Tiles Advanced Features

Author :
Cedric Dumoulin (cedric.dumoulin@lifl.fr)
Date : 11 Feb. 2002
Rev. : 5 Nov. 2002
Draft Version v0.6

Reviewers: Holman Cal (cal.holman@west-point.org)
Yann Cébron (yannc76 at yahoo.de)

1 Introduction

Large web sites often need a common Look and Feel (L&F). If the L&F is hard coded in all pages, changing it becomes a nightmare: you would have to modify nearly all pages. In a good design, we want to separate the L&F and page content. Using a template allows to define a master L&F: position of header, menu body, content, and footer. The page content is defined in a JSP page without worrying about the L&F. The final page is built by passing header, menu, body and footer to the template which in turn builds them as requested. Header, menu and footer can be the same for all final pages. In addition using a stylesheet allows a consistent policy of formats and colors.
Another aspect to web development is often redoing the same things: websites contain many common parts: menus, forms, shopping cart, etc… Each time we have to rewrite or copy & paste the same code. But what happens when you improve the code, or when you discover a bug? You need to modify all your copy & pasted sections ! A solution is to have reusable components that you insert where you need them. You always insert the same components, but with different data. Now, if you modify the component, the modification is spread everywhere you use it is used: you have a central point for modification.
Tiles allow both templating and componentization. In fact, both mechanisms are similar: you define parts of page (a "Tile") that you assemble to build another part or a full page. A part can take parameters, allowing dynamic content, and can be seen as a method in JAVA language.

Note: All examples in this chapter are based on new examples located in the */examples* directory of the Tiles distribution WAR file.

2 Installing Tiles

The Tiles installation process depends on the Struts version you use. If you start a new project, use the latest Struts version. Tiles can also be used without Struts.

To enable Tiles definitions described in one or more files, you need to write these definition's files and to initialize the definition factory. If you don't use definitions, you don't need to initialize the factory.

2.1 Struts1.1

2.1.1 Required Files

Required files are:

- *struts.jar* – in *WEB-INF/lib/.* Tiles are now in the main Struts distribution.
- *tiles.tld* – in *WEB-INF/*
- *all commons-*.jar files needed by Struts* – in *WEB-INF/lib/*

All these files should come with the Tiles or Struts distribution. They are normally located in *WEB-INF/lib/* for .JAR files and *WEB-INF/* for .TLD files.

2.1.2 Enabling Definitions

Use the Tiles plug-in to enable Tiles definitions. This plug-in creates the definition factory and pass it a configuration object populated with parameters explained here after. Parameters can be specified in the web.xml file or as the plug-in parameters. The plug-in first read parameters from web.xml, and then overload them with the one found in the plug-in. All parameters are optional and can be omitted. The plug-in should be declared in each struts-config file:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
  <set-property property="definitions-config"
                       value="/WEB-INF/tiles-defs.xml,
                           /WEB-INF/tiles-tests-defs.xml,/WEB-INF/tiles-tutorial-defs.xml,
                           /WEB-INF/tiles-examples-defs.xml" />
  <set-property property="moduleAware" value="true" />
  <set-property property="definitions-parser-validate" value="true" />
</plug-in>
```

- `definitions-factory-class`: (optional)
  - Name of the class used as definitions factory. User can develop its own definitions factory, and specify it here. If not specified, use the I18n factory.
- `definitions-config`: (optional)
  - Specify configuration file names. There can be several comma separated file names (default: "`/WEB-INF/tileDefinitions.xml`")
- `moduleAware`: (optional)
  - Specify if the Tiles definition factory is module aware. If true (default), there will be one factory for each Struts module. If false, there will be one common factory for all modules. In this later case, it is still needed to declare one plugin per module, and the factory will be initialized with parameters found in the first initialized plugin (generally the one associated with the default module).
    - `true` : One factory per module. (default)
    - `false` : One single shared factory for all modules
- `definitions-parser-validate`: (optional)
  - Specify if xml parser should validate the Tiles configuration file
    - `true` : validate. DTD should be specified in file header. (default)
    - `false` : no validation

Paths found in Tiles definitions are relative to the main context.
You do not need to specify in your struts config file(s) that the processorClass will be TilesRequestProcessor. This is automatically done by the plug-in. If a class is specified in the struts configuration file(s), that processorClass will be used instead.
To specify your own RequestProcessor, extend `org.apache.struts.tiles.TilesRequestProcessor` rather than just `RequestProcessor`. The Tiles plug-in checks check this constraint. Also, you must specify your class in your struts configuration file(s).
For example,

```
<controller processorClass="com.my.TilesRequestProcessorExtended"/>
```

2.2 Struts1.0.x

2.2.1 Required Files

Required files are:

- *tilesForStruts1.0.jar* – in *WEB-INF/lib/*
- *tiles.tld* – in *WEB-INF/*
- *struts.jar, commons-digester.jar*, *commons-collections.jar*, *commons-beanutils.jar* – in *WEB-INF/lib/*
- Struts related files

All these files should come with the Tiles or Struts distribution. They are normally located in *WEB-INF/lib/* for .JAR files and *WEB-INF/* for .TLD files.

2.2.2 Enabling Definitions

You need to use a special servlet extending the Struts servlet. This is specified in the web.xml file of your application:

```
  <servlet>
    <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.tiles.ActionComponentServlet</servlet-class>
        <!-- Tiles Servlet parameter
                  Specify configuration file names. There can be several comma
                  separated file names
                -->
        <init-param>
         <param-name>definitions-config</param-name>
         <param-value>/WEB-INF/tiles-defs.xml</param-value>
        </init-param>
         <!-- Tiles Servlet parameter
                Specify if xml parser should validate the Tiles configuration file.
                true : validate. DTD should be specified in file header.
                false : no validation
                -->
    <init-param>
      <param-name>definitions-parser-validate</param-name>
      <param-value>true</param-value>
    </init-param>
      ...
  </servlet>
```

If you have developed your own servlet extending the standard Struts servlet, consider extending the Tiles servlet instead, or provide methods and hooks enabling Tiles (check the Tiles servlet implementation for more information).

2.3 Tiles Standalone

2.3.1 Required Files

Required files are:

- *tiles.jar* – in *WEB-INF/lib/*
- *tiles.tld* – in *WEB-INF/*
- *commons-digester.jar*, *commons-collections.jar*, *commons-beanutils.jar* – in *WEB-INF/lib/*

All these files should come with the Tiles distribution. They are normally located in *WEB-INF/lib/* for .JAR files and *WEB-INF/* for .TLD files.

2.3.2 Enabling Definitions

Tiles can be used without Struts. To initialize the definition factory, you can use the provided servlet. Declare it in the web.xml file of your application:

```
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.tiles.TilesServlet</servlet-class>


        <init-param>
         <param-name>definitions-config</param-name>
         <param-value>/WEB-INF/tiles-defs.xml</param-value>
        </init-param>
    <init-param>
      <param-name>definitions-parser-validate</param-name>
      <param-value>true</param-value>
    </init-param>
   ...
```

The parameters are the same as for Struts1.1 or 1.0.

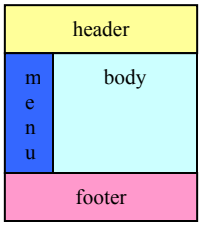
3 Basic mechanisms: Templates, Layouts and simple Tiles

A ***template*** and a ***layout*** are both the same: they describe where parts should be laid out in a bigger part, or in a final page. Both terms are use interchangeably in this document.

Layout = a description of where ***Tiles*** should be laid
Template = a complete HTML page with `<html>`, `<head>` and `<body>` tags and a layout

3.1 Building a Layout/Template

Layouts, or Templates, are described in a JSP page by using available tags. In your layout description, you place special tags where you want to render a Tile.
To build a layout, you need to know where you want to go. First, draw a layout example on a paper and identify parts or Tiles you want to be rendered. Name each Tile with a generic name, like header, menu, body, footer, or up, down, center, left and right.
As an example, we will build a classic layout, made of a header, a menu, a body and footer, as shown in the picture:



Creating such a structure in HTML is easy: you simply create a table with appropriate cells. Transforming it into a layout requires only adding appropriate tags in cells.

This example gives you the complete code for the layout shown above:

```
<%@ taglib uri="/WEB-INF/tiles.tld" 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>
```

You can see that in each cell we use the JSP tag `<tiles:insert attribute="aName" />`. This instructs the Tiles library to insert a Tile identified by the value of the specified attribute. Attribute values are passed to the layout when it is rendered.
This layout can be completed to make a template page by adding appropriate HTML tags, like `<html>`, `<head>` and `<body>`, to build a valid HTML page.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<HTML>
  <HEAD>
    <title><tiles:getAsString name="title"/></title>
  </HEAD>

<body bgcolor="#ffffff" text="#000000" link="#023264" alink="#023264" vlink="#023264">
<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>
```

The page title is dynamically rendered by `<tiles:getAsString name="title"/>`. This tag instructs Tiles to render an attribute value as a String. The JAVA - method `toString()` is applied to the attribute (usually a `String`) and rendered in the resulting page. As you can see, this template contains HTML tags used to structure an HTML page. That's why it is not necessary, and even discouraged, to use these tags in Tiles to be inserted into the template. This means that when you will define a header, menu, body or footer Tile, you will not use tags like `<html>`, `<head>` and `<body>`.

3.2 Usage

A layout or template can be used anywhere in a JSP page. Usually, templates are used by themselves, as they already define the HTML page structure.
Using a layout or template is the same as using a Tile: you insert it where you want it to be rendered. You need to pass attribute names and values required by the layout or template.

To use our previous template, we need to insert it in a page as follows:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:insert page="/layouts/myLayout.jsp" flush="true">
  <tiles:put name="title"  value="Page Title" />
  <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>
```

The tag `<tiles:insert page="/layouts/myLayout.jsp" flush="true">` instructs Tiles to insert a Tile defined in `page` whose URL (relative to the web application) is "`/layouts/myLayout.jsp`". The attribute "`flush`" is set to `true` to flush the generated page immediately after the insert. This is not mandatory in recent web containers, but some older web containers need it to be set to `true`.
The `<tiles:insert …>` and the nested `<tiles:put>` tags are used to specify the attributes of the inserted Tiles. The tag `<tiles:put name="title"  value="Page Title" />` defines an attribute with specified name and value. This attribute is then passed to the inserted Tile. You can define as many attributes as you want, as long as they have different names.
Now it's time to try our template: Call the JSP page containing the previous code in your browser, and you will see a page using the template.
You can reuse a template as many times as you want: just insert it with different attribute values. Generally, we just change the "`body`" and "`title`" value to render different pages with the same look and feel. With this method, you need to create at least two JSP pages for each page of your site: one page for the body, and one page inserting the template and using the body. We'll see later this can be avoided by combining Struts and Tiles.

3.3 Using Stylesheets

Adding a stylesheet can complete our previous template. This is not mandatory, but using a stylesheet helps having consistent formats, policies and colors in all Tiles.

To add a stylesheet to your template, add the following scriplet between `<head>…</head>`:

```
  …
  <HEAD>
    <link rel="stylesheet"
          href="<%=request.getContextPath()%>/layouts/stylesheet.css" type="text/css">
    …
  </HEAD>
  …
```

Another useful trick is to use the following code:

```
  …
  <HEAD>
<base href="http://<%=request.getServerName()%>:
<%= request.getServerPort()%>/<%=request.getContextPath()%>/">
  </HEAD>
  …
```

4 Definitions

Tiles' ***definitions*** allow specifying all attributes required by a Tile in one reusable structure. This structure contains the attributes as well as associated properties (page, name, …) for the Tiles. Definitions can be specified in any JSP page or in a centralized XML file.
Writing a definition requires similar syntax as inserting a Tile. Nearly all properties available in the `<tiles:insert>` tag are available in definitions as well (except for a few).

4.1 Definition in JSP page

4.1.1 Declaration

Declaring a definition looks something like this:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:definition id="definitionName" page="/layouts/myLayout.jsp" >
  <tiles:put name="title"  value="Page Title" />
  <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>
```

This is very similar to the example inserting our template. The difference is the tag name and one extra tag property "`id`".
The tag `<tiles:definition id="definitionName" page="/layouts/myLayout.jsp" >` declares a definition identified by `id="definitionName"`. This definition will be saved as a bean in the JSP page context scope under the specified name. It is possible to specify a different scope with `scope="application | request | page"`.

4.1.2 Usage

To use this definition, you simply do an insert, specifying the bean name and optionally the bean scope:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
  …
<tiles:insert beanName="definitionName" flush="true" />
```

You can also specify the attribute `flush="true"`. This will enforce a flush before the Tiles is inserted. Normally, the flush is automatically done by the web container, but some fail to do it.
Note that the previously defined definition can only be used in the same page, as default scope is "`page`". You can try this definition by putting it and its insertion into a single JSP page.

4.1.3 Overloading Attributes

When inserting a definition, you can overload some or all of its attributes. You can also add new attributes.
This is done naturally by specifying attributes in the `<tiles:insert>` tag. To overload an attribute, `<put>` a new attribute value with the same name. To create a new attribute, `<put>` a new attribute name and associated value.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
  …
<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>
```

Overloading attributes is not restricted to definitions attribute overload. You can always overload or declare new attributes when you use the `<tiles:insert>` tag, independent of the inserted source ("beanName", "attribute", "name", ...).

4.1.4 Extending Definitions

It is possible to describe a definition by extending another existing definition. All attributes and properties are inherited, and it is possible to overload any property or attribute.
To specify a new definition by extending another one, specify your definition as usual and add the property `"extends='definitionToExtend'"`.
[TODO: example of extending]

4.1.5 Reusing Definitions

In order to reuse definitions, you need to be able to declare them in a common place. A possible approach is to declare them in a JSP file, and include this file in each page using the definition.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:definition id="masterLayout" page="/layouts/classicLayout.jsp" scope="request" >
  <tiles:put name="title"  value="My First Definition Page" />
  <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>

  <%--
    Add as many definition as you need ...
  --%>
```

Then you can use it in your JSP pages as follows:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<%@ include file="definitionsConfig.jsp" %>

<tiles:insert beanName="masterLayout" beanScope="request" />
```

```
  <tiles:put name="title"  value="Another Page" />
  <tiles:put name="body"   value="/tiles/anotherBody.jsp" />
</tiles:insert>
```

The key word here is the JSP include tag: `<%@ include file="definitionsConfig.jsp" %>`. It includes the specified file that contains your definitions. Now it is possible to use a definition as usual. In the example, we overload the "`title`" and "`body`" parameters in order to customize our page: we inherit the general template, but modify it with a different title and body.
Any page needing a common place for definitions can use this inclusion mechanism. Definition insertion can overload some parameters to adapt the definitions to your needs.
Using this solution allows a centralized place for definitions, and also one master template definition, defining the default values used by templates. If you change the master definition, e.g. by specifying a new template to use, all pages using it will change accordingly.
This way of reusing and centralizing definitions works fine. But if you have a lot of definitions defined in the file, or if you heavily use definitions, you can encounter some performance problems because the file is included each time you need to use a definition, and so the definitions are (re-)defined each time. This can be solved with a mechanism checking if definitions are already loaded, like a `<logic:present> ... </logic:present>` section. But we will see that Tiles proposes another way to define definitions in a separate and centralized place.

4.2 Definitions in Separate Configuration Files

Definitions can be specified in a separate XML configuration file. The syntax is similar to the one used in JSP files. The XML configuration file is parsed once, normally by the Tiles factory during initialization phase.
Parsing a configuration file creates a "definition factory" containing all described definitions. Definitions are identified by their names, thus each definition should have a unique name. This name will be used by the JSP tags or Struts Action Forwards to retrieve the definition. A definition name is only meaningful on the web server. It is a logical name for the definition, and can't be used as an URL, or as a mapping to an URL.

4.2.1 Syntax Example

A simple configuration file looks like the following example:

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

<!-- Tiles definitions   -->

<tiles-definitions>

  <!-- ======================================================  -->
  <!-- Master layout                                             -->
  <!-- ======================================================  -->

  <!-- Master layout and default tiles used by all pages -->
  <definition name="portal.masterPage" page="/layouts/classicLayout.jsp">
          <put name="title"  value="Tiles 1.1 Dynamic Portal Example" />
          <put name="header" value="/tiles/header.jsp" />
          <put name="menu"   value="portal.menu.bar" />
          <put name="footer" value="/tiles/footer.jsp" />
          <put name="body"   value="/tiles/body.jsp" />
  </definition>

</tiles-definitions>
```

The first statement `<!DOCTYPE …>` specifies the root element used in the file, and the name of the DTD file used to validate this file. The DTD file is given as an external URL, but the parser will not try to load it from its real location. In fact, the parser is instructed to search the file corresponding to the URL in local places. For the Tiles DTD, the parser is instructed to search in *tiles.jar* where a copy of the appropriate DTD resides.
The next statement `<tiles-definitions>` is the root element of the XML file. This element encloses all Tiles definitions.
Next is the definition tag `<definition … >`. This definition is named "`masterPage`", and will use the file `"/layouts/classicLayout.jsp"` as Tiles, layout or template page.
The remaining `<put … />` tags define attributes added to the definition.

4.2.2 Extending Definitions

A definition can be specified by extending another definition. In this case, the new definition inherits all attributes and properties of the parent definition.
The property "`extends`" is used to specify which definition is extended.

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

In the example, we specify a new definition named "`portal.page`", extending the definition "`portal.masterPage`". The new definition inherits all attributes (title, header, menu, body and footer) and properties (page). The attributes "`title`" and "`body`" are overloaded in order to customize the new definition.
This inheritance capability allows root definitions defining default attributes, and extended definitions specializing requested attributes, like title and body. If all your definitions extend one root definition, changing a value in the root definition will change it for all extended definitions. For example, changing the layout in the root definition will change it everywhere.

4.2.3 Multiple Configuration Files

In large projects you often need to have several separate resource files. Tiles allows specifying several file names as configuration file. You specify the filenames in your *web.xml*, under the parameter "`definitions-config`".

```
<servlet>
   …
   <init-param>
      <param-name>definitions-config</param-name>
      <param-value /WEB-INF/tiles-defs.xml,/WEB-INF/tiles-tests-defs.xml,
         /WEB-INF/tiles-tutorial-defs.xml,/WEB-INF/tiles-examples-defs.xml</param-value>
    </init-param>
   …
<servlet>
```

File names are comma "," separated, and leading and ending spaces are trimmed. Each file is parsed in sequence and encountered definitions are added to the same definition factory. If a definition has the same name as a previous one, the latest one replaces the previously described definition. The inheritance mechanism works across different configuration files. Inheritance is resolved after all files have been parsed.

4.2.4 Definition's Name as Struts Forward

With Struts, the original way to prepare data for the view is to do it in an Action and then forward to the appropriate view. The view to forward to is expressed as a logical name in `<action ..>`, and this name is then matched to the action's associated forward. Each forward is a tuple (logical name, URL of the JSP).
If you use Tiles' servlet instead of Struts' one, you can use a Tiles definition name defined in the Tiles configuration file instead of the URL for the JSP.

Your Struts Action configuration will look like this:

```
    <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>
```

Your action's code is exactly the same as before, but your forward mapping now specifies a definition name instead of an URL. When the action forward encounters the definition name, it loads the definition, creates and initializes the Tile's context attributes, and then inserts the corresponding definition.

5 Complex Tiles

Until now we have just covered simple Tile descriptions. We know how to use them in the most common cases. It is now time to learn more about Tiles: what exactly is a Tile ? How to pass dynamic business data to a Tile ? What are the different tags available to pass attributes to a Tile ?

5.1 What's a Tile?

A Tile is an area in a web page. This area is rectangular, and can also be called "region" (like in another template mechanism by David Geary).
Assembling several Tiles can make themselves a new Tile. Tiles in a web page can be build recursively and represented as a tree where nodes represent regions. Root node is usually the page, final nodes or leafs contain page content, and intermediate nodes are usually layouts.
[TODO: Draw simple example]

5.1.1 Attributes

A Tile is parameterizable. This means it accepts some parameters, called "attributes" (to avoid confusion with request parameters). Tile attributes are defined when inserting the Tile, and are visible to the Tile only. They aren't visible in sub-Tiles, or outside of the Tile. This avoids name conflicts when using the same Tile several times in a page. It also allows Tile designers to use the same attribute name for different Tiles. This allows focusing on the design of Tiles, without wasting time with attribute names conflict.

5.2 Preparing data for the view: add a controller

We often need to prepare data to be shown by a JSP page. In the MVC framework, the controller prepares data (in the model) to be shown by the view. Translated to Tiles and Struts, we can use

a Struts Action as a controller, a JSP page as a view, and combine both in a Tile. So, when you insert the Tile, the Action is called before the JSP page is displayed.
Now, your complex web page can be made of Tiles fed by controllers (one sub-controller for each Tile). This approach is better than one single controller for all Tiles of the page, because it really allows building autonomous Tiles, without worrying about how to fed them all.

There are several ways to associate an Action and a View as a Tile:

- Specify the Action classname or action URL in `<insert>` or in `<definition>` (Tiles V1.1)
- Use a Struts Action in *struts-config.xml*, let it forward to a definition name. In *tiles-config.xml* specify one definition with the action URL as page, and one definition with the view as the page. The first definition is used as the complete Tile definition (action + view), the second definition is used as the view definition.
- It is possible to associate more than one view to a controller. Exactly one will be chosen. This is useful when you have a main view and an error view. Same as before, but with more than one forward and view definition.
- [TODO : Do we provide a way to define an Action with several forwards as a Tile Definition in *tiles-config.xml* ? Syntax ?]

5.2.1 One Controller - One View

It is possible to associate a "controller" to a Tile in the `<insert>` or `<definition>` tag.
The simpliest way to associate a controller to a Tile is to specify it in the `<insert>` or `<definition>` tag. You can associate a controller as a local URL or as a class.
This controller is called immediately before the JSP page is displayed. It shares the same Tiles' context as the JSP page. So it is possible to read, modify or add Tile attributes.

This Controller can be used for several purposes:

- Interact with data model before passing it to the view
- Change or add some attributes before passing them to the view

If you use an URL as controller, it should be an URL in the current web application. You can use a Struts Action URL. In this case, your action can extend `org.apache.struts.action.TilesAction` instead of the original Struts Action class. The `TilesAction` class implements the original `perform()` method, and provides a new `perform()` method with an extra "`tileContext`" parameter. This is useful when you need to interact with Tiles' attributes.

If you use a class name as controller, it should extend one of the following base classes or interfaces:

`org.apache.struts.tiles.Controller`
: This is the Controller interface defining the controller method. This method receives as arguments the current Tile's context, and the usual servlet parameters `request`, `response` and `servletContext`.

`org.apache.struts.tiles.ControllerSupport`
: This is a basic implementation with an empty method.

- `org.apache.struts.action.Action` (wrapper `org.apache.struts.action.StrutsActionControllerWrapper` is used)
  If you provide a Struts Action subclass, it will be wrapped with the appropriate class, and Struts' perform method will be called, but the "`mapping`" and "`form`" attributes will be `null`.

Example of a controller inserted as a class:

```
<tiles:insert page="layout.jsp"
           controllerClass="org.apache.struts.example.tiles.test.TestTileController" >
  <tiles:put name="title"  value="Test controller set in insert" />
  <tiles:put name="header" value="header.jsp" />
  <tiles:put name="body"   value="body.jsp" />
</tiles:insert>
```

Class skeleton:

[TODO: missing]

Example of a controller inserted as an URL:

```
<tiles:insert page="layout.jsp"
           controllerUrl="myAssociatedAction.do" >
  <tiles:put name="title"  value="Test controller set in insert" />
  <tiles:put name="header" value="header.jsp" />
  <tiles:put name="body"   value="body.jsp" />
</tiles:insert>
```

Struts Action declaration:

```
    <action     path="/myAssociatedAction"
                type="org.apache.struts.example.tiles.MyAssociatedAction">
   </action>
```

Action skeleton:

```
public final class MyAssociatedAction extends TilesAction {

    /**
     * Process http request (controller)
     * @param context The current Tile context, containing Tile attributes
     * @param mapping The ActionMapping used to select this instance
     * @param form The optional ActionForm bean for this request (if any)
     * @param request The HTTP request we are processing
     * @param response The HTTP response we are creating
     *
     * @exception IOException if an input/output error occurs
     * @exception ServletException if a servlet exception occurs
     */
    public ActionForward perform( ComponentContext context,
                                  ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                            throws IOException, ServletException
    {
        String title = (String)context.getAttribute( "title" );
        System.out.println( "Original title" + title );
        context.putAttribute( "title", "New Title" );
        return null;
    }
}
```

5.2.2 One Controller - Multiple Views

Sometimes, the controller does some kind of logic and depending of some result selects the appropriate view to render data. For example, the controller processes user requests, if all is ok, data is shown, if an error occurs, the appropriate error page is shown.
The current Tiles version doesn't directly support one controller associated to multiple views, but this can be done with Struts' Action Forward mechanism.
The solution consists of writing a Tile with its attributes (insert or definition), and specify a Struts Action as page URL instead of a JSP page. The action plays the role of the controller and then forwards to the appropriate JSP page, which can either be an URL or a definition name. The Tiles context for the controller (action) and the selected page will still be the same, allowing you to modify or set Tiles attributes.

The Tile definition looks like this:

```
<definition name="tileWithActionAsController" path="/actionAsController.do" >
                            //'path' the same as 'page'
  <put name="title" value="Title" />
  <put name="anAttribute" value="aValue" />
</definition>
```

The Action used as controller and dispatcher:

```
    <action     path="/ actionAsController "
                type="org.apache.struts.example.tiles.ActionAsController">
      <forward  name="failure"        path="/failurePage.jsp"/>
      <forward  name="success"        path="success.definition"/>
  </action>
```

In this example, "`failure`" is associated to a JSP page, while "`success`" is associated to a definition name. If "`success`" is selected, the definition is loaded and attributes defined in definition and not actually defined in Tiles' context are added to the current context. This allows default attribute values for the view.

5.3 Attribute types and syntax

Tiles attributes can be defined in different ways: `<put>`, `<putList`, `<bean>` and `<item>`.

5.3.1 <put>

The `<put>` tag is used to associate a value to an attribute. It allows specifying the attribute name and its value. The attribute value will be retrieved by its name.

The value can be specified in several ways:

- as tag property:

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

  Value of the "`value`" attribute can be a String, like in the example, or an object retrieved with a scriptlet, like in `value="<%=anObject%>"`. This is useful when you mix scriptlets in a JSP page.

- as tag body:

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

- as a bean defined in some scope:

```
<tiles:put name="title" beanName="aBean"
           beanScope="page|request|application|tiles"/>
```

The bean identified by "beanName" is retrieved, and its value used as attribute value. "beanScope" is optional, if not specified, bean is searched for in scopes in the following order: page, request and application. Scope "tiles" means the current Tile's context (if any). In this case, "beanName" identifies a Tile attribute name.

- as property of a bean defined in some scope:

```
<tiles:put name="title" beanName="aBean" beanProperty="aProperty"
           beanScope="page|request|application|tiles"/>
```

This is a variant of the previous case. The value is not the retrieved bean, but the returned value of the specified bean property.

It is also possible to specify the value type:

- This type will be used by the <insert> or <get> tag as an hint of what to do with the value

```
<tiles:put name="title"  value="aValue" type="string|page|definition"/>
```

- type="string" - The value - corresponding String is written. The method toString() is called on the value to obtain a String.
- type="page" – Value denotes an URL that is inserted.
- type="definition" – Values denotes a definition name that is inserted.

It is possible to associate a role:

```
<tiles:put name="title"  value="aValue" role="aRole"/>
```

- Role is checked and attribute value is used only if current user is in the specified role.
  Role is checked immediately if <put> is used in an <insert> tag.
  Role is checked when Tile context is initialized if <put> is used in a <definition> tag.

5.3.2 <putList> and <add>

The tag <putList> is used to create an attribute of type java.util.List. Elements are added to the list with the <add> tag. The list name should be specified in the tag:

```
<tiles:insert page="menu.jsp" >
  <tiles:putList name="items">
    <tiles:add value="home" />
    <tiles:add>
          <img src="<%=request.getContextPath()%>/images/struts-power.gif"
               align="right" border="0"></tiles:add>
    <tiles:add value="documentation"/>
  </tiles:putList>
</tiles:insert>
```

This example creates a List attribute named "items".
The tag <add> has the same syntax as <put>, except for the "name" property . Elements are added to the list in the specified order.

The list can be used in the Tile controller, or in the view as in the next example:

```
<tiles:importAttribute />

<table>
```

```
<logic:iterate id="item" name="items" >
<tr>
  <td>
    <%item%>
  </td>
</tr>
</logic:iterate>
</table>
```

Above example starts with `<importAttribute>`. This imports all attributes from the Tile's context to page context. So, the "`items`" attribute becomes accessible in the page as a bean. Then, the `<iterate>` tag accesses the list and walks through all elements.

5.4 Example: RssChannel

The action/controller loads rss data and prepares it for the view. Parameters are passed as Tile attributes : rssData, what else ?

In this example, we will create a Tile reading the content of Rss channels from specified URLs and showing the content using an appropriate layout.
The Tile is made of a controller and a view. The controller reads channel content from an XML file, parses the content, and creates a bean used as model and easily useable by the view. The view shows the channel content stored in the bean. The URLs of the channel feeds are specified as Tiles attributes.

This approach has several advantages:

- It is possible to insert the Tile anywhere in a page.
- It is possible to insert several channel - Tiles in the same page, with different content if desired.
- It is possible to change the way the channel is rendered by simply changing the view.
- It is possible to specify several channel - Tiles, each one rendering a different set of channels.

5.4.1 Definition

The Tile definition is a master specifying the controller, the layout used and the URLs of the channels to render.

```
  <definition name="examples.rssChannel.body" path="/examples/tiles/rssChannels.jsp"
                    controllerUrl="/examples/controller/rssChannel.do" >
      <putList name="urls" >
            <add value="http://www.newsforge.com/newsforge.rss" />
            <add value="http://xmlhack.com/rss.php" />
            <add value="http://lwn.net/headlines/rss" />
      </putList>
  </definition>
```

It is possible and easy to declare several definitions using different channel URLs and/or different layouts.
The controller is specified as an URL of a Struts Action. This implies that the action is defined in the Struts configuration file. The view is specified as the definition page. URLs are specified in a list ("`urls`") passed as attribute of the Tile.

5.4.2 Struts Action Declaration

The controller is declared as a Struts Action. Its specification is as follows:

```
    <action     path="/examples/controller/rssChannel"
                type="org.apache.struts.example.tiles.rssChannel.RssChannelsAction">
   </action>
```

The action doesn't forward to a view, as the view is associated with a Tiles definition.

5.4.3 Controller

The controller is just a JAVA class. It retrieves the list of the channel URLs from the Tile's attribute and reads and parses the channel content using `RSSDigester`. Then, the bean containing contents is used as the model and passed to the view by a Tiles attribute.

Following is a simplified version of the controller code:

```
public final class RssChannelsAction extends TilesAction {


      /** Tile attribute key for saving Channel bean */
    public static final String CHANNELS_KEY = "CHANNELS";
      /** Tile attribute key for getting Channel urls list */
    public static final String CHANNEL_URLS_KEY = "urls";

    /**
     * Main process of class. Reads, parses
     */
    public ActionForward perform( ComponentContext context,
                                  ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                            throws IOException, ServletException
    {
        org.apache.commons.digester.rss.Channel channel = null;

        // -- Retrieve parameters --
        List channels = (List)context.getAttribute( CHANNEL_URLS_KEY );
            List channelBeans = new ArrayList(channels.size());
        for (int i=0; i<channels.size(); i++) {
               RSSDigester digester = new RSSDigester();
               String url = (String)channels.get(i);
               // Add application path if needed
               Channel obj = (Channel)digester.parse(url);
               channelBeans.add(obj);
        } // end loop

       // Use Tile context to pass channels
        context.putAttribute( CHANNELS_KEY,channelBeans);
        return null; //no mapping
    } // ---- End perform ----

} // ---- End Fetch ----
```

The action returns `null` because there is no forward associated with this action.

5.4.4 View

The view renders the contents. It takes the data to be displayed from the Tile's context. Then it iterates on each channel, and renders it.

```
<%--
/**
 * Summarize channels as unadorned HTML.
 *
```

```
 * @param CHANNELS List of channels
 */
--%>

<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<div align="center"><font size="+1"><b>

<tiles:importAttribute name="CHANNELS" scope="page"/>

<logic:iterate name="CHANNELS" id="CHANNEL" >
<TABLE border="0" cellspacing="0" cellpadding="4" width="100%" align="center" >
<TR>
<TD class="spanhd" ><logic:present name="CHANNEL" property="image">
  <a href="<bean:write name="CHANNEL" property="link"/>">
    <img src="<bean:write name="CHANNEL" property="image.URL"/>"></logic:present></a>
</TD>
<TD    class="spanhd"    width="100%"><bean:write    name="CHANNEL"    property="title"/>    <a
href="<bean:write name="CHANNEL" property="link"/>">[home]</a></TD>
</TR>
<TD class="yellow" colspan="2"><bean:write name="CHANNEL" property="description"/></TD>
</TR>

<TR>
<TD class="datagrey" colspan="2">
<logic:iterate name="CHANNEL" property="items" id="ITEM">
<br><b><bean:write name="ITEM" property="title"/></b>
<br><bean:write name="ITEM" property="description"/>
<br>  [ <a href="<bean:write name="ITEM" property="link"/>">more</a> ]
<br>
</logic:iterate>
</TD>
</TR>
</TABLE>
<br>
</logic:iterate>

</b></font></div>
```

This page can easily be adapted to change the way channels are rendered.

6 Available layouts

Here we describe some useful layouts available in the Tiles distribution.
One interesting point of Tiles is that it is possible to reuse existing layouts, define new layouts, and easily exchange a layout by a different one as long as they require the same set of attributes.
For each layout, a short description, a description of requested attributes, an example of usage and the actual implementation is provided.
You can easily customize any layout by starting from the provided implementation. All layouts examples can be found in the */layouts* directory of the Tiles example jar file. They can be tested by pointing to the */examples* directory of the Tiles example jar file.

6.1 Classic Layout

The classic layout is the most often used layout when building a web site. It provides a classic "header, menu, body and footer" rendering. It defines the skeleton of a HTML page with `<header>` and `<body>`, and renders the header, menu, body and footer at appropriate places.

6.1.1 Requested Attributes

The classic layout requires the following attributes:

- `title` - String used to set the title of the browser page.
- `header` - URL or definition name used to render the header part.

- `menu` - URL or definition name used to render the menu part.
- `body` - URL or definition name used to render the body part (content).
- `footer` - URL or definition name used to render the footer part.

6.1.2 Usage

Typical usage of this layout is as follows:

```
<tiles:insert page="/layouts/classicLayout.jsp" flush="true">
  <tiles:put name="title"  value="Browser Title" />
  <tiles:put name="header" value="/header.jsp" />
  <tiles:put name="footer" value="/footer.jsp" />
  <tiles:put name="menu"   value="/menu.jsp" />
  <tiles:put name="body"   value="definition.name" />
</tiles:insert>
```

Attributes and layout can also be specified in a definition:

```
  <definition name="mainLayout" path="/layouts/classicLayout.jsp">
         <put name="title"  value="Browser Title" />
         <put name="header" value="/header.jsp" />
         <put name="footer" value="/footer.jsp" />
         <put name="menu"   value="menu.main" />
         <put name="body"   value="main.portal.body" />
  </definition>
```

6.1.3 Implementation

Following is the implementation of this layout:

```
<%-- Layout Tiles
  This layout creates a HTML page with <header> and <body> tags. It renders
   a header, left menu, body and footer tile.
  @param title String used as page title
  @param header Header Tile (URL of a JSP page or definition name)
  @param menu Menu Tile
  @param body Body Tile
  @param footer Footer Tile
--%>
<HTML>
  <HEAD>
       <link rel=stylesheet href="<%=request.getContextPath()%>/layouts/stylesheet.css"
             type="text/css">
    <title><tiles:getAsString name="title"/></title>

  </HEAD>

<body bgcolor="#ffffff" text="#000000" link="#023264" alink="#023264" vlink="#023264">
<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>
```

The page title is retrieved via `<tiles:getAsString name="title"/>`. Layout parts are inserted with `<tiles:insert attribute="partName" />`. Property "`attribute`" retrieves the Tile's attribute and uses the retrieved value as a name to be inserted. This name can denote either an URL or a definition name.

6.2 Menu Layout

The menu layout is used to render a simple menu with its links. A menu requires a list of "`item`" beans, each "`item`" contains data for one menu entry.
It's possible to develop other menu layouts taking the same attributes, and replace the menu layout by another one, e.g. a vertical menu bar could be built by using the vbox layout (described later).

6.2.1 Requested Attributes

The menu layout requires the following attributes:

- `title` - String used as menu title [optional]
- `items` - List of items. Items are beans with the following properties: `value`, `link`, `tooltip`; `icon`

6.2.2 Usage

Typical usage is as follows:

```
  <!-- Preferences menu definition -->
<definition name="examples.menu.settings" path="/layouts/menu.jsp" >
  <put name="title" value="Preferences" />
    <putList name="items" >
      <item value="my Portal Settings"
            link="/examples/myPortalSettings.jsp"
             classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
      <item value="my Menu Settings"
             link="/examples/myMenuSettings.jsp"
             classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    </putList>
</definition>
```

The definition declares two attributes: a `title` and a `list`. "`list`" is a list of item beans, one for each entry in the menu. Each item bean can contain several properties. Here we use the property "`value`" for the entry title, and "`link`" for the link.

6.2.3 Implementation

Following is a possible implementation for this layout:

```
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
<%@ page import="java.util.Iterator" %>


<%-- Menu Layout
  This layout renders a menu with links.
  It takes as parameter the title and a list of items. Each item is a bean
  with following properties: value, href, icon, tooltip.
  @param title Menu title
  @param items list of items. Items are beans
 --%>

<%-- Push tiles attributes in page context --%>
<tiles:importAttribute />
```

```
<table>
<logic:present name="title">
<tr>
  <th colspan=2>
    <div align="left"><strong><tiles:getAsString name="title"/></strong></div>
  </th>
</tr>
</logic:present>

<%-- iterate on items list --%>
<logic:iterate id="item" name="items" type="org.apache.struts.tiles.beans.MenuItem" >

<%  // Add site URL if link starts with "/"
  String link = item.getLink();
       if(link.startsWith("/") ) link = request.getContextPath() + link;
%>
<tr>
  <td width="10" valign="top" ></td>
  <td valign="top"  >
         <font size="-1"><a href="<%=link%>">
<logic:notPresent name="item" property="icon"><%=item.getValue()%></logic:notPresent>
<logic:present name="item" property="icon">
       <%  // Add site URL if link starts with "/"
         String icon = item.getIcon();
              if(icon.startsWith("/") ) icon = request.getContextPath() + icon;
       %>
<img src='<%=request.getContextPath()%><bean:write name="item" property="icon" scope="page"/>'
      alt='<bean:write name="item" property="tooltip" scope="page" ignore="true"/>' />
</logic:present></a>
         </font>
  </td>
</tr>
</logic:iterate>
</table>
```

This implementation renders a menu in a vertical table with two columns: one used as a leading space, the second used to render the entries. The menu title is rendered only if it exists. Each entry is rendered as a hyperlink. If an icon is specified, it is used in place of the value.

6.3 VBox or VStack Layout

The vbox layout is used to render a list of Tiles vertically.
This layout is used when rendering a menu bar, or in multi - columns layout.

6.3.1 Requested Attributes

The vbox layout requires the following attributes:

- `list` - List of names or URLs to insert.

6.3.2 Usage

Typical usage of vbox layout is as follows:

```
<definition name="examples.menu.bar" path="/layouts/vboxLayout.jsp" >
  <putList name="list" >
    <add value="examples.userMenu" />
    <add value="examples.menu.links" />
    <add value="doc.menu.links" />
    <add value="examples.menu.settings" />
    <add value="doc.menu.taglib.references" />
    <add value="doc.menu.printer.friendly" />
    <add value="examples.menu.admin" />
  </putList>
```

This definition declares a menu bar made of specified menus, which will be rendered vertically.

6.3.3 Implementation

The implementation loops over the list of Tiles and inserts each one using `<tiles:insert …>`. The loop is done with a scriptlet rather than with a `<logic:iterate …>` tag because JSP1.1 forbids the insertion of a page inside the body of a tag implementing "`BodyTag`", like the `<logic:iterate>` tag does.

Following is the implementation of this layout:

```
<%@ page import="java.util.Iterator"%>
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<%-- Layout component
  Render a list of Tiles in a vertical column
  @param : list List of names to insert

--%>

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

<%-- Iterate over names.
  We don't use <iterate> tag because it doesn't allow insert (in JSP1.1)
 --%>
<%
Iterator i=list.iterator();
while( i.hasNext() )
  {
  String name= (String)i.next();
%>
        <tiles:insert name="<%=name%>" flush="true" />
        <br>

<%
  } // end loop
%>
```

The tag `<tiles:useAttribute id="list" name="list" classname="java.util.List" />` is used to declare a servlet variable initialized with a Tile's attribute. Here we declare the variable 'list' (`id="list"`) of type String and initialized with Tile's attribute 'list' (`name="list"`).

6.4 Multi-columns Layout

The multi-columns layout renders lists of Tiles in several columns stacked vertically. Lists of Tiles are passed as Tiles parameters, one list per column. A list can contain application URLs and definition names.
This layout is used to build a portal main body made of several Tiles.

6.4.1 Required Attributes

The multi-columns layout requires the following attributes:

- `numCols` - Number of columns to render and passed as parameter.
- `list1` - First list of Tiles (URL or definition name)
- `list2` - Second list of Tiles (URL or definition name) [optional]
- `list3` - Third list of Tiles (URL or definition name) [optional]
- `list`*n* - *N*-th list of Tiles (URL or definition name), where *n* is replaced by column index [optional].

6.4.2 Usage

Typical usage is as follows:

```
  <definition name="examples.portal.body" path="/layouts/columnsLayout.jsp"
              controllerUrl="/portal/myPortal.do" >
         <put name="numCols" value="2" />
         <putList name="list0" >
           <add value="/examples/tiles/portal/login.jsp" />
           <add value="/examples/tiles/portal/messages.jsp" />
           <add value="/examples/tiles/portal/newsFeed.jsp" />
         </putList>
         <putList name="list1" >
           <add value="/examples/tiles/portal/advert3.jsp" />
           <add value="/examples/tiles/portal/stocks.jsp" />
           <add value="/examples/tiles/portal/whatsNew.jsp" />
           <add value="/examples/tiles/portal/advert2.jsp" />
         </putList>
  </definition>
```

6.4.3 Implementation

Here is an implementation of columns layout:

```
<%-- Multi-columns Layout
  This layout renders lists of Tiles in multi-columns. Each column renders its Tiles
  vertically stacked.
  The number of columns passed in parameter must correspond to the number of list passed.
  Each list contains Tiles. Lists are named list0, list1, list2, ...
  parameters : numCols, list0, list1, list2, list3, ...
  @param numCols Number of columns to render and passed as parameter
  @param list1 First list of Tiles (URL or definition name)
  @param list2 Second list of Tiles (URL or definition name) [optional]
  @param list3 Third list of Tiles (URL or definition name) [optional]
  @param listn Niene list of Tiles (URL or definition name), where n is replaced by column index.
--%>


<tiles:useAttribute id="numColsStr" name="numCols" classname="java.lang.String" />

<table>
<tr>
<%
int numCols = Integer.parseInt(numColsStr);
ComponentContext context = ComponentContext.getContext( request );
for( int i=0; i<numCols; i++ )
  {
  java.util.List list=(java.util.List)context.getAttribute( "list" + i );
  pageContext.setAttribute("list", list );
  if(list==null)
    System.out.println( "list is null for " + i  );
%>
<td valign="top">
  <tiles:insert page="/layouts/vboxLayout.jsp" flush="true" >
    <tiles:put name="list" beanName="list" beanScope="page" />
  </tiles:insert>
</td>
<%
  } // end loop
%>
</tr>
</table>
```

This implementation is a little bit technical: First, it declares a servlet variable by `<tiles:useAttribute id="numColsStr" name="numCols" classname="java.lang.String" />`. The variable name is identified by `id="numColsStr"`, while the attribute `name` used to initialized variable is identified with `name="numCols"`. The variable is declared as a String.

Next, there is a piece of code iterating on each list. The list name is built "on the fly", reading from Tile attributes in page context, and then used in a vbox layout. This renders Tiles in the list in a vertical stack.

This implementation loops on attributes using scriptlet code instead of an `<logic:iterate…>` tag because JSP1.1 does not allow the insertion of another page inside the body of a tag.

6.5 Center Layout

6.5.1 Description

The classical layout consisting of "top, left, center, right, bottom". Left and right parts are optional and can be omitted.

6.5.2 Requested Attributes

The center layout requires the following attributes:

- `header` - URL or definition name used to render the header part.
- `right` - URL or definition name used to render the right part. [optional]
- `body` - URL or definition name used to render the header part.
- `left` - URL or definition name used to render the left part. [optional]
- `footer` - URL or definition name used to render the footer part.

6.5.3 Usage

Usage is similar as the classic layout:

```
<tiles:insert page="/layouts/centerLayout.jsp" flush="true">
  <tiles:put name="header" value="/header.jsp" />
  <tiles:put name="footer" value="/footer.jsp" />
  <tiles:put name="right"   value="/menu.jsp" />
  <tiles:put name="left"   value="/menuLeft.jsp" />
  <tiles:put name="body"   value="definition.name" />
</tiles:insert>
```

Either one of `left` or `right` can be ommited as in the following example:

```
<tiles:insert page="/layouts/centerLayout.jsp" flush="true">
  <tiles:put name="header" value="/header.jsp" />
  <tiles:put name="footer" value="/footer.jsp" />
  <tiles:put name="body"   value="definition.name" />
</tiles:insert>
```

6.5.4 Implementation

Implementation is straightforward :

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<%-- Centered Layout Tiles
  This layout renders a header, left tile, right tile, body and footer.
  It doesn't create <html> and <body> tags.
  @param header Header tile (URL of a JSP or definition name)
  @param right Right center tile (optional)
  @param body Body or center tile
  @param left Left center tile (optional)
  @param footer Footer tile
--%>

<table border="0" width="100%" cellspacing="5">
<tr>
  <td colspan="3"><tiles:insert attribute="header" /></td>
</tr>
<tr>
  <td width="140" valign="top">
    <tiles:insert attribute=right ignore='true'/>
  </td>
  <td valign="top"  align="left">
    <tiles:insert attribute='body' />
  </td>
  <td valign="top"  align="left">
```

```
    <tiles:insert attribute='left' ignore='true'/>
  </td>
</tr>
<tr>
  <td colspan="3">
    <tiles:insert attribute="footer" />
  </td>
</tr>
</table>
```

6.6 Tabs Layout

The tabs layout is used to render a list of Tiles in a tab fashion. The tabs layout has a body area used to render the currently selected Tile, and an indexes area used to render the available tabs or Tiles. This layout requires as argument a list of Tiles and names used in indexes.

6.6.1 Required Attributes

The tabs layout requires the following attributes:

- `tabList` - A list of Tiles URLs or definition names for the tabs. We use `MenuItem` to carry data (Tiles name or URL, indexes name, icon, ...).
- `selectedIndex` - Index of the default selected tab.
- `parameterName` - Name of HTTP parameter carrying the selected Tile info in the HTTP request.

6.6.2 Usage

Typical usage is as follows:

```
  <definition name="examples.tabs.body" path="/layouts/tabsLayout.jsp" >
       <put name="selectedIndex"  value="0" />
       <put name="parameterName"  value="selected" />
    <putList name="tabList" >
      <item value="Doc Home"
                    link="/index.jsp"
                    classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
      <item value="Quick overview"
                    link="/doc/quickOverview.jsp"
                    classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
      <item value="Tutorial"
                    link="/doc/tutorial.jsp"
                    classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    </putList>
  </definition>
```

This definition creates a tab layout with three indexes.

6.6.3 Implementation

The following implementation contains the controller code and a view rendered in the same page (this could be separated into a separate controller and view).
The controller retrieves the value of selected indexes and stores it in the appropriate attribute.
The view renders indexes and highlights the currently selected index. Then, it renders the body area and the currently selected Tiles or URLs.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<%--
  Tabs Layout .
  This layout allows to render several Tiles in a tabs fashion.
  @param tabList A list of available tabs. We use MenuItem to carry data (name, body, icon, ...)
  @param selectedIndex Index of default selected tab
  @param parameterName Name of parameter carrying selected info in HTTP request.
```

```
--%>

<%--
Use Tiles attributes, and declare them as page variable.
These attribute must be passed to the Tile.
--%>

<tiles:useAttribute name="parameterName" classname="java.lang.String" />
<tiles:useAttribute id="selectedIndexStr" name="selectedIndex" ignore="true"
classname="java.lang.String" />
<tiles:useAttribute name="tabList" classname="java.util.List" />
<%
  String selectedColor="#98ABC7";
  String notSelectedColor="#C0C0C0";

  int index = 0; // Loop index
  int selectedIndex = 0;
    // Check if selected come from request parameter
  try {
    selectedIndex = Integer.parseInt(selectedIndexStr);
    selectedIndex = Integer.parseInt(request.getParameter( parameterName ));
        }
   catch( java.lang.NumberFormatException ex )
    { // do nothing
        }
  // Check selectedIndex bounds
  if( selectedIndex < 0 || selectedIndex >= tabList.size() ) selectedIndex = 0;
  String                                  selectedBody                                    =
((org.apache.struts.tiles.beans.MenuItem)tabList.get(selectedIndex)).getLink(); // Selected body

