**SNHU CS 320 - Project 2**

Salah Bedoui

1. **Summary**

My testing approach aligned perfectly to the software requirements as the test cases covered all the methods implemented in the classes that needed to be tested. For each class, the different constructors are tested, and by association the getter methods are tested to ensure all parameters meet the requirements. Setter methods are also tested to ensure that each new or updated value meets the requirements. Test cases make sure that an exception is thrown if any value is null or does not meet the number of characters’ limit.

The overall quality of the tests is good. This is demonstrated by a perfect coverage of 100% of JUnit tests for each test class. While the Contact and ContactService classes showed a 100% coverage after performing the test for the first time, Junit tests for the task, TaskService, Appointment and AppointmentService classes showed coverage rates of 87% and 97%. I added few lines of codes specifically to address the cases of null parameters to achieve a perfect coverage percentage.

To ensure that my code was technically sound, I first declared constant variables to hold values for the length’s limit and other variables to hold initial values for the default constructor. This ensures that the code is more understandable as it grows longer and helps avoid mistyping these values if they you were individually inserted later in the code.

With the help of the Eclipse IDE, I made sure to import any package needed for the good functioning of my tests. I tried to keep my test cases concise and follow a defined logic that ensures all requirements are met. For example, I always started by testing the constructors, grouping all parameters with *assertAll* and using *assertNotNull* and *assertEquals* depending on the constructor to be tested.

To ensure requirements are met, any parameter is tested against all its requirements. For example, the first name is tested to ensure it is not null assertThrows(IllegalArgumentException.class, () -> contact.setLastName(null))

and at the same that it does not exceed the length limit assertThrows(IllegalArgumentException.class, () -> contact.setLastName(tooLongLastName))

For efficiency, I declared testing variables such as firstNameTest and lastNameTest, tooLongPhoneNumber, and tooLongAddress, and initiated them with corresponding values to use them each time a requirement is tested. To further improve readability of my code and improve the efficiency of my test cases, I tried to group logically related tasks in the same test cases. For example, in the ContactTest class, when writing a test case for updating the first name, it checks if the first name is null or is too long. So, with one test case, we can test multiple requirements.

*@Test*

void updateFirstNameTest() {

Contact contact = new Contact();

contact.setFirstName(firstNameTest);

*assertAll*(

"first name",

()

-> *assertEquals*(firstNameTest, contact.getFirstName()),

()

-> *assertThrows*(IllegalArgumentException.class, () -> contact.setFirstName(null)),

()

-> *assertThrows*(IllegalArgumentException.class, () -> contact.setFirstName(tooLongFirstName)));

}

1. **Reflection**
   1. Testing techniques

The application at hand is a small and simple one that does not require the use of many testing techniques. For each of the milestones, I essentially used specification-based (Black-Box) techniques such as equivalence partitioning and boundary value analysis techniques. Equivalence partitioning organizes in “chunks” similar input groups. How these chunks and partitions are defined depends on the software requirements. It can be such as negative and positive numbers form two separate groups, or different number ranges define different partitions for an age-based software content. We can then test one value from each group and assuming every other value in the group will behave similarly. Boundary value analysis ensures that inputs at the boundaries of each partition are tested as they might cause bugs or strange behavior. In the case of our milestones, there were many inputs’ length limits for different parameters. I used equivalence partitioning to test this requirement. Since these values cannot be null, negative values and zero are excluded and we can have two partitions, with the first ranging from 1 to the length limit and the other with values above the limit. A 100% coverage can be achieved for this requirement by only testing one value from each partition. In the specific case of the phone number, three different partitions were needed. Since the phone number needed to have an exact length of 10 characters, we needed a first partition of numbers from 1 to 9, a second partition for the number 10 and another partition for numbers greater than 10.

As I mentioned earlier, this application is relatively short and simple. It did not require extensive testing and the use of many testing techniques. Other techniques were not needed such as decision table and state transition testing. Decision tables can be useful when dealing with a combinations of input conditions that result in different behavior and outputs. A decision table is structured into rows that list all input conditions and columns that list all the possible actions.

Another technique that was not used is the state transition testing. This technique is useful in applications and systems where outputs depend on changing input conditions or the “state” of the system. For example, a button on a watch might have different behavior depending on the mode the watch is in when pressing it. An Edit mode should have a different behavior allowing to set the time, while a regular mode should trigger no output, or a different action not related to setting the time.

The above-mentioned techniques are widely used, but the use of each one of them is needed based on a case-by-case basis depending on the software size, complexity, and requirements. Equivalence partitioning is useful when dealing with a large number of possible inputs that cannot be individually covered. Dividing all these possible inputs into groups ensures an important coverage percentage by only testing few values from these groups.

On the other hand, decision table and state transition testing can be very useful in large and complex systems and applications when different inputs combinations or conditions might trigger or result in different outputs.

* 1. Mindset

When writing my tests, I adopted caution throughout the process to avoid errors. The Eclipse IDE was helpful by highlighting potential errors and problems and with its color coding of keywords, class, method, and variable names. One technique was particularly useful: whenever I applied a change that might affect other parts of the code (especially with method and variable names), I used the “refactor” function in Eclipse to ensure that the change applies wherever necessary. In addition, I used regression testing by running the tests again every time a change was made to the code to ensure that these changes did not affect the expected behavior of the software and the test were still successful.

One of the main challenges a developer might face is when they are tasked with testing their own code. Bias can diminish the quality of their tests and even lead to unnoticed escapements and defects. I tried to adopt a dual personality mindset where I wear a developer’s hat when developing code and a tester’s hat when writing tests. This partition of roles is important as it helps not only in limiting bias but also to derive tests from the software requirement specifications rather then writing them just to pass the already written code. With this strategy, when writing code as developer, the goal is for the code to pass the tests and not the opposite. An analogy to this mindset is when debating on a subject, one should start from arguments that will lead to a conclusion and making an informed opinion rather than believing in some ideas and trying to prove them right with no matter pieces of information you can find (even if they lack credibility).

In general, discipline is a crucial component for any successful process. In an extremely fast-moving tech scene, business driven decisions might place developers on the hot seat to deliver fast results. This may lead to developers trying to cut corners and lowering their guard to meet their deadlines. But at the same time, it increases to risk of technical debt, which might turn out to be much more costly at the end. In Module 7, many examples were introduced of released escapements and defects that led to huge financial losses and human fatalities in some cases. These escapements and defects might have been avoidable in some cases with more careful testing. Humans are never cautious enough. Discipline will help avoid errors and defects as much as possible. As mentioned earlier, writing tests based on the software requirement specifications will not make developers’ job easier, but it will ensure a more efficient testing strategy and higher coverage percentage, and as a result a more reliable software.