Tapestry Contributor's Guide

by Howard Lewis Ship

Tapestry Contributor's Guide

by Howard Lewis Ship
Copyright © 2002, 2003 The Apache Software Foundation

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction

This document is a guide to developers who want to go beyond merely developing applications *using* Tapestry, and want to extend and improve Tapestry itself.

Tapestry has benefitted over the first two years of its development from having a focused vision and, predominantly, a single developer. At the time of this writing, May 2002, the Tapestry community is truly coming alive, with new developers contributing fixes, components and documentation.

The goal is to maintain the stability of Tapestry even as it shifts from a one-man-show to a true community effort. Meanwhile it is vitally important to not to sacrifice quality in either code or *documentation* if Tapestry is to stay on track.

Contributing to Tapestry requires a commitment to produce excellent code, examples and documentation. In fact, proper documentation in JavaDoc and as updates to the tutorials and manuals represents the *dominant* amount of effort when contributing to Tapestry.

Chapter 2. CVS Access

Using Eclipse, obtaining the source code takes only a few steps. Tapestry compiles using some libraries from JBoss 3.0.6 and Jetty 4.x which must be downloaded first.

Eclipse must be configured with the location of JBoss, this is done from the preferences panel. A new entry for JBOSS_DIR should be added.

Figure 2.1. Eclipse: Java Classpath Preferences
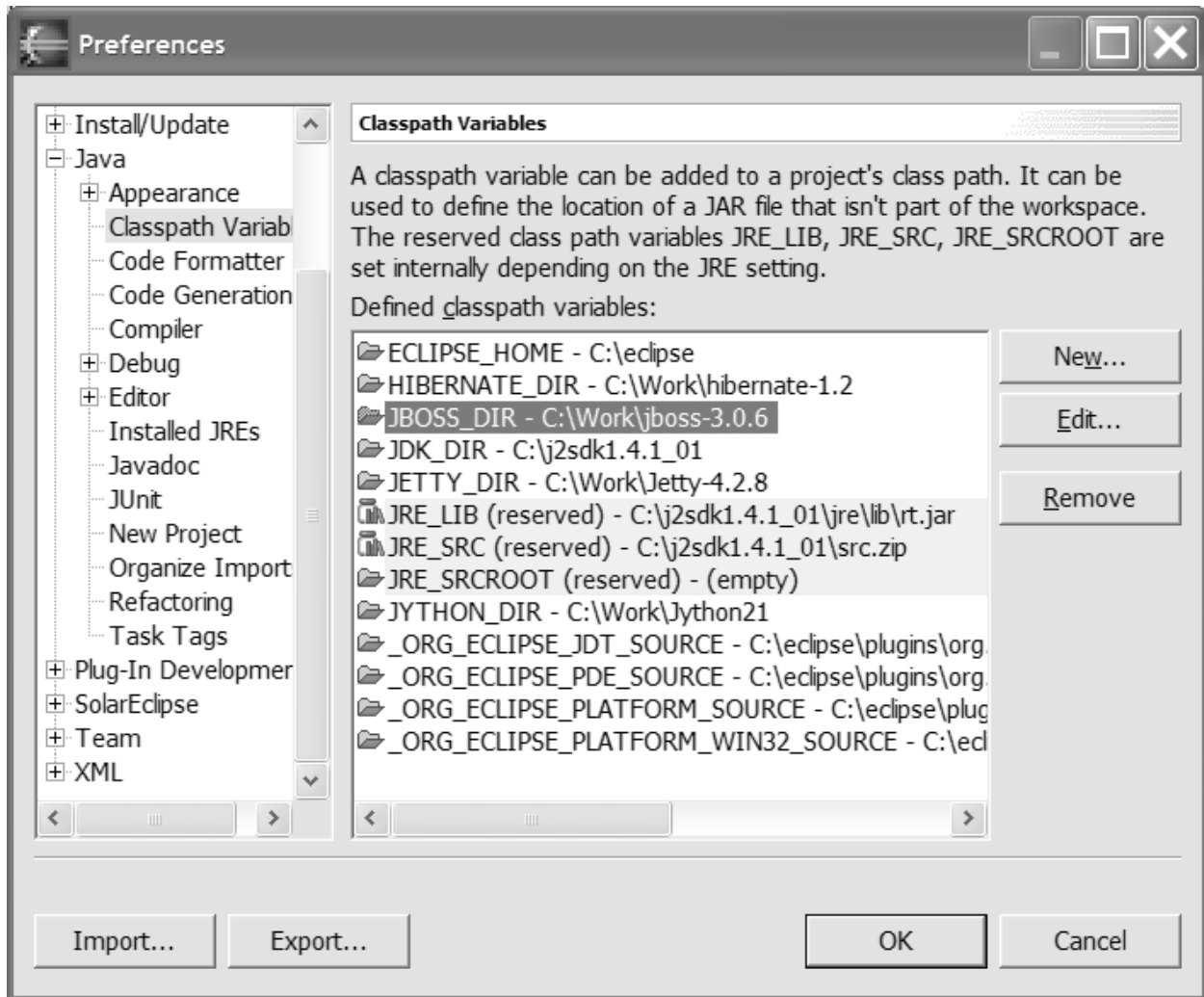
Activate the CVS Repositories view and use the context menu to create a new CVS Repository location. This raises a panel for defining connection information. Fill in your own Jakarta name and password:

Figure 2.2. Eclipse: New CVS Repository Location
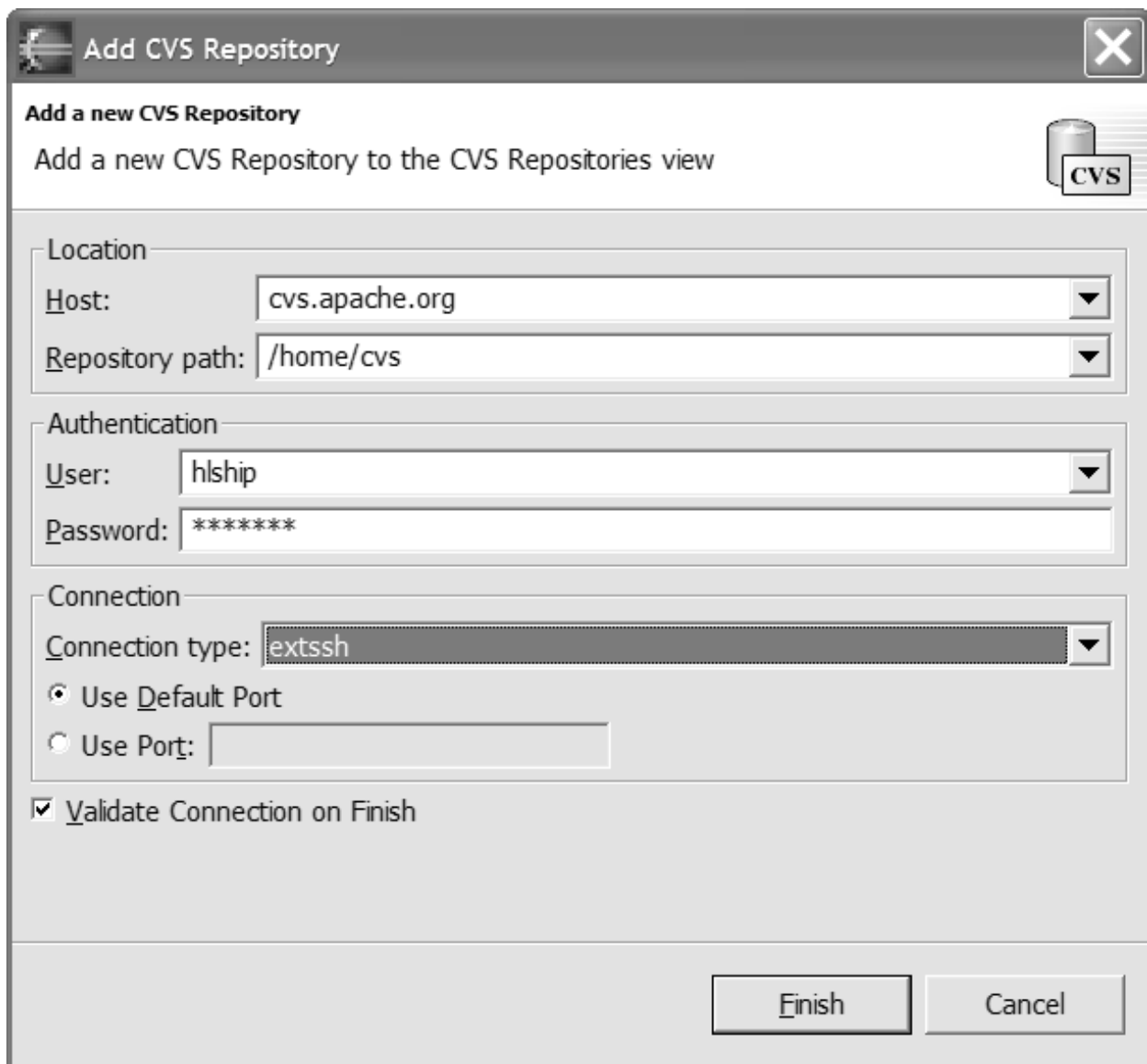
Next, open the new CVS Repository location. Expand the "HEAD" node, then scroll down to the "jakarta-tapestry" module. Right click and select "Check Out As Project".

Figure 2.3. Eclipse: Check Out Project

```
CVS Repositories
   ⊞  jakarta-taglibs.bu
   ⊞  jakarta-taglibs-sandbox
   ⊟  jakarta-tapestry
      ⊞  conf        │ New                          ▶
      ⊞  cont        │
      ⊞  doc         │ Check Out As Project
      ⊞  eclip       │ Check Out As...
      ⊞  exar        │ Check Out Into...
      ⊞  ext-        │
      ⊞  fram        │ Tag as Version...
      ⊞  imag        │ Tag with Existing...
      ⊞  img-        │
      ⊞  JBE         │ Compare With...
      ⊞  junit       │ Compare
      ⊞  lib         │ Add to Branch List...
      ⊞  pack        │ Configure Branches and Versions...
      ⊞  supp        │
      ⊞  web         │ Refresh View
         .classpath 1.31
         .cvsignore 1.4
```
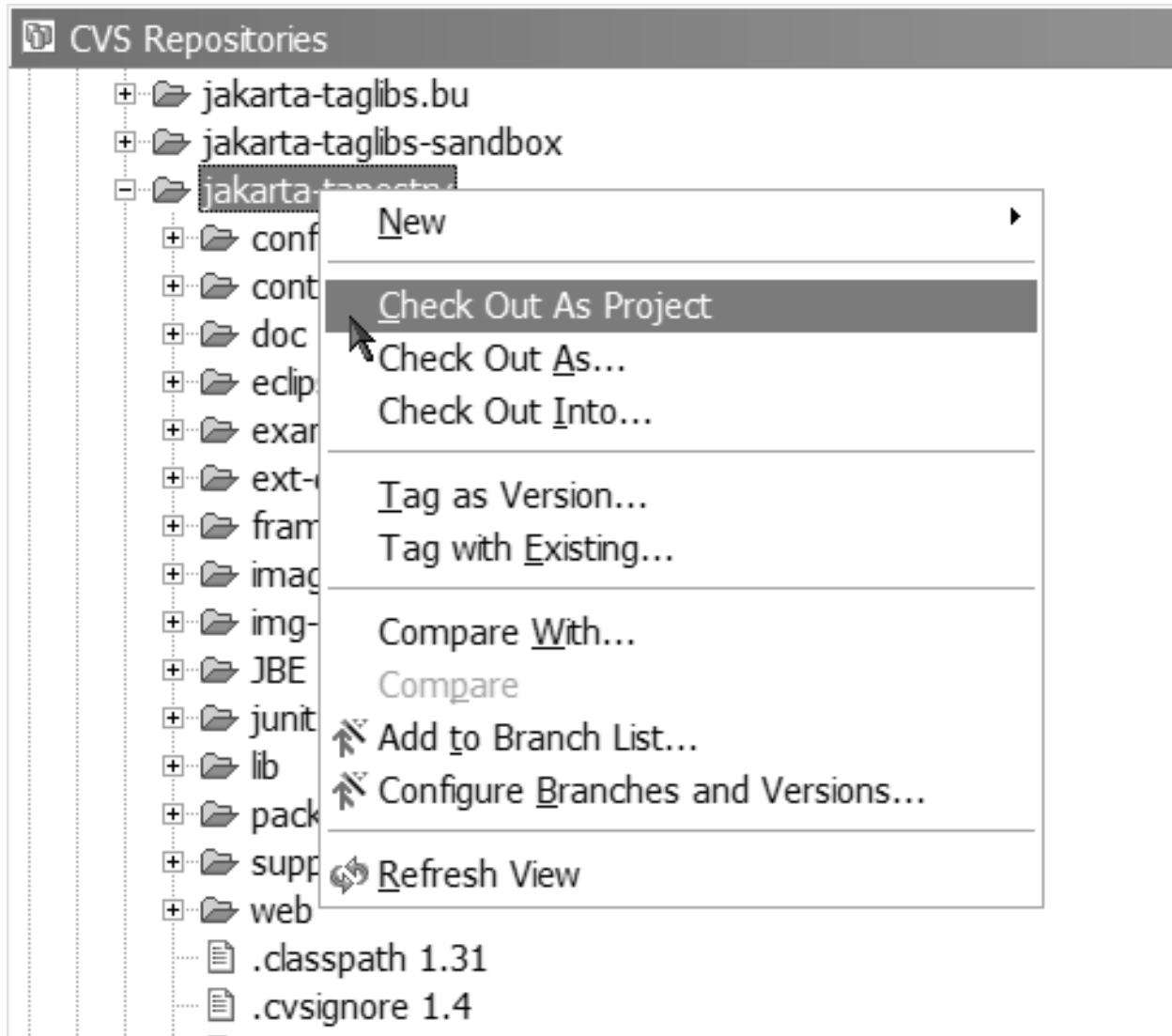
Eclipse will checkout the latest versions of all the Tapestry code and compile it.

You can access the Tapestry repository using command line CVS or other tools, as well. Details for using command line CVS are available at the Jakarta.

Chapter 3. Building Tapestry

Tapestry is built using Ant 1.5. In addition, Tapestry includes the necessary control files to allow development using the excellent open-source IDE, Eclipse.

To perform a full build from the command line, you must have JDK 1.3 or better installed, as well as JBoss 3.0.6.

You must create the file `config/build.properties` (under the Tapestry root directory). This file defines a property, `jboss.dir` that identifies the full pathname to the JBoss installation and the Jetty installation. A sample file is provided.

Tip

Be sure to use forward slashes for the path name, even under Windows. Using backslashes, the escape character in property files, will cause the build to fail, since Ant will be using incorrect paths to the libraries obtained from the JBoss distribution.

Tapestry has some additional, external dependencies on libraries that (due to licensing conflicts) are not supplied in the distribution or stored in CVS. More details available shortly ....

Tapestry Subprojects

The Tapestry source tree contains multiple sub-projects, each in its own subdirectory, with its own Ant build file and own source code tree. A root level build file (described in the next section) performs builds over all sub-projects.

Tapestry Sub-Projects

framework
> Contains the core framework, builds `tapestry-x.x.jar`.

contrib
> Builds `tapestry-contrib-x.x.jar`.

junit
> Builds and runs JUnit tests.

examples/Workbench
> Builds `workbench.war`.

examples/VlibBeans
> Builds `vlibbeans.jar`, the EJBs used by the Virtual Library demonstration.

examples/Vlib
> Builds `vlib.war`, the presentation layer of the Virtual Library demonstration.

examples/VlibEAR
> Builds `vlib.ear` from `vlibbeans.jar` and `vlib.war`.

doc/src/Tutorial
> Builds the Tapestry Tutorial documentation.

doc/src/DevelopersGuide
> Builds the Tapestry Developer's Guide documentation. This guide is out of date, as is being replaced.

doc/src/UsersGuide

Builds the Tapestry Users' Guide (the replacement for the Developer's Guide). This document is still incomplete. See, you just can't win.

`doc/src/ContributorsGuide`
Builds this very documentation.

`doc/src/ComponentReference`
Builds the component reference documentation.

Build Targets

The following Ant build targets are available from the Tapestry root directory:

Root Targets

clean
Cleans each sub-project and deletes derived files (such as the Tapestry framework JAR and examples).

clean-all
As with `clean`, but also deletes all documentation.

documentation
Builds all documentation (see notes below).

install
Performs a full build, by re-invoking `install` in each sub-project.

javadoc
Creates Tapestry API documentation.

junit
Runs all JUnit tests.

clover
Runs all JUnit tests and builds a code coverage report (using the Clover tool).

Documentation Setup

Tapestry documentation, including this manual, is also generated using Ant. Documentation source is in DocBook XML format, and uses XSL transformation to generate readable HTML. Tapestry uses Saxon to generate HTML documentation, and FOP to generate PDF documentation.

- Download and unpack the Saxon distribution, release 6.5.2 exactly (later versions do not work).

- Obtain the latest copies of the two DocBook distributions and place the files in the `ext-dist` directory. Details are in the file `doc/src/common/Readme.html`.

- Copy `saxon.jar` into the Ant `lib` directory.

- Update your `ANT_OPTS` environment variable to add the following two system properties:

  - -Djavax.xml.parsers.DocumentBuilderFactory=org.apache.crimson.jaxp.DocumentBuilderFactoryImpl

- -Djavax.xml.parsers.SAXParserFactory=org.apache.crimson.jaxp.SAXParserFactoryImpl

- Download FOP 0.20.4 and unpack into a permanent directory.

- Update `config/build.properties` and add a `fop.dir` entry, identifying the directory into which you unpacked FOP. Be sure to use an absolute path name, and only forward slashes.

- Get a copy of JIMI (an imaging package from Sun, needed by FOP to process PNG image files), and unpack it to temporary directory.

- Copy `JimiProClasses.zip` into the *FOP*`/lib` directory.

Clover Setup

Clover is a properietary tool that gathers code coverage information and generates reports from it. They have kindly donated a license for Clover to the Tapestry project.

To configure for clover:

- Get a copy of the Clover distribution. Cortex eBusiness has donated a copy of Clover to support Tapestry. The distribution is available from Howard M. Lewis Ship.

- Extract the Clover distribution to a non-temporary directory.

- Modify `config/build.properties` and add an entry for `clover.dir`. As usual, provide the absolute pathname to the Clover directory, using only forward slashes.

- Copy `clover.jar` to the *Ant*`/lib` directory.

The Clover report executes from the `junit` directory, using the Ant target `clover`. It runs builds the clover-enhanced version of the framework classes, and executs the JUnit test suite twice (with all logging enabled and all logging disabled), then generates the HTML report into the `web/doc/clover` directory.

Chapter 4. Development Standards

This chapter covers a number of standards, both in code and in procedure, expected by Tapestry contributors.

Use of Id Symbol

Every file checked into the CVS repository should have the Id symbol inside a comment, near the top of the file. The `$Id$` token is expanded by CVS into a useful header, identifying the revision of the file, date last changed, and name of last user to change the file.

For example, the Id for this document is `$Id: ContributorsGuide.xml,v 1.20 2003/04/21 15:39:20 hlship Exp $`.

Type Comment

Each Java file *must* have a complete and useful type comment. Type comments must come after all `import` statements, and before the start of the class.

Figure 4.1. Type Comment

```
/**
 *  A useful description of the class or interface, especially covering
 *  how it is used, and what other classes or interfaces it interacts with.
 *
 *  @author Your Name
 *  @version $Id$
 *  @since Version
 *
 **/
```

The *Version* should be replaced with the numeric version number of the Tapestry release the type will first appear in. This is the minor release number; for example, a change introduced in release `2.3-beta-3` would be identified as `2.3`.

JavaDoc

All methods should be commented, with the following exceptions:

* Simple accessor methods with no side-effects.

* Methods that are fully described by an interface and don't add any additional behaviors.

Parameters and return values should be identified. `@throws` should identify when any checked exceptions are thrown; additional `@throws` entries should describe any runtime exceptions that may also be thrown.

Methods should always include a `@since` entry, unless the method was added as part of a new Java class or interface, in which case the `@since` for the containing type is sufficient. Use the same version number as type comments when adding individual methods.

Try not to skimp on the comment (it is often best to write the comment before writing any code). Tapestry has some of the best documentation of any open source project and that should be maintained. Remember to try and answer the question *why?*, which is always much more interesting and useful than *how?* or *what?*.

It is appropriate to create JavaDoc comments for variables, even private variables (to at least provide an `@since` value).

Collections (from package `java.util`) should be documented to identify the type of object stored, and for `Map` the type of key. Example: `List of {@link IRender}`, or `Map of {@link IBinding} keyed on String name`.

When a method returns a collection type, the documentation should indicate if it is safe for the caller to modify the collection or not. In general, it is best to always return an immutable copy of a collection, but for efficiency this is not always reasonable.

And don't forget to make liberal use of JavaDoc links (`@link`) which makes the documentation far eaiser to use.

Javadoc Formatting

The standard for formatting Javadoc comments in Tapestry is to close the comment with `**/`. You should attempt to follow this, especially when modifying existing code.
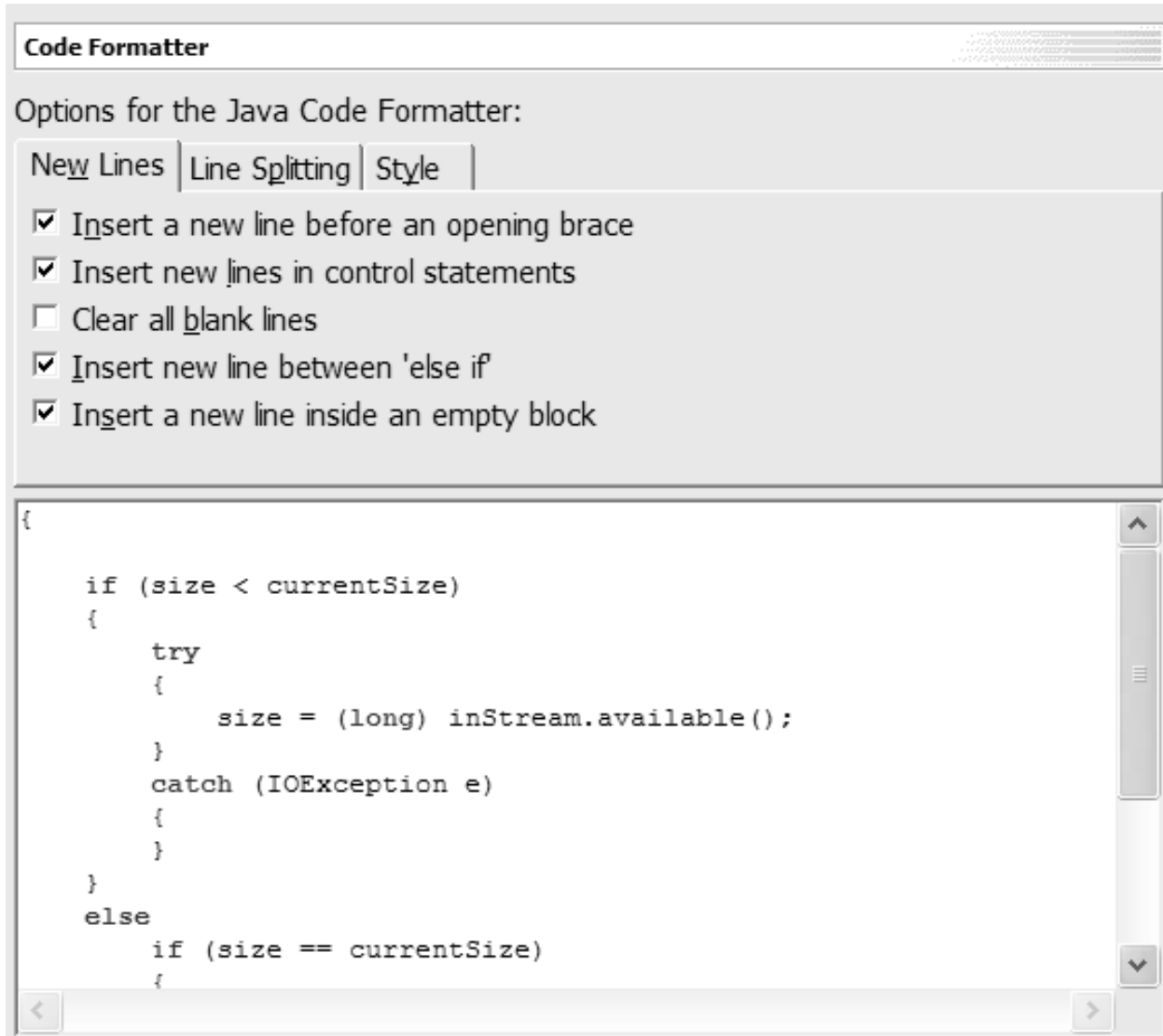
Java Code Formatting

Ah, a *religous issue*. The most important things are to be consistent (an editor that indents code for you is helpful) and to *conform to the existing style* when editting someone else's code.

Tapestry is formatted using *spaces* (not tabs), and an indent of four.

All the code currently in the repository has been formatted using the Eclipse IDE. My personal preference is to include a newline before opening braces. In addition, a maximum line-length of 100 characters has been used. These preferences are easy to setup in Eclipse:

Figure 4.2. Eclipse: Java Code Formatting Preferences

```
Code Formatter

Options for the Java Code Formatter:

New Lines | Line Splitting | Style

☑ Insert a new line before an opening brace
☑ Insert new lines in control statements
☐ Clear all blank lines
☑ Insert new line between 'else if'
☑ Insert a new line inside an empty block
```

```
{

    if (size < currentSize)
    {
        try
        {
            size = (long) inStream.available();
        }
        catch (IOException e)
        {
        }
    }
    else
        if (size == currentSize)
        {
```

Naming Conventions

Standard Java guidelines are expected to be followed. Class names are capitalized (example: `MyClass`). Method-sstart with a lower-case character (example: `myMethod`).

Static final variables used as constants are in upper-case (example: `MY_CONSTANT`).

Private member variables (both instance and static) are named with a leading underscore (example: `_myVariable`). Public member variables are to be avoided.

Naming in transition

I've resisted the leading underscore syntax for a long time; the rationale behind it is to make it possible, at a glance, to visually seperate instance variables from local variables and parameters. Previously, I've always maintained that the problem was methods that were too large; lately I've changed my mind ... the underscore naming helps when debugging and helps avoid a number of naming collisions.

At the time of this writing, 2.1-beta-1, very little of the code used the new naming. Over time, mixed in

with other bug fixes, renaming will occur (Eclipse helps with this greatly). New code will be written to conform.

Interfaces in Tapestry are prefixed with the letter 'I' (example: `IRequestCycle`). Implementations (often in a different package) strip off the 'I' (example: `RequestCycle`). Interfaces related to JavaBean events do not start with an 'I' (example: `PageDetachListener`).

