CoffeeMaker

# Jordan McAleer

CS 1699 – FINAL DELIVERABLE

For the final deliverable, I rewrote CoffeeMaker.java to be more test-friendly. In deliverable 2, the class created JUnit tests for the program. Thus, half the work was already accomplished because I had the original JUnit tests and coverage to compare against the rewritten code tests and coverage. This meant that I would have to use the same type of test, otherwise I would not be accurately showing the growth in testing coverage of the new code. Also, after having gone through the code the first time and getting frustrated with how impossible it was to write tests, I had some ideas on how to change the code.

The JUnit tests determine output and properties of the program methods. For instance, the *House.generateRooms()* is tested that it has the correct number of rooms, an end room with no North exit, an end room with no South exit, all three items contained in the array of Rooms, etc. While all the methods are tested upon their output, they are not all tested on their properties. The properties were not always relevant to the code: *House.java* and the other scripts are being used specifically in the CoffeeMaker script, so as long as no errors occur in the basic use of the program, there was no reason to test the properties. However, if more options were to be added to the program, more property-based testing would not only be suggested but possibly necessary to ensure the program does not break in edge cases. I tried to maintain as much of the original code or the results of the code as possible, so not much was changed about the actual implementation that could have greatly increased the worth of the program.

This assignment was probably the easiest thus far. It was reusing some ideas that had been used in an earlier deliverable and making it more my style—easily testable. The only problem that I was having was juggling keeping the original code, fixing errors, and mainly choosing between the DRY/YAGNI principles or testing different properties of the code. For instance, I wanted to test each of the “get some item” methods in *Player.java* because the *Player.getSugar* method might give a Player cream instead, but testing only if the user has none or all of the items would not detect this problem. To test this feature, I would have to add methods that return each individual boolean of having the specific item. This code would not be useful to the actual implementation of the code, but only the testing. Also, as long as the user can obtain all three items (which is tested when we get each of the objects), the accuracy of the messages is something users could put up with. Other tests I omitted because it felt like repeating the tests for another class. Thus, I really attempted to limit the tests to method returns or relevant properties: I could have exhaustively tested until the sun stopped shining.

Another issues was trying to test the return values on the *CoffeeMaker.run()*. The only way to get the method to return the correct values was to have the methods called by run actually run. I spied the CoffeeMaker object and used Mockito’s doReturn in an attempt to force return values, but they did not seem to work. Thus, I had to write tests, then make the code fit the tests, and make the tests fit the code. That was probably one of the more difficult portions.

I would have liked to do more tests that may have been a bit exhaustive, but ensured that each method was working **exactly** as expected. I also would have liked to property test the program and allow users options like the House size or even just creating a random House size and random locations for the items. Options and unexpectedness lures customers whereas if the program does the same thing every time, it loses enjoyment.

After rewriting the CoffeeMaker program, the program no longer had any errors from the previous deliverables. While I maintained most of the original code, I also edited some bad code areas and included methods that were not originally there. The two main areas fixed were the user input (now completely case insensitive and includes the help instructions) and the North/South return an error if a user attempts to exit through a door that does not exist. The code coverage is almost 100% where the only places we lack are lines of code that return in case of an error and the main method of *CoffeeMaker.*

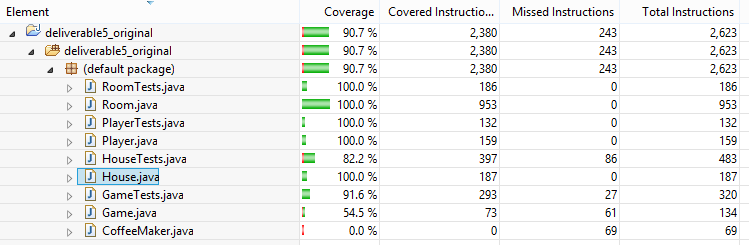
While the program does not have defects, it is only decent at best: it does not have any features that might annoy a user, but the program is predictable. The coffee, cream, and sugar are always in the same places: [SPOILER ALERT] the very beginning, the third room, and the last room [END SPOILER]. Also, the user can simply go North until the end of the Rooms and press “L” (i.e. look) the whole way. With this small amount of work, users will not find any motivation to repeatedly play this game. In order to maintain user attention, the program should be less easy and less predictable. Unless the goal of this program is to be a two second one-time play through, the lack of allure for the user is a source of serious concern. I want people running up and down hallways screaming “I need cream and sugar for my coffee! L! L!”

A few ways this could be accomplished is to make the House more than a straight line of Rooms that go South to North, add dimensions such as East and West, or even change the directions to relative positions (i.e. right, left, forward, back) so that the users have to remember the direction from which they came; the cream, sugar, and coffee should be placed into random rooms. Lastly, rupees. As this program is obviously an ode to the many nights spent drinking coffee in order to finish Ocarina of Time, the program should have rupees. An “R” command could be added in order to ransack the place and collect all the rupees where the number of rupees in the room could be dependent on a chance that is proportional to the likeness of sounds inputted into the mic to the sound of “Lost in the Woods.” Only then will the program be complete.

