CS 320 Project 2: Reflection

To summarize the approach used to test for each section of the mobile application, I started with ensuring that the requirements were understood to the best of my comprehension. For each part of the application, this meant that the bounds of the character limits, types, or uniqueness were understood. To use an example, the appointment class needed to have a date type in its construction. The date could not be in the past or null, so a check had to be created in the constructor to verify it was valid and for the future. If an invalid parameter was entered, an exception was created to be thrown to stop the program from proceeding. When designing the JUnit tests, I made sure to verify that for each of the methods created they correctly threw the exceptions and stored properly entered data. To ensure that this covered as much as possible, I repeated this for each and every method in both the constructor and service classes. There was at least one Junit test for each and every method and service.

While creating the different test classes for the application, I needed to make sure that the code was technically sound and efficient. To do so, I ensured that every parameter for each class was accounted for in the JUnit tests in as few tests as possible. As an example, when testing the task construct, I made sure that each parameter was filled with something that the construct should accept, as shown here:

@Test

**void** testTask() {

Task test1 = **new** Task("12345", "Test1", "This should work");

*assertTrue*(test1.getId() == "12345");

*assertTrue*(test1.getName() == "Test1");

*assertTrue*(test1.getTask() == "This should work");

}

To be efficient, I made sure to test each parameter in the constructor in the same test. This is also shown when testing failure conditions, as shown in the following snippet:

@Test

**void** testFailedTask() {

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Task("12345678910", "Test1", "This should work");

});

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Task("12345", "This should not work for this test.", "This should work.");

});

Assertions.*assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Task("12345", "Test1", "This should not work for this test to be a success.");

});

}

The test was designed to consider the outer bounds of each parameter and show that exceptions would be thrown if exceeded. As with the previous example, multiple parameters were tested in the same test.

To test the methods and classes created for this project, I primarily used unit testing. Unit testing is a dynamic test that tests the parameters of pieces of the code. This was done through creating Junit test cases which would check the parameters given through the requirements. Testing these pieces of code ensures that as the project scales, the individual pieces making it up would be verified as working, potentially saving time down the line. The most common technique used during these tests was that of boundary value analysis, where I took the highest value, lowest value, and a value in between and verified each case produced the correct result.

A form of testing that was not implemented was that of acceptance testing. Another dynamic test, acceptance testing is typically done to ensure that the product is up to the customer’s specifications. Usually, an alpha or beta is based off a completed build of a project and would be able to show that the product meets those requirements. As the scope of this project did not include a functional build with an interface, this kind of testing would be unnecessary. Acceptance testing is something that is useful when designing, say, a mobile game application. The beta would be a near completed product of the game, used to find any instances of the game not performing as expected. This could include anything from buggy code to setting proper parameters, like if it is too easy or hard.

As I was working on the project, I had to remind myself not to skip steps during the design of the unit tests. It would be easy for me to justify to myself that the code would work, but there were many points where the testing proved that the code was not up to par. For example, using the array list library. This library, while relatively straight forward, I found had a few peculiarities that if I did not test would have gone unnoticed, as the code compiled but did not perform as expected. By taking the time to design a test for the basic addContact function, I was able to catch that my original code did not properly add the contact objects to the created array. Similarly, reminding myself to remain cautious allowed me to go through the design process in a much smoother fashion.

To continue with the testing mindset, I also needed to limit my biases as much as possible. In continuing from the previous example of the array list, I wanted to assume that the library worked as I expected and skip the test. It was not until I designed the test that I realized I was about to make a relatively big mistake that would not have been caught until the app was presumedly released. To continue with this mindset, even though I was confident in my ability to create a constructor, I designed a test for each parameter in each constructor to verify the boundaries. Through those tests, I was able to determine that each boundary performed as expected, and kept the code clean as well.

To remain disciplined in coding is to remain in control of the expected outcome. As more corners are cut to save time, the more opportunities there are to make mistakes. These mistakes can be easily overlooked the more lines of code are written. In addition, they can be compounded as more and more code could potentially be built on top of those mistakes. By time the product is ready to be delivered, these mistakes could be so embedded in the code that one would have to almost rebuild the product completely. Simply taking the time to test at all points of the development can save much in time and costs for any project.

References

*SOFTWARE TESTING : an istqb-bcs certified tester foundation guide.* (2019). Bcs, The Chartered Instit.

Boni García. (2017). Mastering software testing with JUnit 5 : comprehensive guide to develop high quality Java applications. Packt Publishing, Birmingham (Gran Bretaña.