**CS 1699 - DELIVERABLE 2: Unit Testing and Code Coverage**

**for Coffee Maker Quest 1.0**

Mike Byrne & Jamie Rossi

mjb187@pitt.edu

wjr18@pitt.edu

https://github.com/wjrossi/CS1699

CS 1699: Software Testing

Bill Laboon

February 17th, 2015

**DESCRIPTION OF ISSUES**

Perhaps the most universal issue in testing is simply figuring out what needs to be tested. In this assignment in particular, this proved difficult because we had not authored the code and were not familiar with it. For most tests, we attempted to provide at least one “normal use” case and at least one edge case; for example, a normal use case for House.generateRooms(int) would constitute a small positive number, while an edge case for that same function would be zero.

However, the nature of many functions such as the ones found in the Room class made even coming up with edge cases; most of the Room class functions simply return the value of a private class-level boolean variable. That leaves an obvious testing set of the case when the variable is true and the case when the variable is false. We were uncertain whether or not there should be additional cases to truly test the code. One suggestion might be to test the function when the class level variable is null, but since these variables are set in the constructor, trying to set them to null would result in code that cannot be compiled due to its errors. Perhaps a mock could assist us here, but then we had to consider if testing these functions in this way was even necessary, as it would be impossible to change the private variable from anywhere but the class level and the variable is guaranteed to have a valid value due to it being set in the constructor.

Additionally, this issue of accessibility came up often while devising tests. Again in the case of the Room class, there is a Room.getDescription() function that returns a description generated from a set list of adjectives using a pseudo-random private static class-level integer. At first glance, it’s obvious that two Room objects can have the same description if this variable is not incremented after use. However, that incrementing code is in the function, and since the variable is static and private, there seemingly cannot be a case where the variable is decremented to cancel out the increment and cause an issue. We decided then that attempting to circumvent these protections with a mock would be unnecessary because we would be testing for an incredibly unlikely scenario outside of something like concurrent accesses.

Another issue that arose while creating unit tests was validating that void functions ran properly. Void functions cannot be mocked to return a value on completion due to the fact that void functions do not return values at all. Thus, these functions had to be allowed to run to completion and checked for execution using Mockito’s verify function. This was tricky to implement, as some void functions modified values while they executed, making testing more difficult.

Finally, the last issue we faced was whether or not to test private methods. In the lectures, it was stated that private methods are generally not tested due to the fact that they will be implicitly tested while testing the public methods; if these methods are not tested implicitly, then our tests either have poor code coverage or the methods in question are dead code. Thus we decided to not test these bits of code. However, it is indeed possible that these functions contain errors that would be difficult to test implicitly, so in a “real world” scenario it would be beneficial to explicitly test these functions.

Our biggest takeaway from this project is that it is difficult write unit tests after a project has been completed, as is the case with CoffeeMakerQuest. Concurrently designing the functions and their unit tests makes the code much easier to test and would most likely decrease the amount of bugs in the final product. Furthermore, limiting the number of void methods would also make the program easier to unit test. Even something as simple as a “return 1 on success” aids in testing and especially mocking due to the fact that there is some kind of return value to check.

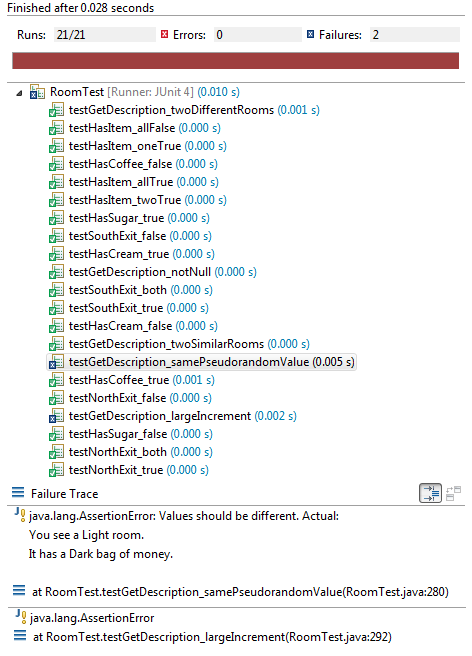
**FAILED TESTS**

**public void testGetDescription\_samePseudorandomValue ()**: this test of the Room.getDescription() function failed because it attempted to check that two rooms returned different, unique strings (as the program’s requirements stated). This fails because the adjectives and nouns for each room and its contained item are generated using a pseudorandom value modulus divided by the size of the array of adjectives or nouns. The pseudorandom value is actually just an integer incremented by one each time it is used in a calculation. By multiplying the length of the adjective array by the length of the noun array and creating that many Room objects, I created a scenario where the first Room created and the last Room created used the same array indexes to generate their descriptions. We were slightly unsure of whether or not unit tests should test according to the program requirements or strictly according to what the function was supposed to do or return, but in the end we decided that a more complete set of unit tests would cover the program requirements as well.

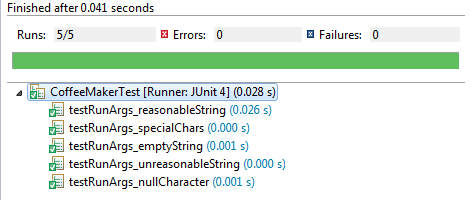
**public void testGetDescription\_largeIncrement()**: this test of Room.getDescription() attempts to abuse the aforementioned pseudorandom counter variable by creating Integer.MAX\_VALUE + 1 rooms and checking to make sure the Room object and its description are not void. In theory, this should fail because the pseudorandom counter will surpass the Integer.MAX\_VALUE limit and wrap around to Integer.MIN\_VALUE, causing an error when attempting to index the adjective and noun arrays used in generating the description. However, we are not completely certain this is the case due to the way the test instantly fails instead of taking a fair amount of time to generate 2,147,483,647 Room objects. The way this functionality is tested calls back to our concern of whether or not we should use mocks to “hack” private variable values (and if such a thing is even possible).

**EXECUTED UNIT TESTS**

RoomTest.java:

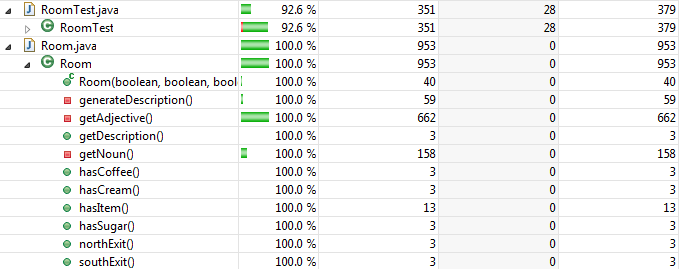


CoffeeMakerTest.java:



**CODE COVERAGE**

RoomTest.java:



CoffeeMakerTest.java:

