Software Quality - ENGR-3980U

*Swift Ticket*

*Phase 5 – White Box Testing*

Created by

**Daniel Smullen**

**Jonathan Gillett**

**Rayan Alfaheid**

04/10/2013

Document Revision 1.1

Table of Contents

[White Box Testing 2](#_Toc353317821)

[Statement Coverage 2](#_Toc353317822)

[Methodology 2](#_Toc353317823)

[Testing Overview 2](#_Toc353317824)

[Test Results and Analysis 2](#_Toc353317825)

[Decision Coverage 4](#_Toc353317826)

[Methodology 4](#_Toc353317827)

[Testing Overview 4](#_Toc353317828)

[Test Results and Analysis 4](#_Toc353317829)

[Loop Coverage 4](#_Toc353317830)

[Methodology 4](#_Toc353317831)

[Testing Overview 4](#_Toc353317832)

[Test Results and Analysis 4](#_Toc353317833)

[Figure 1 - Successful Test Execution 3](#_Toc353318387)

# White Box Testing

Phase 5 required that white-box tests were executed using JUnit for the entire codebase that comprised the Back End. In order to facilitate this process, the Software Requirements Specification was cross-referenced with our code in order to create tests that captured all requirements, as well as provided full statement coverage for all lines of code throughout the codebase.

Decision and loop coverage was conducted for one method each, in addition.

Figure 1 shows the successful execution of all tests in JUnit, after bug fixing and code optimization/cleanup was completed:

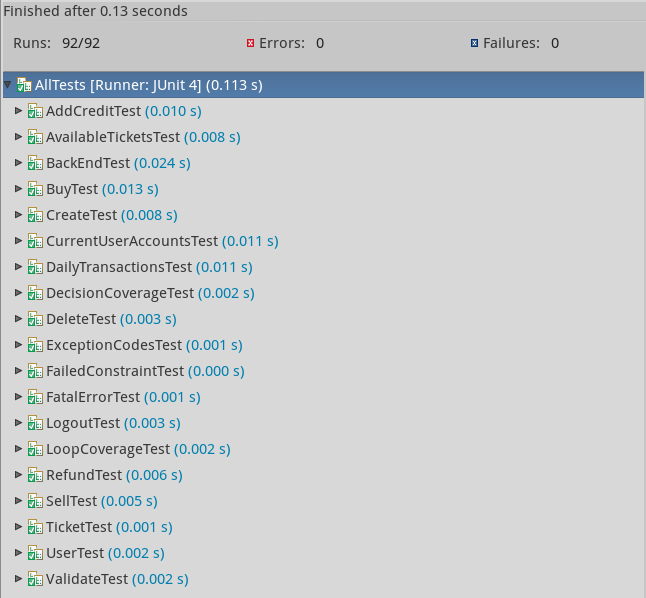


Figure - Successful Test Execution

***Refer to the Master Test List document for specific test details.***

## Statement Coverage

Statement coverage testing was conducted for all lines of code in the Back End. The result was that only one bug was found, but there were several interesting anomalies that were found along the way.

### Methodology

In order to ensure that all lines of code were executed during testing, functional units (classes, then methods within classes) were chosen for tests. These were then analyzed to determine the process flows that would result in all statements being executed. Individual tests were devised that ensure that all flows were executed, with overlapping areas where process flows terminate execution of the code before every line of code in a method is executed.

Notable examples of these types of situations were in the instances where exceptions were thrown. Special tests were required that forced an exception to be thrown, with other tests devised that would not result in exceptions, allowing for the remainder of the non-exception-handling code to be completed.

### Testing Overview

Overall, 93 statement coverage tests were devised, covering 100% of the Back End codebase. An additional tool was used, ECLEmma, in order to verify the statement coverage visually using static analysis.

#### ELCEmma Discussion

Some deficiencies in the statement coverage visualization tool were noted during usage. Because this tool fails to utilize the additional libraries that work with JUnit to provide full coverage of the code, the tool reports that there is not 100% statement coverage. However, manual verification shows that the code is 100% covered.

