Final Project 7-1: Software Design and Engineering Artifact

CS-499-Q5048 20EW5

*Original Assignment Name:*

*Final Project 7-1: Summary and Reflection*

*CS-320-X1984 19EW5*

Jacob Gottschalk

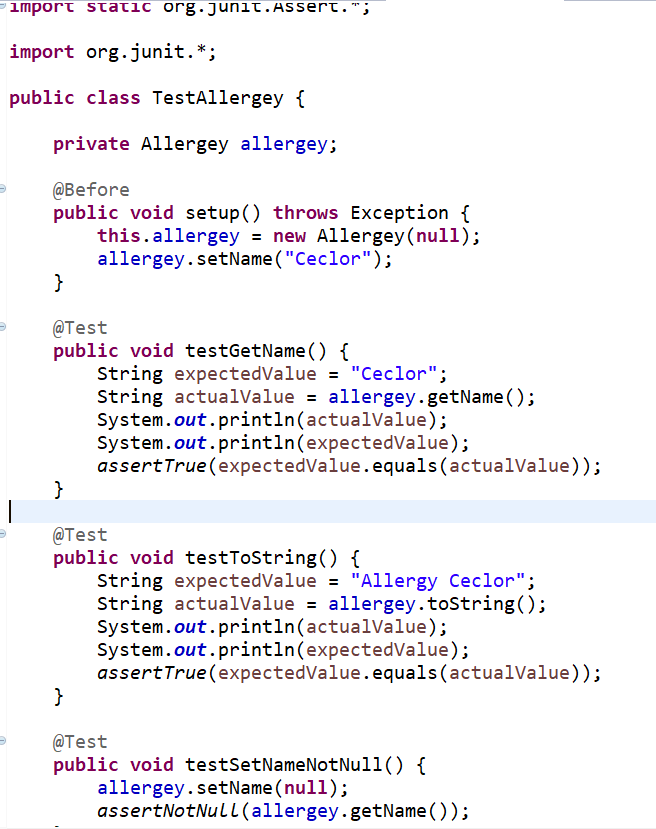
**Summery:**

While writing my JUnit tests I took the approach of writing tests to cover the good, the bad, and the ugly. The definition of good, bad, and ugly would be the programming accepting a valid input (good), accepting inputs of multiple formats (bad), and accepting a null input (ugly). On average I wrote 4.7 tests for each feature (27 tests written, 6 features covered). This may sound low, but there was 1 class that brought that average because there was only one function to test within that class. The returned value from that function was defined in the program code so there was no real failures that could be found within this class. Never the less, I felt like it was important to test this class to ensure the function returned the value expected. For the number of tests written, I feel as though the level of testing was good despite not addressing all the ways the functions within the features could fail. Out of my 27 tests written, 16 found defects within the code. The best practice guideline I was trying to follow (and I believe I achieved) was not writing tests to pass. In relation to testing the complete program, my level of testing was poor given that I did not cover 4 major classes/features during the test.

The organization of each test case started with testing values that would be considered valid entries. The reason why I started with this is because it allowed me to write the @Before test setup to define variables which could be used in multiple test cases. This allowed early tests that should pass to be written in a light weight format with fewer lines of code. Later tests defined the variables for just that test case to catch specific failures. Using this approach follows the best practice of keeping tests small and simple.

During my JUnit testing I did not address all the software requirements. I did ensure that all the functions processed the information in manor as expected (test with valid inputs), but I did not get to writing tests for the more complex classes. I performed some black-box testing prior to writing any tests to get an understanding of the flow through the software. During the black-box testing I found that a doctor could be input into the system with the same ID as another doctor. I started testing basic features to determine if they operated as anticipated with known valid inputs but had a hard time figuring out how to write a test for the doctor ID failure that I knew would be in the software based on the fact that the value returned for comparing this was part of a string data type. I started working on parsing the string to extract just the portion containing the ID number to compare entries, but I ran out of time to complete this test. Black-box testing also revealed that the software did indeed prevent adding medications when there was no treatment added to the patient profile. Despite the black-box testing not falling within the scope of the course, it was helpful to perform some of these tests before jumping into writing the JUnit tests.

My goal while writing the tests was to maintain a simplistic test format. The reason why I kept the tests simple is because simplicity prevents errors within the tests. A test is only effective if the test itself is error free. Writing an overly complex test will be more likely to have errors, and as such it will not test the software the way it was intended (giving a false pass/fail status). The below screenshot shows a test that is syntactically accurate, logical, concise, and modular:



The above screenshot tests the allergey class to ensure it can process a valid string and return the value accurately. It also tests if a null entry can be maid (test fails because there should be no reason to make a null entry despite the system allowing a null entry to be submitted). These tests each contain two to three lines of test code within each test structure (excluding the System.out.println statements to allow comparisons to be made between actual and expected values). Pair this with two lines of code in the @Before and this test class is not only covered completely, but also with syntactically accuracy, logically, concisely, and modularly.

