**Project Two**

Justin Starr

Department of STEM

CS 320 – Software Test Automation & QA

Dr. Ramsey Kraya

October 15, 2023

**Project Two**

**Summary**

As a software engineer for Grand Strand Systems, a mobile application has been created for a client. As part of its deliverables, it was required that Junit tests be written for each of the application’s classes that focused on testing the back-end of the mobile application. The following is an analysis of the various approaches to software testing based on the requirements and appropriate testing strategies that meet such requirements.

The approach taken for unit testing for each of the three features of the application began by first gathering the requirements of the features to have an understanding of what each feature was supposed to do. I was then able to create each that was required for the feature. Each feature had a class specific to the feature object, Contacts had a contact class, tasks had a task class, and appointments had an appointment class. Inside each of these classes, I implemented attributes that the objects would have based on the feature requirements. Contacts, tasks, and appointments had to have a unique ID String that could not be longer than 10 characters, could not be null, and not be updatable. Contact’s had to have a first and last name string that could not be longer than 10 characters and they could not be null but were updateable. Contact’s also had to have a phone string that had to be exactly 10 characters and could not be null. It also had to have an address field that could not be longer than 30 characters and could not be null. Both the phone and address were updateable. Similarly, task objects had to have a task name that could not be longer than 20 characters and it can not be null. Tasks also required a description string that could not be longer than 50 characters and could not be null. Task names and descriptions are updateable. Lastly, appointments had to have a date that could not be in the past and the date could not be null. Appointment objects also had a description string that could not be null and it could not be longer than 50 characters. Once I had created each of the three feature objects, I was able to incorporate lines of code that would throw an exception if any objects were created that did not meet the specifications.

After creating the class for contacts, tasks, and appointments I then began writing their associated service classes. Each of these classes had specific requirements that the service needed to be able to do. The ContactService class needed to be able to add contacts with a unique ID, delete contacts per contactId, and be able to update contact fields (firstName, lastName, phoneNumber, and address) per contactId. Similarly to the ContactService class, the AppointmentService class must be able to add tasks with a unique ID, delete tasks per taskId, and per taskId be able to update a task’s name and description. The AppointmentService class had to be able to add appointments with a unique ID and be able to delete appointments per appointmentId.

After completing the code for each of the features, I then began writing Junit tests that tested each unit of each class. The aim of the tests that were written for each of the classes was to try and prove that the code did not work. Each test that was written was a specific instance of whether or not the code handled errors correctly. For example, when adding contacts, tasks, or appointments it was necessary to first create a new object, and then create a second object with the exact same ID to see whether or not an exception would be thrown. The test expects that an exception would be thrown and because one is thrown when this happens, the test therefore passes. In this case, if no exception had been thrown we would know that it was a bad test and that there is something wrong with the code. The code is not performing properly. Tests also included testing the scope of each object’s attributes. If a uniqueId could not be more than 10 characters, we needed to write a test to see what happens when an ID is less than 10 characters, a test for when an ID is exactly 10 characters, and a test to see what happens when an ID is greater than 10 characters. By writing tests using this approach we see that the approach was significantly aligned to the software requirements. Without requirements, there would be nothing to test for. The program could essentially operate however the developer wanted without regard to the client's needs.

While conducting Junit tests, the classes can be run for line coverage. This tells the developer how much of the code has been tested and helps to identify what portions of the code have not yet been tested. Seeing the coverage can help identify portions of code that may be unused or are written incorrectly. When we achieve 80 percent or better in our test coverage we know that the code is being tested well and that our tests are effective. If we have below 80 percent coverage, this means we have not adequately tested the code or that the tests we have written are not effective. For our client's software, it was essential to get the highest test coverage possible, and in the case of this application, we were able to achieve 100% coverage for all of the features of the application.

The way I was able to ensure my code was technically sound was by ensuring that the code included exception handling. One of the biggest ways this can be seen throughout the code is when objects are created or added through the associated methods for example the addContact method creates a new contact if the ID is unique. The contact class will throw an exception if the contactId is null or if it is greater than 10 characters (Line 16 of Contact.java). We see the same for tasks on line 13 of Task.java and line 11 of Appointment.java. Similarly, in ContactService an exception will be thrown if the contactId is already in use (Line 30 of ContactService.java). We see the same as it applies to tasks on line 25 of TaskService.java and for appointments, we see it on line 30 of AppointmentService.java. The same applies to all of the Contact’s attributes, an exception is thrown if any of the attributes do not meet specifications (Lines 19, 22, 25, and 28 of Contact.java). For tasks, lines 16 and 20 of Task.java, and for appointments, lines 14 and 17 of Appointment.java. Junit tests were written that specifically test each of these exceptions and this helps to prove that the code is technically sound. For example, lines 27, 33, 39, 46, and 52 of ContactTest.java all test the possible scenarios of a contact being created based on the contactId. They test what happens when the ID is null or not null, if it is too long, if it adds an ID that is exactly 10 characters, or if it adds a contact when the ID is less than 10 characters. Similar tests were written for every class and their associated object attributes which all help to ensure the code is technically sound.

Similarly to showing that the code was technically sound, we ensured that it was efficient by incorporating more exception handling for all of the methods of each feature. Junit tests were then written to show whether or not the methods performed the way they were supposed to perform. These tests were written for the service classes of each feature. Similar to how tests were written for objects and their attributes, it was important to test to see that methods would not work if they contained strings that did not meet requirements. For example, line 88 and line 102 of ContactServiceTest.java test what happens when the program tries to add a contact with the same ID. In both tests, an exception is thrown because it is adding a contact that has an ID that is already in use. These types of tests continue throughout all of the service features which helps to ensure the code is efficient.

