CS 320

Professor Toledo Lopez

Michael Thomas

4 August 2024

Summary and Reflections Report

When writing the JUnit tests, I had individual functions for each software requirement. For the Contact Service and Task Service projects, it was specified to limit string character lengths and ensure inputs were not null. When testing in specified java files, there are individual functions for different exceptions like “…NameTooLong” and “…IsNull”. The testing approach aligned perfectly with software requirements, where an individual test existed for each separate requirement.

My JUnit tests encompassed every line of code where the only pieces that were not read are the exception handling aspects. These exceptions are still tested, but the tests are not included in the percentage as the program ran without issue. This is mainly found within the Task and Contact java files with their respective testing files, as some of the exception handling is duplicated for different program use. Otherwise, with the service testing, the resulting percentage is 100% showing that the JUnit tests were extremely effective.

In order to ensure code was technically sound, I employed thorough testing of each aspect of the code. There may be some redundancy found when testing the add and delete functions of Contact and Test Service, but I wanted to verify that the program could handle multiple entries and requests. In all of my testing, I first establish the change, then use assertEquals commands to get the new description after it has been updated. If the two match, then I know the program is working as intended.

The only lack of efficiency found within my code is duplicate exception handling in initial method creation. I attempted to remove each set of handling, but this would cause some of the JUnit testing to fall short. Otherwise, the code is streamlined, with an easy-to-read format. I especially appreciate the combination of restrictions when establishing method aspects. For example, when placing restrictions on user input phone number, all the restrictions are in one line: “if (phone == null || phone.length()!=10 || phone.matches("[0-9]"))” – found in Contact.java, line 20 & 58.

Each of the milestones utilized JUnit 5 testing, where cases are built upon internal code structure. As the tester, I assigned values to each object parameter, then checked to see if those values changed after using program functions. For the testing of adding or deleting Appointments, Contacts, and Tasks I implemented the use of assertEquals clauses. If the test passed, then it showed whenever something was successfully added or removed to the list. As different requirements were established, these JUnit tests allowed me to test specific areas within the code. To ensure that the code was complete and represented all requirements, I created a test case for each individual function in the code. On the opposing side, in order to test the creation of Tasks, Appointments, and Contacts, I utilized assertTrue and assertThrows clauses. AssertTrue allows me to compare the real value of parameters for each class, ensuring they fall within the proper guidelines. It is as important to check if exception handling is working properly, so the use of assertThrows helped. I purposely input incorrect information and then checked for the IllegalArgumentException class, which would notify me if the exception was handled properly.

One of the techniques I did not use was code isolation through mocking. Mockito is the most common framework used for JUnit tests, which can help with more complex methods that may connect externally to databases (Jakubiak, 2018). Another unused technique is JBehave, which is a framework for Behavioral Driven Design (BDD). Instead of being test-based, this changes to behavior-based, where focus is changed to the user’s perspective (Mork, 2015). Tests are written in different acceptance criteria, where the system is observed with the setup in place, when the function occurs, and then the expected results. This is definitely a higher level of testing, and better suited for more complex projects.

I believe JUnit tests work best for smaller programs with simple functions. When a program becomes more complex, like accessing databases or calling functions within functions, it may need a more robust test case technique like JBehave or Mockito. It is especially important to test thoroughly, so implementing some isolation techniques on smaller projects may be worthwhile. Before choosing a technique to use, think about the complexity of functions and the desired output. If a program is meant to be user friendly, with a majority of concern being for their experience, I suggest using JBehave. Taking the shift from Test-driven development (TDD) to BDD will help visualize user perspective, allowing the discovery of the most important features.

As a software tester, it is important to employ caution when working with every line of code. This is why I made sure to create JUnit tests for each and every function found throughout the program. Another important reason for testing thoroughly is due to the interrelationships found within code due to the complexity of programs. To ensure that everything was encompassed properly, there is some duplication in error exception handling. This is mainly found when establishing a new class and then having functions modify those class details. Due to the same trait being modified twice, it is important to test and provide exceptions for both functions individually.

The best way I found myself limiting code review bias was by strictly following the client requirements. Bias is a huge concern, especially when testing one’s own code, as they may feel some areas are not as big of a concern as others. At one point, my code required a user to update all aspects of a class in order to change only one attribute. However, after review from the professor, it was made clear that each attribute needed to be updated individually, so the client could have an easier experience. If I was not made aware of this perspective, I would have never considered making the change, due to my bias in how the program should operate.

It is of utmost importance to have high discipline in program code and test quality. If corners are cut when writing or testing code, numerous vulnerabilities and bugs will occur that can be exploited or cause harm to businesses. In order to avoid technical debt, I plan to employ strong Root Cause Analysis (RCA). This involves finding the why for each error until the core is reached. I believe this will be best employed with other team members, as to have multiple perspectives on why programs are not working properly.