Base classes, classes which are concrete and functional, but often extended, are prefixed with 'Base' (example: `BaseComponent`). Abstract classes are prefixed with 'Abstract' (example: `AbstractEngine`). Classes which are functional and only rarely subclassed are often prefixed with 'Default' (example: `DefaultScriptSource`).

The base package for the framework JAR (`tapestry-x.x.jar`) is `org.apache.tapestry`. The base package for the contrib JAR (`tapestry-contrib-x.x.jar`) is `org.apache.tapestry.contrib`.

Chapter 5. Tapestry Release Numbering

Tapestry release numbering is relatively simple, as long as you don't look back in time (the less managable numbering system used through release 2.0.5 is described shortly).

Tapestry releases consist of a major version, a minor version and a incremental version. The pattern `major.minor-incremental-index` is used, for example: `2.1`, `2.2-alpha-3` or `2.3-beta-1`.

The major version represents large-scale changes in the framework ... short of translating Tapestry to another language (say, Python or Ruby), this is not likely to happen again. Tapestry is currently in major release 2.

The minor version represents a milestone release, encompassing the introduction of new functionality and bug fixes in a stable manner. `2.1` or `2.2` would be examples of milestone releases.

An incremental release represents a transition from one milestone release to the next. Incremental releases are `alpha`, `beta` or `rc` (release candidate). Typically, after a milestone release there will be a series of alpha, then beta, then rc releases, leading up to the next milestone release. A possible sequence is `2.1`, `2.2-alpha-1`, `2.2-beta-1`, `2.2-rc-1`, `2.2`.

Typically, there will be several incremental releases of the same type, numbered from 1 up. Alpha releases contain significant functionality changes, beta releases represent bug fixes to those changes (stabilizing the changes), and rc (release chandidate) releases are expected to be stable versions of the next minor release (though any problems can spur further release candidates).

Through Tapestry release 2.0.5, numbering was a bit different. Under the modern scheme, 2.0.1 would be named `2.1-alpha-1`, 2.0.2 would be `2.1-alpha-2`, and 2.0.5 would be `2.1-beta-1`. Modern release numbering begins with `2.1-beta-2` (the release immediately following 2.0.5).

Chapter 6. Development Procedures

This chapter defines procedures for development of Tapestry. This includes many things not directly related to coding, such as documentation and interacting with the CVS repository.

Deprecating methods and classes

Tapestry is being used by a increasingly large community of developers and it is necessary that they have some stability in their development.

To that end, classes and methods must follow a developer-friendly lifecycle. If a method or class must be deleted, it should be marked as deprecated in one minor release, and can be removed in the following minor release.

For example, a method may be marked as deprecated in release 2.2-alpha-1. This change isn't considered "real" until release 2.2. The method can be removed any time after that, say in release 2.3-alpha-3, and the removal becomes "real" in release 2.3.

Don't simply mark a method as deprecated, give the end-developer the information needed adapt their code. Use the following template as part of the Javadoc comment:

```
@deprecated To be removed in Version.
  Use {@link SomeClass#someMethod(...)} instead.
```

It is also important for the changer to make the transition as simple as possible for the end-developer. Base classes and default implementations should be changed to make use of the new API in such as way that, at most, a recompile of the end-developer's classes is required.

Sometimes, changes require a lack of backwards compatibility. If a method has to change and the old signature can't be maintained, then simply change it ... but be sure to document the change in the Tapestry release notes `web/` `(new.html)`.

JUnit Tests

Tapestry has an excellent JUnit test suite, with code coverage figures over 80% at the time of this writing (2.4-alpha-4). It is *required* that changes to the framework be accompanied by additional JUnit tests (typically, mock tests; see below) to validate the changes. In addition, there is an ongoing effort to fill in the gaps in the existing suite; the suite should reach over 90% code coverage.

Some of the JUnit tests now require Jython. You must download and install Jython 2.1, then configure `jython.dir` in `config/build.properties` to point to the install directory. As usual, use an absolute path and forward slashes only. To run the JUnit test suite within Eclipse, you must set the `JYTHON_DIR` classpath variable.

JUnit test source code is placed into the `junit/src` source tree. The package name for JUnit tests is `org.apache.tapestry.junit`.

Less than half of Tapestry is tested using traditional JUnit tests. The majority of JUnit testing occurs using a system of mock unit tests. Mock testing involves replacing the key classes of the Servlet API (`HttpServletRequest`, `HttpSession`, etc.) with out own implementations, with extensions that allow for checks and validations. Instead of processing a series of requests over HTTP, the requests are driven by an XML script file, which includes output checks.

Generally, each bit of functionality can be tested using its own mini-application. Create the application as `junit/` `contextx`. This is much easier now, using Tapestry 3.0 features such as dynamic lookup of specifications and implicit components.

The Mock Unit Test Suite is driven by scripts (whose structure is described below). The suite searches the directory `junit/mock-scripts` for files with the ".xml" extension. Each of these is expected to be a test script. The order in

which scripts are executed is arbitrary; scripts (and JUnit tests in general) should never rely on any order of execution.

Test scripts are named Test*Name*.xml.

Note

The XML script is not validated, and invalid elements are generally ignored. The class MockTester performs the test, and its capabilities are in fluxx, with new capabilities being added as needed.

A test script consists of an <mock-test> element. Within it, the virtual context and servlet are defined.

```
<mock-test>
  <context name="c6" root="context6"/>

  <servlet name="app" class="org.apache.tapestry.ApplicationServlet">
    <init-parameter name="org.apache.tapestry.engine-class"
        value="org.apache.tapestry.junit.mock.c6.C6Engine"/>
  </servlet>
```

The name for the context becomes the leading term in any generated URLs. Likewise, the servlet name becomes the second term. The above example will generate URLs that reference /c6/app. Specifying a root for a context identifies the root context directory (beneath the top level junit directory). In this example, HTML templates go in context6 and specifications go in context6/WEB-INF.

Following the <servlet> and <context> elements, a series of <request> elements. Each such element simulates a request. A request specifies any query parameters passed as part of the request, and contains a number of assertions that test either the results, generally in terms of searching for strings or regular expressions within the HTML response.

```
<request>
  <parameter name="service" value="direct"/>
  <parameter name="context" value="0/Home/$DirectLink"/>

  <assert-output name="Page Title">
<![CDATA[
<title>Persistant Page Property</title>
]]>
  </assert-output>
```

Warning

As in the above example, it is very important that HTML tags be properly escaped with the XML CDATA construct.

Adding failover="true" to the <request> simulates a failover. The contents of the HttpSession are serialized, then deserialized. This ensures that all the data stored into the HttpSession will survive a failover to a new server within a cluster.

All of the assertion elements expect a name attribute, which is incorporated into any error message if the assertion fails (that is, if the expected output is not present).

The <assert-output> element checks for the presence of the contained literal output, contained within the element. Leading and trailing whitespace is trimmed before the check is made.

```
  <assert name="Session Attribute">
  request.session.getAttribute("app/Home/message").equals("Changed")
  </assert>
```

The `<assert>` element checks that the provided OGNL expression evaluates to true.

```
  <assert-regexp name="Error Message">
<![CDATA[
<span class="error">\s*You must enter a value for Last Name\.\s*</span>
]]>
  </assert-regexp>
```

The `<assert-regexp>` looks for a regular expression in the result, instead of a simple literal string.

```
  <assert-output-matches name="Selected Radio" subgroup="1">
<![CDATA[
<input type="radio" name="inputSex" checked="checked" value="(.*?)"/>
]]>
    <match>1</match>
  </assert-output-matches>
```

The `<assert-output-matches>` is the most complicated assertion. It contains a regular expression which is evaluated. For each match, the subgroup value is extracted, and compared to the next `<match>` value. Also, the count of matches (vs. the number of match elements) is checked.

```
  <assert-output-stream name="Asset Content"
    content-type="image/gif"
    path="foo/bar/baz.gif"/>
```

The `<assert-output-stream>` element is used to compare the entire response to a static file (this is normally associated with private assets). A content type must be specified, as well as a relative path to a file to compare against. The path is relative to the junit directory. The response must match the specified content type and actual content.

```
  <assert-exception name="Exception">
  File foo not found.
  </assert-exception>
```

The `<assert-exception>` element is used to check when an request fails entirely (is unable to send back a response). This only occurs when the application specification contains invalid data (such as an incorrect class for the engine), or when the Exception page is unable to execute. The body of the element is matched against the exception's message property.

Force a failure, then check for correctness

Sometimes the tests themselves have bugs. A useful technique is to purposely break the test to ensure that it is checking for what it should check, then fix the test. For example, adding XXX into a `assert-<output>`. Run the test suite and expect a failure, then remove the XXX and re-run the test, which should succeed.

Documentation

Documentation is much harder than coding, but the ongoing success of Tapestry depends on maintaining the quality of documentation. Tapestry documentation is written using DocBook XML format, using XSL stylesheets to convert to final documentation.

Changes to the framework usually require a change in documentation to the Tapestry Developer's Guide.

Component Documentation

Warning

This section is out of date. In general, each component should include a link to the Component Reference page for the component. The Component Reference page has a format and content similar to what's listed here.

Although there is limited documentation about components in their component specification file, that documentation is designed to be a short reminder, not the complete documentation. Full documentation goes into the component's Java file, as part of its type comment JavaDoc.

Component documentation consists of a table, identifying all the formal parameters of the component. In addition, a note indicating whether informal parameters are allowed, and if the component may have a body (that is, wrap other components) is supplied at the end.

Figure 6.1. Component Documentation Template

```
/**
 *   Type comment documentation ...
 *
 *   <p><table border=1>
 *   <tr>
 *     <th>Parameter</th>
 *     <th>Type</th>
 *     <th>Direction</th>
 *     <th>Required</th>
 *     <th>Default</th>
 *     <th>Description</th>
 *   </tr>
 *
 *   <tr>
 *     <td>name</td>
 *     <td>{@link Type}</td>
 *     <td>in|out|in-out</td> ❶
 *     <td>yes|no</td>
 *     <td>Default value</td> ❷
 *     <td>Full description</td>
 *   </tr>
 *
 *   ...
 *
 *   <p>Informal parameters are [not] allowed.  The component
 *   may [not] contain a body.
 *
 *   ...
 *
 **/
```

❶   This describes how the component uses its binding. in indicates the binding is read, but never updated, which is the most common case. out indicates the binding is updated, but not read; this is rare, but does apply to some parameters of Foreach, for example. in-out is common used with certain form parameters.
❷   If the parameter is required, then this is usually specified as   (non-breaking space).

Recently, seperate HTML component documentation has been created. This will be the standard location for Framework component documentation. Javadoc for the component should simply have a link to the correct Component Reference page.

The component reference is simply HTML (at least, for the time being). There are many examples and a template available, for creating new reference pages.

