**CS 1699 - DELIVERABLE 5: FINAL DELIVERABLE**

**for Coffee Maker Quest 1.0**

Mike Byrne

mjb187@pitt.edu

https://github.com/mjb187/CS1699-Deliverable5

CS 1699: Software Testing

Bill Laboon

April 14th, 2015

**SUMMARY**

For my final deliverable, I decided to rewrite CoffeeMakerQuest to be more compatible with unit testing. Specifically, I attempted to rewrite the program as if it had initially been designed with Test-Driven Development in mind. This included converting void methods into functions that return values and creating a more robust set of unit tests.

In this project, you will find two folders: v1.0 and v2.0. The folder named v1.0 contains the original CoffeeMakerQuest and the unit tests created for it from Deliverable #2. The folder named v2.0 contains the refactored CoffeeMakerQuest code and the new unit tests created for it. Some of these new unit tests are lifted from the second deliverable because they test part of the code that has gone unchanged; others are completely new because the previous iteration was difficult to test in some places. Additionally, I chose to fix bugs identified in the program from deliverables #1 and #2.

I chose to update CoffeeMakerQuest for a few different reasons. For starters, I enjoy working in Java and would like to continue to do so before I move onto a professional career where Java is largely irrelevant. Secondly, I feel that I understand JUnit and Mockito much more thoroughly now than I did for Deliverable #2, so I wanted to revisit this project and see if I could write better unit tests this time around. Additionally, Deliverable #2 was my lowest-scored deliverable (at 91%), so I wanted to see if I could improve this by incorporating instructor feedback and increased understanding of the JUnit and Mockito frameworks into this project. Finally, I chose to refactor CoffeeMakerQuest partially because I strongly identify with the protagonist in his search for coffee during finals week. I undertook two personal coffee-making quests late at night during this project, so I felt this assignment was nothing short of appropriate.

**DESCRIPTION OF ISSUES**

The main issue I faced during this project was deciding where the lines are between “making unit-test friendly”, “fixing bugs”, and “completely rewriting”. I approached this project as if I was in a workplace situation and handed an existing project to review. I did not want to rewrite the project from the ground-up, so I erred on the conservative side and opted to only fix bugs that explicitly violate the program requirements. Bad code was left in place if it was unit-test friendly and not an explicit bug; if I’m being honest, that’s partially due to the fact that I had less time to work on this assignment than I would have liked. Identifying explicit bugs and bad code proved challenging because of the sometimes vague requirements of the program.

Another issue I faced was the question of whether or not I should be using the same unit tests for both the original program and its modified version. I initially felt that I should use the same unit tests across both versions because I keep referring to this assignment as a “refactoring” and that involves the red-green-refactor loop (and the same tests across the loop). I eventually decided that, since I’m changing the structure of the program to be more compatible with unit testing, the modified program would be able to be tested in different ways than the original program. Thus, I opted to use different tests across the two versions. I feel as though the new tests are more thorough than the original tests anyway.

Going forward, I would expect my conservative changes to necessitate another refactoring of the project for both performance reasons and for bad coding conventions. Though they are not defects, bad code should be eliminated whenever possible to improve the performance (and therefore usability) of the program and reduce any safety concerns that may arise due to the bad code. However, since this is a simple, inconsequential program with no real threat to anything, I decided to take this as an exercise in identifying defects versus poor convention.

Additionally, if given more time to work on the rewrite, I would like to take a closer look at the Room class. I did not change anything in this class because it already lends itself well to unit testing. However, there are some areas that could obviously be improved, such as the randomization and possible repetition of the descriptive words and items. As mentioned in the second deliverable, after enough iterations, two Rooms can be generated with the same description; while this is inevitable to an extent, this can be mitigated somewhat, perhaps by adding an additional modulus operation to the array index after all the words have been exhausted once. Furthermore, I would like to take another look at the placement of items within the House class. Instead of the items always being in the first, third, and last rooms, their placement can be slightly more randomized (but repeatable under the same circumstances for testing purposes) by utilizing modulus operations or something of that sort.

**PREVIOUSLY IDENTIFIED DEFECTS**

**Deliverable #1**

*“Help” command does not work as described.*

This has been corrected in the modified program by introducing a helpMenu() method into the Game class that displays the program’s commands and a simple description of what they do.

*The player is able to move when there is no door.*

This has been corrected in the modified program by checking if the player is able to move before allowing them to do so in the moveNorth() and moveSouth() methods of the House class.

