Web Curator Tool
Developer's Guide

by
Kevin Urwin
April 2008

Copyright Oakleigh Consulting April 2008

Contents

1 Introduction 4
1.1 The Control Centre (WCTCore) 4
1.2 Harvest Agents (WCTHarvestAgent) 4
1.3 Digital Asset Store (WCTDigitalAssetStore) 4
2 Webcurator Tool Development Environment 5
2.1 Folder locations 5
2.2 Required packages and libraries 6
2.2.1 XDoclet 6
2.2.2 JUnit 6
2.2.3 Spring Mock Objects 6
2.3 Tomcat 6
2.3.1 Later versions of Tomcat 6
2.3.2 Additional java libraries 7
2.3.3 Switching between multiple database types 7
2.4 Ant 7
2.4.1 Nightly build and regression test script (wct-build-server.xml) 8
2.4.2 WCTCore build scripts 9
2.4.3 WCTDigitalAssetStore build scripts 10
2.4.4 WCTHarvestAgent build scripts 11
2.5 Eclipse 12
2.5.1 Importing ant build files 12
2.5.2 Modifying the build path and including additional jars 13
2.5.3 Debugging with Eclipse 18
3 SourceForge 20
3.1 CVS Software Configuration Management 20
3.1.1 CVS via Ant Scripts 20
3.1.2 Setting up Tortoise CVS 20
3.1.3 Checking out and Committing Code 24
3.1.4 Branching Code 26
3.1.5 Labelling (Tagging) Code 29
3.1.6 Merging Code 30
3.2 Website and Software Releases 31
3.2.1 Creating Release Packages in WCT 31

3.2.2 Creating/Uploading a Release 31
3.2.3 Editing the SourceForge Webcurator Website 34
4 Unit Testing Approach 36
4.1 Base Class 36
4.2 Mock Objects 37
4.3 Xml Data Files 38
4.4 Test Suites 38

1 Introduction

This document outlines the processes and practices employed during the first two iterations of the WCT development project for British Library.

The Web Curator Tool is used for managing a selective web harvesting process. It is typically used at national libraries and other collecting institutions to preserve online documentary heritage.

Unlike previous tools, it is enterprise-class software, and is designed for non-technical users like librarians. The software was developed jointly by the National Library of New Zealand and the British Library, and has been released as free (open source) software for the benefit of the international collecting community.

The tool is a Java based web application which runs within the Apache Tomcat container. It consists of three components as follows:

1.1 The Control Centre (WCTCore)

- The Control Centre includes an access-controlled web interface where users control the tool.
- It has a database of selected websites, with associated permission records and other settings, and maintains a harvest queue of scheduled harvests.

1.2 Harvest Agents (WCTHarvestAgent)

- When the Control Centre determines that a harvest is ready to start, it delegates it to one of its associated harvest agents.
- The harvest agent is responsible for crawling the website using the Heritrix web harvester, and downloading the required web content in accordance with the harvester settings and any bandwidth restrictions.
- Each installation can have more than one harvest agent, depending on the level of harvesting the organization undertakes.

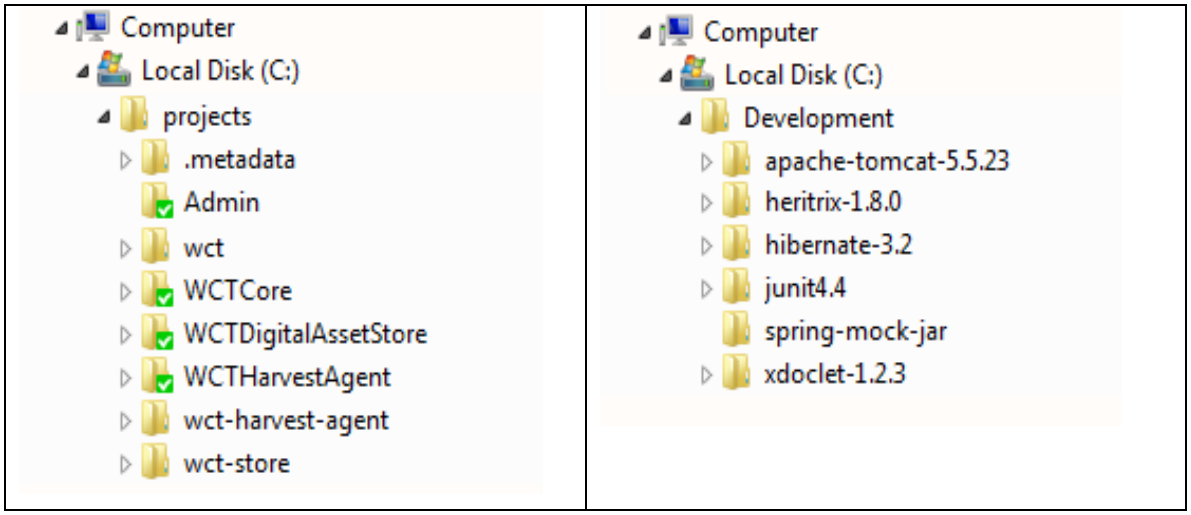
1.3 Digital Asset Store (WCTDigitalAssetStore)

- When a harvest agent completes a harvest, the results are stored on the digital asset store.
- The Control Centre provides a set of quality review tools that allow users to assess the harvest results stored in the digital asset store.
- Successful harvests can then be submitted to a digital archive for long-term preservation.

2 Webcurator Tool Development Environment

2.1 Folder locations

Development and deployment of the WCT application on a workstation takes place across two local folders as shown below:



The *projects* folder is the build folder. This contains the root source directories for each of the three WCT web applications, as checked out from SourceForge CVS; WCTCore, WCTDigitalAssetStore, and WCTHarvestAgent. The Admin folder (also checked out of SourceForge CVS contains the overall Ant build scripts for building, testing and deploying the whole suite. The Admin folder is mainly used for performing nightly build and regression testing on the build server. The remaining folders are generated by Eclipse; these are based on the project names defined in the ant build files for each of the applications. It is convenient that these folders are named slightly differently to the source folders as removing and completely replacing the source (such as to build on a different branch) does not affect the Eclipse project settings.

The *Development* folder is the deployment folder. We are deploying the WCT suite of applications to Apache Tomcat, and this is the location defined in the build scripts. It is possible to change the location of Tomcat by modifying the build.properties files for each of the applications, but to save unnecessary modification of checked out properties files on development machines, the above location is preferred.

The other folders are third party packages. Some (such as heritrix and hibernate) are present for reference only as the jar files are moved to the source folders for inclusion in the web application (war) files, while others are not included in the application but are used during the build process itself. These are described in section 2.2.

2.2 Required packages and libraries

2.2.1 XDoclet

XDoclet is used during the build process to generate the hibernate.cfg.xml file which maps the database schema to the application class hierarchy. This is achieved by reading tags which mark up the entity java classes.

The WCT build files require that XDoclet 1.2.3 is installed (see 2.1). One issue is that the version of Xjavadoc-1.1.jar provided with the download needs to be replaced by xjavadoc-1.5-snapshot050611.jar. This can be obtained, along with xdoclet itself, at http://sourceforge.net/project/showfiles.php?group_id=31602. In order to maintain the integrity of the build we found it necessary to rename the downloaded file to xjavadoc-1.1.jar.

2.2.2 JUnit

JUnit 4.4 is used for the unit and regression tests. This can be obtained from http://sourceforge.net/project/showfiles.php?group_id=15278. The build scripts expect this to be installed in a folder called 'junit4.4' in the development folder (see 2.1).

2.2.3 Spring Mock Objects

Spring Mock Objects are used by the unit and regression test classes. The spring-mock.jar library can be obtained in zip format from http://www.java2s.com/Code/Jar/Spring-Related/Downloadspringmockjar.htm. The build scripts expect this to be extracted into a folder called 'spring-mock-jar' in the development folder (see 2.1).

2.3 Tomcat

The Web Curator Tool as released is designed to run under Tomcat 5.5.23. This should be deployed under the development folder (see 2.1).

2.3.1 Later versions of Tomcat

It is possible, with some modification of build.properties in the build folders of WCTCore, WCTHarvestAgent and WCTDigitalAssetStore to deploy and run the project under Tomcat 6.0 or later versions. To do this the following properties should be changed to map to the installed location of the desired Tomcat instance:

tomcat.lib=c:/Development/apache-tomcat-5.5.23/common/lib

tomcat.webapps=c:/Development/apache-tomcat-5.5.23/webapps

tomcat.conf=c:/Development/apache-tomcat-5.5.23/conf/Catalina/localhost

2.3.2 Additional java libraries

In addition to the jar files deployed as standard in the tomcat lib folder, the following jar files should be present:

- jta.jar
- mysql-connector-java-5.1.5-bin.jar (when using a mysql database)
- ojdbc14.jar (when using an oracle database)
- postgresql-8.1-404.jdbc3.jar (when using a postgreSQL 8.1 database)

Note that jta.jar and postgresql-8.1-404.jdbc3.jar are provided in the wct_package_bin_v1.3.0.zip download. If you need to use a later version of the PostgreSQL database, or one of the other databases, the relevant JDBC driver should be downloaded and copied into the tomcat lib folder.

2.3.3 Switching between multiple database types

During testing, it is often desirable to run WCT against each supported database type. In order to do this there are a number of steps to take.

- Install the database, and use the scripts provided with the wct_package_bin_v1.3.0.zip download to build the database schema as described in the Web Curator Tool System Administrator Guide.
- Edit the local_<database>_config.properties file which can be found in \projects\WCTCore\build. Setting should be changed to match the installed database configuration. Most notably schema.name, schema.url, schema.user and schema.password.
- Edit build.properties to uncomment the appropriate ‘system’ property. It is important that only one ‘system’ property is uncommented at a time.
- Build the WCTCore using the ‘war’ target (use command *ant war* at the command line in the \projects\WCTCore\build folder)
- Copy the resulting wct.war file into the tomcat webapps folder.

It is important to deploy WCTCore using a war file when changing databases, as this forces tomcat to replace the wct.xml database context file in the \development\apache-tomcat-5.5.23\conf\Catalina\localhost folder.

2.4 Ant

Apache Ant 1.7.0 is used for scripting of all build and deployment operations for WCT outside of the Eclipse IDE. Scripts are run at two levels – building and deploying the individual web applications, and nightly build and regression test of the whole project.

2.4.1 Nightly build and regression test script (wct-build-server.xml)

This script is located in the \projects\Admin folder and is used to build all targets and run all unit tests during a nightly build. There is also functionality to move war files to an ftp folder location and send confirmation emails to interested parties. Targets in this script are as follows:

- **cvsWCTCore** – performs an anonymous checkout of a specified branch/label of the WCTCore project from SourceForge CVS.
- **cvsWCTHarvestAgent** – performs an anonymous checkout of a specified branch/label of the WCTHarvestAgent project from SourceForge CVS.
- **cvsWCTDigitalAssetStore** – performs an anonymous checkout of a specified branch/label of the WCTDigitalAssetStore project from SourceForge CVS.
- **buildWCTCore** – builds the WCTCore project using the build.xml file in WCTCore\build. This includes building all unit tests.
- **buildWCTHarvestAgent** – builds the WCTHarvestAgent project using the build.xml file in WCTHarvestAgent\build. This includes building all unit tests.
- **buildWCTDigitalAssetStore** – builds the WCTDigitalAssetStore project project using the build.xml file in WCTDigitalAssetStore\build. This includes building all unit tests.
- **testWCTCore** – runs all unit tests for the WCTCore project.
- **testWCTHarvestAgent** – runs all unit tests for the WCTHarvestAgent project.
- **testWCTDigitalAssetStore** – runs all unit tests for the WCTDigitalAssetStore project.
- **buildWAR** – builds war files for the WCTCore, WCTHarvestAgent and WCTDigitalAssetStore projects.
- **copyWAR** – calls buildWAR, then copies the resulting war files to a folder based on the date and build number in a specified root folder on the filesystem (for example an ftp root folder).
- **ftpWAR** – copies the specified war files to a folder based on the date and build number in a specified ftp location.
- **notifyMail** – sends an email to interested parties to inform them of the status of a completed build/regression test run. Copies of log files related to the build are zipped and then attached to the mail.
- **cvsGetAll** – calls cvsWCTCore, cvsWCTHarvestAgent and cvsWCTDigitalAssetStore.
- **buildAll** – calls buildWCTCore, buildWCTHarvestAgent and buildWCTDigitalAssetStore.
- **testAll** – calls testWCTCore, testWCTHarvestAgent and testWCTDigitalAssetStore.
- **fullTask** – this is the default task which calls cvsGetAll, buildAll, testAll and notifyMail.

2.4.2 WCTCore build scripts

The main script is located in the \projects\WCTCore\build folder and is used to build and deploy the WCTCore web application. There is also functionality to build zip and tar.gz files for release, which can be found in the build-package and build-package-src folders. Targets in the main script are as follows:

- **clean** – deletes the target folder
- **prepare** – creates the target and test\classes folders
- **hibdoclet** – builds the hibernate.cfg.xml config file using the tags defined in the java source classes
- **schemaexport** – builds the wct-schema-<schema name>.sql files in the db folder.
- **copy-config** – replaces tokenised information in config files, and copies then to the target folder. Also copies web content files to the target folder.
- **test_hib** – calls hibdoclet, copy-config and schemaexport
- **compile** – compiles java source in the src-app and src-api folders, placing the classes in the target\wct\classes folder
- **compiletests** – compiles java source in the src-test, src-app and src-api folders, placing the classes in the target\test\classes folder
- **double_escape_yes** – if browse.double_escape property is set, uncomment the regex in wct-browse-servlet.xml which escapes characters previously unescaped by ModJK.
- **double_escape_no** – unless the browse.double_escape property is set, comment out the regex in wct-browse-servlet.xml which escapes characters previously unescaped by ModJK.
- **double_escape** – calls double_escape_yes and double_escape_no.
- **deploy** – copies the contents of the target folder to the tomcat webapps folder.
- **war** – builds WCTCore and creates a wct.war file in the target folder
- **war-with-sql** – calls war and schemaexport.
- **quick-compile** – calls compile, copy-config and deploy.
- **all** – calls clean, hibdoclet, compile, copy-config, deploy and compiletests. This is the default target.
- **all-with-sql** – calls all and schemaexport.
- **deploy-jsps** – copies the web files to tomcat webapps
- **javadoc** – builds javadoc for the application.
- **release-bin** – builds wct-package-bin-v<version>.zip and wct-package-bin-v<version>.tar.gz files. However, it is recommended that the separate build files in the build-package folder are used for this task.

- **release-src** – builds wct-package-src-v<version>.zip and wct-package-src-v<version>.tar.gz files. However, it is recommended that the separate build files in the build-package-src folder are used for this task.
- **build-das** – calls the build file in \projects\WCTDigitalAssetStore\build.
- **build-agent** – calls the build file in \projects\WCTHarvestAgent\build.
- **dist** – calls war-with-sql, build-das, build-agent, javadoc, release-bin and release-src

2.4.3 WCTDigitalAssetStore build scripts

The main script is located in the \projects\WCTDigitalAssetStore\build folder and is used to build and deploy the WCTDigitalAssetStore web application. Targets in the main script are as follows:

- **clean** – deletes the target folder
- **prepare** – creates the target and test\classes folders
- **compile** – compiles java source in the src and src-api folders, placing the classes in the target\wct-store\classes folder
- **compiletests** – compiles java source in the src-test, src and src-api folders, placing the classes in the target\test\classes folder
- **copy-config** – replaces tokenised information in config files, and copies then to the target folder.
- **customise-archive-oms** – replaces oms archive related tokens in wct-das.xml with values defined in the local-config properties file which is itself defined in build.properties.
- **customise-archive-file** – replaces file archive related tokens in wct-das.xml with values defined in the local-config properties file which is itself defined in build.properties.
- **customise** – calls copy-config, customise-archive-oms and customise-archive-file.
- **deploy** – copies the contents of the target folder to the tomcat webapps folder.
- **war** – builds WCTDigitalAssetStore and creates a wct-store.war file in the target folder
- **all** – calls compile, customise, deploy and compiletests. This is the default target.

2.4.4 WCTHarvestAgent build scripts

The main script is located in the \projects\WCTHarvestAgent\build folder and is used to build and deploy the WCTHarvestAgent web application. Targets in the main script are as follows:

- **clean** – deletes the target folder
- **prepare** – creates the target and test\classes folders
- **compile** – compiles java source in the src and src-api folders, placing the classes in the target\wct-agent\classes folder
- **compiletests** – compiles java source in the src-test, src and src-api folders, placing the classes in the target\test\classes folder
- **copy-config** – replaces tokenised information in config files, and copies then to the target folder.
- **deploy** – copies the contents of the target folder to the tomcat webapps folder.
- **war** – builds WCTHarvestAgent and creates a wct-agent.war file in the target folder
- **all** – calls clean, compile, copy-config, deploy and compiletests. This is the default target.

2.5 Eclipse

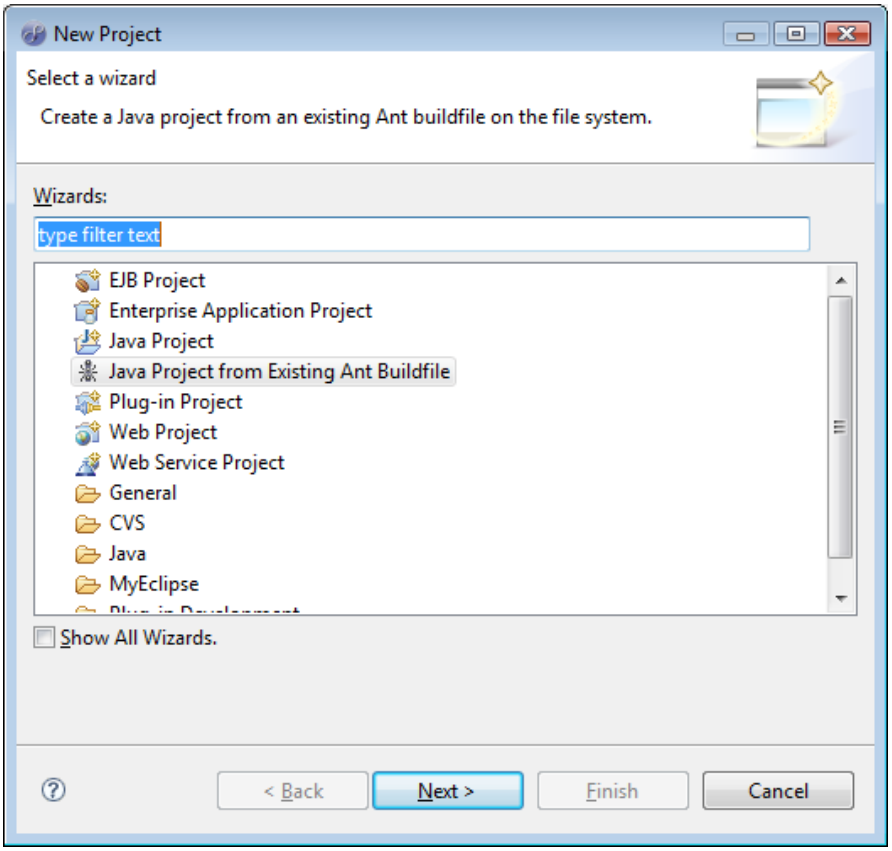
During the development phases of WCT, the preferred development environment has been MyEclipse 6.0. Although this version of Eclipse bears a subscription cost, it was found that this was easily recouped by the presence of a stable consistent version of Eclipse and its plugins, which enabled simple, consistent deployment across multiple development workstations.

2.5.1 Importing ant build files

The quickest way to set up Eclipse to build the WCT is to import the three main ant build files described in sections 2.4.2, 2.4.3 and 2.4.4. The following process should be used to import each project:

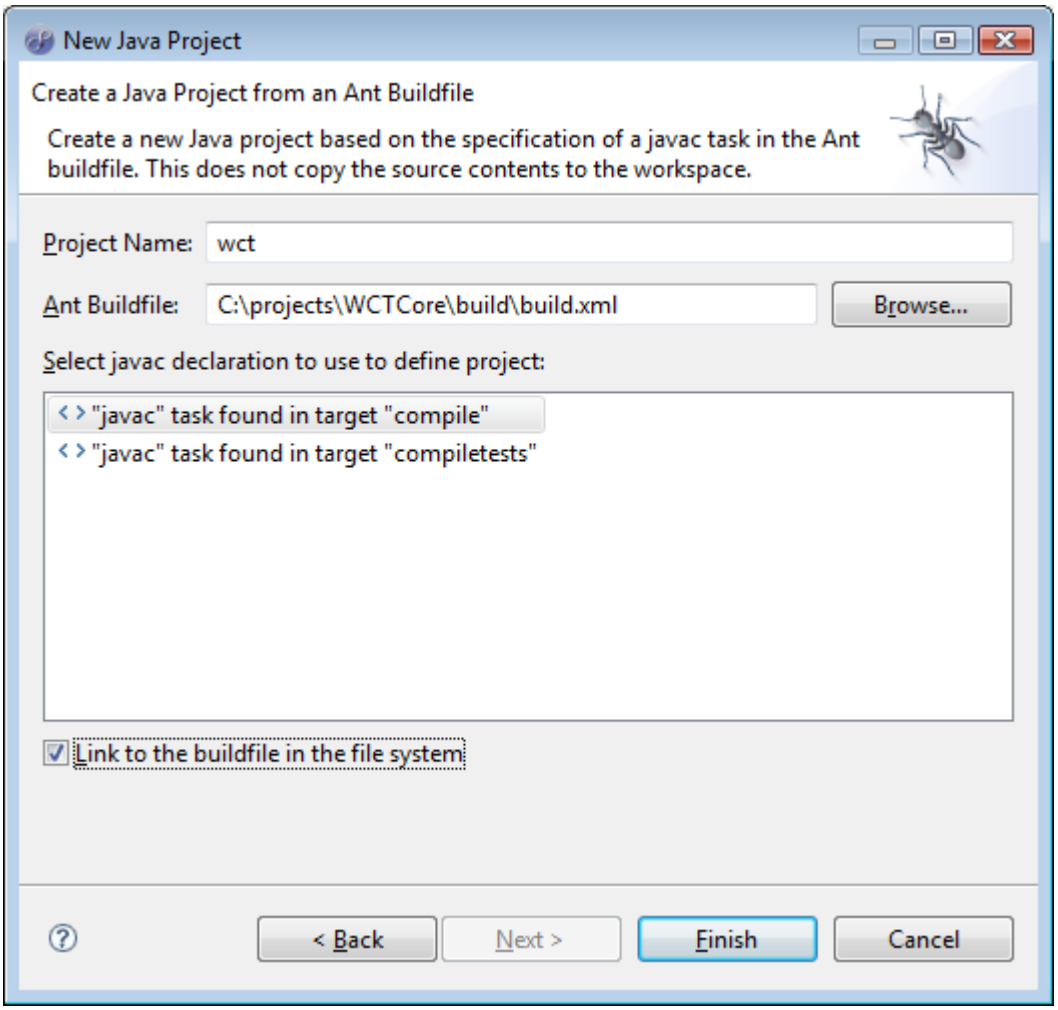
i. Choose File->New->Project... from the Eclipse Workbench

ii. Choose 'Java Project from existing Ant build file'



iii. Browse to the build.xml file.

iv. Check 'Link to the buildfile in the file system.



