Test Results Analyzer Plugin Extension

CS-427 Final Project

Team Spartacus

December 2015

Contents

1 Introduction **2**
1.1 Project Goals 2
1.2 Test Results Analyzer Plugin 2

2 Architecture Background: Test Results Analyzer Plugin **3**
2.1 Backend 3
2.2 Frontend 3

3 Obtaining Source Code **4**

4 Defined User Stories **4**

5 Design & Implementation **5**
5.1 Frontend Tables 5
5.2 Backend Data 5
5.3 Filtering 6
5.4 Extra 8

6 Usage **9**

1 Introduction

1.1 Project Goals

The Test Results Analyzer Plugin extension was completed for the final assignment of course CS427-Software Engineering at University of Illinois Urbana Champaign. The purpose was to modify/extend an existing Jenkins plugin using the "Extreme Programming" software development methodology as a team of 8 people. More specifically the selected project goal was defined as follows:

> Create/improve a dashboard to track individual test failures/passes over time. Jenkins currently shows an aggregate view of all the tests in a project, in terms of number of passing/failing tests. The aim in this project is to create a finer-grained view whereby developers may track the history of a single test over time to show its pass/fail history.

1.2 Test Results Analyzer Plugin

The "Test Results Analyzer Plugin" was chosen as the starting point for the final project. The plugin is avilable here:

`https://wiki.jenkins-ci.org/display/JENKINS/Test+Results+Analyzer+Plugin`

The Test Results Analyzer Plugin is a Jenkins plugin that shows history of test execution results in a tabular format. Specifically the plugin works with freestyle projects using JUnit tests.

Chart	See children	Build Number ⇒ Package-Class-Testmethod names ⇓	16	15	14	13	12	11	10	9	8	7	6	5
		org.common.samplea	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
		SampleATest	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
		testA	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
		testB	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
		testC	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
		testD	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
		org.common.sampleb	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	FAILED	N/A
		org.common.samplec	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
		SampleDTest	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
		testA	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A
		testB	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	N/A
		testC	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	N/A
		testD	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	N/A

Figure 1: Test Results Analyzer Plugin default output.

Reasons for selecting the plugin as the base for extension were as follows:

- The plugin appeared simple.
- Team Spartacus was already familiar with Junit.
- The popularity of the plugin was rising.

2 Architecture Background: Test Results Analyzer Plugin

2.1 Backend

Representing Data
Jenkins source represents an abstract "result" with a class called CaseResult. The test results analyzer plugin uses Jenkins to retrieve all of the CaseResults for every build of the project. These case results are then read into a custom backend structure. The method by which the plugin creates this custom backend is a bit convoluted. There is an abstract class called ResultData which holds all the data for any type of result as fields, with getters and setters to modify said fields. This abstract class is constructed by sublcasses for each of the result data types (plugin, class, test). The constructor of each sublcass simply populates the fields of the abstract class.

Retrieving the Data
The frontend of the plugin is primarily javascript, limiting communication between the frontend and backend. The custom backend structure has methods to represent the data in a universal JSON format acceptable by the frontend. The abstract class (ResultData) contains the methods for retrieving the data as a JSON object, but since different subtypes populate different data fields the method is verbose. Retrieving the JSON this way creates JSON objects with many key-value pairs that have null values.

2.2 Frontend

.jelly Files
The plugin's main HTML page is setup with a .jelly file. Jelly is an open and customizable XML processing engine, and in this case the .jelly file can be treated almost like a glorified HTML file. Notably, the jelly file imports necessary javascript files, defines the main layout for the HTML page of the plugin (ie: placement of buttons and major divs etc.), and sets up any button actions/function calls for user input.

