Definition, our `<insert>` tag specifies a new page `title` and a different `body`, and throws in an `extra` attribute that was not used in the original Definition. This implies that the layout can manage without this tile but can also display it when provided. A layout can indicate an optional tile using the `ignore` property. When `ignore` is set to true, an error will not be reported if the attribute is not present.

This is how the myLayout.jsp from the original Definition would specify the `extra` tile:

```
<tiles:insert attribute="extra" ignore="true" />
```

Reusing Definitions with JSPs

In the preceding example, we had to repeat the Definition before we could overload any of the attributes. Of course, this is not a very practical approach to reuse. A slightly better approach, shown in listing 11.9, is to declare and reuse Definitions by using a utility page and including that page wherever any of the Definitions are needed.

Listing 11.9 Including a Definition

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<%@ include file="definitionsConfig.jsp" %>
<tiles:insert beanName="definitionName" beanScope="request" />
  <tiles:put name="title"  value="Another Page" />
  <tiles:put name="body"   value="/tiles/anotherBody.jsp" />
</tiles:insert>
```

The key here is the standard JSP `include` tag that brings in the file containing our Definitions. The Definitions are created normally; you can use any of the Definitions in the usual way. The example also overloads the `title` and `body` attributes to customize the page.

This approach to reuse is fine if you have a small number of Definitions, but it doesn't scale well. Each time the file is included, by default all the Definition objects are re-created, draining performance. This can still be a useful approach during development, since any change to the Definitions will be reflected in the next page load.

In production, one workaround would be to create the Definitions in application scope and protect the block with the Struts `<logic:notPresent>` tag (or equivalent), as shown in listing 11.10.

Listing 11.10 Including a Definition using <logic:notPresent>

```
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<logic:notPresent name="definitionName" scope="application">
<tiles:definition id="definitionName" page="/layouts/myLayout.jsp">
  <tiles:put name="title"  value="Hello World" />
  <tiles:put name="header" value="/tiles/header.jsp" />
  <tiles:put name="footer" value="/tiles/footer.jsp" />
  <tiles:put name="menu"   value="/tiles/menu.jsp" />
  <tiles:put name="body"   value="/tiles/helloBody.jsp" />
</tiles:definition>
<%-- … other definitions … --%>
</logic:notPresent>
```

If the Definitions have already been created and stored in application scope, they will not be created again. Each page still has to include the Definitions, and each page will be looking to see if they exist, but at least they won't be continually re-created.

Tiles offers a better approach to loading and reusing Definitions. The Definitions can be declared and loaded once from an XML document. Internally, Tiles renders the Definitions directly from ActionForwards. This is a truly excellent way to manage Tiles Definitions, which we explore in the next section.

1.0 vs 1.1 Tiles for Struts 1.0 subclasses the ActionServlet to render the Definitions. In Struts 1.1, Tiles subclasses the RequestProcessor to do the same thing.

11.3.3 Configuration file declarations

By declaring your Definitions in an XML configuration file, you can make your application load the file at startup and create a "Definition factory" containing your Definitions. Each Definition is identified by the `name` property that should be unique to all your Definitions. Other components, like ActionForwards, can then refer to the Definition by name.

Declaring Tiles Definitions from a configuration file requires some additional setup to enable support for reading the configuration file when the application initializes. See chapter 4 for more about installing Tiles with Struts 1.1.

The XML configuration file is read when the application is initialized and parsed into a Definition factory that contains an instance of each declared Definition. Each Definition should have a unique name so that it can be referenced by

JSP tags or the Struts ActionForwards. The Definition name is an internal reference only. It is not a URI and can't be referenced directly by a client.

The process of using Definitions declared from an XML document is no different than using includes from a JSP file. The main difference is how they are created, how they can be extended, and how the Definitions can be used as Struts ActionForwards.

Creating the configuration

The overall format of the XML configuration is similar to the Struts configuration (since they are both proper XML documents). Unsurprisingly, the syntax used within the XML configuration file is similar to the Tiles `<definition>` tag, as shown in listing 11.11.

Listing 11.11 The Tiles XML configuration file

```
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
       "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">
<tiles-definitions>
<definition name="definitionName" page="/layouts/myLayout.jsp">
  <put name="title"  value="Hello World" />
  <put name="header" value="/tiles/header.jsp" />
  <put name="footer" value="/tiles/footer.jsp" />
  <put name="menu"   value="/tiles/menu.jsp" />
  <put name="body"   value="/tiles/helloBody.jsp" />
   </definition>
  <!-- ... more definitions ... -->
</tiles-definitions>
```

An empty Tiles Definitions file is provided with the Blank application on the book's website [Husted].

NOTE The name and location of the Tiles Definitions file can be specified in the web application deployment descriptor (web.xml). We recommend creating a conf folder under WEB-INF to store the growing number of configuration files that a Struts application can use.

Extending Definitions

A Definition can be declared as a subclass of another Definition. In this case, the new Definition inherits all the attributes and properties of the parent Definition. The property `extends` is used to indicate the parent Definition:

```
<definition name="portal.page" extends="portal.masterPage">
     <put name="title"  value="Tiles 1.1 Portal" />
     <put name="body"   value="portal.body" />
</definition>
```

In this code segment, we specify a new Definition named `portal.page` that extends the Definition `portal.masterPage`. The new Definition inherits all the attributes and properties of its parent. In the preceding fragment, the attributes `title` and `body` are overloaded. This inheritance capability allows us to have root Definitions—declaring default attributes—and extended Definitions with specialized attributes (like `title` and `body`). If all your Definitions extend one root Definition, changing a value in the root Definition will change that value for all Definitions extended from that root

Extending is similar to overloading but adds persistence. Overloading describes the process where we specify a Definition and pass it new parameters (or attributes), rather like making a call to a method and passing it parameters. But a `<tiles:insert>` tag cannot call another `<tiles:insert>` tag, so the overloaded Definition cannot be referenced and reused. By using the `extend` property, you are creating a new Definition. This new Definition can then be inserted and overloaded and even extended by another Definition. It is still linked back to its ancestor Definitions through the usual type of inheritance tree, as shown in figure 11.5.

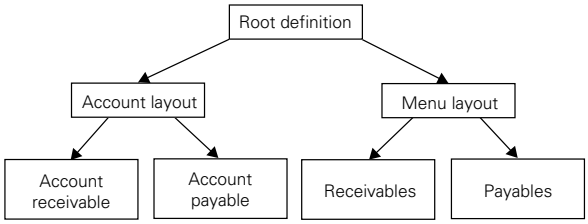

Figure 11.5 Definitions can be extended to create new Definitions.

Extending and overloading Definitions can dramatically reduce the amount of redundant information in your page declarations. Each markup, navigation, and content component in your website schema need be declared only once. The component can then be reused wherever it is needed.

While this is all quite cool, we would still need an extra page to host the Definition. This means to add a new page of content, we need to add the content page and then another page to insert the Definition that specifies the new content. A conventional application will have 60 pages for 60 pages of content. A templated application will use at least 120 pages to cover the same ground. Each of the

template pages are smaller and simpler than their conventional counterparts, but file management can be an issue.

A good solution to the page boom is to host the Definitions as Struts ActionForwards.

11.3.4 Using Definitions as ActionForwards

In a Struts application, most pages are not referenced directly but encapsulated by an ActionForward object. The ActionForward is given a unique logical name along with a URI that usually refers to a presentation page. A Struts Action selects and returns an ActionForward to the controller servlet. The ActionServlet then forwards control to the URI specified by the ActionForward's `path` property. (For more about ActionForwards, see part 1 of this book.)

The Tiles package includes an ActionServlet subclass that also checks the `path` property against your Definitions. If the Definition `id` and ActionForward `path` properties match, then the Definition bean is placed in the request scope, control forwarded to the layout, and your assembled templates are displayed.

Accordingly, you can define ActionForwards that use Definition names instead of URIs:

```
<action
  path="/tutorial/testAction2"
  type="org.apache.struts.example.tiles.tutorial.ForwardExampleAction">
  <forward
    name="failure"
    path=".forward.example.failure.page"/>
   <forward
    name="success"
    path=".forward.example.success.page"/>
</action>
```

Of course, you could also name your Definitions using the traditional slash instead of a dot:

```
<action
  path="/tutorial/testAction2"
  type="org.apache.struts.example.tiles.tutorial.ForwardExampleAction">
  <forward
    name="failure"
    path="/forward/example/failure.page"/>
  <forward
    name="success"
    path="/forward/example/success.page"/>
</action>
```

However, the second naming scheme could be confused with other identifiers related to Actions and page URIs. A good practice is to use dot separators (the first scheme) for the Tile Definitions and the slash separators for ActionForwards. This ensures that the names do not intersect.

The code in your Action is exactly the same as before. If you are switching from presentation page URIs in an ActionForward to a Tiles Definition, most Action classes would *not* need to be rebuilt. Typically, Action classes ignore the ActionForward path and just deal with the ActionForward by its name.

You can mix and match conventional ActionForwards and Tiles-Definition-ActionForwards in an application. The Tiles ActionServlet deals with each request on its own terms.

When ActionForwards are used this way, the number of template pages in an application drops dramatically. To host 60 pages of content, we just need 60 content pages, plus a small number of utility tiles to provide the standard navigation and layout features. But creating page 61 can mean creating only one more content-only JSP and one more XML Definition, with the latter often being a single line.

Deploying Tiles Definitions as ActionForwards gives you all the power and flexibility of dynamic templates without the usual red tape.

11.4 Tile attributes

Being able to extend Definitions and overload attributes is a very powerful feature. But so far, we've only shown examples where the attributes are static values hardcoded into the pages. What if you would like to specify an attribute at runtime? Shouldn't an Action be able to pass the value of an attribute to a tile?

Yes, it can. The secret is that Tiles stores the attributes in its own context. Just as JSP stores attributes in a page context, Tiles stores its attributes in a Tiles context associated with the user's request.

There is nothing mysterious about the Tiles context. It is simply a collection that Tiles creates in the request for the use of its components. Specialized contexts are a popular technique for managing the various objects components create and then share with others, especially when control may pass from one application layer to another. Several of the Tiles tags are designed to be used with the Tiles context, including `useAttribute`.

11.4.1 useAttribute

The `<tiles:useAttribute>` tag makes one of the Tiles context attributes available through the page context. This makes the attribute available to other tags that read the page context, like the Struts `<bean:write>` tag.

The same attribute name can be used in either context

```
<tiles:useAttribute name="myAttribute" />
```

or another name can be specified:

```
<tiles:useAttribute attributeName="anAttribute" name="myAttribute" />
```

Once Tiles has put the attribute into page scope, the Struts `bean` tag can refer to the message in the usual way:

```
<bean:write name="myAttribute" />
```

The `<useAttribute>` tag can also be used to declare a scripting variable for use in a JSP scriptlet:

```
<tiles:useAttribute id="list" name="myAttribute"
                    classname="java.util.List" />
```

In general, the `<useAttribute>` tag corresponds to the `<useBean>` action and the Struts `<bean:define>` tag but allows access to the attributes in the Tiles context.

Note that each tile is an individual JSP and therefore has its own page context. To export an attribute so that it is available to other tiles that may make up the completed response page, specify another scope. For example,

```
<tiles:useAttribute name="myAttribute" scope="request"/>
```

puts the attribute in request scope where a tag in another tile would be able to find it.

Since each tile is technically a different "page," each is in its own page scope. If you want to avoid conflicts with attributes on another tile, you can use page scope. If you want an attribute to be shared with another tile, you can use request scope instead.

The `<useAttribute>` operations are rendered sequentially. A tile rendering further down the page would be able to use the attribute, but one rendering earlier would not be able to find it (since it doesn't exist yet).

11.4.2 importAttribute

By default, the `<tiles:importAttribute>` tag imports all of the Tiles context attributes into the page context:

```
<tiles:importAttribute/>
```

Any and all attributes stored in the current Tiles context would now be available through the standard page context.

Optionally, a single attribute or another context may be specified:

```
<tiles:importAttribute name="myAttribute" scope="request"/>
```

But, unlike `<useAttribute>`, `<importAttribute>` does not support renaming attributes or exposing them as scripting variables.

11.4.3 put

The `<tiles:put>` tag is used to associate a value to an attribute. The `name` property will usually be specified as a simple string, but the value may be specified in a variety of ways: as a tag property, as a tag body, and as a JavaBean.

put as tag property

When used in a JSP, this is the most common form of the `put` command. The value is usually set using a simple string but can also be a runtime value:

```
<tiles:put name="title"  value="My first page" />
```

or

```
<tiles:put name="title"  value="<%=myObject%>" />
```

put as tag body

As is the case with many JSP tags, the value property can also be set as the tag's body:

```
<tiles:put name="title">My first page</tiles:put>
```

This approach can also be used to nest the output of other JSP tags:

```
<tiles:put name="title"><bean:write message="first.pageTitle"/></tiles:put>
```

put as a bean defined in some scope

Like many of the Struts tags, the attribute can be passed via a bean:

```
<tiles:put name="title" beanName="myBean" />
```

The object identified by `beanName` is retrieved, and its value used as the attribute value. If `myBean` is not a `String`, the Struts or Tiles JSP tags will automatically call the object's default `toString()` method so that the result of `myBean.toString()` will be used to set the `value` property.

By default, the scopes will be searched in the usual order, until the first instance of `myBean` is found. You may also specify a particular scope:

```
<tiles:put name="title" beanName="myBean" beanScope="session"/>
```

This would ignore any instance of `aBean` in the page or request scope and check the session instead. Tiles adds a context of its own, called `tiles`, that is checked after the standard scopes. The values accepted by `beanScope` are `page`, `request`, `application`, and `tiles`.

put as a property of a bean defined in some scope

Again like many of the Struts tags, you can also specify a certain property on a JavaBean. The `<put>` tag will then call that method to set the attribute value:

```
<tiles:put name="title" beanName="myBean" beanProperty="myProperty"/>
```

This would call the equivalent of `myBean.getMyProperty()` to set the value of the `title` attribute.

Specifying the attribute type

The `<put>` tag is used to set an attribute that will be used by another tag, usually either `<tiles:insert>` or `<tiles:get>`. The corresponding tag may use the value passed in various ways. It may represent a direct string, a page URL, or another Definition to insert. The type of data the attribute is meant to represent can be specified in the `put` tag with the optional `type` property. This can help the corresponding tag insert the value as intended:

```
<tiles:put name="footer" value="/tiles/footer.jsp" type="page"/>
```

The type, when specified, can be any one of the tokens: `string`, `page`, or `definition`. Table 11.2 provides a description of each of the tokens.

Table 11.2 Valid tokens for the `type` property of the `put` tag

Token	Description
`string`	The `value` property denotes a `String`.
`page`	The `value` property denotes a URL.
`definition`	The `value` property denotes a Definition name.

Specifying the security role

When container-based authentication is being used, you can also specify the role for a tile. If the user is not in the specified role, then the attribute value is not set. This allows you to specify a tile for each security role and let the framework select the appropriate one for the current user:

```
<tiles:put name="title"  value="myValue" role="myManager"/>
<tiles:put name="title"  value="myValue" role="myStaff"/>
```

If `<put>` is being used in a `<tiles:insert>` tag, the role is checked immediately. If `<put>` is being used within a `<tiles:definition>` tag, the role is checked when the Tiles context is initialized.

11.4.4 putList and add

In addition to accepting single objects, Tiles attributes can be of type `java.util.List`. You can specify a series of objects for an attribute and pass them as a single attribute. The `<tiles:add>` tag is nested within `<tiles:putList>` to specify the items to be placed on the list:

```
<tiles:insert page="menu.jsp" >
  <tiles:putList name="items">
    <tiles:add value="home" />
    <tiles:add value="documentation"/>
  </tiles:putList>
</tiles:insert>
```

The `<putList>` tag is often used with the `<useAttribute>` or `<importAttribute>` tag to make the list accessible to other tags on the page:

```
<tiles:importAttribute/>
<TABLE>
<logic:iterate id="item" name="items" >
<TR>
  <TD>
    <bean:write name="item">
  </TD>
</TR>
</logic:iterate>
</TABLE>
```

11.5 Migrating an application to Tiles

At the beginning of this chapter we portrayed Tiles as a means of making your applications more consistent and easier to use. We also mentioned that consistency is a hallmark of good design. This is because consistency implies reuse. Reuse leads to applications that are hard to break and easy to maintain.

Often, you will have an existing application that you would like to adapt for Tiles. You may want it to improve the look so it is more consistent, or you may want to improve its functional design—or both. Improving the design of existing software is called *refactoring* [Fowler].

If you are familiar with conventional refactoring techniques, migrating an application to Tiles resembles the *Extract Method*. The tiles equate to methods that are called to render the page. When we are done, pages are reduced to a layout and a punch list of tiles to call. This works much like the central method in a class that just calls one member method after another.

Once we have extracted our tiles and reduced the page to a punch list, we can take the process a step further and replace the page with an XML element. Struts can use this element to render the tiles directly without bothering with a punch list page. But like all good refactorings, ours begins with the smallest step.

11.5.1 Setting up the Tiles framework

First, make an extra backup of everything, regardless of how many backups are made in the normal course. Migrating to Tiles can be tricky at first, and, realistically, you may need to make more than one pass before everything clicks into place. So be ready to roll back and try again. An extra backup on your desktop may be enough to lure you away from a three-hour debugging session spurred by some incidental typo.

Struts 1.0

If you haven't done so, the first step is to install the Tiles 1.0 package and load the Tiles servlet through your application's deployment descriptor. (See the package for details.) The 1.0 version of Artimus (see chapter 16) is based on Tiles and can be used for a working example. Then, test your application to be sure all is well by clicking through a few pages.

Also, be sure to set up a skeleton Tiles Definitions XML document so you can start filling it out as the refactoring proceeds.

Struts 1.1

Tiles is integrated with Struts 1.1. We cover the steps for enabling Tiles in Struts 1.1 in section 4.8 of this book.

11.5.2 Testing the default configuration

Set the `debug` and `detail` parameters in the deployment descriptor (web.xml) to level 2, and restart the application. Check the log entries carefully for any new error messages. Run any unit tests and click through the application to confirm that operation is still nominal.

11.5.3 Reviewing the pages

With Tiles up and running, the next thing is to take a good hard look at your pages and determine the overall layout types and the regions within each layout. It's also time to start thinking about a naming scheme. Each component will need its own identifier.

Identifying the layouts

When you look through an application, you will find varieties of menu or dialog pages—list pages, view pages, and edit pages—among others. Here the focus should not be on what the pages contain but how the parts of the page fit together. Does it have a header or a footer? Does it have a menu? Is the menu along the side or at the top? At this point, the relative position of elements like the navigation bar is more important than the actual links on the bar.

From this group, try to identify some common layouts. Again, the focus is on the visual layouts used throughout your application, not on the page content.

Identifying the tiles

Next take a closer look at each of the common layouts and identify the individual tiles. Determine how the parts of the page fit together. Then take a look at the page itself and identify the individual tiles. The presentation code for each tile will be saved as a separate file. We can use it in multiple layouts or edit it without touching the other tiles.

As mentioned, this process is similar to applying the Extract Method to a large Java class. One clue is to look for any existing comments in the presentation code, like `<!-- menu starts here -->`. A block of markup prefaced with a comment is often a tile waiting to happen, like the block shown in listing 11.12.

Listing 11.12 A good candidate for a tile

```
<%-- messages --%>
<TR class="message">
<TD colspan="3">
<logic:messagesPresent>
   <bean:message key="errors.header"/>
   <html:messages id="error">
      <LI><bean:write name="error"/></LI>
   </html:messages>
   <bean:message key="errors.footer"/>
</logic:messagesPresent>
</TD>
</TR>
```

Style changes in a page are also a good indicator of a potential tile. If the designer put in a style to set off a portion of the page, that portion of the page may make for a good tile.

The best candidate is a block of self-contained code with a single, coherent purpose—again, not unlike a Java method.

If the presentation code does not have comments like these, it can be helpful to add them to some representative pages before making a pass to extract the tiles. If a page segment looks like a tile, except that each page prints something different on the tile, don't despair. Tiles can also pass string constants to a tile so that the rest of the markup can be reused. For now, just put in a marker, like `${subtitle}`, where the replacement string would go.

Naming the candidates

It can help to start working on a naming scheme early in the process. What we call a component can crystallize its purpose. We will need names for the individual regions, the general layouts, and each page in the system.

Table 11.3 shows the Tiles nomenclature for these entities: tile, layout, and definition.

Table 11.3 Tiles nomenclature

Entity	Description
Tile	A reusable page fragment with HTML markup and JSP code.
Layout	A JSP that describes where to position the tiles on a page.
Definition	A JavaBean that represents a particular page. The Definition combines a layout with a set of tiles and other runtime options to generate a distinct page. The Definition may be expressed as a JSP or contained in an XML configuration file.

The *tiles* are coherent fragments of text and markup that you will extract from your existing pages. Tiles can host static content or markup, or a mix of both. Most tiles represent navigational controls and other common assets and will be reused between pages. Common assets might include a table that positions a logo somewhere on a page or a set of buttons, like Save and Cancel, that are used on several forms. If a style is already associated with the asset, consider naming the tile after the style.

DEFINITION *Markup* is the collection of commands placed in a file to provide formatting instructions rather than visible text or content. HTML uses a tag-based markup system.

Other tiles will contain content that is used on a single page. These tiles are often found at the center of the page surrounded by other tiles that provide the menu bars and other HTML chrome.

DEFINITION The *chrome* is that part of the application window that lies outside a window's content area. Toolbars, menu bars, progress bars, and window title bars are all examples of elements that are typically part of the chrome. HTML chrome is created by markup and lies within the application window but serves the same purpose.

You can use the same conventions to name the tiles that you would use for any HTML or JSP. It can be useful to separate the shared tiles from those designed for a particular page. An easy way to do this is to open a tiles folder next to the folder you would usually use for your pages:

```
/pages
  ./article/Form.jsp
  ./channel/Channels.jsp
/tiles
  ./header.jsp
  ./message.jsp
```

If a file resides in the *tiles* folder, it means that the file is meant to be used in more than one Definition. If a file resides in the *pages* folder, it means that the file contains unique content and is not expected to be shared.

The *layouts* are the containers for the tiles that describe where each component appears on the page. The layouts can be reused between Definitions. To keep everything together, you can create a layouts subdirectory beneath the tiles folder:

```
/tiles
  ./header.jsp
  ./message.jsp
  ./layouts
     ./Base.jsp
     ./Remote.jsp
```

The *Definitions* are stored as elements in an XML document. Each Definition element needs its own name. In practice, the Definitions share namespace with the

ActionForwards. The scheme should ensure that the ActionForward and Definition names do not collide. It can be helpful to group Definitions together in a directory/subdirectory type structure, so a name separator is also needed. One convention is to use a dot (.) for a separator in Definition names where many people might use a slash (/) in an ActionForward. To be sure there is no conflict with file system names, a leading dot can also be used where a leading slash would usually go:

```
<definition name=".account.logon"> . . . </definition>
```

Of course, any convention will do. You could use @ signs instead of dots, or preface each Definition with TILES: and then use slashes. Since these are logical references, any valid URI will do.

Once you have a general idea of what tiles, layouts, and Definitions you will need, and what you will call them, the next step is to extract the tiles and refactor a page.

11.5.4 Refactoring a page with <tiles:insert>

As with any refactoring, it's best to start slow, making one small change after another, until the first iteration is complete. As the process continues, you can take larger and larger steps, building on the prior work (and learning from your mistakes).

In most cases, the goal will be to reduce the pages to a Definition that can be declared in the Tiles configuration file and called from within an ActionForward (see section 11.3 for details). This approach saves creating an extra punch list page to insert the tiles. But to get started, it's simplest to build a page using the `<tiles:insert>` tags.

To get off on the right foot, remember to:

- Select a good starter page.
- Thoughtfully extract the tiles from that page.
- Develop a set of good (and bad) practices along the way.

Selecting a good starter page

It's best to start with a simple page, extract the common components, and insert them one at a time back into the original page. Your application's welcome or logon page can be a good candidate. These tend to be relatively simple pages with a nice mix of reusable and custom content. An interior page can also be a good choice, if it does not contain too much chrome. Listing 11.13 shows an interior page from the Artimus example application [ASF, Artimus] before it was migrated to Tiles.

Listing 11.13 Our starter page: /pages/View.jsp

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/request" prefix="req" %>
<!-- HEADER -->
<HTML>
<HEAD>
<html:base/>
<LINK rel="stylesheet" type="text/css" href="<html:rewrite
   forward='baseStyle'/>">
<TITLE>Artimus - Article</TITLE>
</HEAD>
<BODY>
<TABLE class="outer">
<TR>
<TD>
<TABLE class="inner">
<!-- MESSAGE -->
<TR>
<TD class="message" colspan="3" width="100%"><html:errors/></TD>
</TR>
<TR>
<TD class="heading" colspan="3">
<H2><bean:write name="articleForm" property="title"/></H2></TD>
</TR>
<TR>
<TD class="author" colspan="3">by <bean:write name="articleForm"
   property="creator"/>
</TD>
</TR>
<TR>
<TD class="article" colspan="3">
<bean:write name="articleForm" property="content" filter="false"/></TD>
</TR>
<%-- CONTRIBUTOR PANEL --%>
<req:isUserInRole role="contributor">
<TR>
<TD colspan="3"><HR /></TD>
</TR>
<TR>
<%-- DELETE --%>
<logic:equal name="articleForm" property="marked" value="0">
<html:form action="/admin/article/Delete">
<TD class="input"><html:submit >DELETE</html:submit></TD>
<html:hidden property="article"/>
</html:form>
</logic:equal>
<%-- RESTORE --%>
<logic:equal name="articleForm" property="marked" value="1">
```

```
<html:form action="/admin/article/Restore">
<TD class="input">
<html:submit>RESTORE</html:submit>
</TD>
<html:hidden property="article"/>
</html:form>
</logic:equal>
<html:form action="/admin/article/Edit">
<TD class="button" colspan="2">
<html:hidden property="article"/>
<html:submit>EDIT</html:submit>
<html:cancel>CANCEL</html:cancel>
</TD>
</html:form>
</TR>
</req:isUserInRole>
<!-- NAVBAR -->
</TABLE>
</TD>
</TR>
<TR>
<TD class="navbar">
<html:link forward="done">DONE</html:link>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

Once you've selected your starter page, go through it and extract each logical block into its own tile and insert it back again. After each extraction, test the page to be sure it still renders. Listing 11.14 shows a fragment extracted into its own tile.

Listing 11.14 An extracted tile: /tiles/header.jsp

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<HTML>
<HEAD>
<html:base/>
<LINK rel="stylesheet" type="text/css" href="<html:rewrite
  forward='baseStyle'/>">
<TITLE>Artimus - View Article</TITLE>
</HEAD>
<BODY onload="document.forms[0].elements[0].focus();">
<!-- OUTER TABLE -->
<TABLE class="outer">
<TR>
<TD align="center">
```

```
<!-- INNER TABLE -->
<TABLE class="inner">
<TR>
<TD class="navbar" colspan="3">View Article</TD>
</TR>
```

Listing 11.15 shows how the View.jsp can include the Header tile again after it has been extracted.

Listing 11.15 Inserting an extracted tile: /pages/article/View.jsp

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/struts-bean" prefix="bean" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<%@ taglib uri="/tags/request" prefix="req" %>
<!-- HEAD -->
<tiles:insert page="/tiles/header.jsp"/>
<!-- MESSAGE -->
<TR>
<TD class="message" colspan="3" width="100%"><html:errors/></TD>
</TR>

<!-- ... -->

</HTML>
```

As soon as your first tile is extracted and inserted back, be sure to test opening the page before moving on to the next tile.

When this process is complete, the page will consist of a series of insert tiles, as shown in listing 11.16.

Listing 11.16 A refactored page: /pages/View.jsp (completed)

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:insert page="/tiles/header.jsp"/>
<tiles:insert page="/tiles/message.jsp"/>
<tiles:insert page="/tiles/view.jsp"/>
<tiles:insert page="/tiles/navbar.jsp"/>
```

If the text in some of the tiles has to be customized, say with the page title, you can use the `<tiles:put>` tag to send a custom value along to the tile. The `<tiles:getAsString>` tag can then write it out when the page renders. Listing 11.17 shows

how to send the text to the tile; listing 11.18 shows how to write the dynamic text out again.

Listing 11.17 Inserting dynamic content: /pages/View.jsp (revised)

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:insert page="/tiles/header.jsp">
<tiles:put name="title" value ="Artimus - View Article"/>
<tiles:put name="subtitle" value ="View Article"/>
</tiles:insert>
<tiles:insert page="/tiles/message.jsp"/>
<tiles:insert page="/tiles/view.jsp"/>
<tiles:insert page="/tiles/navbar.jsp"/>
```

Listing 11.18 Writing dynamic content with getAsString

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<HTML>
<HEAD>
<html:base/>
<LINK rel="stylesheet" type="text/css" href="<html:rewrite
   forward='baseStyle'/>">
<TITLE><tiles:getAsString name="title"/></TITLE>
</HEAD>
<BODY onload="document.forms[0].elements[0].focus();">
<!-- OUTER TABLE -->
<TABLE class="outer">
<TR>
<TD align="center">
<!-- INNER TABLE -->
<TABLE class="inner">
<TR>
<TD class="navbar" colspan="3"><tiles:getAsString name="subtitle"/></TD>
</TR>
```

If you prefer, the Struts `<bean:write>` or `<bean:message>` tag can also be substituted for `<tiles:getAsString>`. The attributes are being saved to the standard contexts, so any standard tag will work as well.

Extracting tiles

The main work of the refactoring is to determine which part of the page is part of which tile, moving that fragment into its own file and then inserting it back again. Here's a step-by-step checklist:

1. Select and cut the block.

2. Open a new file.
3. Paste in the block.
4. Save and name the file as a JSP (or HTML file if you can).
5. Insert any taglib import statements the tile requires.
6. Close the new tile.
7. Place a `<tile:insert page="/path/to/new/page"/>` tag where the segment used to be.

NOTE The Tiles Definitions represent the set of pages that were in use before the refactoring began. The pages are being composed differently, but the content and appearance should remain the same. Most often, the Definition includes a body or content tile as part of the Definition. Some distinct pages may not have a unique *tile* but are represented by a unique *list* of shared tiles. You may have a form that uses different buttons under different circumstances. One page may include the set of buttons for creating a new record. Another page may include the set of buttons for updating a record. But other pages may share the sets of buttons and other tiles in the Definition. Distinct pages may also be created from the same Definition by passing a title string and the dynamic content retrieved from a data service. You might have several different ways of searching for records but display them all using the same Definition.

Extraction practices

Here are some caveats and practice notes regarding the extraction process. Since this will become a routine, it is important to have clear practices and to learn from past mistakes:

- All custom tags used by the tile must be imported into the tile.
- All custom tag elements must begin and end in the same tile.
- Avoid carrying HTML elements over to another tile.
- Mark up by contract.
- Consider trade-offs.
- Leverage technology.

