**Project 2**

**Jonathan Wolanyk**

**CS-320: Software Test Automation & QA**

**Professor Linda Gaston**

**August 14, 2022**

**Summary**

To develop the backend of an application to the standard of the client, aligning the development and tests of a project to the given software requirements is integral. While developing and testing back-end services for Grand Strand Systems, I began by copying each software requirement and pasting each requirement into the appropriate areas of code as comments. By placing each requirement verbatim into my code, I was sure to align the development of the program and tests to the software requirements. Examples of this practice can be found throughout my submissions since I leave the requirements as comments in my code. A specific example of this practice can be found in my contact service application from Module Three. One requirement for the contact class was that “The contact object shall have a required unique contact ID string that cannot be longer than 10 characters. The contact ID shall not be null and shall not be updatable.” I had an if statement to validate each contact ID to develop my program around this requirement. If the passed-in contact ID were too long or less than one character (null), it would throw an illegal argument exception. By throwing an illegal argument exception, I developed tests around the requirement to validate that the proper error was thrown when invalid IDs were passed in for the contact ID. Each software requirement throughout the course was implemented in the way mentioned above.

My JUnit tests were crafted to test each function created to address the software requirements. The goal was to ensure that each function was covered by at least one JUnit test. Whenever possible, I isolated the test to one feature at a time. This practice made troubleshooting easy since, when a test failed, it was simple to determine what had to be altered in the code. Additionally, isolating tests produced fewer false passes or fails than bundling tests. To create high-quality tests, I ensured no prior data would enter each test. For example, I utilized the Singleton Pattern in my program to maintain consistency across various services meant to track a mutable list of entries; clearing this list before testing was essential to prevent false passes or fails. As a result, I cleared the program’s ArrayList used to track such entries before each test began. This practice ensured that tests were isolated and provided accurate results. These practices provided a coverage percentage of 88% across the entire project, with most classes being covered well over 90%. After reviewing what was not covered by my JUnit tests, I determined that further coverage would be redundant. Most of the missing coverage related to internal workings of functions that were not relevant to the workings of the overall program.

**Reflection**

I utilized various software testing techniques throughout the project to eliminate errors and ensure the software requirements were met. The primary technique used for testing requirements was Whitebox testing. Whitebox testing involves the test constructor being aware of the source code and tests for specific scenarios. Since I was both the creator and tester of the code, Whitebox testing was the natural choice for testing. An instance of Whitebox testing in the project exists in the contact service class. I created unit tests to determine whether the program throws errors for invalid inputs using my knowledge of the aforementioned if statements to test passed-in values.

Some testing techniques were not used throughout testing. An example of such a technique would be exploratory testing. Exploratory testing is the practice of using a program as intended to discover errors. Since the program developed throughout the class was concentrated on the back-end exclusively, the program could not be operated as intended by the end-user. This testing can be helpful when a code base is large, resulting in combinations of actions that would not be obvious to test constructors. Through exploratory testing, errors that would not have been discovered through tests alone can sometimes arise. A secondary testing technique known as acceptance testing was not employed either. Acceptance testing involves having end-users test a program and determine whether it meets the client's requirements. Since the client, in this case, is theoretical, this testing technique cannot be used.

Caution is important while developing projects, especially as the project scales. In the case of my program, I was careful to separate classes based on the software requirements. An example of this exists within my contact service and contact classes. Initially, I considered having the validity of inputs for a new contact takes place in the contact service class. Once validated, I would have had the values passed into the contact class. However, the software requirements guided me to make the contact class validate passed-in values rather than accept whatever values were given from the contact service class. This kept the classes encapsulated and closer to the given software requirements.

While developing JUnit tests for my program, I was careful to develop tests that validated each section of my code with all possible inputs. I had some functions initially I felt could have been refactored to account for a broader range of inputs and ensured that my tests would prove or disprove whether these functions were sufficient. An example would be in the appointment class where I tested whether a date had passed using a deprecated version of time in Java. My code felt as though it was barely held together after searching through the documentation on how to use the outdated input for time. When constructing my tests, I ensured that past, present, and future dates were given to test whether my code could handle such requests appropriately. It would have been easy to simply test future dates and maintain 80% or better coverage, but I wanted to ensure my code would satisfy the requirements of the hypothetical client.

Discipline is an integral characteristic of an exceptional software engineer. The nature of software development is such that problems constantly arise that tax one’s brain. Over time, it can be quite easy to overlook problems at the moment with the promise of refactoring later. This approach can cause problems across numerous areas (security, efficiency, compiling). If one is disciplined and follows software requirements throughout the development process, programs are developed linearly and efficiently. Part of this development methodology includes developing tests early that test new code. Rather than creating an entire program and saving testing for last, it is beneficial to test a program throughout to address bugs and errors. An example of cutting corners lies in the disaster of Ariane Flight V88. In the tragedy, a rocket carrying European Space Agency research satellites self-destructed after an error linked to the reuse of code from a previous iteration of the rocket was used for the final product (Ariane Flight V88, 2022). Many developers fall into the trap of copying and pasting code in their projects, only to face compiler issues or logic errors later in the development process. By using discipline and a consistent design approach, programs are developed that stand the test of time.

**References**

Wikimedia Foundation. (2022, May 29). Ariane Flight V88. Wikipedia. Retrieved August 11, 2022, from https://en.wikipedia.org/wiki/Ariane\_flight\_V88