%>

<table border="0"  cellspacing="0" cellpadding="0">
  <%-- Draw tabs --%>
<tr>
  <td width="10""> </td>
  <td>
    <table border="0"  cellspacing="0" cellpadding="5">
      <tr>
<logic:iterate id="tab" name="tabList" type="org.apache.struts.tiles.beans.MenuItem" >
<% // compute href
  String href = request.getRequestURI() + "?"+parameterName + "=" + index;
  String color = notSelectedColor;
  if( index == selectedIndex )
    {
        selectedBody = tab.getLink();
        color = selectedColor;
        } // enf if
  index++;
%>
  <td bgcolor="<%=color%>">
  <a href="<%=href%>" /><%=tab.getValue()%></a>
  </td>
  <td width="1" ></td>

</logic:iterate>
      </tr>
    </table>
  </td>
  <td width="10" > </td>
</tr>


<tr>
  <td height="5" bgcolor="<%=selectedColor%>" colspan="3" > </td>
</tr>

  <%-- Draw body --%>
<tr>
  <td width="10" bgcolor="<%=selectedColor%>"> </td>
  <td>
  <tiles:insert name="<%=selectedBody%>" flush="true" />
  </td>
  <td width="10" bgcolor="<%=selectedColor%>"> </td>
</tr>

<tr>
  <td height="5" bgcolor="<%=selectedColor%>" colspan="3" > </td>
</tr>

</table>
```

6.7 Developing your own layouts

Starting from the provided examples, you can start developing your own layouts.
Keep in mind that if you want to reuse or replace one Tile with another one easily, you should clearly identify what the required attributes are. Always separate the controller and view parts (MVC model).

7 Dynamic Portal Example

7.1 Description

Features:

- Portal Page:
  It shows different Tiles: static ones, rss channels, …
- Choice of Tiles displayed in portal:
  A preferences page allows users to add/remove/arrange Tiles in the portal.
- User menu:
  The user can configure the menu by adding/removing menu items.
- General L&F by using a master template
- User customizable L&F

7.2 User Customizable Portal

7.2.1 Static Portal

Building a static portal with Tiles is very easy: all you need to do is to develop each Tile constituting the portal, and then assemble them, e.g. using a multi - columns layout.
When developing individual Tiles, you only concentrate on the Tiles you're working on. You don't need to bother about other Tiles in the portal, or about the portal itself. Each individual Tile is generally a piece of HTML code with no `<header>` or `<body>` tags. Individual Tiles can be complex. Any Tile, JSP or HTML page can be used as individual Tile of a portal.

Following is an example of a simple individual Tile:

```
<table  width="100%">
  <TR>
    <TD>This is a very simple Tile that can be displayed in a portal</TD>
  </TR>
  <TR>
    <TD></TD>
  </TR>
</table>
```

Assembling Tiles in a portal fashion requires you to declare a definition with lists of Tiles for your portal:

```
  <definition name="doc.portal.body" path="/layouts/columnsLayout.jsp">
    <put name="numCols" value="2" />
    <putList name="list0" >
      <add value="/doc/portal/welcome.jsp" />
```

```
      <add value="/doc/portal/features.jsp" />
      <add value="/doc/portal/documentation.jsp" />
      <add value="doc.menu.bar" />
    </putList>
    <putList name="list1" >
      <add value="/doc/portal/news.jsp" />
      <add value="/doc/portal/download.jsp" />
      <add value="/doc/portal/tilesCompsTemplates.jsp" />
      <add value="/doc/portal/strutsIntegration.jsp" />
      <add value="/doc/portal/comments.jsp" />
      <add value="/doc/portal/revisions.jsp" />
    </putList>
  </definition>
```

This declares a portal body with two columns. Each column displays the Tiles stated in its list. Lists can, as usual, contain local URLs as well as definition names identifying a Tile.

The previous definition is declared in a configuration file, but it is also possible to do the same in a JSP page using `<tiles:definition …>`, or directly via an `<tiles:insert>` tag:

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>

<tiles:insert page="/layouts/columnsLayout.jsp" flush="true">
  <tiles:put name="numCols" value="2" />
  <tiles:putList name="list0" >
    <tiles:add value="/tutorial/portal/login.jsp" />
    <tiles:add value="/tutorial/portal/messages.jsp" />
    <tiles:add value="/tutorial/portal/newsFeed.jsp" />
    <tiles:add value="/tutorial/portal/advert2.jsp" />
  </tiles:putList>
  <tiles:putList name="list1" >
    <tiles:add value="/tutorial/portal/advert3.jsp" />
    <tiles:add value="/tutorial/portal/stocks.jsp" />
    <tiles:add value="/tutorial/portal/whatsNew.jsp" />
    <tiles:add value="/tutorial/portal/personalLinks.jsp" />
    <tiles:add value="/tutorial/portal/search.jsp" />
  </tiles:putList>
</tiles:insert>
```

Our previous portal definition can be used anywhere a definition name can be used. It is possible to use it in an `<tiles:insert name="doc.portal.body" />` tag, or in a template definition providing a common L&F:

```
  <definition name="doc.mainLayout" path="/layouts/classicLayout.jsp">
        <put name="title"  value="Tiles Library Documentation" />
        <put name="header" value="/common/header.jsp" />
        <put name="menu"   value="doc.menu.main" />
        <put name="footer" value="/common/footer.jsp" />
        <put name="body"   value="doc.portal.body" />
  </definition>
```

To change your portal page, edit portal definition lists and restart your web server; or use the Tiles reload administration page.

7.2.2 Dynamic Portal

We have seen how to build a static portal. It's possible to modify the example above in order to load dynamic lists of Tiles constituting the portal. We will describe this feature in a user customizable portal.
In this example we'll start with a default portal page and call a "Settings" page showing all available individual Tiles and the actual portal configuration. Buttons in the page allow the users to move Tiles around in the columns.

Our example is made of two Tiles, each one associating a view and a controller. These Tiles are declared as definitions in the configuration file. The example also requires an ActionForm class used to read the new user settings, a "user setting class" and a "catalog class" used to display all available Tiles.

7.2.2.1 Dynamic Portal View

The dynamic portal view is a static portal view associated to a controller which sets dynamic portal lists.
The portal body definition declares lists of Tiles used as default values, and labels shown in the settings page. Labels are optional; values of Tiles lists are used if not overridden.

```
<definition name="examples.portal.body" path="/layouts/columnsLayout.jsp"
            controllerUrl="/portal/myPortal.do" >
        <put name="numCols" value="2" />
        <putList name="list0" >
          <add value="/examples/tiles/portal/login.jsp" />
          <add value="/examples/tiles/portal/messages.jsp" />
          <add value="/examples/tiles/portal/newsFeed.jsp" />
          <add value="examples.menu.bar" />

        </putList>
        <putList name="list1" >
          <add value="/examples/tiles/portal/advert3.jsp" />
          <add value="/examples/tiles/portal/stocks.jsp" />
          <add value="/examples/tiles/portal/whatsNew.jsp" />
          <add value="/examples/tiles/portal/advert2.jsp" />
        </putList>
          <!-- labels used by catalog [optional]-->
        <putList name="labels0" >
          <add value="Login" />
          <add value="Your Messages" />
          <add value="News Feed" />
          <add value="Menus Bar" />
        </putList>
        <putList name="labels1" >
          <add value="Advert 3" />
          <add value="Stock" />
          <add value="What's new" />
          <add value="Advert 2" />
        </putList>
</definition>
```

7.2.2.2 Portal Controller

The controller retrieves the current user settings and creates lists to pass to the portal body view.
In our implementation, user settings are stored in a "settings object" of class `PortalSettings`; stored in the user session context.
The controller is a Struts Action extending TilesAction. It starts by retrieving the current user settings by calling the appropriate method. Then, it sets the attribute required by the portal layout (in real life the multi-columns layout).

Following is a code abstract of the action:

```
public final class UserPortalAction extends TilesAction {

   // …
  public ActionForward perform( ComponentContext context,
                                ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
                         throws IOException, ServletException
   {
     // Get user portal list from user context
   PortalSettings settings = getSettings( request, context);
```

```
      // Set parameters for tiles
    context.putAttribute( "numCols", Integer.toString(settings.getNumCols()) );
    for( int i=0; i<settings.getNumCols(); i++ )
      context.putAttribute( "list"+i, settings.getListAt(i) );

    return null;
    }

  public static PortalSettings getSettings( HttpServletRequest request,
                                            ComponentContext context )
  {
    // …
  }
}
```

The method `getSettings(…)` retrieves the "settings object" from the user's session, or creates it if it does not already exist. You can modify this method to retrieve settings from another source, like a DB.

The controller is declared as a Struts Action, so we need to declare the action in the Struts configuration file:

```
  <action     path="/portal/myPortal"
              type="org.apache.struts.example.tiles.portal.UserPortalAction">
  </action>
```

7.2.2.3 Portal Settings View

The portal settings view takes responsibility for the rendering of available Tiles and the rendering of actual settings. HTML `<select>` tags are used for this purpose: one for all available Tiles, and one for each column of current settings. Additional buttons and some JavaScript are used to move selections around columns.
The portal settings body requires a "settings form object" as attribute. This object contains the current user settings and available Tile keys and labels.
It's associated to a controller responsible for "settings form object" initialization.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<script language="javaScript1.2">
function selectAll( )
{
for( j=0; j<selectAll.arguments.length; j++ )
  {
  col1 = selectAll.arguments[j];
  for(i=0; i<col1.options.length; i++ )
    {
        col1.options[ i ].selected = true;
        }
  } // end loop
  return true;
}

function move( col1, col2)
{
  toMove = col1.options[ col1.selectedIndex ];
  opt = new Option( toMove.text, toMove.value, false, false );
  col1.options[col1.selectedIndex ] = null;
  col2.options[col2.length] = opt;
  return true;
}

function remove( col1)
{
  col1.options[ col1.selectedIndex ] = null;
  return true;
}
```

```
function up( col1 )
{
  index = col1.selectedIndex;
  if( index <= 0 )
    return true;

  toMove = col1.options[ index ];
  opt = new Option( toMove.text, toMove.value, false, false );
  col1.options[index] = col1.options[index-1];
  col1.options[index-1] = opt;
  return true;
}

function down( col1 )
{
  index = col1.selectedIndex;
  if( index+1 >=  col1.options.length )
    return true;

  toMove = col1.options[ index ];
  opt = new Option( toMove.text, toMove.value, false, false );
  col1.options[index] = col1.options[index+1];
  col1.options[index+1] = opt;
  return true;
}
</script>

<html:form action="/examples/myPortalSettings.do"  >

  <html:select property="remaining" multiple="true" >
    <html:options property="choices" labelProperty="choiceLabels" />
  </html:select>

  <html:button property="v" value="v" onclick="move(remaining,l0);return true;"/>
  <br>
  <table>
  <tr>
    <td>
          <html:select property="l0" multiple="true" size="10">
            <html:options property="col[0]" labelProperty="colLabels[0]"/>
          </html:select>
        </td>
        <td>
          <html:select property="l1" multiple="true" size="10">
            <html:options property="col[1]" labelProperty="colLabels[1]"/>
          </html:select>
        </td>
  </tr>
  <tr>
  <td align="center">
  <html:button property="right" value="^"   onclick="up(l0);return true;"/>
  <html:button property="right" value="del"   onclick="remove(l0);return true;"/>
  <html:button property="right" value="v"   onclick="down(l0);return true;"/>
  <html:button property="left" value=">"     onclick="move(l0,l1);return false;" />
  </td>
  <td align="center">
  <html:button property="right" value="<"    onclick="move(l1,l0);return true;"/>
  <html:button property="right" value="^"   onclick="up(l1);return true;"/>
  <html:button property="right" value="del"   onclick="remove(l1);return true;"/>
  <html:button property="right" value="v"   onclick="down(l1);return true;"/>
  </td>
  </tr>
  <tr>
    <td colspan="2"  align="center">

      <html:submit   property="validate"   value="validate"   onclick="selectAll(l0,   l1);return
true;"/>
       </td>
  </tr>
  </table>

</html:form>
```

The portal settings body contains a `<html:form>` with an action URL. This URL will be used when settings are validated. It denotes the site entry point of the page, not setting the controller URL or body URL.

The action entry point is declared as follows:

```
  <form-beans>

    <form-bean      name="myPortalSettingsForm"
                    type="org.apache.struts.example.tiles.portal.PortalSettingsForm"/>

  </form-beans>
    …
 <action     path="/examples/myPortalSettings"
                 type="org.apache.struts.tiles.actions.NoOpAction"
                      name="myPortalSettingsForm" >
      <forward  name="success"              path="examples.portal.settings.page"/>
  </action>
```

The action only forwards to the logical name "`success`", which is associated to the page definition. This page contains a nested Tiles rendering settings. It is mandatory to specify the form name because Struts uses it as data source in the HTML tags.

The portal settings body is associated with its controller in a definition:

```
  <definition name="examples.portal.settings.body" path="/examples/tiles/myPortalSettings.jsp"
              extends="examples.portal.body"
                      controllerUrl="/examples/controller/myPortalSettings.do" >
  </definition>
```

7.2.2.4 Portal Settings Controller

The dynamic Portal Settings controller is responsible for "settings body form" initialization and user settings validation. The controller starts by retrieving the current user settings and the catalog of all available Tiles to be displayed as choices. Then, it checks if the validate button has been pressed. If true, the new user settings are stored in the "user settings object".