All custom tags used by the tile must be imported into the tile. The tile will inherit HTML assets, like style sheets. It will not inherit references to JSP assets, like tag libraries. The web browser applies the style sheet when the page renders, but

since each tile is in fact a stand-alone JSP servlet, it needs its own references to JSP resources.

All custom tag elements must begin and end in the same tile. If you use the `<html:html>` tag in your file, place the elements for this tag at the top and bottom of your layout (which is a single tile). This restriction does not apply to HTML elements. You can, for example, open the `<BODY>` element in a header tile and close the `</BODY>` element in a footer tile, but custom tags are validated when the tile is compiled. JSP tag elements must begin and end within the same tile.

Avoid carrying HTML elements over to another tile. In practice, you may decide that one tile should open an element, like a table; a second should provide the content, like the rows in a table; and a third should close the element. When this pattern is useful, you should use it but still try to begin and end as many elements as possible in the same tile. This makes finding markup errors much easier. Even if the middle tile only provides the rows for a table, it can do so as complete `<TR>...</TR>` rows.

Mark up by contract. When you decide to use something like Tiles, you are also deciding to treat markup like code. Accordingly, all the usual paradigms apply, like assertions and program by contract. When you design your tiles, think about the preconditions and postconditions for each tile, just as you would for a method in a class. Don't hesitate to document the conditions for each tile, just as you would for a method. A good approach is to just use standard JavaDoc conventions in a JSP comment, which avoids creating a new wheel:

```
<%--
/**
 * Global page header, without automatic form select.
 * See headerForm.jsp for version with form select.
 * Imports global style sheet.
 * Opens HEAD, BODY, and TABLE. Another tile must close these.
 * @author Ted Husted
 * @license ASF 1.0
 */
--%>
```

For HTML tiles, just use HTML comment braces instead. Unlike JSP comments, HTML comments will be visible in the source of the page. You may wish to be brief and discreet when using HTML comments.

Consider trade-offs. Dissembling a page into tiles is much like normalizing a database. You can strip out absolutely all the redundant information, but if you do,

some of the parts become so small that maintenance and performances issues can appear. In practice, you may have two or three varieties of header or footer files that may replicate markup. But if the markup changes, it is still easier to conform 3 files than 30 or 300. As with any programming task, the usual trade-offs apply.

Leverage technology. If you are also introducing style sheets and other markup changes at the same time, don't hesitate to create skeleton files and use editing macros with search-and-replace to automate some of the work. It's easy to forget how much time you can save with these old friends.

11.5.5 Extracting the <tiles:insert> tags into a Definition

After you have applied the process in section 11.5.3 to your starter page and have confirmed that it still works, you can finish the job by extracting the `<tiles:insert>` tags into an XML Definition. This is a four-step process:

1. Move the page to the layouts folder.
2. Rename the body tile.
3. Convert the insert tag to a layout and Definition.
4. Update the ActionForward.

Move the page to the layouts folder

Move the refactored page to the location you've chosen for layout tiles, under /tiles/layouts, for example. At this point, the refactored page should be a punch list of `<tile:insert>` tags, and the original content of the page should have been reduced to a set of tiles.

Rename the body tile

One of the extracted tiles probably represents the body, or "guts," of the original page. If so, consider renaming it as the original page you just moved. The implication is that the core content of the original page will still be where the original page stood. This helps to minimize change. If you need to edit the page's content, the content will still be where it always was. If you do this, update the `<tiles:insert>` tag after moving or renaming the file. Working from listing 11.16, we would change

```
<tiles:insert page="/tiles/view.jsp"/>
```

to

```
<tiles:insert page="/pages/view.jsp"/>
```

Convert the insert tag to a layout and Definition

Add a `name` property to each of the insert tags. This will often match the name of the JSP. The exception might be the tile representing the original body of your page. This will usually have a more generic name, like *content*.

Copy the `<tiles:insert>` statements from your refactored page into the starter tiles.xml configuration file that we set up at the beginning of this section and place them inside a `<definition>` element. If you used any `<tiles:put>` elements, you can promote those to top-level elements now:

```
<definition>
      <tiles:insert put="title"    value ="Artimus - View Article"/>
      <tiles:insert put="subtitle" value ="View Article"/>
      <tiles:insert name="header"   page="/tiles/header.jsp"/>
      <tiles:insert name="message"  page="/tiles/message.jsp"/>
      <tiles:insert name="content"  page="/pages/view.jsp"/>
      <tiles:insert name="navbar"   page="/tiles/navbar.jsp"/>
  </definition>
```

Then, rename the `<tiles:insert>` tags as `<tiles:put>` elements and the `page` attribute as a `value` attribute. To the `<definition>` element, add `name` and `path` properties. The `path` property should be to your layout page (see step 1). The `name` property can correspond to the name of your original page, but substitute dots for the slashes. Listing 11.19 shows our complete tiles.xml file.

Listing 11.19 A Tiles configuration file: /WEB-INF/conf/tiles.xml

```
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration//EN"
       "http://jakarta.apache.org/struts/dtds/tiles-config.dtd">

<tiles-definitions>
    <definition name=".article.view" path="/pages/tiles/layouts/Base.jsp>
      <tiles:put name="title"    value ="Artimus - View Article"/>
      <tiles:put name="subtitle" value="View Article"/>
      <tiles:put name="header"   value="/tiles/header.jsp"/>
      <tiles:put name="message"  value="/tiles/message.jsp"/>
      <tiles:put name="content"  value="/pages/view.jsp"/>
      <tiles:put name="navbar"   value="/tiles/navbar.jsp"/>
   </definition>
</tiles-definitions>
```

Now, in the layout page, change the `<tiles:insert>` tags to `<tiles:get>` tags and delete the `page` attribute (since that is now part of the Definition)

Any `<tiles:put>` tags can be changed to `<tiles:useAttribute>` tags. Keep the `name` attribute, but delete the `value` attribute (since the value is part of the Definition now).

Listing 11.20 shows a complete layout JSP.

