Project Two: Summary and Reflections Report

Joshua Pardue

CS 320: Software Test Automation & QA

Professor Matthew McHann

June 16, 2022

Summary and Reflections Report

**Summary**

*To what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.*

While challenging, quite a bit was learned in coding the contact and task service. I tried to format my testing approach in a way that verified the specific requirements of each service were separately met. This was no easy task as the requirements were particular so the tests must be particular as well. In the case of JUnit testing, it was necessary to make the tests repeatable. Test cases were written calling the different methods reflecting and aligning with the requirements. By asserting the return value against the expected value, the test case verifies certain parameters. For the task service test, the Task class takes three arguments in the constructor. This constructor has conditional checks to run on the task ID, task name, and task description. If these objects do not meet their requirements, an illegal argument exception is thrown. For the JUnit tests, I created objects and used assertions provided in the constructor from the get methods to test data. To test if an object is too long, an exception is thrown. Assertions.assertThrows(IllegalArgumentException.class,() -> identifies the exception expected to be thrown while new Task(“01”, “task special”, “yellow”) provides the expected behavior that causes the exception. So per the test, if assertTrue(task.gettaskID().equals(“01”) is longer 11 characters and not 10 (using the requirements), an exception is thrown.

*Defend the overall quality of your JUnit tests for the contact service and task service. In other words, how do you know that your JUnit tests were effective on the basis of coverage percentage?*

Coverage is a very useful tool in determining how effective the JUnit tests are. In running the coverage of my code I was able to ascertain which statements were executed in the test case and which ones were not. For the task class coverage, 100% was covered so the test cases I coded were exercised when those specific tests were run. The contact and task service were almost identical with the exception of a few minor requirements. These minor differences, in turn, had to be reflected in the JUnit tests. Even though they were very similar, I had more issues coding the contact service tests. The green highlighted portions mean the line of code has been covered by the test. The yellow highlighted portions indicates that not all the branches in that particular code have been reached, and the red highlighted portions show that line of code has not been reached or covered in the JUnit. On the contact service, my JUnit tests were less effective as my coverage percentage was substantially lower and more red highlights were present.

*How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.*

In ensuring my code is technically sound I need to be aware of bugs and/or errors which might crash the program. It also means using JUnit tests to experiment with what code follows the exact set requirements. I am able to make sure the proper portions of code interact with the necessary counterparts. The coverage tool can also be used to determine how technically sound the code is. For my task class, my constructor public Task(String taskID, String taskName, String taskDescription) is in green so that portion has been covered by the test and seems to sync up where it needs to. My “if” statement of if(taskID == null || taskID.length() > 10 is in yellow so not everything in this statement was reached. I had to write a null test as well, which then allowed the entire statement to be covered. TaskID was first checked for null and (or) then if it was less than 10 characters in length. It was then highlighted green.

*How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.*

Efficient code is something I work hard at and the more ways I learn to write, the more I understand what is efficient and what might not be as structured. One of my main sources of efficiency from this project is in the lambda expressions used in JUnit tests. This particular lambda expression is in the task class test:

@Test

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Task("0111111111111111111111", "task special", "yellow");

});

}

By using this lambda expression, I don't need to define the method again for providing the implementation. The constructors I wrote ideally were written with efficiency in mind. My Task test covers all the objects initially through:

@Test

void testTask() {

Task task = new Task("01", "task special", "yellow");

assertTrue(task.gettaskID().equals("01"));

assertTrue(task.gettaskName().equals("task special"));

assertTrue(task.gettaskDescription().equals("yellow"));

While I am still learning newer and more efficient ways of coding, I understand varying efficiency techniques such as eliminating unessential operations, avoid declaring unnecessary variables, appropriate algorithms, and minimization.

**Reflection**

*Software Testing Techniques Employed*

There were a few techniques I was able to recognize and utilize to complete the testing for the various requirements. Of the testing techniques, specification-based, structure-based, and experience-based, I seem to be using white-box or structure-based testing with JUnit. With this process, the various elements of the structure are exercised to ensure their correct operation. I used the process of *decide on a test condition*, *design a test case to verify the specific condition,* and *write a test procedure to execute the test.* Utilizing the white-box technique allowed me to further incorporate characteristics of the stages of this process and uncover errors or issues in the code. Using the coverage tool allowed me to verify which lines of code were functional and which parts of the code were actually tested and to what extent.

