**JUSTIN CROUCH – Module 7: Project 2 Reflection**

The testing approach was aligned well to the software requirements. For example, the software requirements stated that the contact class shall have a mandatory first name with type string that is no longer than 10 characters and never null. There are numerous tests that check for these requirements, such as setting the first name of a contact object to a string of exactly 10 characters, exactly 11 characters, an empty string, as well as null. The tests assume that any supplied first name that do not fall within the requirements is set to the string “NULL”, since the requirements do not specify what should happen to invalid first names. Similar testing is done for the other requirements, with the assumption that a default string value of “NULL” (or “2222222222” for a contact’s default phone number) should be provided to invalid arguments.

The JUnit tests were effective based on coverage percentage due to using black-box testing techniques, such as boundary value analysis, and white-box testing techniques, such as statement coverage testing. Take, for example, a contact’s phone number requirements: a contact’s phone number must be 10 digits long, with type of string, and not be null. After using boundary value analysis, three tests check for supplying a phone number with a 9-, 10-, and 11-digit string. After using statement coverage testing, two more tests check for supplying a phone number with a null and empty string value. Although thorough, the current tests do not check for some unwritten requirements, such as verifying if a supplied phone number is, in fact, a string of numbers and not a string of 10 random characters.

The code was ensured to be technically sound by following Object-Oriented principles. For example, in the Contact.java class file, lines 19 to 30 declare private member variables. These lines showcase the principle of encapsulation, meaning that these variables are contained within the class and hidden from users outside the class. In the same file, there are various mutators and accessors that reference these private member variables. For example, lines 77 to 80 declares the Contact class's getLastName accessor method, which returns the initialized Contact’s last name attribute. These mutators and accessors are examples of abstraction, where a user can interact with the object in expected ways without knowing how it is actually accomplished.

The code was ensured to be efficient by utilizing a hash map data structure in the ContactService.java and TaskService.java class files. In these class files, a hash map stores the ID of a contact or task object, respectively, with the reference to the respective object of that ID. A hash map was chosen due to its O(1) performance when inserting and accessing elements, no matter how many elements are stored in the data structure (*HashMap*, 2025). This allows for the service classes to efficiently and quickly store new objects while also being able to quickly access any element with the object’s ID. In ContactService.java, line 52 uses the hash map to check if a Contact’s ID is already stored, while line 58 uses the hash map to insert a new Contact object with its ID as its key.

Each milestone utilizes the same testing techniques to ensure effective JUnit tests for their respected milestone. To begin, the Equivalence Partitioning testing technique is used to group similar input and output values together, which practically reduces the number of tests required for proper coverage. By combining similar test conditions, Equivalence Partitioning prevents the need to test for every possible input and reduces the number of necessary tests to one value that fall within each test case (Hambling, 2019, pgs. 95-97). For example, in Milestone 1, the requirements for the Contact class’s first name are stated as being a mandatory string value with length no longer than 10 characters. The input groups to test for are valid strings with length 1 to 10 characters, invalid strings with length 0 and above 10 characters, as well as an invalid non-string value. In addition to Equivalence Partitioning, the Boundary Value Analysis technique was used, which consists of testing values that lie on either side of the range of valid inputs (Hambling, 2019, pgs. 98-99). This technique verifies that edge cases are handled correctly within the program. For example, a program that returns if water is liquid at some temperature, given in Celsius as an integer, should be checked for proper classification when given temperatures -1, 0, and 1 at the lower boundary, and temperatures 99, 100, and 101 at the higher boundary.

Although numerous testing techniques should be used for testing software, there were two techniques that were not entirely appropriate to the milestones. The first technique, Decision Table Testing, is a testing technique that uses a table to analyze what action a program should perform given a set of conditions. The collection of various conditions to actions, appropriate to the business requirements, are labeled as business rules (Hambling, 2019, pgs. 99-100). Decision Table Testing is effective at finding every test case that fall under the business requirements since each business rule defines what each condition state must be to reach a specific action that the program performs. The other technique, State Transition Testing, involves testing the output provided by a change in some input. This technique is usually performed by following a State Transition Diagram (Hambling, 2019, pgs. 100-102). A common use case of this technique is checking the flow of a project’s various UI states. For example, a game tester begins on the main menu of a videogame. Then, she interacts with the onscreen *Options* button and verifies that the *Options* screen is displayed. From there, she verifies that the onscreen *Close* button transitions back to the main menu. The game tester continues this process until each state and corresponding transition within the provided State Transition Diagram is reached.

When acting as a software tester for this project, I employed caution during specific steps of development. Specifically, I was cautious when writing JUnit tests since these tests would guide my understanding and development of the actual production code later. Also, the JUnit tests had to be specially crafted so to effectively and efficiently verify the functionality of the production code components, which was aided by thoroughly appreciating the interrelationships of the tests and respective production code.

I limited the amount of bias in the self-review of the code in varies ways. For one, I measured the success of my code by how closely the functionality matched the requirements given for this project. This allowed me to be more objective when analyzing the code for defects. However, this technique did not push me to write more performant code. To help with this issue, I thoroughly reviewed my code after a few days of writing it, allowing me to analyze with a fresh mind and prevent bad performing code from remaining in the codebase. I believe bias is a concern when tasked with testing your own code since little to no new ideas are being generated, which can lead to unseen defects being pushed to production.

Being disciplined in producing high quality software is highly important. Not only is producing high quality software within the job description of a software engineering professional, but low quality software can lead to risking the lives of people and generating hundreds of thousands of dollars in general damage. As a software engineering practioner, I plan to avoid technical debt in a few ways. For one, when a defect arrises within a project, I plan to eliminate that defect as early as possible to prevent it from imbedding itself deeply within the codebase. Also, if at any time during development that I feel overwhelmed by my responsibilities, I plan to obtain help from my team so to not add friction to the project’s progress.

**References**

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.

*HashMap (Java Platform SE 8)*. (2025). Oracle. <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>