*Lowercase ‘n’ is not recognized as a valid input.*

This has been corrected in the modified program by adding case insensitivity into the doSomething() method of the Game class.

*Items can be picked up repeatedly.*

This has NOT been corrected in the modified program because it is not explicitly laid out in the requirements of the program. Because of this ambiguity, I was unsure of whether or not this is a true defect. Since this is a relatively harmless consequence of the way the program is coded and it does not interfere with the player’s progression, it has been judged as a bad coding practice and left in the program.

**Deliverable #2**

*Rooms can have the same descriptions after enough iterations.*

This has NOT been corrected in the modified program because it required 324 Room objects to be generated before this scenario occurred. It has been decided that this number is sufficiently large enough to be considered an unlikely occurrence; thus this has been judged as a bad coding practice and left in the program. See the “Description of Issues” section for ideas on how to fix this issue in a further rewrite.

*Problems can occur when Integer.MAX\_VALUE Rooms are created in succession*

This has NOT been corrected in the modified program because it, too, was considered sufficiently large enough to be an unlikely occurrence. Additionally, it is more likely than not that this problem is caused by memory issues that arise when creating an unnecessarily large amount of objects in quick succession.

**QUALITY ASSESSMENT**

**Failed Tests or Problem Areas**

*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. I was 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 I decided that a more complete set of unit tests would cover the program requirements as well.

Additional problem areas are discussed in the “Previously Identified Defects” and “Description of Issues” sections in regards to unchanged parts of the program that have previously been identified as potential issues. Minor problems are discussed in the following sections.

**Overall Quality**

In regards to quality between v1.0 and v2.0, I would say that v2.0 is significantly more testable than v1.0, and important defects that violate the program requirements have been eliminated in the modified version of the program. However, in terms of overall objective quality, there still are some issues in the program. In addition to the aforementioned repeatability of Room descriptions and unchanging nature of the item placement, there are other issues that make the program rather rigid and not user-friendly. For example, every generation of the Rooms will result in the same descriptions because the stored words are used linearly. The program is also hardcoded to create a House of six Rooms when that is something that can easily be parameterized by the user in the command line. The instructions for the game are only displayed when a command line argument is entered; this is not communicated to the user and intentionally obtuse given that the simple instructions could be displayed once at the beginning of every game. Add into the mix some inelegant coding practices and I would have to say the overall quality of this program is poor, though it has significantly improved from its previous iteration.

**Status**

**User-Interface** – Clunky and minimalist, but generally effective.

**System Performance** – Unoptimized, but functional enough.

**Game Layout** – Houses are always six Rooms long, could be parameterized; movement limited to two directions; item placement is static among Games.

**Interactivity** – Items can be collected more than once; player interactivity is very limited.

**Flavour Text** – Rooms always have the same descriptions due to linear “randomization”; possible repetition of descriptions if enough Rooms are generated (failed unit test); some wording and phrasing is awkward due to description generation.

**Requirements** – All explicit requirements have been met.

**Recommendation for Release**

Honestly, I would say that this product is not fit for release. Though it is properly functional and meets all of its requirements, significant areas for improvement exist that require our attention. Coming off the success of Rent-A-Cat, releasing CoffeeMakerQuest in its current state would only hurt the company’s prestigious reputation.

To be blunt, CoffeeMakerQuest is simply a primitive game; more impressive products have come out of 24-hour Game Jams. One would expect that, to compensate for its complete lack of graphics, CoffeeMakerQuest would provide users with a more robust experience with branching paths, two-dimensional movement, and multiple ending scenarios. While I agree that we are sitting on a conceptual gold mine with the premise of a distraught student searching desperately for coffee, the elephant in the room is that the game is simply not very engaging. We cannot expect CoffeeMakerQuest to pull in the same level of revenue as the Rent-A-Cat service at the moment.

Therefore, I am recommending CoffeeMakerQuest be given a serious change in direction going forward in order to meet potential profit expectations. Specifically, I would recommend the project be reworked into a point-and-click adventure game (in the vein of classic Sierra Entertainment videogames) for modern mobile devices. I strongly believe that the cost of bringing on an art and design team will be offset by the profits available to us on the mobile platform.

**Code Location**

https://github.com/mjb187/CS1699-Deliverable5

**EXECUTED UNIT TESTS**