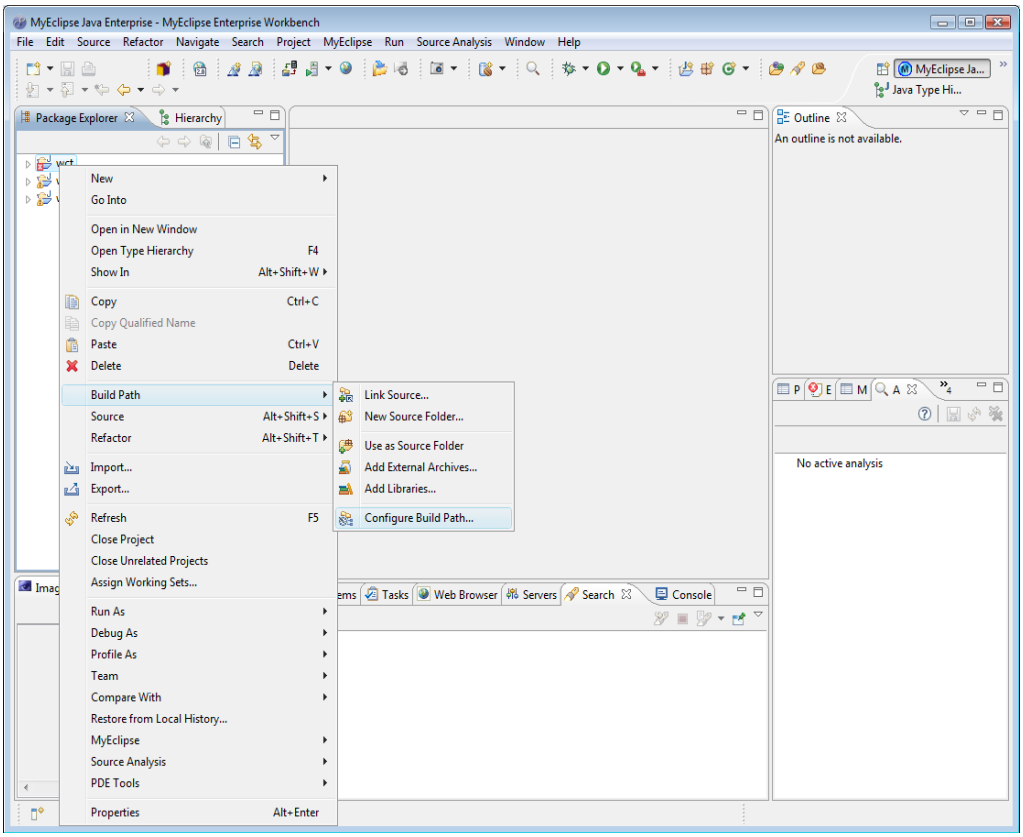
v. Click Finish to create the project in Eclipse.

Please note that at the time of writing there is an error in the build.xml file for WCTDigitalAssetStore which causes the project names to be incorrectly displayed as wct-harvest-agent. Simply change this to wct-store before clicking Finish.

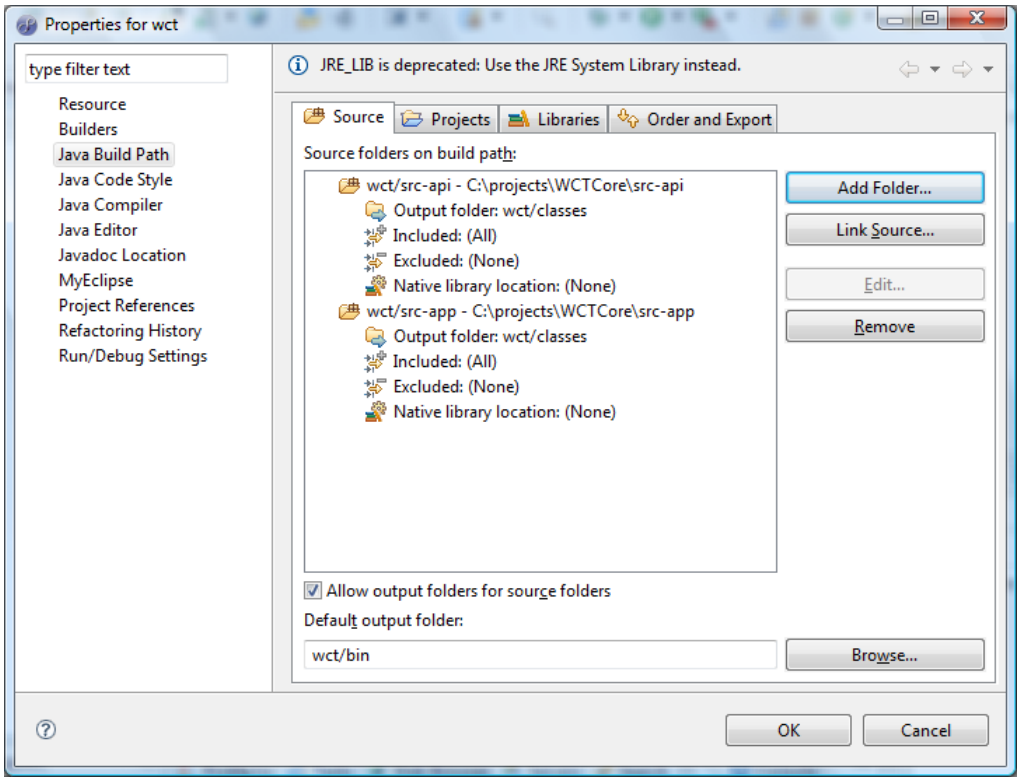
2.5.2 Modifying the build path and including additional jars

After importing the build files, the wct project is likely to show errors. There are further changes required to the build path within eclipse in order to correctly build the project.

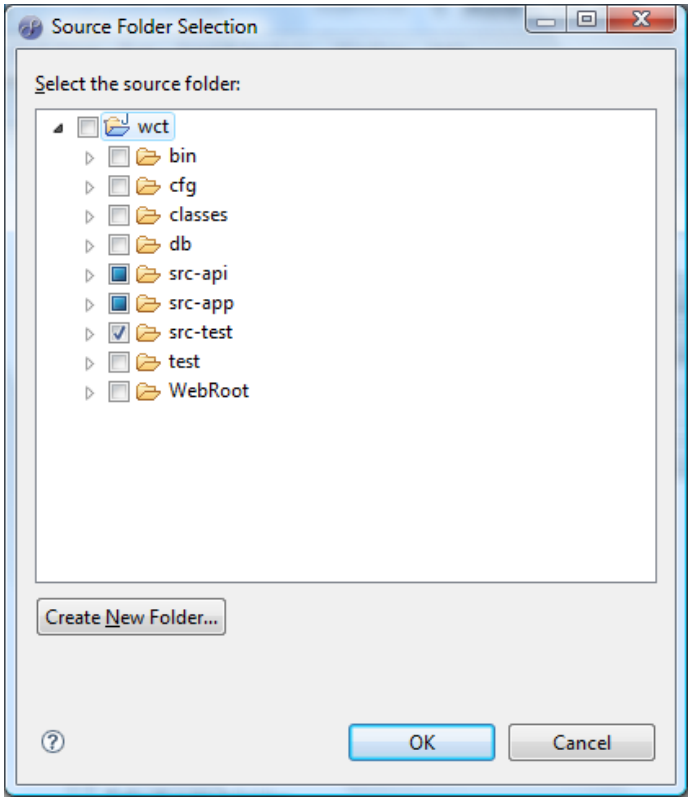
i. Right click on the wct project to select the Build Path->Configure Build Path option.



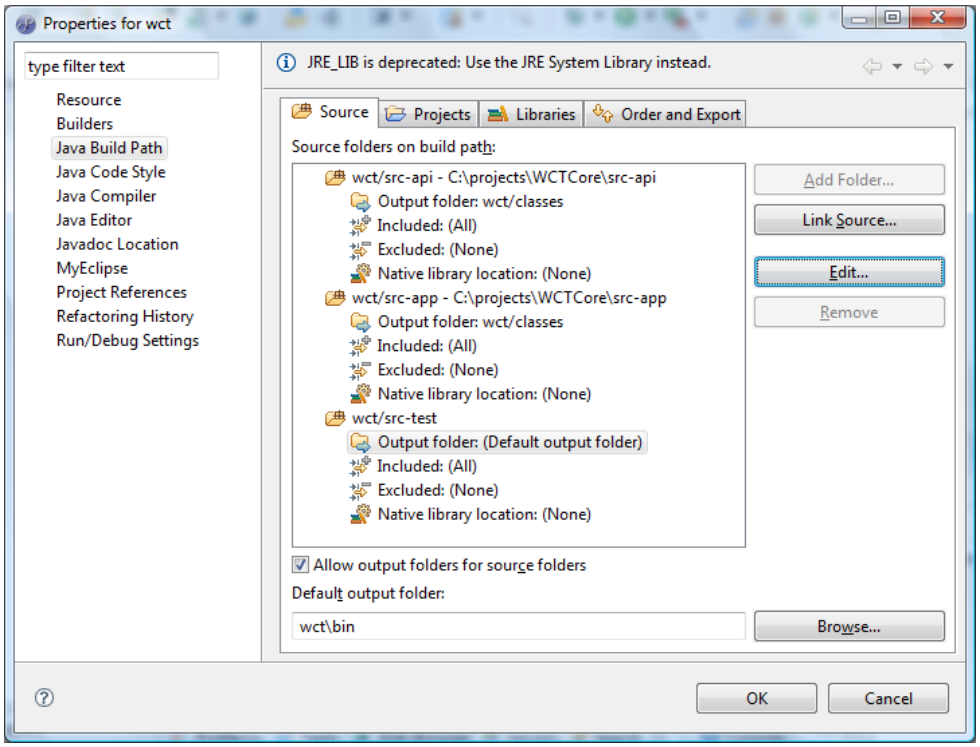
ii. On the ‘Source’ tab, click ‘Add Folder’



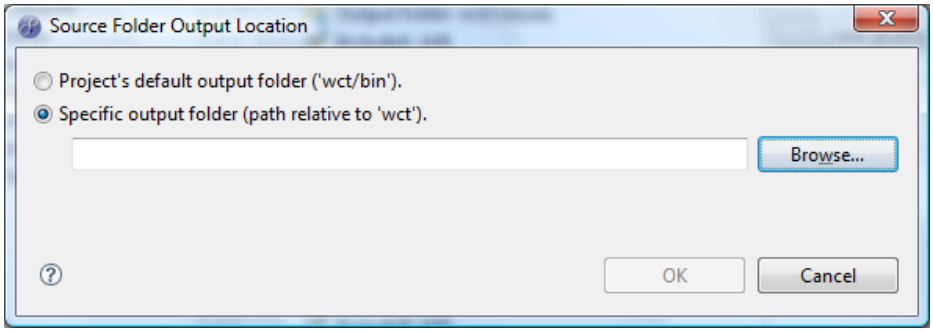
iii. Add the src-test folder as a source folder



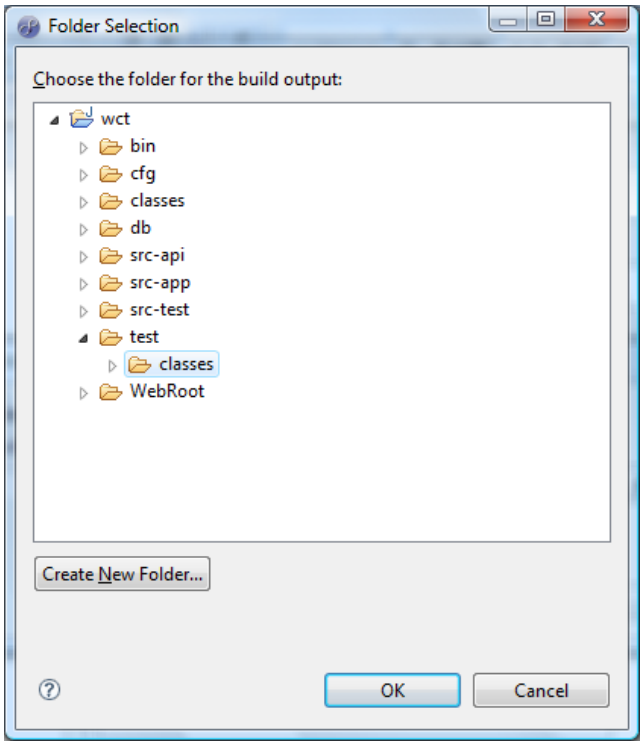
iv. Select 'Output folder:' under wct/src-test and click Edit.