```
public final class UserPortalSettingsAction extends TilesAction {

    public ActionForward perform( ComponentContext context,
                                  ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                           throws IOException, ServletException
  {
  PortalSettingsForm prefsForm = (PortalSettingsForm)form;

      // Get user portal settings from user context
  PortalSettings settings = UserPortalAction.getSettings( request, context);
  PortalCatalog catalog =
              UserPortalAction.getPortalCatalog( context, getServlet().getServletContext() );

  if( prefsForm.isSubmitted() )
    {  // read arrays
      // Set settings cols according to user choice
    for( int i=0;i<prefsForm.getNumCol(); i++)
      {
      settings.setListAt( i, catalog.getTiles( prefsForm.getNewCol(i)) );
      } // end loop

    prefsForm.reset();
         //return null; // (mapping.findForward("viewPortal"));
    }

      // Set lists values to be shown
    for( int i=0;i<settings.getNumCols(); i++ )
      {
      prefsForm.addCol(settings.getListAt(i) );
      prefsForm.addColLabels(catalog.getTileLabels( settings.getListAt(i)) );
      } // end loop
```

```
    prefsForm.setChoices(catalog.getTiles() );
    prefsForm.setChoiceLabels(catalog.getTilesLabels() );

    return null;
    }
}
```

In this implementation, ”`catalog`”’ is initialized from lists specified in the settings body definition. It’s possible to initialize it from other sources, like a DB, by rewriting the method `getPortalCatalog(…)`.

The controller is a Struts Action using a Struts Form. It should be declared in the Struts configuration file:

```
  <action     path="/examples/controller/myPortalSettings"
                 type="org.apache.struts.example.tiles.portal.UserPortalSettingsAction"
                        name="myPortalSettingsForm" >
  </action>
```

7.2.2.5 Using Dynamic Portal Tiles

The two definitions "portal body" and "portal settings body" can be used anywhere a Tiles definition can be used. Generally they are used as body of a classic layout.
The portal Tile is made by using an appropriate layout (multi-column layout), the list of Tiles to be displayed is built in an action associated to the layout.

Solution one:
The list is retrieved from session context. If not found, we initialize it by trying to load it from storage or create and initialize it from a definition.

Second solution :
(Planned for Tiles1.1) We use a definition stored in user definitions factory. If found, we use it, if not found, the application definitions factory is used. [TODO:] How and when do we initialize such a definition?

7.3 User Customizable Menu

This example shows how to build menus and menu bars with a "user menu" configurable by the user. Users can add any menus entries to their customizable menu.
Upon start, the menu shows a "customize" link. When chosen, the user settings page appears with a list of all available entries and the current user settings. A set of buttons allows adding, removing and arranging user entries. After validation the user menu shows the selected entries.

We start by describing a static menu, and then describe how to add a "user menu". Finally, we study the user settings page allowing the user to customize its menu.

7.3.1 Static Menu and Menu Bar

Menu and menu bar layouts have been described previously. To build static menus and menus bar, define them as definitions in a configuration file. Each menu definition contains the menu title and a list of entries with entry label, link and optionally an icon and a tool tip.

```
<definition name="examples.menu.root" path="/layouts/menu.jsp"  >
</definition>

<definition name="examples.menu.links" extends="examples.menu.root" >
  <put name="title" value="Examples" />
  <putList name="items" >
    <item value="Examples Home"
                      link="/examples/index.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="Portal"
                      link="/examples/portal.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="my Portal"
                      link="/examples/myPortal.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="my Portal Settings"
                      link="/examples/myPortalSettings.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="my Menu Settings"
                      link="/examples/myMenuSettings.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="Tabs (choosen pages)"
                      link="/examples/tabs.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="Tabs (Summaries)"
                      link="/examples/summariesTabs.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="Rss Channels"
                      link="/examples/rssChannels.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
  </putList>
</definition>

  <!-- Preferences menu definition -->
<definition name="examples.menu.settings" extends="examples.menu.root" >
  <put name="title" value="Preferences" />
    <putList name="items" >
    <item value="my Portal Settings"
                      link="/examples/myPortalSettings.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    <item value="my Menu Settings"
                      link="/examples/myMenuSettings.jsp"
               classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    </putList>
</definition>

  <!-- admin menu definition -->
<definition name="examples.menu.admin" extends="examples.menu.root" >
  <put name="title" value="Admin" />
  <putList name="items" >
    <item value="Reload"
          link="/admin/tiles/reload.do"
          classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
  </putList>
</definition>
```

In this example, we use the `/layouts/menu.jsp` layout. Note that menu definitions don't declare this layout as property, but instead extend one definition declaring the layout. This approach allows changing the layout for all menus in one single location.

The menu bar definition is responsible for rendering all menus in one single vertical bar. The menu bar contains a list of menu definition names and uses the vbox layout to render menus.

The definition is as follows:

```
<definition name="examples.menu.bar" path="/layouts/vboxLayout.jsp" >
  <putList name="list" >
    <add value="examples.userMenu" />
    <add value="examples.menu.links" />
    <add value="doc.menu.links" />
    <add value="examples.menu.settings" />
    <add value="doc.menu.taglib.references" />
    <add value="doc.menu.printer.friendly" />
```

```
    <add value="examples.menu.admin" />
  </putList>
</definition>
```

Note that definition names used are not necessarily defined in the same configuration file, they can be defined in other configuration files as well.
This menu bar can be used or rendered anywhere a definition name can be used. It is usually used as ”menu” parameter of a classic layout but is not restricted to that usage; e.g. it can also be used as a Tile of a portal.

7.3.2 User Menu Tile

The customizable user menu is a Tile associating a view and a controller. Classic menu layout is used for the view. The controller retrieves the user settings, and initializes the list of entries for the view layout.

Tile definition declaration is as follows:

```
<definition name="examples.userMenu" extends="examples.menu.root"
            controllerClass="org.apache.struts.example.tiles.portal.UserMenuAction"  >
  <put name="catalogSettings" value="examples.myMenu.catalog.settings"/>
  <put name="catalogName" value="examples.portal.menuCatalog"/>
  <put name="title" value="My Menu" />
    <putList name="items" >
      <item       value="customize"                          link="/examples/myMenuSettings.jsp"
classtype="org.apache.struts.tiles.beans.SimpleMenuItem" />
    </putList>
</definition>
```

The definition declaration doesn't specify directly which layout is used. Instead, it extends the "root menu" which specifies the layout used by all menus.

The definition contains four attributes used to initialize the user settings and the menu catalog:

- catalogSettings - Used to initialize the catalog. This attribute can be a definition name denoting a menu bar or a menu. It can also be a list similar as the one found in menu bar. This attribute is mandatory.
- catalogName - Name used to store the catalog in application scope [optional].
- title - Title of this menu.
- items - List of items used as default.

Following is an abstract of the controller code:

```
public final class UserMenuAction extends TilesAction implements Controller {
      /**
       * Tiles Action perform method.
       */
    public ActionForward perform( ComponentContext context,
                                  ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                           throws IOException, ServletException
    {
    perform( context, request, response, getServlet().getServletContext() );
          return null;
    }

   /**
    * Controller perform method.
```

```
    */
   public void perform(ComponentContext context,
                       HttpServletRequest request, HttpServletResponse response,
                       ServletContext servletContext)
        throws ServletException, IOException
    {
      // Load user settings from user context
    MenuSettings settings = getUserSettings( request, context);
      // Set parameters for rendering page
    context.putAttribute( USER_ITEMS_ATTRIBUTE, settings.getItems() );

    }
}
```

The controller implements two `perform()` methods: the one from `TilesAction` and the one from `Controller` interface where the first one calls the second. This construct allows using the action either as a Struts Action or directly as a Tile controller class.
The controller retrieves the user settings and sets attributes for menu layouts.
The user menu Tile definition can be used in the menu bar list, just like the other menus.

7.3.3 User Menu Setting Tile

The menu settings Tile is made of a view and a controller. The view renders all available entries and actual user settings. `<html:form>`, `<html:select>` and some JavaScript are used for this purpose. A set of buttons allows rearranging the user menu.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<%--
      Render a page allowing user to customize its menu.
          @param choiceItems Available menu entries proposed as choice
          @param userItems Actual user menu entries
--%>

<script language="javaScript1.2">
function selectAll( )
{
for( j=0; j<selectAll.arguments.length; j++ )
  {
  col1 = selectAll.arguments[j];
  for(i=0; i<col1.options.length; i++ )
    {
       col1.options[ i ].selected = true;
       }
  } // end loop
  return true;
}

function move( col1, col2)
{
  toMove = col1.options[ col1.selectedIndex ];
  opt = new Option( toMove.text, toMove.value, false, false );
  col1.options[col1.selectedIndex ] = null;
  col2.options[col2.length] = opt;
  return true;
}

function remove( col1)
{
  col1.options[ col1.selectedIndex ] = null;
  return true;
}

function up( col1 )
{
  index = col1.selectedIndex;
  if( index <= 0 )
    return true;

  toMove = col1.options[ index ];
  opt = new Option( toMove.text, toMove.value, false, false );
  col1.options[index] = col1.options[index-1];
```

```
  col1.options[index-1] = opt;
  return true;
}

function down( col1 )
{
  index = col1.selectedIndex;
  if( index+1 >=  col1.options.length )
    return true;

  toMove = col1.options[ index ];
  opt = new Option( toMove.text, toMove.value, false, false );
  col1.options[index] = col1.options[index+1];
  col1.options[index+1] = opt;
  return true;
}
</script>

<tiles:importAttribute name="catalog" />
<tiles:importAttribute name="userItems" />

<html:form action="/examples/myMenuSettings.do"  >
   <table align="center">
  <tr>
    <td align="right">
          Items Choice
          <br>
          <html:select property="selectedChoices" multiple="true" >
            <html:options collection="catalog" property="link" labelProperty="value"/>
          </html:select>
        </td>
        <td>
          <html:button   property=">"   value=">"   onclick="move(selectedChoices,selected);return
true;"/>
        </td>
    <td align="left">
          My Items
          <br>
          <html:select property="selected" multiple="true" size="10">
            <html:options collection="userItems" property="link" labelProperty="value"/>
          </html:select>
          <br>
      <div align="center">
          <html:button property="right" value="^"   onclick="up(selected);return true;"/>
          <html:button property="right" value="del"   onclick="remove(selected);return true;"/>
          <html:button property="right" value="v"   onclick="down(selected);return true;"/>
          <br>
      <html:submit   property="validate"   value="validate"   onclick="selectAll(selected);return
true;"/></div>
       </td>
  </tr>
  </table>
</html:form>
```

The view starts by importing Tiles attributes into the page context. These attributes will be used in `<html:options>`. There are two attributes: "`catalog`" and "`userItems`". The first one contains all available items, and the second one user items.
The view contains a `<html:form>` tag associated to a Struts URL Action. This URL is called when the user presses the validate button. The action does nothing except forward to the definition name describing the whole page containing the menu settings Tile.

Action description in Struts configuration file is as follows:

```
  <action     path="/examples/myMenuSettings"
                type="org.apache.struts.tiles.actions.NoOpAction"
                      name="myMenuSettingsForm" >
      <forward  name="success"              path="examples.userMenu.settings.page"/>
  </action>
```

The User Menu Setting Controller initializes attributes requested by the view. It loads the user settings and the catalog. It also receives user modification events and sets user settings accordingly.

```
public class UserMenuSettingsAction extends TilesAction
{

    public ActionForward perform( ComponentContext context,
                                  ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                           throws IOException, ServletException
    {
    MenuSettingsForm actionForm = (MenuSettingsForm)form;

        // Load user menu settings and available list of choices
    MenuSettings settings = UserMenuAction.getUserSettings( request, context );
    List catalog = UserMenuAction.getCatalog( context, request,
                                              getServlet().getServletContext() );

      // Check if form is submitted
      // If true, read, check and store new values submitted by user.
    if( actionForm.isSubmitted() )
      {  // read arrays
      settings.reset();
      settings.addItems( getItems(actionForm.getSelected(), catalog) );
      actionForm.reset();
      } // end if

      // Prepare data for view Tile
    context.putAttribute( "userItems", settings.getItems() );
    context.putAttribute( "catalog", catalog );

    }
}
```

Association of the view and controller is done in a definition:

```
  <definition name="examples.userMenu.settings.body" path="/examples/tiles/myMenuSettings.jsp"
              extends="examples.userMenu"
                        controllerUrl="/examples/controller/myMenuSettings.do" >
  </definition>
```

This definition extends the user menu definition in order to inherit default setting attributes ("`catalogSettings`" and "`catalogNames`").

