**Project Two**

Brandon Cook

Southern New Hampshire University

CS 320: Software Test, Automation & QA

Dr. Cross

October 20, 2024

**Project Two**

The contact, contact service, task, and task service class all call for very specific requirements. For contact and task, variables need to be initialized and checked that they are not null, and they do not exceed a certain length requirement; with the exception of the phone number variable in contact, which requires specifically 10 characters. My testing approach ensured that these requirements were met. For example, in TaskTest.java, I tested each variable to ensure that an exception was thrown when any variable is either set to null or exceeds the maximum length. An assertion verifies that, when an object is created with either a null or a value exceeding length requirement, an exception is thrown. The tests also verify that when an object is created correctly, each variable is assigned with the correct value and the object exists. A similar approach was taken in the service tests, ensuring that multiple objects could be added to a list of objects, that they could be deleted from a list and that the list was updated accordingly, and that values in each object could be updated individually while still meeting the requirements for the specific variable.

My tests are effective in covering all of the requirements. As explained above, they ensure that each requirement is satisfactorily implemented. All requirements were tested separately, avoiding bad practices of testing different functions simultaneously. Test cases were carefully crafted for accuracy, and proofread to ensure that there were no errors that would lead to a false positive or negative. If a test came back negative, there was an error in the original code that was being tested. By taking this much care, it led to technically sound code that is free of any major issues. For example, here is the original code for adding a new task to a list:

A close-up of a computer code

Description automatically generated

This code takes the arguments needed to build a new object and adds it to an in-memory list. The test for this code:

A computer screen shot of a task

Description automatically generated

This test not only tests that a new task can be added to a list, but it tests it on two separate tasks to ensure that the in-memory array list is working as intended. Additionally, this example shows the efficiency of my code as well. In the test cases, there is a @BeforeEach clause that will establish a new TaskService object before each test, which is deconstructed following each test. This prevents objects from lingering in memory following the tests, as shown below.

A close-up of text

Description automatically generated

The software testing techniques employed for each milestone fall under both black-box testing and white-box testing. While I did not employ every testing strategy from each, a combined approach to testing ensured better coverage. The black-box testing technique that was used is equivalence partitioning, which is a technique where a single value is chosen to represent an entire group of results (Hambling et al., 2019); in the case of my milestones, the values needed to represent true or false to indicate whether the correct value was returned or not. From white-box testing, decision testing was employed, which was used to determine that, when given the correct input, the program produced the expected output. It tests that the decisions and logic in the code are correct.

The testing techniques not employed include boundary value analysis, decision table testing, state transition testing, use case testing, and statement testing. Boundary value analysis, in retrospect, may be beneficial to include to ensure that edge cases work as expected. Testing the boundaries requires testing values surrounding the edges of expected values (Hambling et al., 2019); in the case of my milestones, for example, if a variable required no more than 10 characters, boundary analysis would require testing the response with 10 characters and 11 characters. Decision table testing is better suited for more complex logic and decisions, where multiple factors are taken to determine the output. In its current form, the program does not transition between states, which is what state transition testing covers. Statement testing exercises every line of code, which is better suited for larger programs with many lines that may not always run. It is used to determine, when the lines of code run, that they work as expected.

Each testing technique has its own benefits and uses, which are not necessary for every project. Employing any of the strategies should be justified with what the system is required to do. While extensive testing would be ideal, it may not always be possible. Possible test cases should be analyzed and reviewed to determine the ideal approach.

When writing this program, I took care to ensure that I was meeting all of the requirements. Caution was employed, meaning that I triple checked my test cases and code to make sure that the original code being tested met all of the requirements. It is very easy to make a mistake when writing test cases and overlook some logic that leads to either a false positive or negative. In order to prevent this, I first started by writing down each test case on a piece of scrap paper, which gave me a point of reference when working on the project to look back on and compare to the final tests. There were a few instances where, for example, I was testing if having the value of an object name being too long would throw an exception, but it was throwing the exception due to invalid input in another field.

Bias is a difficult concept to overcome when reviewing your own code. It wasn’t something I could do the same day I wrote the code; I already knew the ins and outs of the program and the lines of code that I wrote, so I was more partial to defending my choices. Taking a few days away and returning to it gave me a more impartial perspective on the code I wrote, and allowed me to look at it with a more subjective mindset.

Cutting corners when writing code can lead to unintentional errors. This can be a problem if you end up with logical errors in your code which can be much more difficult to troubleshoot. Care should always be taken when writing code; speeding through functions and tests will accumulate plenty of technical debt later on. This can be costly and time consuming to fix. It is much easier and more efficient to fulfill requirements the first attempt rather than later. Fixes and troubleshooting should be performed for the reasonable oversights and mistakes that weren’t due to excessive speed and a lack of care and attention.

**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. <http://ebookcentral.proquest.com/lib/snhu-ebooks/detail.action?docID=5837074>