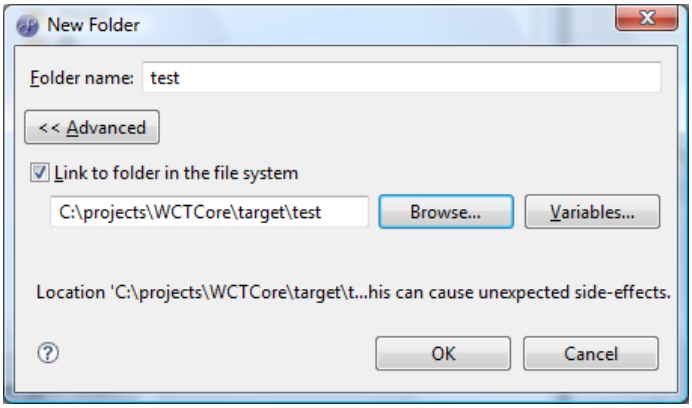
v. Choose ‘Specific output folder and click ‘Browse’



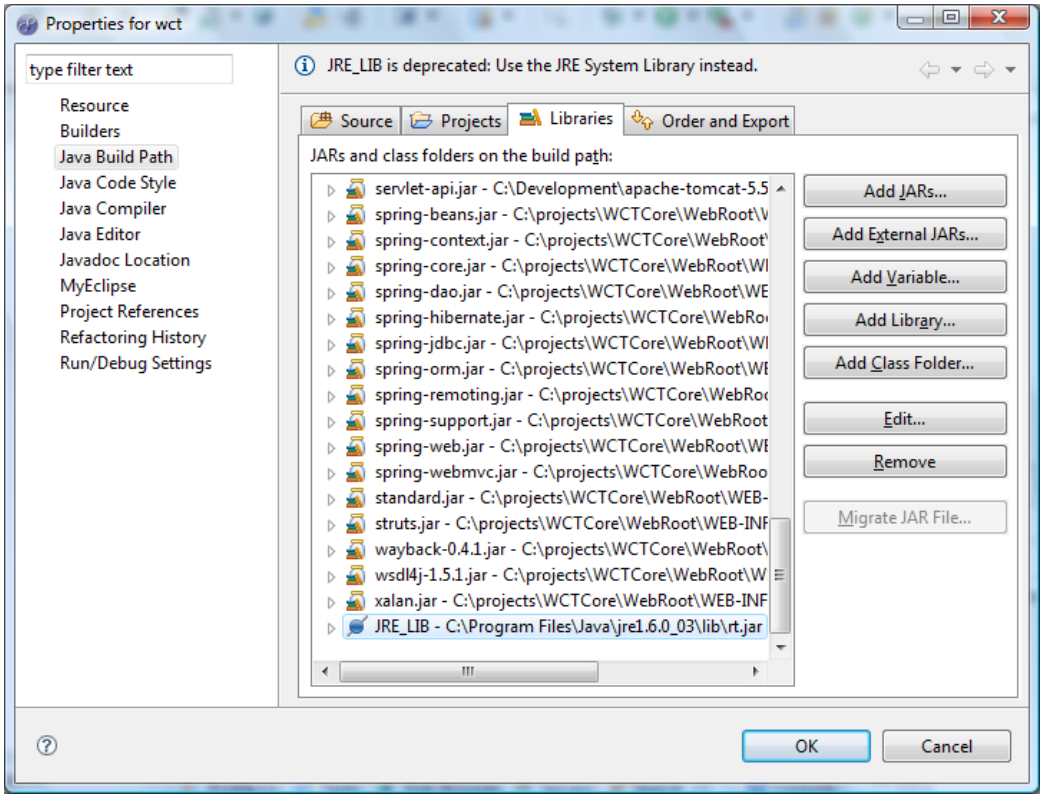
vi. Browse to wct\test\classes and click OK.



vii. There may be a need to create the test folder if it does not already exist. Use the 'Create New Folder' button and use the following options (note that it is advisable to already have built WCT using the ant scripts described in section 2.4 so that the target folders exist on the file system):



viii. Once the 'Source' tab is complete, move to the 'Libraries' tab. Firstly select and remove the JRE_LIB entry, as this method of referencing jars is deprecated in Eclipse.



ix. Now add the following External JARs:

- From the lib folder of the JRE:
  - rt.jar
  - jsse.jar
- From the JUnit4.4 folder:
  - junit4.4.jar
- From the spring-mock-jar folder:
  - spring-mock.jar

x. Repeat the above steps (i – ix) for the wct-agent and wct-store projects, ensuring that the src-api, src and src-test folders are correctly configured as source folders, with targets of 'classes', 'classes' and 'test\classes' respectively.

All three projects should now build in Eclipse (albeit with warnings). It is a good idea to check Project->Build Automatically for a continuous build to the target folders as files are modified.

2.5.3 Debugging with Eclipse

As a fully featured development environment, the Eclipse tool offers in place debugging facilities. This section describes two common scenarios which are used during the normal development cycle.

Debugging a JUnit unit test

This scenario allows for the local debugging of one or more JUnit test classes, and the code they call. In order to debug a specific class (or suite of classes) take the following steps:

i. In the MyEclipse Java Enterprise Workbench perspective, select the class to be debugged in the Package Explorer and double click to open it.

ii. Right click in the left margin of the code window to toggle a breakpoint on the required line(s) of code.

iii. Right click the class name in the Package Explorer and choose Debug As->JUnit Test

iv. The test should run, and stop on any breakpoints, switching to the debug perspective.

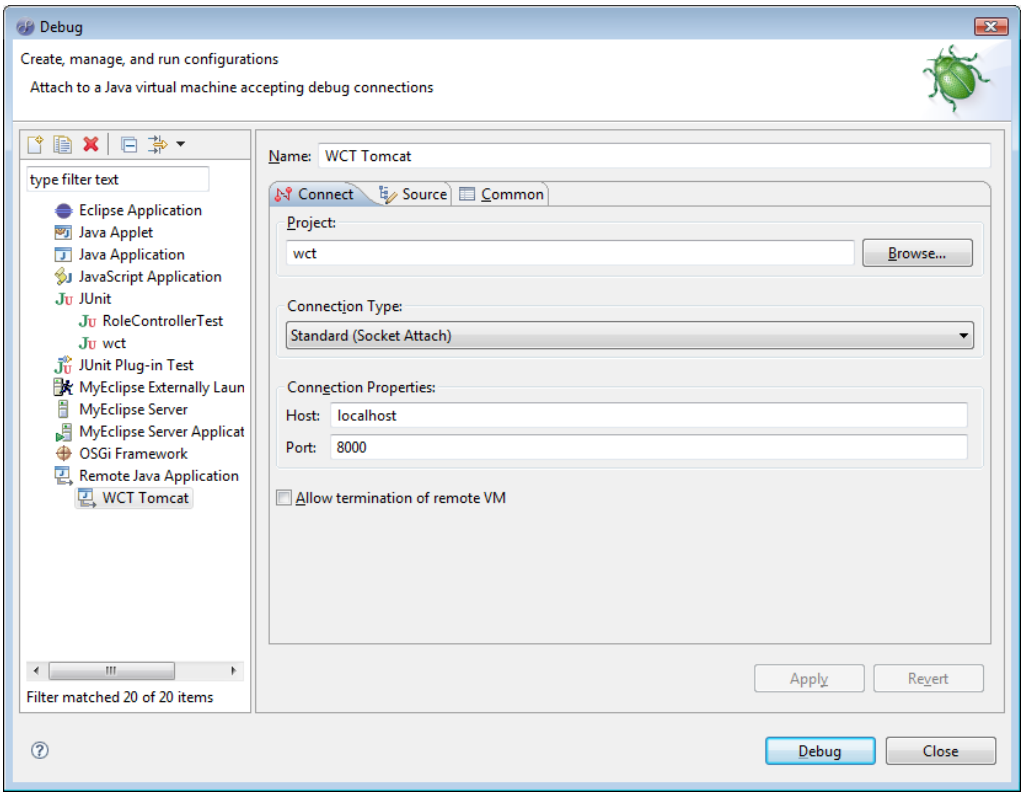
v. If you see red log4j errors in the console window, it is likely that the config files in the target folder have been deleted by Eclipse as part of a clean operation. To correct this, run the WCTCore ant build file from a command prompt, by changing directory to \projects\WCTCore\build and typing 'ant'. Once the build has finished, the config files will be restored, and the output in Eclipse will be correct.

Debugging the WCT running in a local Apache Tomcat instance

This scenario allows for the debugging of the WCT whilst running in a local Tomcat instance. To set up debugging, take the following steps:

i. In the MyEclipse Java Enterprise Workbench perspective, select the Run->Open Debug Dialog... option.

ii. On the left hand side of the dialog that is shown, select 'Remote Java Application'.



iii. Fill the dialog out as shown above – this should mostly be already done.

iv. If you wish to debug a Tomcat instance running on a different machine, enter the machine's IP address or network name in the 'Host' box.

v. Click 'Debug' to debug (Make sure tomcat is running!).

vi. If you have set breakpoints, the debug perspective will open when a breakpoint is hit. Otherwise, you can open the debug perspective manually.

vii. The configuration will be saved in the Debug dialog for future debugging.

3 SourceForge

The Web Curator Tool is an open source application which is managed through the SourceForge online resource. The home page for WCT on SourceForge can be found at http://webcurator.sourceforge.net/. SourceForge provides a number of services to the open source projects that it hosts, including code and documentation version management, software release management and publishing and news and Wiki facilities for communication within the community.

3.1 CVS Software Configuration Management

Concurrent Versions System (CVS) is the open source software configuration management tool used by SourceForge. As this is an open source environment, there can be no 'locking' of files, so due consideration must be given to concurrent development, branching and merging of code.

Access to CVS is either anonymous, in which case it is read only, or signed on as a project developer, in which case code can be modified and checked in.

3.1.1 CVS via Ant Scripts

During the development of the nightly build Ant scripts, there was a requirement to retrieve the latest version of the source code from CVS in order to build it. As there was no requirement to check code in to CVS from the nightly build environment, it was decided to use anonymous access for this purpose.

The Ant tag 'cvs' was used, as shown below:

```
<cvs cvsRoot=":pserver:anonymous@webcurator.cvs.sourceforge.net:/cvsroot/webcurator"
package="WCTCore" tag="BritishLibrary-Jan2008" dest="${basedir}"
output="${basedir}/admin/CVS.${build.log}" error="${basedir}/admin/CVS.${build.log}"
append="true" />
```

Interesting tags here are:

- **cvsRoot** – the protocol (pserver), user and root location of the source code
- **package** – the software package within the project
- **tag** – if the code is branched, entering the branch name here will ensure that the current tip of the required branch is retrieved, rather than the tip of the trunk. This is also useful when fetching a specific version to build a release.

It is worth noting that there are issues with Ant preventing the use of the cvs tag with the ext: (ssh) protocol required by SourceForge for specific user logon and to check code in.

3.1.2 Setting up Tortoise CVS

In order to facilitate user logon to CVS and to ease the checking in and out of code, the Tortoise CVS tool was used by the WCT project. This tool is freely available and can be obtained at http://sourceforge.net/projects/tortoisecvs.

After installing the tool there are a number of steps to follow to get set up on SourceForge.

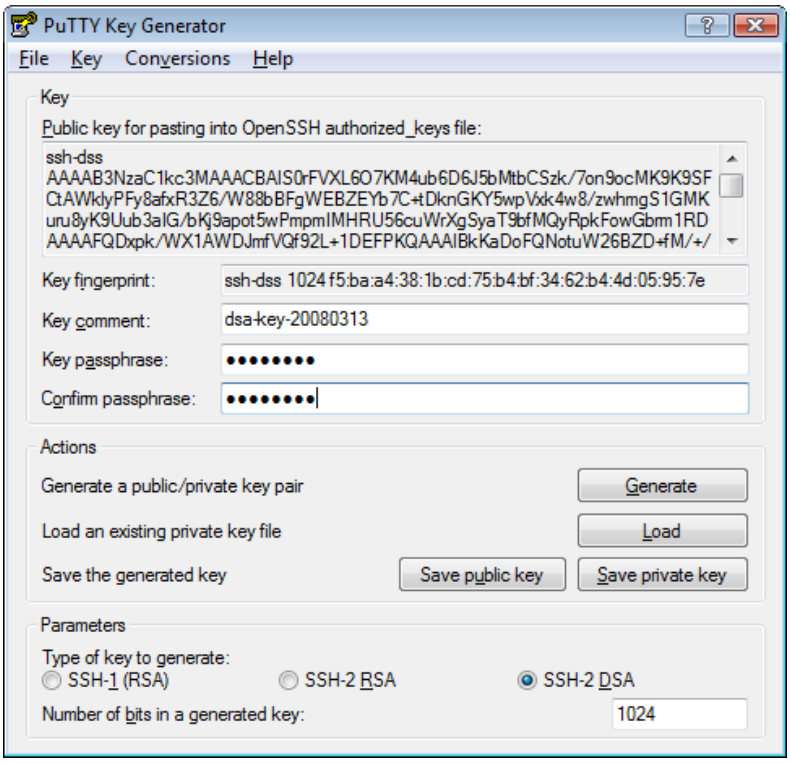
Set up authorised SSH

i. Install PuTTY. PuTTY is a free implementation of Telnet and SSH for Win32 and Unix platforms, along with an `xterm` terminal emulator. The PuTTY authentication agent (Pageant) is used to provide private key authentication over SSH when communicating with CVS on SourceForge. This removes the need for password authentication.

ii. Use PuTTYgen to generate a private key. Make sure that SSH-2 DSA is selected:

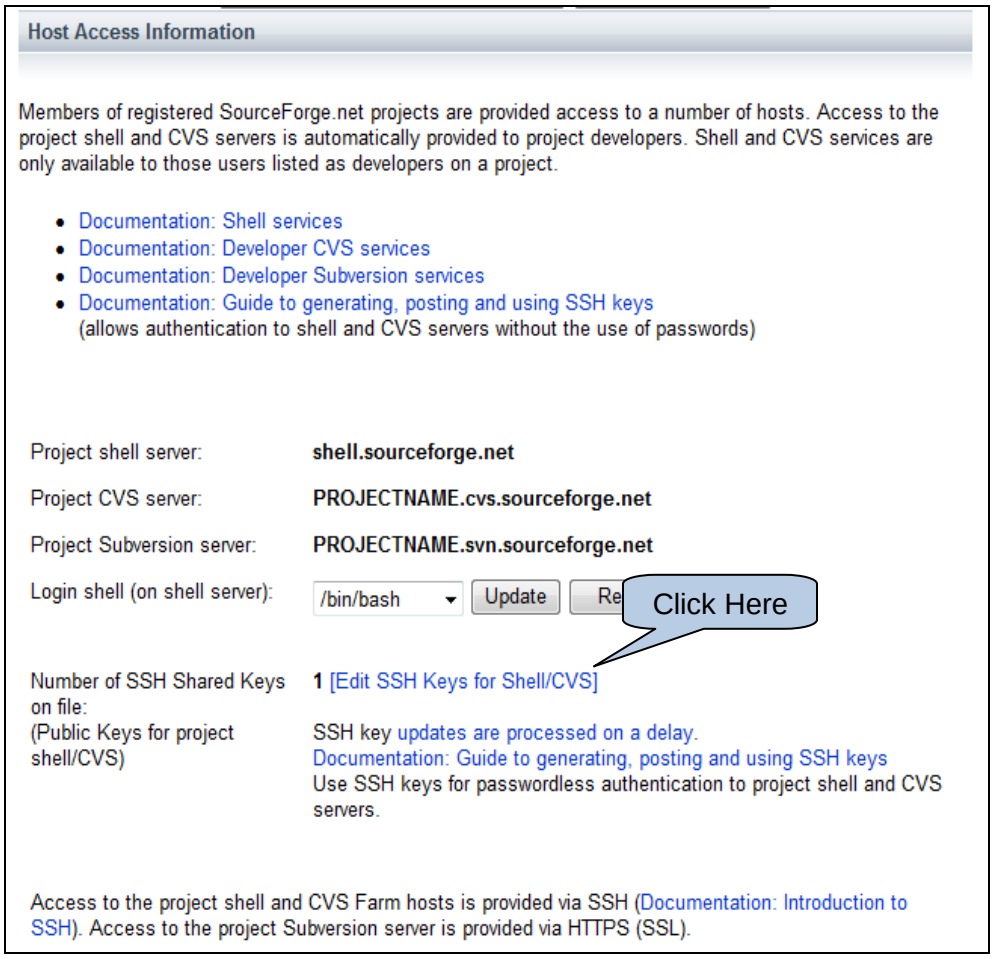
iii. Click ‘Generate’ and follow on screen instructions to generate the key.

iv. Enter a pass phrase.



v. Save the private key in a known location (using a .ppk extension). *Keep PuTTYGen open as the public key will need to be cut and pasted later*.

vi. The next step is to log in to SourceForge and go to your account page which can be found at https://sourceforge.net/account/. Scroll down to the 'Host Access Information' section, and click [Edit SSH Keys for Shell/CVS]:



vii. Paste the public key from PuTTYGen (from the box labelled 'Public key for pasting into OpenSSH authorized_keys file") into the text box provided. Note that there is a delay before the key becomes active.

viii. Finally, run Pageant. Clicking the "Add Key" button allows the private key file saved from PuTTYGen to be selected. Pageant provides authorised SSH using this key.

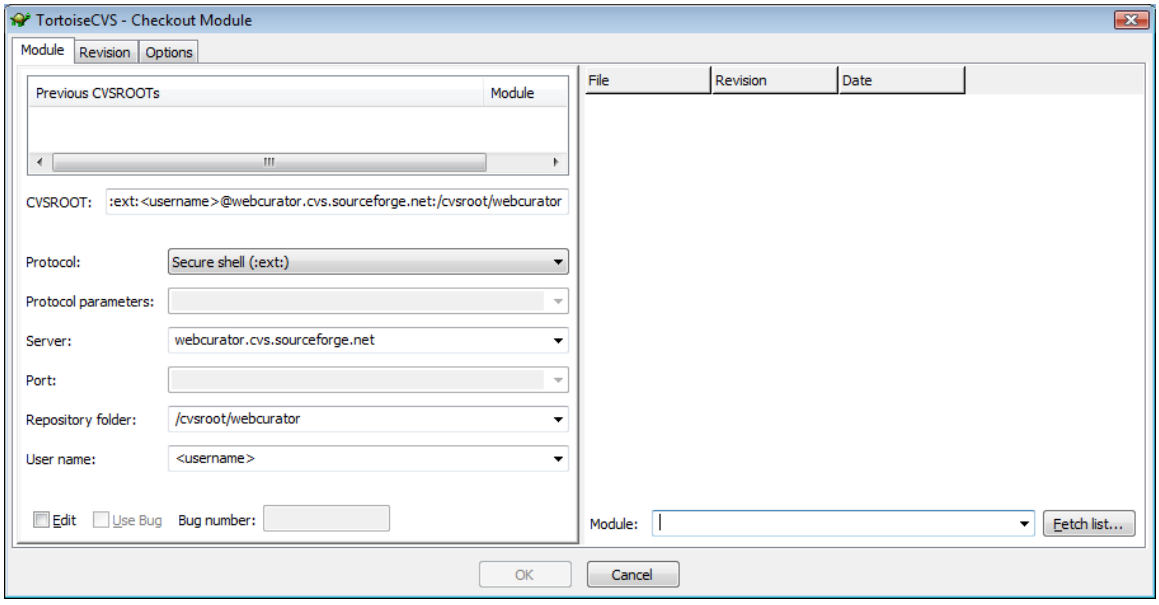
ix. To avoid having to manually run Pageant and add the key in each session, a shortcut can be added to the Windows startup group to run Pageant with the correct key. The command line should be similar to:

```
"C:\Program Files\PuTTY\pageant.exe" "<fullpath>\<keyfile>.ppk"
```

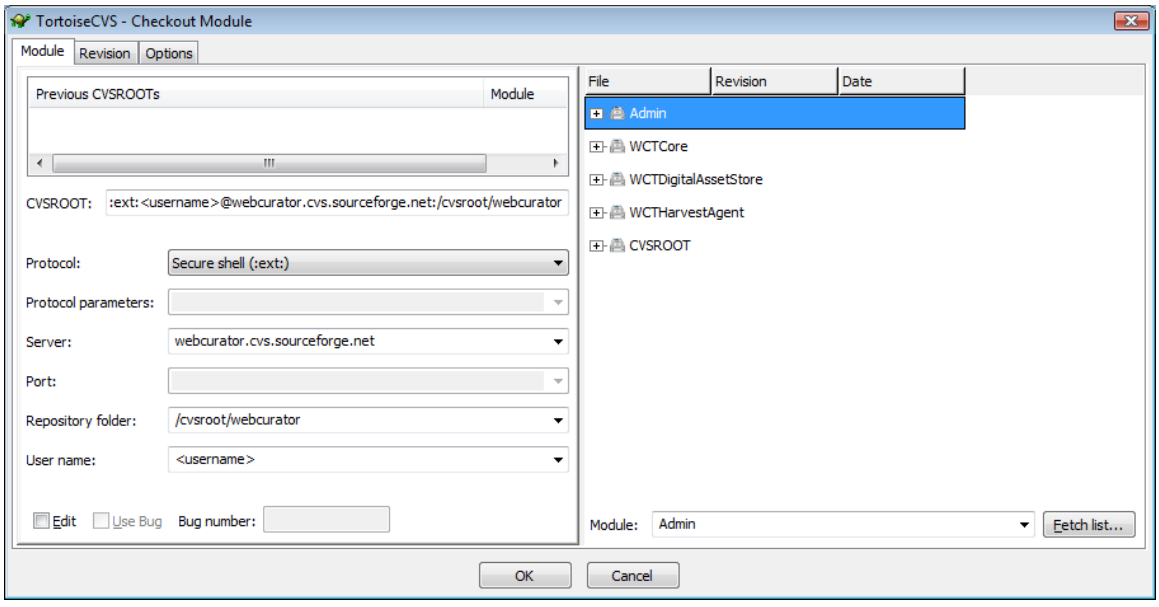
3.1.3 Checking out and Committing Code

Checkout the Source Code

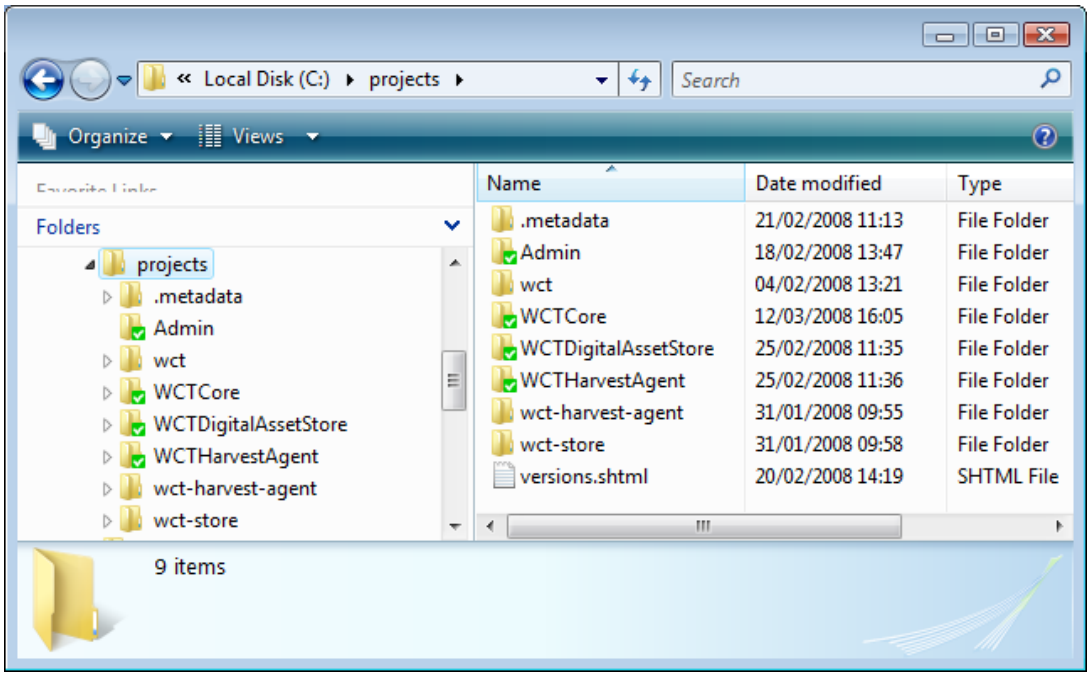
i. Open the '\projects' folder in Windows Explorer. Right click on the right hand pane and choose 'CVS Checkout...'



ii. Enter details as above, replacing <username> with your SourceForge username. Click 'Fetch List...' on the bottom right.



