**Project Two: Summary and Reflections Report**

Sarah C Jodrey

Southern New Hampshire University

CS-320-T6629: Software Test Automation & QA

Professor Linda Gaston

August 14, 2022

**Project Two: Summary and Reflections Report**

The mobile application which consists of the Contact Service, Task Service, and Appointment Service classes all had specific requirements that needed to be met in order to satisfy the customer’s request. To ensure that these requirements where met I had to create JUnit tests that would confirm each requirement was in fact a part of the application. Failing to meet any of these requirements would result in a product that the customer would be rightly unsatisfied with.

**Alignment to Requirements**

All the requested services had similar requirements regarding unique ID’s, null fields, restrictions for length of inputs, the ability to modify, and specific to the Appointment Service class, ensuring dates entered where current. These requirements guided my approach as I made sure to check that each one was covered via different JUnit tests. I also made sure to check for failures and coverage as I wrote each test.

An example of how my approach aligned with requirements is the Appointment Service class. This feature class required a unique ID, appointment date, and an appointment description. The unique ID could not be null, greater than 10 characters, a duplicate of another ID, or updated. The appointment date could not be in the past, or null. The appointment description could not be null, or longer than 50 characters.

To check Appointment Service requirements, I initially filled the list with valid data in addAppointmentsToList(). This was then confirmed using assertAll() and assertEquals() in testAddNewAppointment(). One test, testInvalidAppointmentValues(), was written to verify that an exception was thrown when any of the values did not meet the stated requirements. The data I provided included a null ID, an ID that exceeded character limits, a date that was null, a date that was in the past, a null description, and a description that exceeded character limits. Additionally, I wrote testForDuplicateIDs() which used a for loop to confirm that no ID was ever the same. A for loop was also used to confirm that appointments were successfully deleted using their unique ID in testDeleteAppointment().

**Effective Tests**

According to general standards, striving for a test coverage rate of 80% or greater is considered reasonable (Garcia, 2017). The lowest file coverage I received was 94.8% in the Appointment Service class which is well above the standard. The range of coverage for all files was between 94.8% and 100% as shown in Figure 1. For this reason, in addition to my approach for aligning with the project requirements, I can confidently state that the quality of my JUnit tests was above average.

Figure 1

*JUnit Overall File Coverage*

Graphical user interface

Description automatically generated with low confidence

**Technically Sound Code**

To ensure my code was technically sound I practiced proper naming conventions, concise comments for each method, and organized the objects for readability. The variable names clearly describe their purpose, as do the names of each method. For example, testInvalidContactValues(), testDeleteTask(), and testUpdateTaskDescription(). Another example is using @BeforeAll and @BeforeEach so that each test would have sample data to work with. This way only the ID would need to be referenced instead of creating a new task for each test. Additionally, the variables for testing were made private to ensure controlled access for testing as exemplified in the Appointment Service Test class’s “private static Date validate = new Date(2022, 10, 19)”.

**Efficient Code**

To ensure my JUnit test code was efficient, I used methods from existing libraries that allowed for testing more requirements with less code. One example was using @BeforeAll and @BeforeEach in all three feature classes. This allowed me to set up data to be used in each test case. Another example was using the assertAll() method to check for thrown exceptions and to confirm that values equaled what was expected. The assertAll() method made my code more efficient because it checked that each value returned true instead of failing the test after one incorrect value. This made for less code and more efficient reading of the test results. An example from the Task Service Test class is provided in Figure 2.

Figure 2

*Task Service Test class test for invalid task values*

Text

Description automatically generated with medium confidence

**Reflection: Testing Techniques**

**Techniques Employed**

For this project, I employed both static and dynamic software testing techniques to ensure that my program behaved as expected. The static, or black-box, techniques I used were equivalence partitioning, boundary value analysis, and decision table testing. The dynamic, or white-box, techniques I used were statement testing, decision testing, and testing coverage.

The equivalence partitioning technique essentially chunks potential inputs into groups of similar values. To implement this, I partitioned the inputs of my program into three subsets with an additional subset for the unique ID – valid, too long, null, and duplicate for the unique IDs. A negative value wasn’t used because I am dealing with Strings which, as far as I know, cannot have a negative value. I didn’t exhaust the boundary value analysis technique, which tests for values just outside the boundaries, but I did make sure to test for inputs of varying lengths that fell within the range of valid for the given value.

The decisions table testing technique lists all the possible input conditions that can occur and all the actions that might arise from them (Hambling et al., 2015). This was extremely helpful to me as I mapped out the different test cases based on conditions for the values of the program. For example, the Task Service class required a task name with certain requirements that had to be met for the input to be accepted. If the input was null or more than 20 characters, it would not be added to the task list. This technique was also of significant advantage when writing the white-box decision tests.

By definition, statement testing exercises the executable statements in the code (Hambling et al., 2015). To implement this, I designed tests with inputs that would force my program to execute the statements within my code. An example of this was in my testDeleteAppointment() test where it provided the input to call deleteAppointment() from the Appointment Service class. This in turn called the indexOfAppointmentID() resulting in the execution of for loop that verified the ID’s as being truly unique.