Checkin Procedures

Run JUnit tests before doing a checkin. You should always have a SourceForge bug or feature request. When checking code in, use the SourceForge request as the checkin comment.

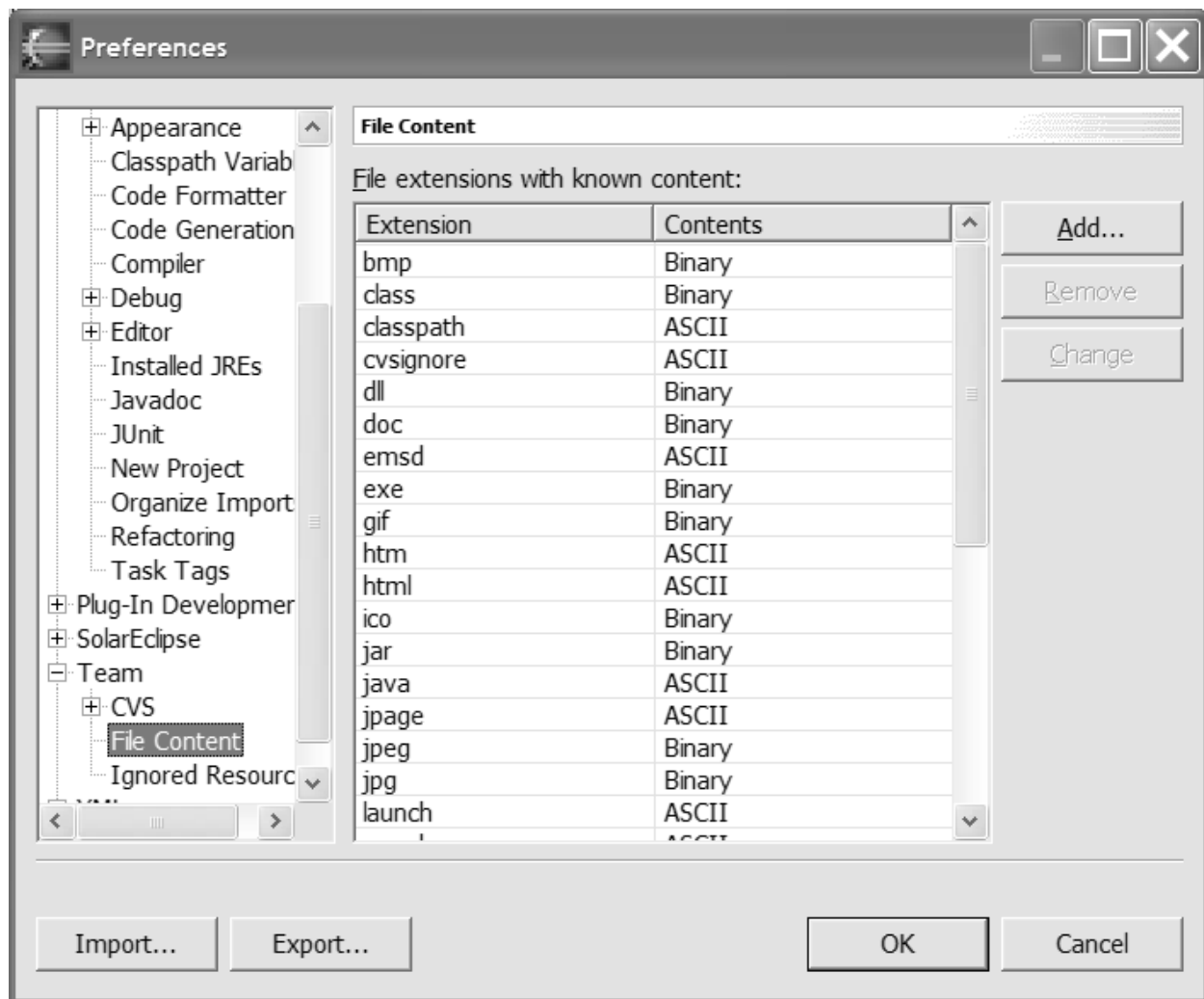**Example 6.1. Example checkin comment**

```
[ 553310 ] Set properties from parameter bindings
```

In addition, update the Tapestry release notes, the file `web/new.html`, to identify the feature request.

Be very careful when checking files in that they are checked in with the correct keyword substitution type. Files should be either binary or text; text should be checked in with keyword expansion turned on (this is the `-kkv` option).

When new files are added using Eclipse, it must decide whether they are binary or text. Eclipse always assumes *binary* unless specifically informed that a file is text. Use the Team preferences panel to set this.

Figure 6.2. Eclipse: Team Preferences

Creating Examples

Extending the Workbench application to demonstrate new features or components is expected for any significant changes or additions to the framework, or to the contrib library.

Updating Copyrights

All source code stored in the repository must contain the standard Apache copyright and license. A copy of the license, as a comment block, is stored as `support/license.txt`

The contents of this file can be pasted in directly before the `package` statement of a Java source file.

Alternately, a Python script is provided which can locate all Java source files within a directory tree and ensure that the leading comment block is correct. It modifies any source files where the leading comment doesn't match, but does not modify any files where the leading comment matches.

To use the script, execute the command **python support/update-copyrights.py LICENSE.txt *directory ...*.**

You may specify any number of directories, though the script is fast enough that just using "." (for current directory) is easiest.

Cygwin Python

On my computer (running Windows XP and/or 2000), when using the Cygwin version of Python, it is necessary to execute the script from the Bash shell, not the standard Windows command line.