**Reflection – Testing Techniques**

The software testing technique employed for the project was Junit testing, a form of white-box testing where the test cases were based on the internal structure (Jakubiak, 2022). For the project, I was able to choose inputs that exercise specific paths through each of the classes, and I configured assertions that would then validate the output. Another way that white-box testing was used was to test component structures, which at this level, program structures such as program statements and decisions were tested (Hambling, B., et al., 2019). Statement tests that were written exercised the executable statements in the code. It forces the program to execute specific statements by setting the program to a start state and then giving it input that forces the program to execute the required statements (Hambling, B., et al., 2019). Similarly to statement tests being conducted, decision tests were conducted to test the ability of the program to make appropriate decisions based on input that the tests were given.

Software testing techniques that were not used were black-box testing. This form of testing is done when the tester does not know how the code works or what the structure of the code is that is being tested (BrowserStack, 2023). Black-box testing examines the external behavior of the software and then determines, based on specified requirements, whether the software performs or functions as it should (BrowserStack, 2023). Creating JUnit tests can sometimes be thought of as black-box testing; however, it depends on how the tests were created. For this project, we did not consider this black box testing because we were concerned about the inner workings of the code. The tests were created after each class was written, which then proved it met the requirements or specifications and functionality. Had the tests been written first, then we could say that we absolutely developed or created the tests, first, based solely on the requirements documentation.

When considering when it would be useful to implement white-box testing, its practical uses, and its implications for different software development projects and situations we would consider situations where it is essential to know what the inner workings of the software are. It is helpful when trying to identify problems with the developer's code and if it does what it is supposed to do like making the correct decisions at the correct times. It can help to identify code that is necessary but perhaps written in a way that is unreachable. This helps ensure that all parts of the code are tested. Another way white-box testing is useful is that it can help identify defects early on in the developmental process, which can help to save money, and also depending on the type of system being developed, it can help minimize the risk of catastrophic failures that could result in projects being scrapped or even worse, the potential for it to cause harm.

Black-box testing is practical in situations when we are concerned with user-focused testing (Sereda, I, 2023). This is essentially making sure that the software performs how users would expect the software to work, for example when testing the software’s UI (User interface). Again, black-box testing tests the behavior of a system and isn’t concerned with the inner workings of the software or its code. Another example of when black-box testing could be beneficial to a project is if the project is running out of time and needs to be able to have testing conducted quickly. This again, is possible because tests are written not based on the code but rather on if the software does what it is supposed to do based on behavior. Because black-box testing requires less technical expertise, it can be performed by testers with less technical knowledge (Sereda, I., 2023).

**Reflection – Mindset**

The extent to which caution was employed while acting as a software tester was knowing that I needed to try to prove that the software did not work. By trying to accomplish this, in the end, I end up showing that the software does work. The caution existed from the very first test that was written to the last. I knew that the more tests I wrote the better the code would be so I had to ensure that I was writing as many tests as possible that would test the scope and limits of the functionality of the code. In fact, in the beginning, I thought that I had written the code correctly and that I had also written all of the appropriate Junit tests that verified this. After finding out that certain aspects of my code were in fact incorrect and also that I had not adequately tested every functional aspect of the code, the more I realized I needed to exercise more caution to ensure I would arrive at the desired result. Using more and more caution throughout the project enabled me to identify areas of the code that were not working correctly, which in turn also helped me to write better and better tests.

Because I was the developer who created the code, admittedly, there did exist some bias not only while I wrote the code but more importantly when I wrote the JUnits tests. In the beginning, I had thought that just because I had 100% coverage, I thought that had also meant that I had written enough tests for the code. It was throughout the course with much re-work that I started to realize that there were problems with the code itself and also that I had not written enough tests for each of the classes. This illustrates that as a developer who might write their own tests for their code, bias can and will exist. I have learned a valuable lesson to be more open-minded when writing my code and to realize that it is possible and likely that I will have problems. It is essential to keep an open mind when writing code and to think of as many possible outcomes to situations so that the programs I write are not only more effective but that they are technically sound and efficient.

The reason why it is important to be disciplined in my commitment to quality as a software engineer is that I would not want to produce a product that leads to some sort of failure. Many systems that exist today are systems that people rely on for safety. When these types of systems fail they could result in harm including but not limited to the loss of life. When corners are cut, especially as it concerns testing software, problems or faults in the software can be missed leading to tremendous failures. No matter what software we are developing, whether life may be dependent on it or not, we also want to ensure we are meeting the requirements of our clients. For example, it is essential that if were are writing software that handles sensitive information, security is taken into consideration so that the information is adequately protected. If information is not protected, not only could harm be caused to the client but also to the people who rely on the client’s software. Therefore, no matter what software is being developed, it is essential that developers keep an open mind, and stay disciplined not only in the production of code but also when testing it to help ensure that only the best working products are delivered to live environments. One way that I plan to avoid technical debt as a practitioner in the field is to always ensure I am following requirements and that during the requirements gathering phase of the SDLC, I attempt to gather as many requirements as possible. Also, I intend to keep an open mind to try and think of ways that my code could potentially function improperly to help avoid defects that can lead to failures (Hambling, B., et al., 2019).

**References**

BrowserStack. (2023, July 31). *What is Black Box Testing: Types, Tools & Examples*. App &

Browser Testing Made Easy. <https://www.browserstack.com/guide/black-box-testing>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing*

*: An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

Jakubiak, N. (2022, December 6). *JUnit Tutorial with Examples: Setting Up, Writing, and*

*Running Java Unit Tests*. Parasoft. <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

Sereda, I. (2023, July 5). *Black-Box vs White-Box Testing: The Pros and Cons of Each*. Solvd.

<https://www.solvd.com/blog/black-box-vs-white-box-testing>