Handlebars
By default the Test Results Analyzer Plugin has only one major table for visualizing test history. The .jelly file specifies a locator div for this main history chart, but on load this div is unpopulated. The plugin works by creating an HTML table and inserting it into the div. To ease the process of populating a table with variable data, the plugin uses handlebars to create the dynamic HTML. Handlebars is a minimal HTML templating system that makes it very easy to define variables in an HTML template and then link a context to populate those variables with data. The template looks like regular HTML with some embedded handlebars expressions. The context for the variable data can be literal values or a JSON object. When additional processing is required for a data insertion, handlebars supports custom helper functions to modify the input before inserting in the template.

Any desired tables for the final plugin page are created as handlebar templates and supplied a context via a JSON object containing all the necessary data from the backend. After the handlebars templates are linked with their context, raw HTML is generated for the tables and inserted at the appropriate location in the main HTML page (defined by the .jelly file).

All other frontend processing is handled with javascript.

3 Obtaining Source Code

The entirety of the code written for this project can be found at the following repository:

https://subversion.ews.illinois.edu/svn/fa15-cs427/_projects/Spartacus/tags/FinalSubmission/

4 Defined User Stories

The following user stories were drafted at the beginning of the XP process.

User Story Title	#	Description
View Specific Test Info	1	Users can click on a result (package, class, test) and see detailed information about it.
View Stability Statistic	3	Users can see a stability statistic (pass ratio) for any result given the selected builds.
View Builder ID	4	Users can see the Jenkin's user id that initiated a build.
View Assertion Console Output	5	The assertion output (console output of expected/actual) of a failed test is displayed.
View Exception Type	6	The exception type of a failed test is displayed.
Filter by Build #	7	Users can enter the specific build numbers or a range of build numbers so that only those builds will be displayed in the table.
Sort By Stability Statistic	8	Users can sort the main history table by the stability statistic (pass ratio).
Filter by Builder ID	9	Users can type a csv of jenkins' or svn usernames and only builds associated with those usernames will be displayed.
Filter by Exception Type	10	Users can select a specific exception type from a list of exceptions types. Only builds that have a test which fails with that exception type will be displayed.
View Coverage Information	11	Users can view a table of all the Cobertura coverage information for the displayed builds.
View Autobuild SVN Commit Username	13	If the build was triggered by an SCM poll, the table will display the svn username that had committed a change.

5 Design & Implementation

5.1 Frontend Tables

Result Details Table
Goal: To be able to click on a result (row of the main table) and see detailed information about it. This should be displayed in a new table which presents detailed information about the result (package, class, test) for all selected builds #'s.

Implementation Overview: A separate div for the table was defined in the .jelly file for the main HTML page. A custom handlebars template was added to define the layout for the new table (which varies depending on the hierarchy level of the result type). An on-click for the rows of the main chart was added. The invoked function populates the new template using the correct JSON object for the clicked result and then generates the HTML for the table and replaces any HTML currently in the locator div for the new table.

Special Notes: The context for the details table is a subtree of the main JSON object for the entire history table. When retrieving this subtree, it must be pulled from the existing HTML as a string and then built back into a JSON object for the context of the new table. For some reason the retrieved JSON string has some unnecessary double quotes before and after any brackets. A custom workaround involved a simple function called "fixJsonStringify" which removed the double quotes around the brackets such that the string could be successfully converted to a JSON object.

5.2 Backend Data

Builder Name
Goal: To see the Jenkins user ID and/or SVN user ID that initiated the build

Implementation Overview: A list of causes can be pulled from project builds called in getJsonLoadData. The retrieved causes (which are only caused when a user runs a build from Jenkins), are parsed to return the list of Jenkins users that caused the run. If the list is empty, the user is subbed with anonymous. An alternate method was used to extract the SVN commit ID from the build. Formatting functions were added in order to flag the username's origin as either SVN or Jenkins. These functions take the Jenkins usernames and SVN usernames and automatically formats the final output for the frontend to parse. These methods include addSuffixes() and userNameFormat().