Listing 11.20 A Tiles layout page: /tiles/layouts/Base.jsp

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<tiles:useAttribute name="title"/>
<tiles:useAttribute name="subtitle"/>
<tiles:get name="header">
<tiles:get name="message"/>
<tiles:get name="content"/>
<tiles:get name="navbar"/>
```

Update the ActionForward

Finally, replace the ActionForward that referred to the original JSP with a reference to the Tiles Definition:

```
<action
        path="/article/View"
        type="org.apache.scaffold.struts.ProcessAction"
        parameter="org.apache.artimus.article.FindByArticle"
        name="articleForm"
        scope="request"
        validate="false">
       <forward
            name="success"
            path=".article.View"/>
</action>
```

As shown in figure 11.6, the Tiles ActionServlet will intercept the reference and apply the Definition to the layout page.

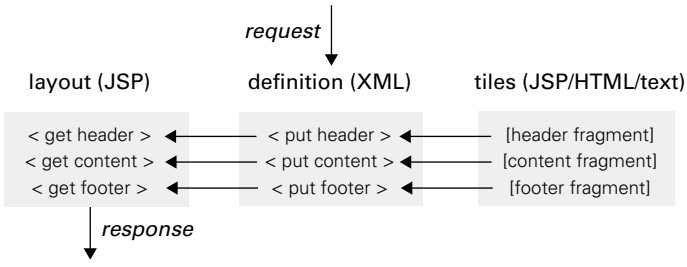

Figure 11.6 The Tiles ActionServlet uses the definition to help create the response.

If you were forwarding directly to the JSP before, you can use the standard Scaffold SuccessAction to route control through the controller so the Definition can be used to select and render the layout. You can use the Definition anywhere you were using a system path to the JSP. If there are enough references, you can even use the search-and-replace feature of your editor to change them all automatically.

At runtime, the Tiles ActionServlet will intercept the ActionForward and check its path against the Definitions in the Tiles configuration. If it finds a match, it includes each tile of the Definition in the response. The container then processes each included tile normally. The HTML tiles are rendered by the container's HTML service, and the JSP tiles are rendered by the container's JSP service.

If the ActionForward path is not the name of a Definition, the ActionServlet handles it as a normal URI, same as always.

NOTES If your refactored page does not display the same as the original, first make sure that you have imported any needed tag libraries in the tiles. If the taglib is not included, the tag will not be rendered and will be ignored by the browser (and you will see it in the HTML page source). If that is not the problem, create a new page and start reassembling the page by hand to uncover the error. Most often, you will find that a tile broke its "API contract" by opening or closing an element improperly. Another API contract to check is the path for the input property of the ActionMapping. This should also point to the Definition name rather than the physical JSP page.

If you expose the *Path must be absolute* error after switching over to Tiles, it means that you've tried to use a Definition as the path for a forward but it was not found in the Tiles configuration. After checking for a Definition, Tiles passes the path up to the super class method, and Struts treats it like a system path. Our leading dot is being interpreted as a relative reference to a directory, hence the *Path must be absolute* advice. The bottom line is there is usually a typo in either the Struts or Tiles configuration file.

To test your changes, be sure to reload the application so that the current Struts and Tiles configurations are loaded into memory. If you've been following along, you can try this now.

Once you have gone through the process of refactoring a page with `<tiles:insert>` and then converting it to a Definition, you may wish to convert other pages directly to a Definition. To do this, you:

- Copy an existing Definition using the same layout and give it a new name.
- Clip out and save the content segment of the page you are refactoring.
- Change the new Definition to refer to the segment you just saved and converted.
- Test and repeat.

NOTE When you first switch a block of pages over to Tiles, it may take a little extra time for the pages to render at first. This is because the JSPs for the new tiles are being created *in addition to* the usual one you need for the content. Once the JSP for each tile is created, it will not be re-created until it is changed, and everything goes back to normal. In the future, if you edit the content tile, only that one JSP will be recompiled.

11.5.6 Normalizing your base layout

Back in listing 11.20, we showed the layout file as a series of `<tiles:insert>` tags. If you like, you can also use regular HTML and JSP code on your layout page. This is a good place to put the topmost tags, like `<HTML>` or `<html:html>`, so that these do not need to be put inside other tiles, which might really be intended to do other things.

Listing 11.21 shows a revised base layout page that extracts the topmost elements from the header and navbar tiles and puts them on the base layout. We've also renamed it from Base.jsp to Article.jsp to indicate its role as the Article layout tile. Other pages in the application may need to use a different layout.

Listing 11.21 Revised layout tile (/tiles/layouts/Article.jsp)

```
<%@ taglib uri="/tags/tiles" prefix="tiles" %>
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<html:html>
<HEAD>
<html:base/>
<LINK rel="stylesheet" type="text/css" href="<html:rewrite
   forward='baseStyle'/>">
<TITLE>Artimus - <bean:write name="title"/></TITLE>
</HEAD>
<tiles:useAttribute name="title"/>
<tiles:get name="header"/>
<tiles:get name="message"/>
<tiles:get name="content"/>
<tiles:get name="navbar"/>
</BODY>
</html:html>
```

11.5.7 Refining your Definitions into base and extended classes

As you commute your pages to layouts and Definitions, it's easy to end up with sets like this:

```
<definition name=".article.View" path="/tiles/layouts/Article.jsp">
  <put name="title"       value="View Article" />
  <put name="header"      value="/tiles/header.jsp" />
  <put name="messages"    value="/tiles/messages.jsp" />
  <put name="content"     value="/pages/articles/view.jsp" />
  <put name="navbar"      value="/tiles/navbar.jsp" />
 </definition>

<definition name=".article.View" path="/tiles/layouts/Article.jsp">
  <put name="title"       value="Search Result" />
  <put name="header"      value="/tiles/header.jsp" />
  <put name="messages"    value="/tiles/messages.jsp" />
  <put name="content"     value="/pages/articles/result.jsp" />
  <put name="navbar"      value="/tiles/navbar.jsp" />
 </definition>
```

If you look closely, you'll see that the first and third items differ but the others are the same. A better way to write them, and any like them, is to create a base Definition. The Tiles Definition supports an `extends` property, which lets a Definition inherit attributes and overload only what needs to be changed. Here, we extend `.article.Base` and overload `title` and `content`:

```
<definition name=".article.Base" path="/tiles/layouts/Article.jsp">
      <put name="title"     value="${title}"/>
      <put name="header"    value="/tiles/header.jsp"/>
      <put name="message"   value="/tiles/message.jsp"/>
      <put name="content"   value="${content}"/>
      <put name="navbar"    value="/tiles/navbar.jsp"/>
   </definition>
   <definition name=".article.View" extends=".article.Base">
      <put name="title"     value="View Article"/>
      <put name="content"   value="/pages/article/view.jsp"/>
   </definition>
   <definition name=".article.Result" extends=".article.Base">
      <put name="title"     value ="Article Search Result"/>
      <put name="content"   value="/pages/article/result.jsp"/>
   </definition>
```

With the base Definition in place, we now have to supply only two lines for each of our subDefinitions. The other settings fall through and do not need to be specified. If there are attributes that will be used throughout your site, you can put those in a base Definition and extend everything else from that. Then, if any of the base attributes change, you need to make the change in only one place.

As a convention, we put markers in for the values of the first and third items (`title` and `content`) to indicate that these are extension points that subDefinitions need to override. If the base Definition were used directly, then these markers would just print out as literals. The `${}` markers have no special meaning to Tiles.

Another convention shown here is to use an initial capital letter for the layout JSP but an initial lowercase letter for the tile JSPs. This is to indicate that the layout page can be called directly, because it is a fully formed JSP class. The tile pages are like methods invoked by the layout JSP, and so use the same naming convention as a method. But this is only a convention; any other consistent naming scheme would work just as well.

11.5.8 Developing a routine

After the first few pages, you should be able to develop a routine that goes something like this:

1. Create a new Definition (in tag.xml), often by copying a similar one.
2. Update the Definition with the path to the existing page, page title, and any other custom information.
3. Open the existing page.
4. Delete the top and bottom, leaving the core content and tag import statements.
5. Review and revise the core content to ensure that the markup keeps its API contract with the tiles before and after it in the Definition. One tile may need to open an element, like a `<TABLE>`, and another tile may need to close it. Remove any unneeded tag import statements. Optionally, add a comment block.
6. Update the paths in the Struts configuration (struts-config.xml) to reference the new Definition, including any input properties.
7. Reload the tag configuration and the Struts configuration.
8. Review the page.
9. Rinse and repeat.

At first, you will probably start with pages as they appear in your application's flow. Once you have the procedure down, it is not difficult to step through the page tree and refactor each page in turn. This will ensure that you don't miss any. It may also uncover some obsolete pages left over from prior development efforts.

DEFINITIONS A popular way to view a method's signature is as a *contract* between the method and its caller. The caller agrees to provide certain parameters and the method agrees to provide a certain result based on those parameters. Since an API is a collection of method signatures, the paradigm of seeing interactions between components as a binding agreement is generally referred to an *API contract*.

API is an acronym for application programming interface, any set of routines generally available for use by programmers. The operating system, for example, has APIs for a variety of disk/file-handling tasks. APIs are written to provide portable code. The programmer only has to worry about the call and its parameters and not the details of implementation, which may vary from system to system. [CKNOW]

11.5.9 Managing the migration

Moving an application over to Tiles is not difficult but neither is it trivial. Be sure to schedule enough time for the project. The first few pages may take several hours, but once the pattern is established, additional pages may take only a few minutes each.

If you also need to apply a new look to the site, it may be best to first convert it to Tiles and then apply the new design so that you are not doing two new things at once. Once the layout is migrated to Tiles, bringing up the new design will go more quickly.

A good time to schedule a migration is when you know a visual redesign is coming but don't have the new design in hand yet. If you can have the application migrated to Tiles while the redesign is being finalized, applying the changes will be a smoother process. Doing both at once is not recommended—especially for your first migration.

So what's the bottom line of a migration? Your mileage will vary, but a small application with 25 presentation pages consuming about 140,000 kbytes of markup code was migrated to 55 tiles of about 120,000 kbytes—the 15% difference being redundant markup that was removed.

Moving forward, new pages for the application can now be created much more quickly and will be more consistent with existing pages. To change the overall layout, you can edit the layout or a few individual tiles instead of every page on the site.

11.6 Summary

Dynamic template systems, like the Tiles framework, can bring familiar programming patterns to the presentation layer of a web application. They let us slice and dice the HTML markup and JSP commands into manageable pieces. We can then assemble a page by calling the individual pieces, or tiles, the way we would call Java methods to perform a larger process. A tile encapsulates a block of markup, much like a method encapsulates a block of Java code.

The pages of a web application are built around a common look and feel, or layout, that helps users navigate the site. When we assemble pages using Tiles, we start with a base layout that defines where the tiles are placed and gives each position a logical name. The paths to the tiles a particular page uses can then be passed at runtime. A typical page may use five or six tiles, with only one or two of those changing from page to page.

In Tiles, a complete page, including the layout and the paths to its tiles, can be represented as an object called a Definition. To assemble a particular page, we can simply refer to its Tiles Definition. Like the Struts framework components, the Definitions can be configured using an XML document and loaded at startup.

The Struts framework uses ActionForwards to encapsulate paths to system resources, including presentation pages and Action classes. The framework's controller, the ActionServlet, uses ActionForwards to route control. The other components refer to the ActionForward by name and rely on the controller to invoke the resource indicated by the forward's path.

A standard extension to the controller allows Definitions to be used as the ActionForward path. When a Definition is the target of an ActionForward, the controller includes the fragments in a combined response. The standard service for each fragment then finishes the job.

When an application is migrated to Struts, one consequence is that system paths become encapsulated in the Struts configuration. The presentation pages can then refer to other resources using a logical name rather than an actual path or URI. This lets us focus on what the page wants to do rather than the particulars of how it is done.

In the same fashion, when an application is migrated to Tiles, the configuration encapsulates the system paths. Struts can then refer to the page Definition by name and leave the particulars of assembling the page to the Tiles framework.

Tiles can be of most use to larger systems with dozens or hundreds of pages. Like decomposing a process into constituent methods, refactoring an application to use Tiles creates more component parts, but each individual part is simpler to

understand and maintain. More important, the individual tiles can be reused, avoiding the need to make the same change in multiple places. As applications grow, the need to eliminate redundancy becomes increasingly important.

The focus of this chapter has been to provide you with enough information to put Tiles to work in your application. But it by no means covers everything that is possible with the Tiles framework. The Artimus example application (see chapter 15) is based on Tiles and demonstrates several of the best practices described in this chapter.

In the next chapter, we explore another "optional" component, the Struts Validator.