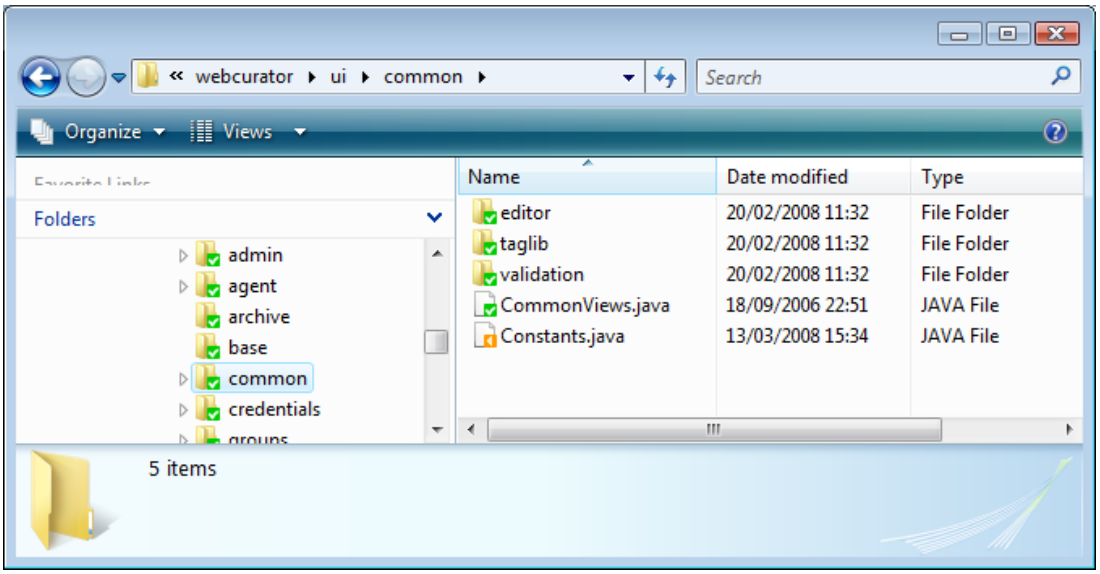
iii. Select 'Admin' and click OK. The Admin folder will be created, and code checked out. Repeat steps (ii) – (iii) for each of WCTCore, WCTDigitalAssetStore and WCTHarvestAgent.



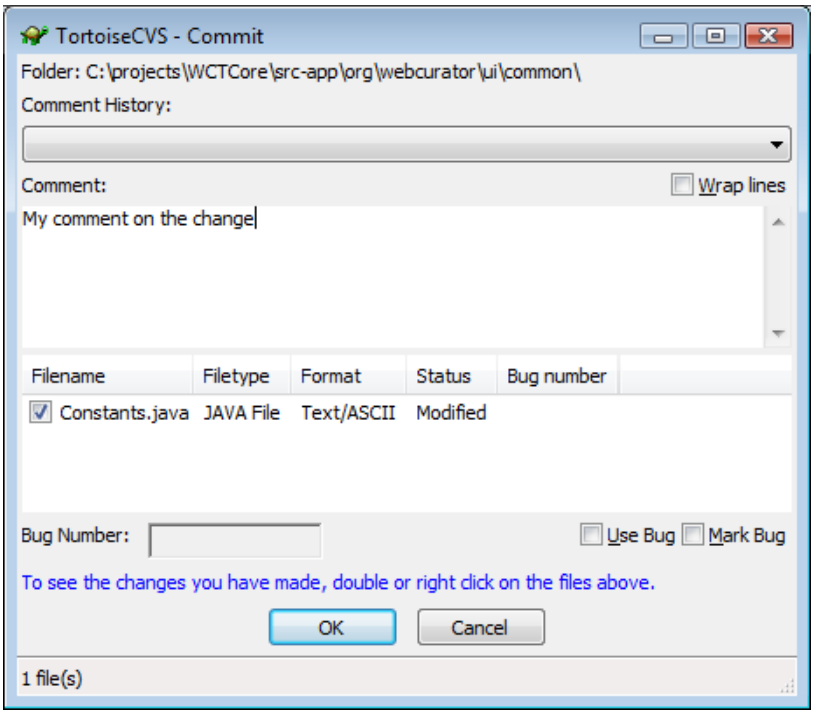
iv. The projects folder should now resemble the one above. The green ticks show folders and files that are checked out from CVS and are unchanged.

v. Note – refreshing the checked out folders to the latest version from CVS can be achieved by right clicking each folder and choosing 'CVS Update'. This will not overwrite any changed files, but will highlight files in need of manual merging.

Committing a modified file



i. The folder view above shows that Constants.java has been modified. To commit this file to CVS, right click and select 'CVS Commit...'



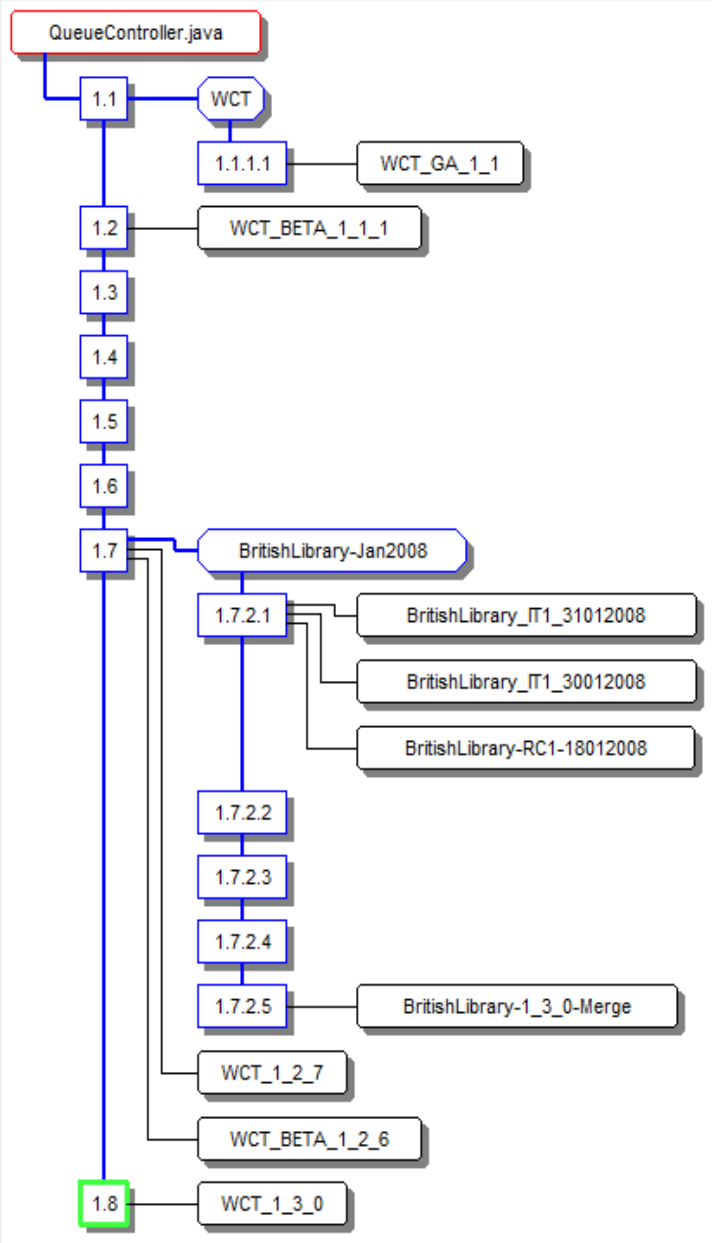
ii. Enter a comment and click OK to commit this change.

iii. To commit multiple files, right click a parent folder for all of the files and choose 'CVS Commit...' as before. Multiple files appear in the bottom list. Check the ones that are to be committed to CVS and enter a single comment to apply to all files. Then click OK to commit these files.

3.1.4 Branching Code

The Web Curator Tool is an open source project. This has an impact on the development of the system in that many organisations and individuals could be modifying the code on simultaneous or overlapping development cycles. Unlike development practice within a single organisation, it is not feasible to 'lock' files to a single user or project while they are under development.

For this reason, branching within CVS is used for development cycles.

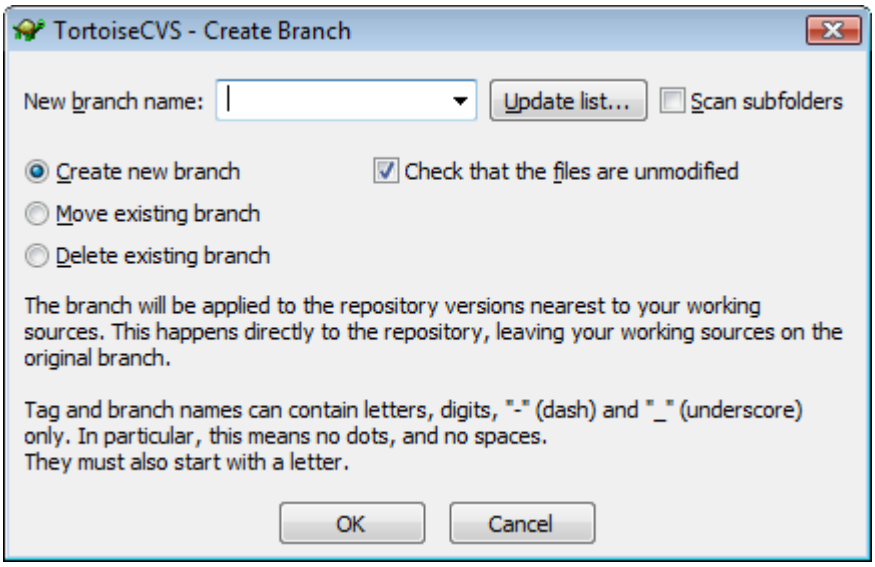


The above diagram shows the revision graph for a single java file, QueueController.java. At the start of development in January 2008, a branch was created in CVS for all source files, called 'BritishLibrary-Jan2008'. Development continued on this branch for each of the iterations, until being merged back into the trunk prior to release. In this case, version 1.7.2.5 was merged into version 1.8.

In order to create a branch, and then begin development on that branch there are a number of (fairly laborious) steps required:

Creating a branch using Tortoise CVS

Once a full checkout of the trunk version of the project has been completed (see section 3.1.3), right click on the root folder (e.g. WCTCore) and choose CVS->Branch...

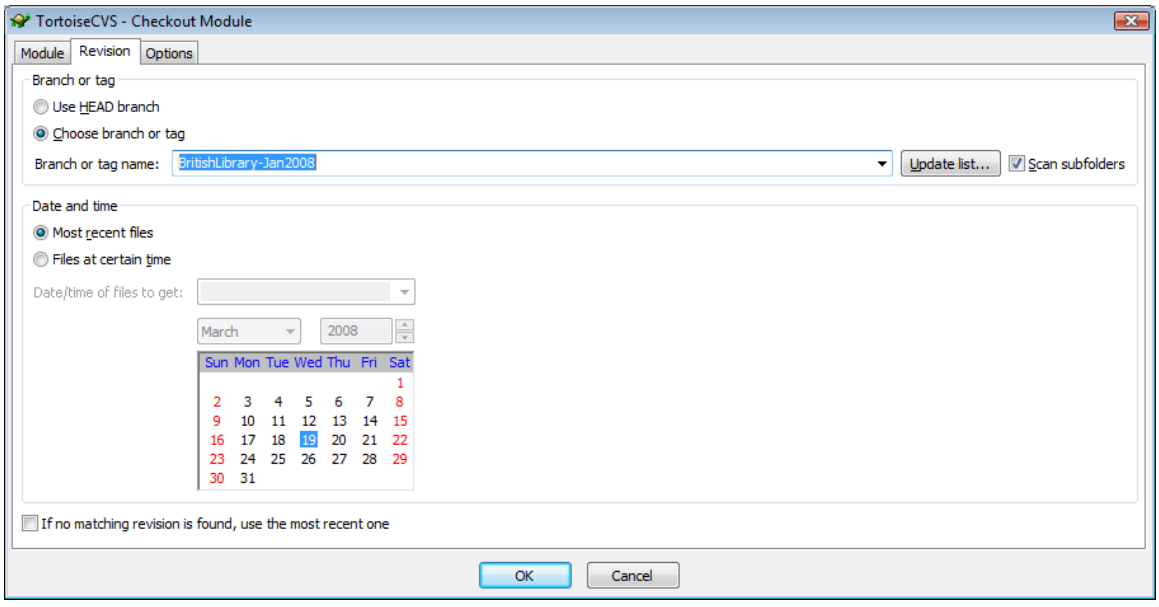


The above dialog is displayed. Enter the new branch name (The labelling convention for the branch is <client>-<developmentStartMonthYear> - e.g. BritishLibrary-Jan2008) and click OK. This creates the new branch in SourceForge CVS, based on the current revision of each file, but leaves the source on the local machine based on the trunk. In other words, *checking out, modifying and committing a file will create a new version on the trunk, not on the new branch*.