Overall my JUnit tests were very effective. I was able to catch far more than 5 errors with 16 errors being found in the 27 tests performed. Most of the errors that I found were related to either formatting or variables accepting null values. Null values being accepted in most applications would likely be considered bad practice, but especially for a system being released to the medical field as this could cause a life or death situation for a patient. Even after the limited test coverage within the total applications this application has a long way to go before it can be released to the customer.

**Reflection:**

The main testing technique I utilized during this testing was error guessing. This technique is one where the tester uses previous experiences to guess where the troublesome coding mistakes may be. Even though I do not have previous experience I used this technique because I had read through the code and determined that the data types used (strings) made it hard to use other techniques. I did utilize boundary testing as well. I did this because I knew the strings should not accept null values. With both testing techniques, I would go back to the development team with the findings and make some suggestions about further refinements that could be made to this code (not necessarily found with the tests themselves) that could improve the quality of the program.

Two techniques that I didn’t use but could have if I had performed further testing on the who system, would have been using decision tables and state transitions. Both techniques could have been useful while testing the log in and integration portions of the software testing. My testing stopped with testing just the basic classes that are used to control the features of the software.

It is important to test the code of verifying and validating the code is to ensure the code operates exactly as expected as well as meeting the expectation of the customer or end user. If the code is not properly tested, the customer may decline the final product presented to them which will either lose the customer completely for the developing company or be costly to the developing company due to having to restart the development phase. Furthermore, if the software is not tested fully, but is accepted by the customer, then the performance of the software may cause functionality concerns for the users or sensitive data corruption/breaches that could be costly for the customer who purchased the software.

Following the best practices during the planning, testing, and monitoring stages will help to keep the program testing on track. Developing good test cases will give structure to the testing to be performed. Utilizing tools to monitor test quality as well as test quantity will assist with making sure the most important aspects of the testing are met prior to completing the testing phase. Keeping tests small and simple will ensure the quality of the test to prevent either ineffective or unnecessary tests from being performed. One of the most important aspects of software testing is knowing that it is not possible to catch every bug but making sure to test the as toughly as possible to ensure the biggest risks are mitigated is the key to completing a quality program.

My favorite tools that I utilized during testing were the JUnit test suites and the code coverage tools. The code coverage tool was helpful to knowing how many executable statements were performed during the tests. The key to this tool, however, is not focusing solely on reaching 100% coverages, but using it as a guideline to know how much code is tested, what code was skipped, and what sections of code failed to perform their tasks. This tool allowed me to focus more diligently on the areas of code which I might find more errors in. Using the JUnit suite, I was able to track how many tests were performed as well as how many tests failed. Viewing the tests that failed also allowed me to analyze these tests and theorize how there might be failures of this type in other similar classes. This then allowed me to write a few more tests than originally planned which caught the same bugs in other areas of the software.

The mindset that I took on during this project was part detective and part developer. I was looking at the code requirements thinking about the critical parts of the code such as how the code would have to be structured as well as the data types, then I reviewed the code to see which variables were used, how they were used, and what data types were selected for these variables. This investigation into the code structure then allowed me to determine how to start writing my tests for this code to verify it met the requirements. Finally, I was able to put all of this together to write multiple tests for each feature to ensure the code accepted valid inputs as well as not accepting invalid inputs. This detective work was the find the “culprits” that would cause the program to behave in ways outside of the defined requirements.

I feel I was successful in limiting my bias because my goal was to verify the software was working efficiently and effectively within the requirements of the customer. I maintained distance with the code because I was not the one who wrote the code, but I also didn’t want to push back on the developer simply because I thought the code should be written differently than what was provided. Had I developed this code myself I would likely have a hard time being objective with testing because I would feel that I wrote code to meet the requirements of the program without creating any weak points. From this perspective I may have written fewer tests which would only verify the areas where I thought the concerns would be as opposed to writing the tests to verify all potential troublesome areas of the code.

Ensuring quality when writing and testing code is important since errors caught in later stages of code development end up being more costly to correct. Cutting corners to save time during the development or testing phases will likely come back to bite the development team because a product could be released which causes annoying performance concerns for the end user, could cause important data to be leaked, or allow for the safety of people to be put at risk due to errors in the code. These errors can also cause damage to the reputation of the software’s developing company as well as the company who purchased the software as well as could cause legal issues for both companies. When creating or testing software in the future, I plan to follow the best practice rules as well as diligently review my code/tests to ensure I prevent as many bugs as I can from reaching the release phase of the software.