The controller is a Struts Action and should be declared in the Struts configuration file:

```
  <form-beans>
    <form-bean      name="myMenuSettingsForm"
                    type="org.apache.struts.example.tiles.portal.MenuSettingsForm"/>

  </form-beans>
    …
  <action     path="/examples/controller/myMenuSettings"
                 type="org.apache.struts.example.tiles.portal.UserMenuSettingsAction"
                        name="myMenuSettingsForm" >
  </action>
```

7.4 User Customizable L&F

In this example we will see how to have a consistent L&F for the entire site and how to change this L&F easily: first statically by changing the configuration file, then dynamically by letting a user choose among available "layouts".

7.4.1 Consistent L&F

Having a consistent L&F throughout an entire web site is easily done by using the same layout for all pages. Also, it is recommended to use a stylesheet defining fonts, paragraph styles and color classes.
To ease maintenance and avoid repetitive declarations, it is a good habit to declare a "master page" definition, and let all pages inherit from it. The "master page" definition declares the layout used, and default values for title, header, menu, footer and body. Each page then extends the "master page" definition and overloads appropriate attributes, usually title and body. All other attribute values come from the "master page" definition.

A master definition could look like this:

```
<definition name="examples.masterPage" path="/layouts/classicLayout.jsp">
        <put name="title"  value="Tiles 1.1 Examples" />
        <put name="header" value="/examples/tiles/header.jsp" />
        <put name="menu"   value="examples.menu.bar" />
        <put name="footer" value="/examples/tiles/footer.jsp" />
        <put name="body"   value="/examples/tiles/body.jsp" />
</definition>
```

A page definition is declared as follows:

```
  <!-- Index page -->
<definition name="examples.index.page" extends="examples.masterPage">
        <put name="title"  value="Tiles 1.1 Example Summaries" />
        <put name="body"   value="examples.index.portal.body" />
</definition>
```

This way of centralizing attribute values can also be used in other kinds of layouts, like menu layouts.

In its simplest form, a web site with a consistent L&F is made of the following pages or Tiles: one header, one footer, one menu, one layout and several bodies. Consistent pages are assembled in a configuration file as explained previously. Page entry points are declared as Struts Actions forwarding to a definition name, or as JSP that inserts the definition name.

7.4.2 Static L&F Change

L&F change in a site built as explained previously can be done in several ways:

- Modify directly the layout used to render pages. This is the simplest way to change the L&F.
- Create a new layout to render L&F, and change the layout used in the "master page" definition.
  As all your page definitions inherit from the “master page” definition, the layout will be changed for all pages in one go. This solution allows several layouts and the ability to change the one used from time to time.

It is possible to specify the stylesheet used in the layout. Each layout can use a different stylesheet defining the same sets of fonts, paragraphs and colors, but in different ways.

7.4.3 Dynamic L&F

Changing the L&F of a site dynamically requires a way for changing the layout used dynamically.
It's not possible to programmatically change the configuration file, nor the layout code.
The proposed solution consists of a "switch layout" that switches to the appropriate layout according to some logic. The "switch layout" is written as an action. The URL of this action is used in place of the URL of the layout.

```
   <definition name="examples.masterPage" path="/examples/switchLayout.do">
         <put name="title"  value="Tiles 1.1 Examples" />
         <put name="header" value="/examples/tiles/header.jsp" />
         <put name="menu"   value="examples.menu.bar" />
         <put name="footer" value="/examples/tiles/footer.jsp" />
         <put name="body"   value="/examples/tiles/body.jsp" />
  </definition>
```

A simple implementation of "switch layout" is as follows:

```
public class SimpleSwitchLayoutAction extends TilesAction
{
    public ActionForward perform( ComponentContext context,
                                  ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                           throws IOException, ServletException
    {
    String layoutDir = "/layouts/";
    String userSelection = getUserSettings( context, request );
    String layout = "classicLayout.jsp";

    String layoutPath = layoutDir+userSelection+ "/" + layout;

    RequestDispatcher rd = getServlet().getServletContext().getRequestDispatcher( layoutPath );
    if(rd==null)
      {
      layoutPath = layoutDir + layout;
      rd = getServlet().getServletContext().getRequestDispatcher( layoutPath );
      if(rd==null)
        throw new ServletException( "SwitchLayout error : Can't find layout '"
                                    + layoutPath + "'." );
      }
    rd.include(request, response);
    return null;
    }

  public static String getUserSetting( ComponentContext context, HttpServletRequest request )
  {
  HttpSession session = request.getSession( false );
  if(session==null)
    Return null;
  return (String)session.getAttribute(DEFAULT_USER_SETTINGS_NAME);
  }
```

The URL of the layout to use is built by concatenating the layouts directory, user selected customized layout directory and layout name. The switch tries to load the URL, if it fails, the default layout is used. The layout sub-directory selected by the user is stored in session context. In this implementation, the name of the layout is coded in the class. This prohibits its reuse for other kind of layouts.

We can add flexibility by transforming the switch code to read the value of the layout to use from the Tile context:

```
public static final String LAYOUT_ATTRIBUTE = "layout.name";
 …
 String layout = (String)context.getAttribute( LAYOUT_ATTRIBUTE );
 if(layout==null)
   throw new ServletException( "Attribute '" + LAYOUT_ATTRIBUTE + "' is required." );
```

This transformation can also be applied to other parameters like "`layoutDir`".

The definition using "switch layout" must now declare an attribute specifying "`layout.name`":

```
<definition name="examples.masterPage" path="/examples/switchLayout.do">
      <put name=" layout.name"  value="classicLayout.jsp" />
      <put name="title"  value="Tiles 1.1 Examples" />
      <put name="header" value="/examples/tiles/header.jsp" />
      <put name="menu"   value="examples.menu.bar" />
      <put name="footer" value="/examples/tiles/footer.jsp" />
      <put name="body"   value="/examples/tiles/body.jsp" />
</definition>
```

The "switch layout" can be improved in order to let itself select a set of layouts declared in the configuration file. Each set declares layouts participating in one L&F. A set can be seen as a L&F or "skin". An implementation of such a "switch layout" is provided in the Tiles examples.

The example set of layouts can be declared as follows:

```
    <definition name="examples.customizable.layouts.root"
                path="/examples/controller/layoutSwitch.do"   >

      <put name="catalogSettings"   value="examples.available.skins" />
      <put name="layout.attribute"              value="page.layout" />

      <put name="title"  value="Tiles 1.1 Examples" />
      <put name="header" value="/examples/tiles/header.jsp" />
      <put name="menu"   value="examples.menu.bar" />
      <put name="footer" value="/examples/tiles/footer.jsp" />
      <put name="body"   value="/examples/tiles/body.jsp" />
    </definition>

  <!-- Available skins -->
<definition name="examples.available.skins" >
   <putList name="skin.list" >
        <add value="examples.default.skin" />
        <add value="examples.menuleft.skin" />
      </putList>
</definition>

  <!-- Default skin values -->
<definition name="examples.default.skin" >
  <put name="skin.label" value="Default" />
  <put name="page.layout" value="/layouts/classicLayout.jsp" />
  <put name="menu.layout" value="/layouts/menu.jsp" />
  <put name="menuBar.layout" value="/layouts/vboxLayout.jsp" />
</definition>

  <!-- Default skin values -->
<definition name="examples.menuleft.skin" extends="examples.default.skin" >
  <put name="skin.label" value="Left Menu" />
  <put name="page.layout" value="/layouts/skin1/menuLeftLayout.jsp" />
  <put name="menu.layout" value="/layouts/menu.jsp" />
</definition>
```

The first definition is used as "master layout". It declares the "layout switch", the definition used to initialize the catalog of skins ("`catalogSettings`"), and the layout identifier in each skin ("`layout.attribute`"). It is also possible to specify names used to store user settings in session context and skin catalog in application context.
The second definition declares a list of available skin definitions.

The remaining definitions are the skin declarations. Each skin specifies real layouts to use for this skin. There is also a label used in the page allowing the user to choose a skin setting.
To add a new skin, create a new skin definition, and add it to the list of available skins. It will appear automatically in the skin setting page.
To use the "switch layout" do the following: extend the "switch layout" master definition. You need to declare one "switch layout" master definition for each kind of layout (page, menu, …). They only differ by the "`layout.attribute`" value.

Another approach to provide customizable L&F is to use Tiles' channels capability: Layout master root definitions for each L&F are declared in a particular configuration file. Each additional L&F is declared in another file with the same name extended with channel suffix. Each file contains the same definitions, but with different values. Tiles uses the appropriate definitions file according to the channel name which is set in the user's session context.
This approach is much simpler than the switch layout. It can only be used if you don't already use channel or I18N capabilities.

Example: default layouts are declared in *layouts-config.xml*:

```
<definition name="examples.masterPage" path="/layouts/classicLayout.jsp">
        <put name="title"  value="Tiles 1.1 Examples" />
        <put name="header" value="/examples/tiles/header.jsp" />
        <put name="menu"   value="examples.menu.bar" />
        <put name="footer" value="/examples/tiles/footer.jsp" />
        <put name="body"   value="/examples/tiles/body.jsp" />
</definition>
```

Another L&F is declared in *layouts-config_skin1.xml* as follows:

```
<definition name="examples.masterPage" path="/layouts/skin1/classicLayout.jsp">
  <!–attribute are inherited from definition declared in default configuration file  -->
</definition>
```

To use this solution, you should use a different definition factory than the default one (this method is explained later). An example of a customized factory is available in the "channels" example. This factory can be used as is for the customizable layout. You just need to change the channel names to fit the configuration file extensions (see definition "`menu.channel`" in *componentDefinition.xml* from channels example).

8 Advanced Usages

8.1 Reusable Tiles: Components

Tiles can be used to develop "reusable components". Good candidates for reusable components are parts of pages that are repeated several times throughout a site. Developing reusable components allows only one set of source for all occurrences of this part. Modification or enhancement done in the component is automatically reflected wherever this component is used. This facilitates site maintenance: imagine that your e-commerce site displays addresses in several different pages, or the user basket. Your manager now asks you to change the way addresses and user baskets are rendered. If you have to use copy&paste to render them, you need to modify all occurrences manually. If you used components, you only need to change the component in one place.

As an example, we describe an address component used two times in the same form: one for the billing address and one for the shipping address. The form inserts the same Tiles component twice, but with different attributes.
First we describe the address component, then the HTML form inserting it and finally the Struts Form and Action associated to the HTML form.

The address Tile component is made of a JSP view (*tutorial/invoice/editAddress.jsp*):

```
<%@ taglib uri="/WEB-INF/tiles.tld"    prefix="tiles" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<%-- Edit an Address object
  @param bean An address object to edit.
  @param beanName The path to add between the bean and the properties to edit.
--%>
<%-- Retrieve parameters from component context, and declare them as page variable --%>
<tiles:useAttribute id="beanName" name="property" classname="java.lang.String" ignore="true" />
<tiles:importAttribute name="bean" />

<%-- Add a '.' separator to the path (beanName), in order to access the property from the given
bean --%>
<% if( beanName == null ) beanName = ""; else if (beanName !="" ) beanName = beanName + "."; %>

<table border="0" width="100%">

  <tr>
    <th align="right" width="30%">
      Street
    </th>
    <td align="left">
          <%-- Declare an html input field.
                --%>
          <%-- We use the bean passed as parameter.
                --%>
          <%-- Property name is prefixed by the sub-bean name if any.               --%>

          <html:text name="bean" property='<%=beanName+"street1"%>' size="50"/>

    </td>
  </tr>

  <tr>
    <th align="right">
      Street (con't)
    </th>
    <td align="left">
        <html:text property='<%=beanName+"street2"%>' size="50"/>
    </td>
  </tr>

  <tr>
    <th align="right">
      City
    </th>
    <td align="left">
        <html:text name="bean" property='<%=beanName+"city"%>' size="50"/>
    </td>
  </tr>

  <tr>
    <th align="right">
      Country
    </th>
    <td align="left">
        <html:text property='<%=beanName+"country"%>' size="50"/>
    </td>
  </tr>

  <tr>
    <th align="right">
      Zip code
    </th>
    <td align="left">
          <html:text property='<%=beanName+"zipCode"%>' size="50"/>
```

```
    </td>
  </tr>

</table>
```

This Tile requires two attributes: "property" and "bean". “bean” is the name of a bean in some scope containing address properties; “property”, if specified, is the name of a property in the previous bean. This property provides the properties for the address (city, country, …).
A Struts <html:text> tag is used to render all address properties. The name of the bean to use can be specified, or omitted if it is the same as the one used by Struts <html:form>. Properties are prefixed by the bean's property name if any.