Checking out on a branch using Tortoise CVS

The laborious part of the process is that in order to work on the new branch, all current source code needs to be deleted or moved, and a fresh copy checked out on the branch.

Delete the current version of the source code tree and follow the steps detailed in section 3.1.3 to check out a new code tree, only this time, make the following change to the Revision tab on the Checkout Module dialog



Making sure ‘Scan subfolders’ is checked, click ‘Update list...’ to fetch a list of tag and branch names defined in CVS. Select the new branch created in the previous step and click OK. All code is checked out again, only this time, committed revisions will be created on the new branch.

3.1.5 Labelling (Tagging) Code

From time to time during the development process, a need arises to label a version of the source code. This is usually for one of the following reasons:

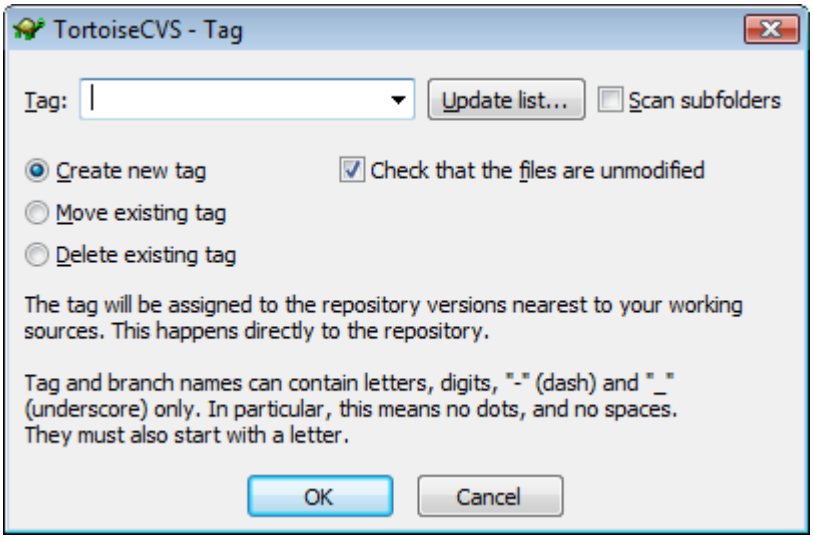
- A release has been created and the release version is labelled.
- A merge is about to take place, and the merge point on the branch is labelled.
- A version of the code is labelled prior to release for system test.
- A version of the code is being labelled to mark the start or end of an iteration.

The labelling convention used for each of these is described in the following table:

Reason	Labelling Convention	Example
Release	WCT_<version>	WCT_1_3_0
Merge Point	<Client>-<version>-Merge	BritishLibrary-1_3_0-Merge
System Test (Release Candidate)	<Client>-RC<candidateNumber>-<ddMMyyyy>	BritishLibrary-RC1-18012008
Iteration Version	<Client>-IT<iterationNumber>-<ddMMyyyy>	BritishLibrary-IT1-31012008

Labelling using Tortoise CVS

Right click on the root folder (e.g. WCTCore) and choose CVS->Tag...



The above dialog is displayed. Enter the new tag name according to the labelling convention in the table above, and click OK.

3.1.6 Merging Code

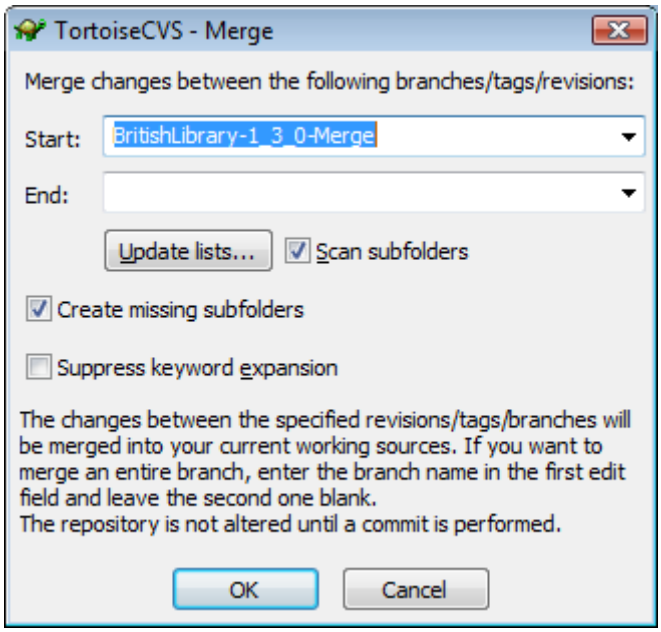
When a release is being prepared, it becomes necessary to merge the source code on the working branch back onto the trunk ready for labelling and building the distribution. The merge operation using CVS is carried out using two folders containing different versions of the source code. The following steps detail the procedure; the starting point is that the \projects folder contains code folders based on the branched version of the code:

i. Label the current branched version of the code as described in section 3.1.5.

ii. Create a \projects\HEAD folder.

iii. Within the newly created HEAD folder, check out the trunk (HEAD) version of the code following the steps defined in section 3.1.3.

iv. On each of the root source folders (e.g. WCTCore) within the HEAD folder, right click and choose CVS->Merge...



v. In the 'Start' field of the above dialog, select the merge tag name created in step (i). It may be necessary to click 'Update lists...' with 'Scan subfolders' checked to see the label in the drop list. Click OK.

vi. Tortoise CVS merges all changes made on the selected branch into the code in the HEAD folder. Note that this does not modify SourceForge CVS in any way until a commit is issued on the changed files.

vii. Any merge conflicts are written to the files, which are highlighted by Tortoise CVS as requiring manual merge. The original HEAD version is copied to a file named '.#<filename>.<version>' (e.g. '.#AddParentsController.java.1.1'), and the original branch version is still in the \projects folder.

viii. Complete any manual merging, so that the version to be checked in for release is in a subfolder of the HEAD folder.

ix. On each of the root source folders (e.g. WCTCore) within the HEAD folder, right click and choose CVS Commit... and commit the changed files with an appropriate comment as normal.

x. Label the release in the HEAD directory as described in section 3.1.5.

3.2 Website and Software Releases

After the final changes have been made, and the release merged and labelled in SourceForge CVS, the release of WCT can be packaged into the distributable files and published on the Web Curator Tool website on SourceForge.

3.2.1 Creating Release Packages in WCT

The first job for publishing the release is to package the binaries and source files into the release distribution files.

Four files are released, a .zip and a .tar.gz version of each of the binary and source code distribution files.

The easiest way to build the release distribution files is to use the specialised Ant build files within the WCTCore directory structure. The binary distribution files are built using \projects\WCTCore\build-package\build.xml and the source distribution files are built using \projects\WCTCore\build-package-src\build.xml.

Release specific documentation filenames such as Web Curator Tool Data Dictionary (WCT 1.3).doc need to be included in the release, and some of the files from the previous release need to be excluded. It is a good idea to make these modifications to the build.xml files to allow easy re-packaging of the release, although it is also possible to manually modify the contents of the generated target-package and target-package-src folders and re-run the *zip-compress* and *tar* targets respectively.

3.2.2 Creating/Uploading a Release

Creating a release, and uploading files to SourceForge is a fairly convoluted process. The steps below detail this procedure and identify some of the pitfalls:

i. The first step is to upload the release files to SourceForge using anonymous ftp to *upload.sourceforge.net* in the *incoming* directory. This is a shared location and the following rules apply:

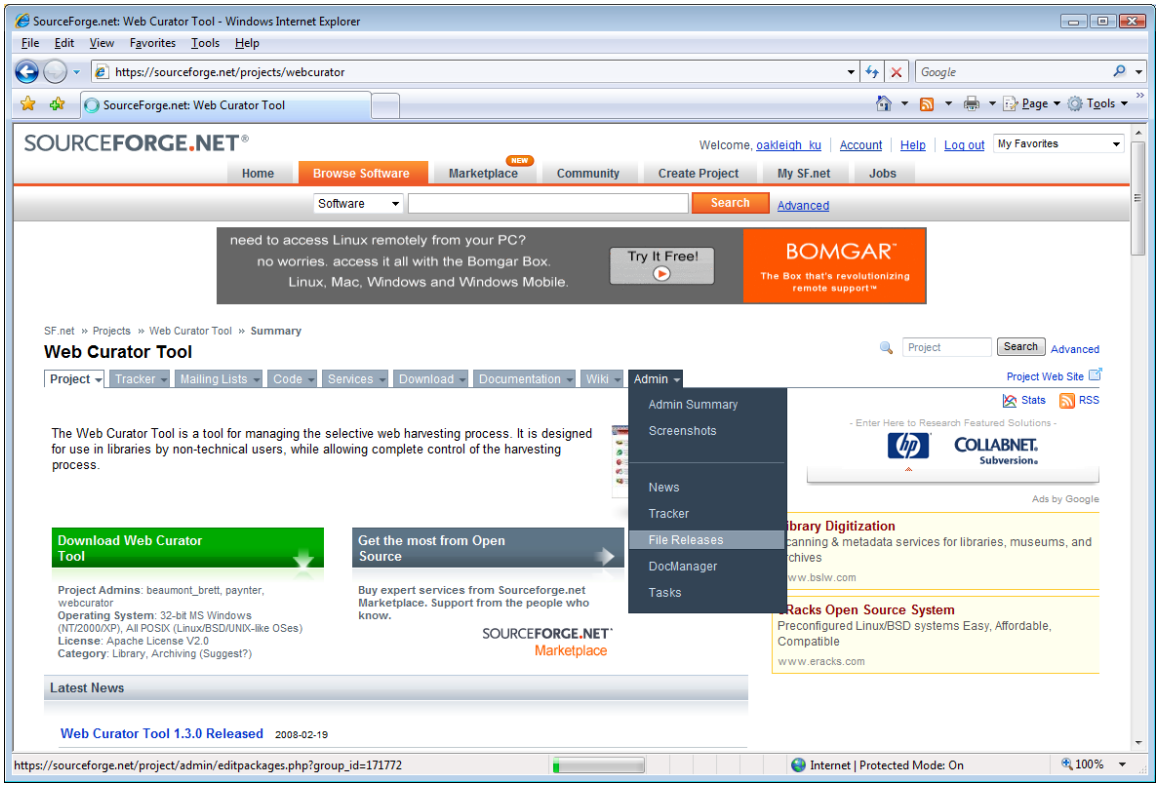
    a. Once uploaded, files or directory listings cannot be viewed or deleted. Files are automatically removed after 24 hours.

    b. Files cannot be overwritten, so if there was a problem with the upload, or if the wrong version of a file was mistakenly uploaded, then a file with that filename cannot be uploaded again for 24 hours.

    The best approach is to use a specialised FTP package such as FileZilla, and queue the files to uploaded one by one. A high bandwidth link is best for upload as it was found that home broadband links often dropped out, causing either a change to the filename or a delay of 24 hours. The optimal settings for upload are specified below:

- Connect to upload.sourceforge.net.
- Login using 'anonymous' for the username and your email address for the password.
- Switch to binary mode.
- Switch to passive mode, if non-passive mode doesn't work.
- Change local directory to the directory containing the files to upload.
- Change remote directory to /incoming.
- Allow only sequential uploads. Filenames may only contain alphanumeric characters, periods, dashes or underscores.

ii. Once the files have been uploaded, log in to the SourceForge site, and click on the Web Curator Tool project on your Personal Page.



iii. On the Admin tab, click on ‘File Releases’. When the page displays, click on the ‘Add Release’ link.

iv. Give the release a name (1.3GA means 1.3 General Availability) and click ‘Create this release’.

v. Upload or cut/paste the release notes and change log information.