Special Notes: We did many refactorings to make the code cleaner, less repetitious, and more robust such as adding formatting methods and a mapUsersToBuild method. Because the getJsonLoadData() method is only called at run-time, it was difficult to test the method from the frontend. Instead, dummy instances of causes are used to test getUsersFromCauseList(). The tests for addSuffixes, userNameFormat, and mapUsersToBuild were easy.

Assertion Output
Goal: To view the assertion details (console output of expected/actual) for a failed test.

Implementation Overview: There are two main functions. One to retrieve the expected value and another to retrieve the actual value using java pattern and matcher classes. When a test failed, the jenkins stack trace includes the expected and actual values. After using the matcher class, assertion detail information is retrieved.

Special Notes: We were not familiar with pattern class, so we had to use online resources to learn about java pattern class.

Exception Type
Goal: To see the Exception Type of any failed tests.

Implementation Overview: The exception type is retrieved through the stack trace originally obtained from TestCaseResultData.java. The first line of the stack trace conveniently contains the exception type. The stack trace is stripped of any any assertion messages only meaningful for the display of assertion output, leaving just the exception type.

Special Notes: Because the exception type only pertains to the "test" result level, the details table template was modified to only include exception type data if the result was a test.

Pass Ratio
Goal: To calculate a stability statistic (in the form of a pass ratio) of all tests based off the currently selected build #'s.

Implementation Overview: For each result at any hierarchy level (package,class, test), the implementation adds up the number of builds that passed and divides by the total number of builds considered. This produces a ratio of $\frac{passed}{total}$. The ratio is inserted into the JSON object passed to the front end, once per hierarchy level. Ie: the ratio is stored with a result, not with a build. Think of a result as a row in the main table, and a build as a column.

Special Notes: Because the ratio depends on the total number of builds being considered, the calculation must be performed at runtime and not stored as a constant value in the backend. The code gets convoluted in how it retrieves the JSON object for the specified builds, but the main entry point is getJsTree in JsTreeUtil.java. This method creates a giant json object representing all the builds. A second method called createJSON, from the same class, then extracts from that object only the builds specified. Therefore the acquisition of the backend JSON data occurs before any calculation could realistically use info about the selected builds. As a workaround, the calculation was performed in this createJSON function. This is accomplished by iterating through the considered builds and tallying those that passed. After the specified builds have been extracted, the statistic for that result is calculated and inserted into the JSON.

5.3 Filtering

Filter the displayed builds by User ID
Goal: To filter the viewable tests by jenkins' or svn user ID that triggered the test.

Implementation Overview: The original plan revolved around filtering out the usernames at the highest level, before even creating the data structure associated with a specific build. However, this function was being executed before any input was even provided by the user. As a result, it was necessary to make use of a HashMap that associated all usernames with build numbers they had triggered.

In order to get the user input, a text field was added and the input was passed to the plugin's backend. This text field supports a comma separated list of usernames which a custom function then parses into a list of usernames. With the HashMap of usernames to build numbers it was simple to find the list of allowed builds and intersect that with any other filters.

Special Notes: We wish we knew there can be multiple users or svn commits for a single build. Once we learned this, we had to go back and extend our code to display multiple users for a single build.

Filter the displayed builds by Exception Type
Goal: To filter the builds based on Exception Type.

Implementation Overview: Data related to the builds and the results of tests are organized in a hierarchical model. At the top of the hierarchy are the packages, followed by the classes under each package, ending with the individual test cases within each class. The plugin makes use of a series of Info objects, one for each level of the hierarchy: PackageInfo, ClassInfo, and TestCaseInfo. Excluding the TestCaseInfo objects, each hierarchy uses a Map to store elements of the level below it. For example a PackageInfo object has a Map to store all the ClassInfo objects related to it. Likewise, a ClassInfo object stores all the TestCaseInfo objects related to it. Information pertaining to the Exception that occurred is stored at the lowest level: TestCaseInfo. The original plan was to iterate through each Map to reach that level, and then note the builds in which the desired Exception occurred. Instead a class was created with a static Map object to map each Exception Type to a set of build numbers associated with it. Each time a project runs a new build, all the Info objects are reconstructed. At the construction of each TestCaseInfo object, both the build number and the Exception Type data are available. Code was added to the constructor of each TestCaseInfo object that would add that data to the static Map. Not only does this provide a mapping from each Exception Type to the set of associated builds, but it also tells which Exception Types occurred in the project. This information was used to make a dropdown menu on the frontend for users to select Exception Types rather than type them into a dialogue box.