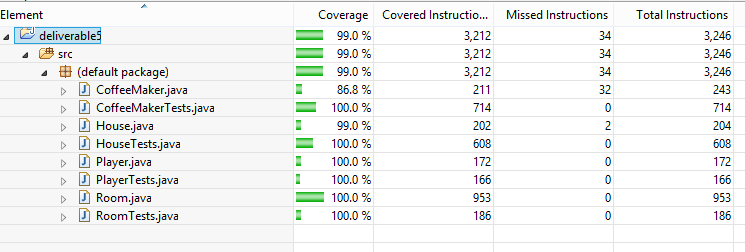
CODE SOURCE: <https://github.com/bluejayy19/deliverable5>

* The files in MyCoffeeMaker are the rewritten code.

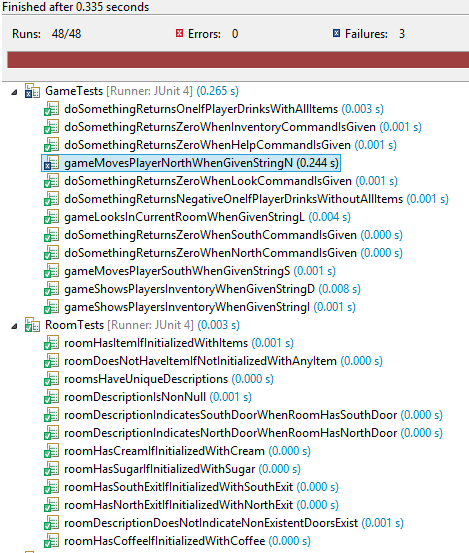
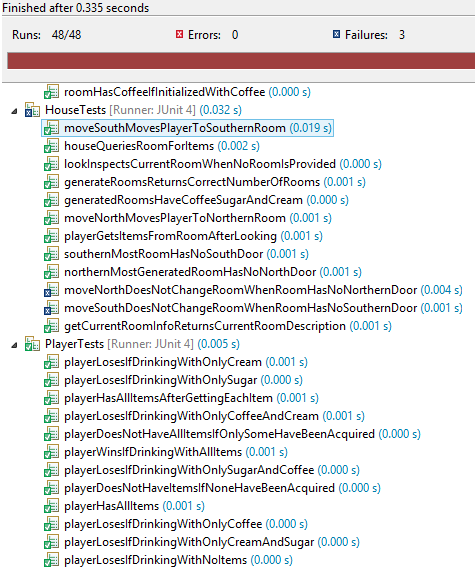
**Code Coverage prior to rewrite:**



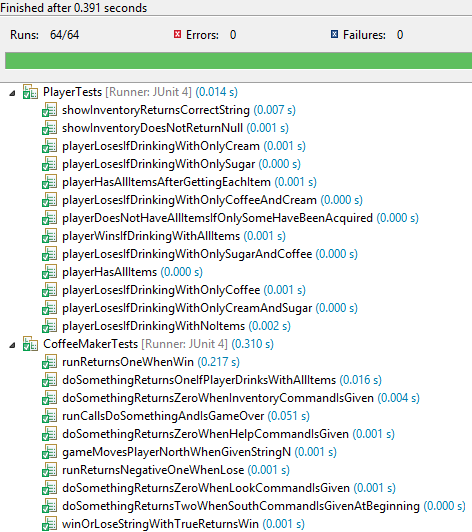
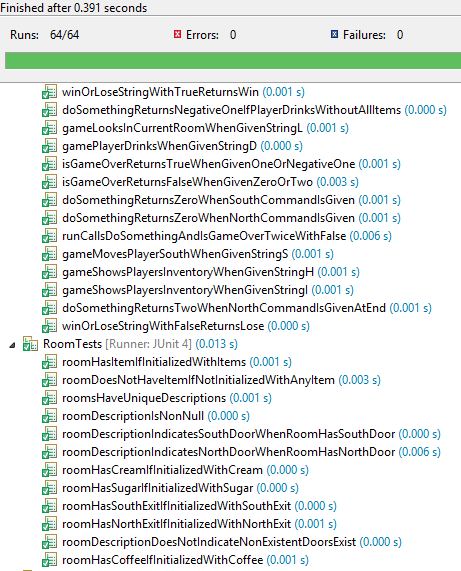
**Code Coverage after rewrite:**

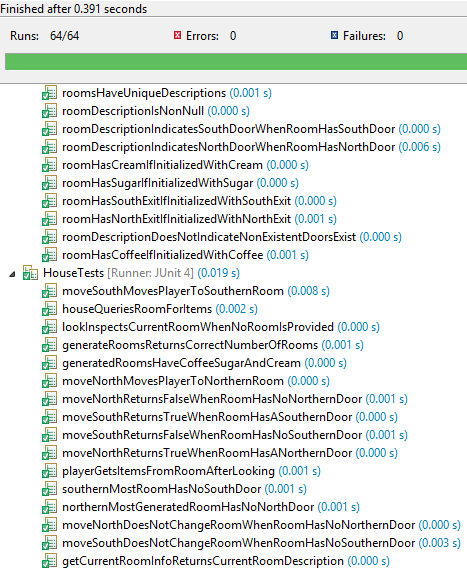


**JUnit Tests prior to rewrite:**

**JUnit Tests prior to rewrite:**



And here we began to see money in all the green test.