Step 2: Add Files To This Release

Next, choose your files from the list below. Choose **ONLY YOUR** files. If you choose someone else's files, they will not be able to access them and they will be rightfully upset.

You can upload new files using **anonymous** FTP to **upload.sourceforge.net** in the **incoming** directory. When you are done uploading, just hit the refresh button to see the new files. Further information regarding this process may be found in our Guide to the File Release System; please refer to this document if you encounter any difficulties in performing this file release.

PLEASE NOTE: Filenames may only contain alphanumeric characters, the period ("."), dash ("-"), or the underscore character ("_"). This means that a space, parenthesis, or other commonly used characters, such as "~", "&", or "#" will cause the file to be rejected.

- Asus3.vmx
- MillenniumBSA-installer-community-4.0.5GA.jar

- Skim-1.1.dmg
- camba-2.1.0.20080319-linux-complete.zip
- carvfs-0.4.0.tar.gz
- eGroupWare-gallery-1.4.003.tar.bz2
- eclipse_office_0_1_12.zip
- etlclient-0.9.7-bin.zip
- etlserver-0.9.7-bin.zip
- gnuplothelper-0_0beta.tar.gz
- gtk-cffi.tar.bz2
- jboss-4.2.2.GA_josso-1.7_console.tar.gz
- jboss-4.2.2.GA_josso-1.7_console.zip
- libcarvpath-0.2.0.tar.gz
- oscmembership_46.tar.gz
- pantheios-1.0.1-beta110.md5
- pantheios-1.0.1-beta110.zip
- pi4soa1.7.0-eclipse3.3.2-linux-gtk.tar.gz
- pi4soa1.7.0-eclipse3.3.2-linux-gtk2.tar.gz
- pi4soa1.7.0-eclipse3.3.2-linux.tar.gz
- rubikprogram-080319-103542.tar.bz2
- ssax-sxml-bigloo-071202.tgz
- ssax-sxml-bigloo-071203.tgz
- tsk-cp-0.4.1.tar.gz
- xtuple-Build1713-Linux.tar.bz2
- xtuple-Build1713-Windows.zip

Add Files and/or Refresh View

vi. Select the uploaded files from the list in ‘Step 2: Add Files To This Release’. If the page was opened before the files were uploaded, click ‘Add Files and/or Refresh View’ to see them. Once all the release files have been selected, click ‘Add Files and/or Refresh View’ to add them to the release.

Step 3: Edit Files In This Release

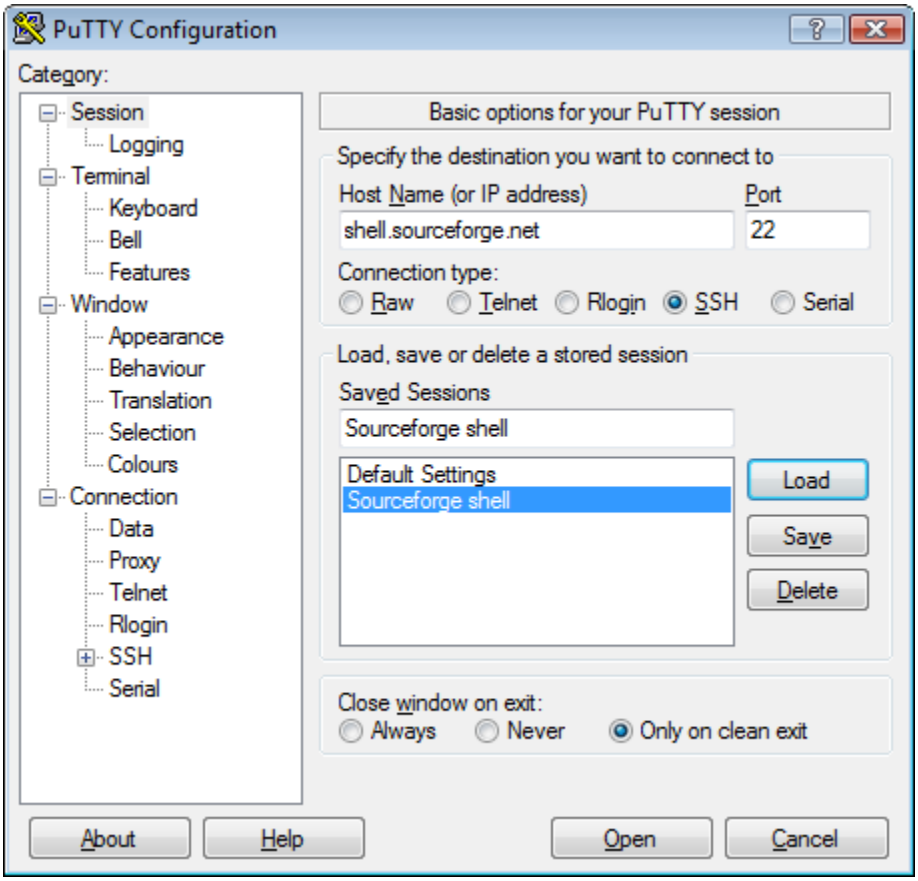
Once you have added files to this release you **must** update each of these files with the correct information or they will not appear on your download summary page.

Filename Release	Processor Release Date	File Type Update
wct_package_bin_v1.3.0.tar.gz	Platform-Independent	.gz
webcurator : 1.3GA	2008-02-28	Update/Refresh
		Delete File I'm Sure
wct_package_bin_v1.3.0.zip	Platform-Independent	.zip
webcurator : 1.3GA	2008-02-28	Update/Refresh
		Delete File I'm Sure
wct_package_src_v1.3.0.tar.gz	Platform-Independent	Source .gz
webcurator : 1.3GA	2008-02-28	Update/Refresh
		Delete File I'm Sure
wct_package_src_v1.3.0.zip	Platform-Independent	Source .zip
webcurator : 1.3GA	2008-02-28	Update/Refresh
		Delete File I'm Sure

vii. Finally, update step 3 as in the above diagram, clicking 'Update/Refresh' for each row.

3.2.3 Editing the SourceForge Webcurator Website

Once the release has been created, the SourceForge Web Curator website can be edited to include the release.



To connect to the SourceForge shell, use PuTTY to establish an SSH session to *shell.sourceforge.net*.

Login using your SourceForge user id. If the Pageant SSH daemon is running, the system should not require a password.

Change to the html documents folder using:

cd /home/groups/w/we/Webcurator/htdocs

All content of the website is available for editing in this folder, and its subfolders. After a release, the following should be changed or updated:

- Update the latest news item ‘latest-news-fragment.shtml’ to advertise the new release.
- Add a new folder under the htdocs/docs folder to include the pdf documentation for the release.

The PuTTY secure copy client can be used at the command line to copy files locally for editing, and to copy them back afterwards. This is run from a command prompt on the local machine with the following syntax:

To copy from SourceForge shell to the local machine

pscp <username>@shell.sourceforge.net:/home/groups/w/we/Webcurator/htdocs/remotefilename localfilename

To copy modified files back to SourceForge shell

pscp localfilename <username>@shell.sourceforge.net:/home/groups/w/we/Webcurator/htdocs/remotefilename

4 Unit Testing Approach

During the first iteration of Oakleigh's work on the WCT project, test first programming based on JUnit unit testing was introduced. The JUnit 4.4 based unit tests are developed under a *src-test* source folder and are run with every nightly build (see section 2.4.1).

4.1 Base Class

All new unit tests are based on the org.webcurator.test.BaseWCTTest base class. This class uses generics to provide the framework provided by JUnit 4.4.

Each test class should be defined in a similar way to the class that follows:

```
public class GroupSearchControllerTest extends
                                        BaseWCTTest<GroupSearchController>{

      public GroupSearchControllerTest()
      {
            super(GroupSearchController.class,
                  "/projects/WCTCore/src-
test/org/webcurator/ui/groups/controller/groupsearchcontrollertest.xml");
      }

      @Test
      public final void testGroupSearchController() {
            assertTrue(testInstance != null);
      }

      @Test
      public final void test2() {
      }
}
```

Key points about the BaseWCTTest base class:

- The file path passed into the constructor is the path to the xml sample data file explained in section 4.3.
- *testInstance* is the instance of the class under test. A new instance of *testInstance* is allocated prior to the execution of each test method (annotated with `@Test`).
- The JUnit `@BeforeClass` method is *BaseWCTTest.initialise()*. This method is called at the start of all tests for the current class and is static (i.e. called prior to instantiation of the class). The base class implementation sets the application context and creates a current logged in user based on the definition in */projects/WCTCore/src-test/org/Webcurator/test/BaseWCTTest.xml*. Note that it is a good idea to have the same user defined in the xml file for the test class. This method can be overridden, but a call to *super.initialise()* should be made unless the functionality in the base class is not needed or is being replaced.

- The JUnit `@AfterClass` method is *BaseWCTTest.terminate()*. This method is called after all tests for the current class have been executed and is static (i.e. called after destruction of the class). The base class implementation simply writes out log messages to confirm that all tests are complete. This method can be overridden, but a call to *super.terminate()* should be made at the end of the overridden method to write out the log messages.
- The JUnit `@Before` method is *BaseWCTTest.setup()*. This is an instance method which is called before each test for the current class is executed. The base class implementation writes out log messages to identify the test being performed, and instantiates a new instance of *testInstance*. This method can be overridden to provide additional initialisation prior to each test, but a call to super.setup() should be made at the start of the overridden method to write out the log messages and to instantiate *testInstance*.
- The JUnit `@After` method is *BaseWCTTest.tearDown()*. This is an instance method which is called after each test for the current class is executed. The base class implementation simply sets *testInstance* to null to allow garbage collection. This method can be overridden to provide additional teardown after each test, but a call to super.tearDown() should be made at the end of the overridden method to write set *testInstance* to null.
- Four methods on the BaseWCTTest class are provided to allow privileges to be assigned or removed from the current user as tests are executed. These are:
    - **protected void** addCurrentUserPrivilege(String privilege)
    - **protected void** addCurrentUserPrivilege(**int** scope, String privilege)
    - **protected void** removeCurrentUserPrivilege(**int** scope, String privilege)
    - **protected void** removeAllCurrentUserPrivileges()

  See the associated JavaDoc for more information on these methods.

4.2 Mock Objects

In order to facilitate unit testing without the need to implement a database or HTTP stack for each test, a number of *mock objects* are employed. These fall broadly into four categories:

i. Spring mock objects such as MockHttpServletRequest etc. (see section 2.2.3).

ii. Mock manager classes such as MockSiteManagerImpl.java. These classes basically override the constructor of the standard manager class to initialise sub objects that would normally be initialised using Spring dependency injection.

iii. Mock DAO classes such as MockSiteDAO.java. These classes provide a DAO (Data Access Object) implementation based on the xml data files (see section 4.3) rather than using an underlying database via hibernate.

iv. Mock utility classes such as MockMessageSource.java and TestAuditor.java.

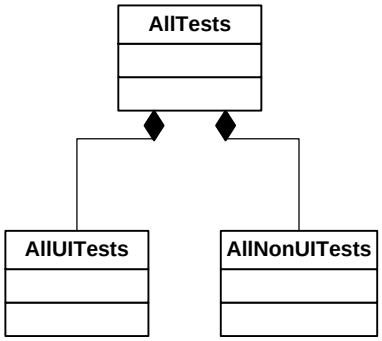
4.3 Xml Data Files

Each unit test should be provided with an xml data file. The file contains the state of the business entities which are provided during the test by the mock DAO objects (see section 4.2 iii).

Although a formal schema has not yet been defined for these files, an implied schema is determined by the code in the mock DAO classes used to load the information from the files into a series of key–value pair arrays. Each element in the xml file which represents an entity has an *id* attribute. This is the *oid* (object id) for the entity. Where the entity is defined with child nodes, this is the definition of the entity. Where the entity is specified with no child nodes, it represents a reference to the entity. It is important that no entity has more than one definition within the file.

4.4 Test Suites

Unit tests under JUnit 4.4 can be grouped into logical suites of tests. The WCT test suites are defined in *org.webcurator.test* and have the following hierarchy:



The AllTests class is called by the nightly build script (see section 2.4.1) and all defined tests should be called from either AllUITests or AllNonUITests. These are not added automatically, and it is the developer's responsibility to add their new test class to these parent classes once they are happy that the tests will pass.