*Other Software Testing Techniques*

The other techniques that I am aware of and have not used yet are specification-based and experience-based. Specification or black-box testing techniques examine the application without knowing the internal design, structure, or implementation of the project. While black-box techniques can be performed at the unit level as well, I have not necessarily tested the functionality as a whole of any of the projects yet.

Experience based techniques are based on the skill and experience of testers, experts, users etc.…These techniques would be utilized if the requirements/specifications were not available, requirements were insufficient, limited knowledge of the project, or time constraints did not allow for a structured approach.

*Uses and Implications of Techniques*

The techniques I have described in this journal have varying practical uses and specific noteworthy situations. For *structure-based (white box)* testing, I noticed some of my code was highlighted in yellow such as *if appointID == null | | appointID.length() > 10*, and that I was only checking the length of the appointment ID and not its null properties. Once I properly tested the null statement as well, it was highlighted in green. By having an understanding of the functionality of the code and testing with this specific technique, I was able to determine the classes and methods being called and check the structure of the code.

*Specification testing (black-box)*, I would not be concerned with the internal code but only the input/output of the contact, task, or appointment programs. I would verify the output based on the input. If I wanted to test the functionality of the actual Contact Service program, I would input a specific contact ID and/or attempt to add, delete, or update and verify the proper output without necessarily concerning the code.

*Experience-based* techniques would involve e**rror guessing** and applying experience to guess the areas in the application that are subject to error. This also involves **exploratory testing** which explores the application and uses experience to navigate functionalities, and **checklist-based testing** which applies a tester's experience to create a checklist of functionalities and use cases. While I have little to no experience, I am still able to minimally scan for errors and assess functionality of the Contact, Task, and Appointment Services. The listed requirements serve as a checklist of sorts where I am able to verify functionality through Junit and coverage testing.

*Employing Caution*

As in any sort of development role, caution was exercised with time and efficiency in mind. It was important to note the complexity and interrelationships of the code due to the specific portions of the methods that were being tested apart from others. As in the white box testing, it was imperative to write tests that covered the purpose of the method such as the Contact Test checking the length of the ID and throwing an error if it is too long. This had to be paired with the Contact class constructor in order to operate properly with “Invalid contactID” string:

Void testIDtoolong(){

Assertions.assertThrows(IllegalArgumentException.class. () -> {

new Contact(“longtoolong”, “Joshua”, “Pardue”, “1234567890”, “12 Pine”);

});

}

*Bias Limiting*

Bias was not necessarily an issue on this project due to it being more of a learning platform but I could see where problems might arise on the professional side of things. Approaching the individual tests with a specific mindset is key with the goal of testing the behavior of the methods inside the discrete classes. There are a few biases I came across when researching this topic such as confirmation, congruence, survivorship, fundamental attribution error, and negative bias. I suppose any of these biases could have been present on this project but it seems negativity bias was most recognized. Due to a lot of these requirements and assignments that were requested, the initial inclination was to project a negative bias on the project such as, the phoneNumber object having issues throwing and IllegalArgumentException when all other avenues seem to be exhausted with resolving the error. The negative bias was quickly resolved with the understanding that the right syntax does exist and there is a systematic way of testing for such things.

*Discipline Importance*

This topic required some persistence on my part mainly due to it being a learning platform so I was not as familiar with what disciplines to concentrate on and what disciplines were not as necessary. I feel that I wrote meaningful names for variables, functions, and classes with clear intent that were self-explanatory. It is important not to cut corners because you do not want a poor reputation or to be known for rushing requirements. For these requirements I used explicit naming for what the function of the class actually does such as contact, appointment, and task along with packaging all of the tests together for ease of use and locating. In addition to proper naming and location of classes and files, I avoided anonymous functions that might be difficult to go back and understand. While this was a relatively simple structure, I feel that the processes and tools used managed things with minimal complexity.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 4.1.1.1 The Test Development Process (K3).* BCS The Chartered Institute for IT. Retrieved from   
<https://app.knovel.com/hotlink/pdf/id:kt00UC2IS4/software-testing-an-istqb/test-development-process>

Shenk, G. (2022, March 9). The impact of cognitive bias on software testing. AI Test Automation with Machine Learning. Retrieved June 15, 2022, from <https://www.functionize.com/blog/the-impact-of-cognitive-bias-on-software-testing>

García, B. (2017). Mastering Software Testing with JUnit 5. Packt Publishing.

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/junitcoverage.html. (n.d.).