Similar to statement testing, decision testing exercises the decisions in the code in particular ways (Hambling et al., 2015). This means testing for all possible outcomes when met with a decision statement. In my program, this was implemented in the testInvalidTaskValues(). This test provided invalid inputs, such character lengths that were too long and null values, that forced the “if” and “or” decisions to execute.

Testing coverage is an important technique to implement as it provides vital feedback. It let me know the quality of my tests by providing a coverage percentage of my files. These results allowed me to improve the tests by adding areas that were overlooked initially.

**Other Techniques**

There are plenty of useful techniques to employ when testing a programs functional and non-functional behaviors. However, they might not all be applicable for a given project. It is important let the type of project guide which techniques are used and to what extent they are exercised.

Since, I had no prior experience working with JUnit testing I did not implement the experienced-based technique. The experienced-based technique is best suited for someone who is well vetted in testing. It is their long history of working knowledge that enables them to write tests for issues that are common in occurrence. They are often issues that would likely be overlooked by a less experienced individual.

Another technique that I did not use in this project was state transition diagrams. The black-box state transition diagram technique depicts the behavior of a system using two symbols – a static state circle and a transitional arrow (Hambling et al., 2015). This is a technique that I could have been implemented but chose not to. System testing was also left out because it was beyond the scope of the project as it used local storage and my own operating system.

**Uses and Implications of Techniques**

Black-box techniques are great for looking at the functional and non-functional aspects of a programs given a set of specifications. It is particularly useful because the tester doesn’t need to know the minute details of the internal components or structure being tested. In other words, using black-box techniques, like equivalence partitioning and boundary value analysis, allows the testers to focus on achieving the desired behavior based on the specifications and not the developers code. This results in a greater chance that a problem will be uncovered and can be rectified earlier in the project. It also makes it a great option for projects that are more informal in approach, or for ones that only need to test specific functions or features.

In my opinion, statement testing, decision testing, and their coverage are techniques that should be implemented on all projects. They verify that internal structures, and code, of the program works as it should. It is an excellent way to measure the progress one has made toward their goal regardless of the development model used. The coverage results provide essential feedback to the testers so they know where to focus their efforts so they can achieve quality coverage. It also provides an opportunity to improve upon the existing tests, so they are more efficient. Neglecting to test the coverage, and functionality, of these decisions and statements could result in released escapements, defects, or all together failure.

**Reflection: Mindset**

**Caution**

I adopted two different mindsets when working on this project. The first mindset was that of a developer, whose goal was to write eloquent and correct code. However, this mindset made a significant transition as I entered the role of tester. My goal as the tester was to uncover as many defects as possible.

As a software tester employing caution is important not only when looking for issues but in communicating these findings to your team and the developer. I also needed to make sure that I had great understanding of what the requirements were before writing any tests. This approach allowed me to systematically approach the tests for more efficient performance.

In hand with understanding the requirements, I needed to be able to understand how the code interacted with one another as well as its’ complexity. For example, the TaskService class called upon information within the Task class. If I did not account for this interaction, I could have missed an important test like confirming that an ID of excess length or null value could not be added. Overlooking the complexity of updating a value in the Contact Service could have also resulted in values that were not tested as the method called upon others to complete its function.

**Bias**

Personally, I did my best to limit bias in the project by understanding that each role had a different set of goals. As a tester, the intent was to find defects and issues so that the code was as clean as possible. Any defects that were found, and many were, just enabled me, as developer, to continually improve upon the existing program.

Bias is tricky because it can sometimes lead to arrogance and willful ignorance. It’s understandable that a person would not want to admit failure out of fear of embarrassment, or in the case of a software developer, appear incompetent. In terms of testing their own code, this could easily lead to avoidance of certain tests because they know they will fail or uncover a flaw. I think that’s why it’s important for all team members to be humble and understand that failures are really just lessons presenting us with the opportunity to improve.

**Discipline**

As a person who holds themselves to high standards, it is difficult for me to even consider cutting corners. Yet, I can understand how some developers and testers, when under the pressure of a deadline, might succumb to these actions. I’m glad that this course has provided me with the data, if ever needed, to defend why a deadline was not met. For example, in the case that more time is needed to deliver truly clean code, the team can be assured that we are saving time and money in the long run.

Being aware of the impact my role as software developer has on the outcome of a software free of defects is just one way I plan to avoid technical debt. This awareness, along with being able to take on the mindset of a tester, will enable me to write code that accounts for as many inputs and scenarios as possible. Ideally, this will result in high-quality code that reduces the technical debt and eliminates the need for a rewrite.

**References**

Garcia, B. (2017). *Mastering Software Testing with JUnit 5: Comprehensive Guide to Develop High Quality Java Applications*. Packt Publishing.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide* (3rd ed.). BCS Learning & Development Limited.