Special Notes: We had some issues where the static Map containing information about the Exception types was carrying over to other projects. This was because each project had a TestAnalyzerAction object associated with it, but there was only one Map being used for all projects. The Map was populated each time a project was built, so it would only contain data on the most recently built project. To solve this, we added a private Map to TestAnalyzerAction for storing Exception Type data. Our static map was used to accumulate data for all builds for a project. We then made a deep copy to assign to the private Map of the TestAnalyzerAction object.

Sorting the Main Table by Pass Ratio
Goal: Filter the main history chart such that the displayed tests under each class were arranged in the order of increasing Pass Ratio.

Implementation Overview: All other sorting operations were meant to operate on the separate details table. The pass ratio sorting operation was uniquely targeting the main history chart. For the main chart this involves sorting the rows. Test rows are sorted underneath their parent class row, while class rows are sorted underneath their parent package rows etc. Each row in the table is a div, and a child of the base div for the entire table. Because of this, the sorting order is critical. Packages rows must be sorted first so that the classes can be placed underneath their rearranged packages, with test rearrangement coming in last. In that order, each hierarchy level was sorted using the parentname attribute of the row div and a custom comparator which looked at the pass ratio attribute of the pass ratio div (the third child of every row div). An additional button

Special Notes: The .jelly file uses $ (a common jquery invocation syntax) for a different purpose. In the .jelly file, an alternate invocation variable was setup ($j) with the command "var $j = jQuery.noConflict()". After this change, all the normal jquery operations are accessible with $j in place of $. Unfortunately this was a source of a lot of confusion until the variable change was discovered. Additionally, the base div of the main history table caused a bit of confusion. The history table uses a container div with the id "projectoverview", that is the parent of all other divs. Using the built in DOM structure to identify parent divs was impossible without reorganizing the table template. Instead, the "parentname" attribute of the row divs was used to identify the hierarchy and accomplish the rearranging. The packaged were all rearranged under the fixed header row, which was the first child of the base div.

For the same reasons the calculation of pass ratio had to be done after the backend data retrieval, the pass ratio filtering was conducted downstream as well. The decision to filter this statistic in the frontend was obvious, but made testing difficult. Previously frontend javascript testing was accomplished by invoking the js with an engine written in java. But this was only performed on js functions not related to HTML. The issue here is that all the javascript is linked to an HTML document that cannot be unlinked during testing. Attempts were made to retrieve document elements from an intermediate parser, passing them directly to the functions to avoid any intelligent interaction between the javascript and the HTML document. That did not work because all the java parsers for HTML aren't actual HTML, but rather pseudo representations of DOM objects that can't be interpreted properly by javascript. Additionally the javax engine was unable to recognize any of the jQuery script. It would load other scripts fine, but the jQuery script likely needs to be linked directly to the HTML document. Therefore it cannot be loaded without one. After a lot of creative attempts, unit testing this function in java was infeasible.

5.4 Extra

View SVN Commit UserName
Goal: To display the username of the SVN commit that triggered the build (by Jenkins polling)

Implementation Overview: For each Abstract Build in the Abstract Project, the Causes and Change Sets are collected and used to get the author of manual builds and svn commits respectively. This information is mapped to each build then stored in the JSON object.

Special Notes: When we started this User Story, there was no clear way to get svn information from jenkins, there were multiple classes that had svn information that could be retrieved as functions, but no clear way to create or access an object of that class. Once we found out that an Abstract Build contained a Change Set it was easy to get the user from that.

Display Cobertura Coverage Info
Goal: To display basic cobertura coverage information about a test.

Implementation Overview: Handlebars was used to display the coverage information on the front-end. This required a JSON object that contained the same build numbers as the other tables as well as the Cobertura coverage information for each build.
An @Javascript function was added to take in the various filter information. Based on this information, the program only searched for the coverage.xml files that were associated with build numbers that passed those filters. After that file was found, it was a simple matter of parsing the file and creating a JSON that contained the information needed for display.

Special Notes: At the start of this project, we had originally planned to find the number of lines and branches a specific test suite covers and display it in the test details table. However, we learned that Cobertura only keeps track of how many times a line is hit or branch is taken across **ALL** test suites but doesn't track the rest responsible. This means we had to change our userstory to reflect this new information. Additionally, we wanted to deserialize a serialized Cobertura file and gather the information from a single file. This turned out to be a big problem since it would have required digging into the Cobertura source code to understand how it was serialized and what data was stored within it. Instead, we parsed the coverage.xml files for each build that Cobertura outputs which contained all the information we wanted to include.

6 Usage

Setup

1. Checkout the Test Results Analyzer source code from the svn repository and build it using mvn install.
2. Install the generated .hpi file to your Jenkins instance.
3. Create a Freestyle Project. **Note:** Test Results Analyzer does not operate as expected with maven builds.
4. As a post-build action, choose "Publish JUnit test result report" with "**/target/surefire-reports/*.xml" as the location for test report XMLs.
5. Similarly, if you wish to view Cobertura coverage, choose "Publish Cobertura Coverage Report" with "**/target/site/cobertura/coverage.xml" as the Cobertura xml report pattern.
6. The plugin itself has no configuration options and should now be selectable from your Freestyle Project page.

Using the plugin

1. Navigate to the job from the jenkins dashboard and click on the plugin to open the main plugin page. No tables will be displayed by default.
2. Input the desired build range or leave the field blank to retrieve all builds (note: it is recommended to deal with smaller build ranges when the build count gets high). Hit the get build report button to create test history charts for the specified builds. Combinations of specified builds or filters that do not exist will result in a displayed message that indicates no builds have been selected.

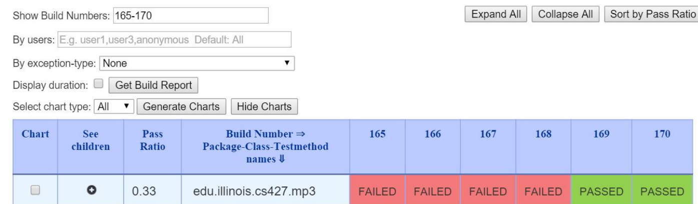

Figure 2: Get a build report for a specified set of builds.

3. Use the various buttons in the upper right hand of the page to expand, collapse and sort the main table.

Chart	See children	Pass Ratio	Build Number ⇒ Package-Class-Testmethod names ⇓	165	166	167	168	169	170
		0.33	edu.illinois.cs427.mp3	FAILED	FAILED	FAILED	FAILED	PASSED	PASSED
		0.50	BookTest	PASSED	FAILED	FAILED	FAILED	PASSED	PASSED
		1.00	testBookConstructor1	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
		1.00	testBookConstructor2	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
		0.83	testBookFunction1	PASSED	PASSED	PASSED	FAILED	PASSED	PASSED
		0.67	testBookFunction2	PASSED	FAILED	FAILED	PASSED	PASSED	PASSED
		0.50	CollectionTest	FAILED	FAILED	PASSED	FAILED	PASSED	PASSED
		1.00	testCollectionFunction1	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
		0.83	testCollectionFunction2	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED
		1.00	testCollectionFunction3	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED

Chart	See children	Pass Ratio	Build Number ⇒ Package-Class-Testmethod names ⇓	165	166	167	168	169	170
		0.33	edu.illinois.cs427.mp3	FAILED	FAILED	FAILED	FAILED	PASSED	PASSED
		0.50	BookTest	PASSED	FAILED	FAILED	FAILED	PASSED	PASSED
		0.67	testBookFunction2	PASSED	FAILED	FAILED	PASSED	PASSED	PASSED
		0.83	testBookFunction1	PASSED	PASSED	PASSED	FAILED	PASSED	PASSED
		1.00	testBookConstructor1	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
		1.00	testBookConstructor2	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
		0.50	CollectionTest	FAILED	FAILED	PASSED	FAILED	PASSED	PASSED
		0.67	testCollectionFunction5	PASSED	FAILED	PASSED	FAILED	PASSED	PASSED
		0.83	testCollectionFunction2	FAILED	PASSED	PASSED	PASSED	PASSED	PASSED

Figure 3: **Left:** Use "Expand All" button to see the build history for all results. **Right:** Use the "Sort By Pass Ratio" button to sort the nested results by increasing pass ratio.

4. Click on the name of a case result name (located in the 3rd column of any row) to open the result details table for that package, class or test.

testBookFunction1

Build Number ⇒ Attributes ⇓	165	166	167	168	169	170
Status	PASSED	PASSED	PASSED	FAILED	PASSED	PASSED
UserID	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)
Exception Type				java.lang.AssertionError		
Assertion Details				Expected: 1,Actual: 69		
Test Duration (sec)	0.178	0.075	0.085	0.041	0.089	0.088

Figure 4: Click result name on the main history table to bring up a test details table for that result.

5. To further filter the results, use the other input filter boxes and drop down menus under the text box for build numbers.

Show Build Numbers: 165-170
By users: rgisle2
By exception-type: None
Display duration: Get Build Report
Select chart type: All Generate Charts Hide Charts
Expand All Collapse All Sort by Pass Ratio

Chart	See children	Pass Ratio	Build Number ⇒ Package-Class-Testmethod names ⇓	165	166	167	168	169	170
		0.33	edu.illinois.cs427.mp3	FAILED	FAILED	FAILED	FAILED	PASSED	PASSED

edu.illinois.cs427.mp3

Build Number ⇒ Attributes ⇓	165	166	167	168	169	170
Status	FAILED	FAILED	FAILED	FAILED	PASSED	PASSED
UserID	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)	rgisle2(jnkns)
Package Duration (sec)	0.307	0.163	0.141	0.134	0.105000004	0.174

Show Build Numbers: 165-170
By users: E.g. user1,user3,anonymous Default: All
By exception-type: java.lang.NullPointerException
Display duration: Get Build Report
Select chart type: All Generate Charts Hide Charts
Expan

Chart	See children	Pass Ratio	Build Number ⇒ Package-Class-Testmethod names ⇓	166	167
		0.00	edu.illinois.cs427.mp3	FAILED	FAILED

Figure 5: **Left:** Provide a username and hit GetBuildReport to retrieve only the builds initiated by the specified users. **Right:** Select an exception type from the drop down menu and hit GetBuildReport to retrieve only builds that contained at least one result failure of the specified exception type.

6. A cobertura report is automatically displayed underneath any other tables.

Cobertura Coverage

Build Number ⇒ Coverage Report ⇓	165	166	167	168	169	170
Branch Rate	58.33%	20.83%	60.42%	25%	64.58%	64.58%
Branches Covered	28	10	29	12	31	31
Branches Valid	48	48	48	48	48	48
Line Rate	65.29%	41.32%	64.46%	39.67%	69.42%	69.42%
Lines Covered	79	50	78	48	84	84
Lines Valid	121	121	121	121	121	121

Figure 6: A cobertura coverage report is displayed beneath the other tables. It provides a report summary for any builds specified.