Figure 2 shows the resultant statement coverage report, generated using ELCEmma:

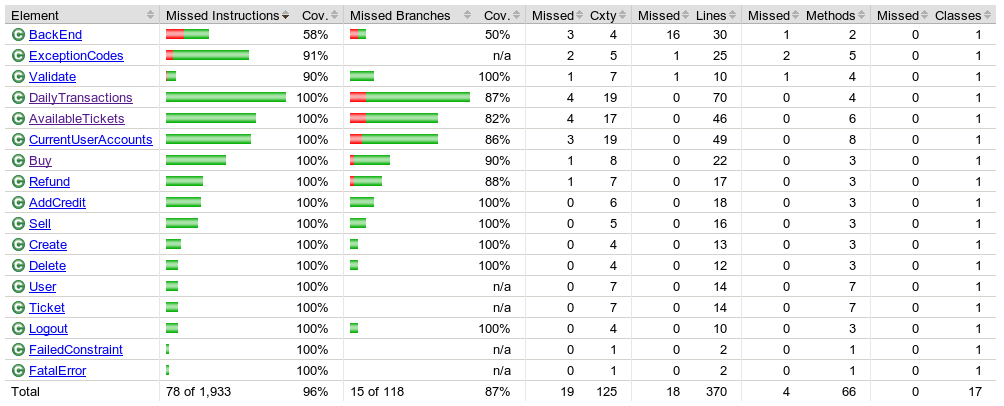


Figure - Statement Coverage Report

### Test Results and Analysis

Statement coverage testing, combined with static analysis, yielded several interesting results, and one bug:

#### RefundTest.testExecute4()

This test found an error in the Refund class, where the following line was incorrect:

|  |
| --- |
| if (cua.getUser(seller).getCredit() - credit > 0.00) |

This statement would fail, indicating that a user has negative credit even if the credit was 0.00. Naturally, the bug was as a result of incorrect usage of the inequality symbol.

#### AvailableTickets.parse() & CurrentUserAccounts.parse()

Static analysis of these two methods yielded a potential code cleanup for the following ‘smelly’ code snippet:

|  |
| --- |
| while ((line = reader.readLine()) != null)  {  /\* Stop if END of file reached \*/  if (reEnd.matcher(line).matches())  {  break;  }  ... |

This code snippet was changed to the following:

|  |
| --- |
| while ((line = reader.readLine()) != null && !reEnd.matcher(line).matches())  {  ... |

Overall, the analysis yielded an improved statement which removes the break, and merges the statement into one of the conditions of the while loop.

#### DailyTransactionsTest.testParse7()

The statement coverage static analysis yielded an error whereby the testing process for this particular method did not have a required logout transaction, which the look-ahead code section of this method was looking for. The particular code snippet where this was relevant follows:

|  |
| --- |
| /\* If buyer is found in lookahead, add transaction with buyer's name \*/  if (match.matches())  {  transactions.add(new Buy(event, match.group(2), seller, volume, price, entry));  } |

#### AvailableTickets.testParse1()

Static analysis of the statement coverage yielded an error in the JUnit tests for this method. In particular, the @Test tag was missing, causing this test to fail to execute. This was caught because the statement coverage visualization showed that this was dead code. The tag was added, the test executed without error, and passed.

## Decision Coverage

Decision coverage tests were conducted for one method. In particular decision coverage was implemented for the Delete.execute() method. This method was chosen due to its simplicity, providing a simple example where decision coverage would be useful – a method with one basic if structure. The single if condition in the method was evaluated with separate tests for both true and false, using testDecision1() and testDecision2() to ensure full decision coverage.

### Test Results and Analysis

Both testDecision1() and testDecision2() passed without finding any bugs or deficiencies in the code.

## Loop Coverage

Loop coverage was implemented for CurrentUserAccounts.hasUser() providing loop coverage for zero, once, twice, and many executions of the body of the loop. These were tested within testLoop1, testLoop2, testLoop3, and testLoop4 for each case, respectively.

### Test Results and Analysis

All loop coverage tests passed without finding any bugs or deficiencies in the code.