The HTML form using previous address is as follows (*tutorial/invoice/editInvoice.jsp*):

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/tiles.tld"    prefix="tiles" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html:errors/>

<html:form action="/tutorial/invoice/editInvoice.do" >

<font size="+1">Edit Customer Informations</font>

<table border="0" width="100%">

  <tr>
    <th align="right" width="30%">
      First Name
    </th>
    <td align="left">
        <html:text property="firstname" size="50"/>
    </td>
  </tr>

  <tr>
    <th align="right">
      Last Name
    </th>
    <td align="left">
          <html:text property="lastname" size="50"/>
    </td>
  </tr>


  <tr>
    <th align="right" >
      Shipping Address
    </th>
    <td align="left">
           
    </td>
  </tr>
  <tr>
    <td align="center" colspan="2">
          <%-- Include an "address editor" component.                                    --%>
          <%-- Pass the component name and component value as parameter     --%>
          <%-- Value comes from the form bean --%>
          <tiles:insert page="/tutorial/invoice/editAddress.jsp" >
            <tiles:put name="property" value="shippingAddress" />
            <tiles:put name="bean" beanName="invoiceForm"  />
          </tiles:insert>
    </td>
  </tr>

  <tr>
    <th align="right" >
      Billing Address
    </th>
    <td align="left">
           
    </td>
```

```
  </tr>
  <tr>
    <td align="center" colspan="2">
        <tiles:insert page="/tutorial/invoice/editAddress.jsp" >
        <tiles:put name="property" value="billAddress" />
        <tiles:put name="bean" beanName="invoiceForm" />
        </tiles:insert>
    </td>
  </tr>

  <tr>
    <td align="right">
        <html:submit>
          save
        </html:submit>
        <html:submit>
          confirm
        </html:submit>
    </td>
    <td align="left">
        <html:reset>
          reset
        </html:reset>
       
      <html:cancel>
        cancel
      </html:cancel>
    </td>
  </tr>
</table>

</html:form>
```

The address Tile is inserted twice. Attribute values differ by the name of the property used to feed address properties.

The form and a Struts Action are declared as follows:

```
  <form-beans>

    <form-bean      name="invoiceForm"
                    type="org.apache.struts.example.tiles.invoice.InvoiceForm"/>

  </form-beans>
   …
  <action     path="/tutorial/invoice/editInvoice"
                 type="org.apache.struts.example.tiles.invoice.EditInvoiceAction"
                       name="invoiceForm" >
      <forward  name="success"        path="/tutorial/invoice/index.jsp"/>
  </action>
```

The action itself does nothing particular. The form object is automatically populated by Struts.

The invoice form class contains properties for an invoice. The two addresses properties return and set an address object:

```
public final class InvoiceForm  {

   // …
  public Address getShippingAddress()
   {
    return shippingAddress;}

  public void setShippingAddress(Address aShippingAddress)
    {
    shippingAddress = aShippingAddress;
    }

  public Address getBillAddress()
    {
    return billAddress;
```

```
    }

  public void setBillAddress(Address aBillAddress)
    {
    billAddress = aBillAddress;
    }

  public double getAmount()
    {
    return amount;
    }
  //…
}
```

A reusable component is a little bit more complicated than its counterpart used when doing copy&paste. You usually need some extra information or scriptlet in order to make the component reusable. This little extra work is really worth it as you will have one single definition of the Tile component. Also, a well developed component can be reused in different web applications.

8.2 How to link directly to a Tiles Definition Name?

After playing for a while with Tiles' definitions, you will try to access a Tiles definition name directly from a client page. This is often done by using the Struts `<html:link>` tag. This is not possible because Tiles definition names aren't URLs, but logical names.

There are several solutions:

- Create a page with an `<tiles:insert>` specifying the definition name.
  - Drawback: you need to create one JSP page for each definition.
- Create a Struts Action forwarding to a Tiles definition. The action does nothing, only the forward declaration is used. Tiles provides such an action as a standard one (`*.tiles.actions.NoOpAction` in Tiles1.1).
  - Drawback: you need to declare one action for each definition.
- Create an action taking a definition name as parameter and forwarding to this definition. Tiles provides such an action as a standard one (`DefinitionDispatcherAction` in Tiles1.1).
  - Drawback: possible security hole. Can be circumvented by adding name filter in the action.
  -

Note: A future version of Tiles will provide a property to specify the URL path allowing direct access to a definition by an URL. The implementation of this behavior will create a Struts Action with the specified path, and a forward associated to the definition name. To be short, the implementation will automatically perform the Action creation that we actually do "by hand".

8.3 Internationalizing Tiles

You can use all I18N capatibilities of Struts to internationalize your Tiles.
But Tiles additionally provides a way to select a Tile according to the Java Locale (language and country) using definitions and definitions files/factories. You can have one definitions factory per Locale. Each factory is linked to (loaded from) a definition file. The default load mechanism allows files starting with the same name, but ending with a Locale suffix, as with Java *.*properties* files.

When you request a definition by its name, the appropriate factory/file is selected, and the named definition is used. If a definition is not found in a localized factory/file, it is searched for in the nearest factory/file, and finally in the default factory/file (same rules as Java **.properties* files). The I18N factory is the default factory used by Tiles.
An example showing this is available in the Struts example: different configuration files (*tiles-tutorial-defs.xml*, *tiles-tutorial-defs_**fr**.xml*, …), language selection in menu, corresponding action (`*.examples.lang.SelectLocaleAction`).

8.4 Writing your own Definition Factory [TODO: Out of date]

Warning: The following description is out of date. The new Definition Factories should now extend DefinitionFactory.

It is possible to use a different definition factory than the default one. You can write your factory by implementing appropriate interfaces. You can reuse parts of the default implementation (XML file reader, I18N factory set, …).
Writing your own default factory allows you to change its behavior, e.g. it's possible to write a factory that selects a Tile definition using its name and a channel type (HTML, WAP, XML, …). Another example is a factory selecting a Tile using its name and a company number. This allows to have the same site used by different companies: a common directory contains common Tiles, and each company has its own directory containing specific Tiles.

To change the definition factory, use the appropriate parameter in *web.xml*:

```
    <init-param>
      <param-name>definitions-factory-class</param-name>
      <param-value>org.apache.struts.example.tiles.channel.ChannelFactorySet</param-value>
    </init-param>
```

This allows you to specify the class of the factory to use.

The interface to implement by your factory is simple:

```
package org.apache.struts.tiles;

/**
 * Component repository interface.
 * This interface allows to retrieve an definition by its name, independently of the
 * factory implementation.
 * Implementation must be Serializable, in order to be compliant with web Container
 * having this constraint (Weblogic 6.x).
*/
public interface ComponentDefinitionsFactory extends Serializable
{

   /**
     * Get a definition by its name.
     * @param name Name of requested definition.
     * @param request Current servelet request
     * @param servletContext current servlet context
     * @throws DefinitionsFactoryException An error occur while getting definition.
     * @throws NoSuchDefinitionException No definition found for specified name
     * Implementation can throw more accurate exception as a subclass of this exception
   */
   public ComponentDefinition getDefinition(String name, ServletRequest request, ServletContext
servletContext) throws NoSuchDefinitionException,DefinitionsFactoryException;

   /**
     * Init factory.
     * This method is called exactly once immediately after factory creation in
```

```
     * case of internal creation (by DefinitionUtil).
     * @param servletContext Servlet Context passed to newly created factory.
     * @param properties Map of name/property passed to newly created factory.
     * Map can contains more properties than requested.
     * @throws DefinitionsFactoryException An error occur during initialization.
   */
   public    void    initFactory(ServletContext    servletContext,    Map    properties)    throws
DefinitionsFactoryException;
}
```

Another option is to write a factory that reads definitions from other sources, like a DB. I hope Tiles will provide such a factory soon.

8.5 How to apply Tiles to legacy applications

First, write the layout, header, menu and footer. Transform existing pages into "body pages" by suppressing `<header>` and `<body>` tags.
Then create page definitions in your configuration file. Start with a master page definition and let others inherit from it.
Finally create page URL entry points in your Struts configuration file.

8.6 Admin facilities

Commands: Reload, view

Reload allows reloading Tiles definition files without restarting the web server. It must be defined in the Struts configuration file:

```
    <action    path="/admin/tiles/reload"
               type="org.apache.struts.tiles.actions.ReloadDefinitionsAction"/>
    <action    path="/admin/tiles/view"
               type="org.apache.struts.tiles.actions.ViewDefinitionsAction"/>
```

View allows printing out the current factory definitions. It is useful to check if definitions are loaded as expected, check inheritance, …
Note: actually nothing is outputted ;-)

9 Extra

9.1 How it works

A very short explanation of how Tiles works:

- Request arrives
- Insert (short description):
    - Definition is retrieved (if specified)
    - A context is created, attributes are set
    - Old context (if any) is saved, new context replaces it
        - Controller (if any) is called
        - forward to specified URL
            - Called URL uses context (attributes),
    - Context is discarded, old context is restored, request continues
- Insert (complete description):
    - Definition is retrieved (if specified)

    - A context is created, attributes are set (first from definition, then overloaded by attribute found in `<insert>` tag)
    - Old context (if any) is saved, new context replaces it
        - Controller (if any) is called
        - forward to specified URL
            - Called URL uses context (attributes),
    - Context is discarded, old context is restored, request continues
- Struts Forward to a definition name (special case):
    - Definition is retrieved
    - If no context exists, context is created
    - Context is populated with attributes in definition that are not already present in context
    - Forward to local URL specified in definition
- Struts Forward to an URL, but request context contains a definition (special case):
    - Definition is retrieved from request context
    - If no Tile context exists, create a context
    - Tile context is populated with attributes in definition that are not already present in context
    - Forward to local URL specified in definition
- Struts servlet (complete)
    - [TODO]

9.2 Definition file syntax

The syntax for the Tiles config file is nearly the same as the syntax for JSP tags.

Following is a quick description of tags available in Tiles config files:

- definition (syntax is a mix between JSP tags `<insert>` and `<definition>`)
    - `path` - Relative path of the layout to use (equivalent to `<tiles:insert>`, 'template' or 'component' or 'page').
    - `page` and `template` - Synonyms to page, don't need to use them.
    - `name` - Name of this definition
    - `extends` - Name of a definition used as ancestor. All attributes from the parent will be inherited by this definition; existing attributes can be overloaded.
    - `role` - Role to be checked when definition will be inserted in a page.
    - `controllerUrl` - Relative URL of a controller. This controller will be called with the proper Tiles context, before the page defined by "`path`" is inserted. This attribute usually denotes a Struts Action URL. Only one of `controllerClass` or `controllerUrl` should be used.
    - `controllerClass` - JAVA class of a controller. Appropriate method of this class will be called with the proper Tiles context before the page defined by "`path`" is inserted. Class can be a Struts Action or implement appropriate interface. Only one of `controllerClass` or `controllerUrl` should be used.
- put
    - Equivalent to JSP `<tiles:put>` tag.
    - `name` - Name of the attribute.

  - `value` or `content` - Value of the attribute (both are synonyms). Can have a body replacing the value attribute
  - `type` [optional] - Can be `string` | `page` | `template` | `definition`. Defines the type of content. See JSP `<tiles:put>` tag for explanation.
  - `direct` [optional] - Can be `true` | `false`. Defines the type of content. See JSP `<tiles:put>` tag for explanation.
- putList
  - Equivalent to JSP `<tiles:putList>` tag. Can accept nested `<add>` and `<item>` tags.
  - `name` - Name of the list attribute
- add
  - Equivalent to JSP `<tiles:add>` tag, which is the same as `<put>` minus the `name` attribute. Used inside a `<putList>`.
  - `value` or `content` - Value of the attribute (both are synonyms). Can have a body replacing the value attribute.
  - `type` [optional] - Can be `string` | `page` | `template` | `definition`. Defines the type of content. See JSP `<tiles:put>` tag for explanation.
  - `direct` [optional] - Can be `true` | `false`. Defines the type of content. See JSP `<tiles:put>` tag for explanation.
- item
  - Used to define menu entries, tabs index, ... Not (yet) available as JSP tag. This tag fills a bean defined by `classtype`. There is a default bean class (`org.apache.struts.tiles.beans.SimpleMenuItem`).
  - `value` - The menu item.
  - `link` - The menu link.
  - `classtype` [optional] - JAVA classtype used to hold tag's attributes. Default: `org.apache.struts.tiles.beans.SimpleMenuItem`
  - `icon` - URL of an icon used instead of the value.
  - `tooltip` - Text used as tooltip