**1-Summary**

The contact Service class, the task service class, and the appointment service class totalize eight requirements. Each class's constructor defines a list and implements a method to add, delete, and update based on a unique identifier. The Unit test has considered the success and the failure case per requirement; in other words, a total of sixteen test cases developed to ensure that all three objects will run as expected. I counted eleven requirements for the class contact, task, and appointment for twenty-six test cases. For example, I designed the failure and the success case for the requirement the task service shall add tasks with a unique ID; The same pattern has been followed for the other requirements.

The percentage of the test covered on contact Service, task Service, and Appointment Service objects was 100%. The high percentage justifies that the tests written per requirement are effective with no errors and failures. Every single part of the requirements has been covered, which means adding a task, adding a contact, delete a task, delete a contact, updating a task, updating a contact, and adding appointment are developed as expected.

The strategy used to ensure that the code was technically sound was to create a test method preceded with the annotation @Test. Inside the method, I declared an in-memory data structure that can store the same elements, such as contact or task. Specifically, I choose an ArrayList that supports the methods add and remove compliant to the requirements. Second, I instantiated three unique objects of the same type that will be used to test the requirements. In example 1, the ArrayList is "tserv," and the objects are "t\_One," "t\_Two," and "t\_Three."

Example 1:

@Test

**void** testTaskServiceAddPass() {

/\*create a Task service array to holds Task objects\*/

TaskService tserv = **new** TaskService();

/\*create three Tasks object to be added to TaskService\*/

Task t\_One = **new** Task("12AD", "Zancran", "Togbe");

Task t\_Two = **new** Task("12BD", "Zamora", "Kevin");

Task t\_Three = **new** Task("12CD", "Zamorano", "Smith");

/\* -----others instructions--------------\*/

}

The next step is to test the efficiency of each method, add, delete and update.

As mentioned early, I considered the success and failure cases to develop each object's test case on each requirement. Once the ArrayList defined and the objects instantiated, I used JUnit's assertEquals() method to test both cases. In the success case, the Boolean value of the method is set up to "True." In the failure case, the Boolean value is set up to "False" at least for one instruction. In example 2, I was trying in the failure case to add a task object "cTwo" that has been already stored in the ArrayList "tserv."

Example 2:

@Test

**void** testTaskServiceAddFail() {

/\*-------------instructions------------------------\*/

/\*add Tasks objects to TaskService\*/

*assertEquals*( **true**, tserv.addTask(cOne));

*assertEquals*( **true**, tserv.addTask(cTwo));

*assertEquals*(**true**, tserv.addTask(cThree));

/\*adding an existing task object\*/

*assertEquals*(**false**, tserv.addTask(cTwo));

}

**2- Reflexion:**

**a-i:** We utilize **specification-based (black box)and structure-based (white) software testing** techniques during modules 3, 4, and 5. In modules 3, 4, and 5, we created the contact class, the task class, and the appointment class based on the specifications on the size of each attribute and the null value. The **equivalence partitioning** is the specification-based technique used to test the range of acceptable values of each attribute in the corresponding test class. For example, in module 3, the string value of the iD of the contact class shall not be longer than ten characters. If no specification, the iD attribute will allow an infinite size of a string. Thus, our test case for the iD string attribute took one valid input ( less than ten characters) and one invalid input (greater than ten characters).

In modules 3,4, and 5, we used **statement testing and coverage and decision testing and coverage** as white-box techniques. Thus we explored in module 4 each method of the taskService class by developing two cases due to decision control (for loops and if clause). For example, to add a task, each instruction of the method add a task is explored, the for loop and the if clause inside the for loop. We developed in taskService test some inputs covering each branch of the decision control to ensure all internal components are exercised adequately.

**a-ii:** Among the other software testing technique that we did not use in the milestones are:

**experience-based techniques (error guessing)**: which are based on the skill and experience of the testers and experts. It is conducted in an ad-hoc manner because proper specifications are not available to test the application. For example, pressing the Esc key might have crashed a similar application in the past or pressing the back button or enter key on a webpage, javascript errors.

**Decision table testing**: test the system behavior for different input combinations. For example, username and password are combined to access an application.

**Boundary value analysis**: it's used to find errors at the lower and upper boundary. For example, if the program requires 100 dollars, check if 99.9 dollars will be accepted or 100.01 dollars will be accepted.

**a-iii:** The practical uses of the white box technique can be explained as follow:

1-identify the component or the program to be tested; 2-plow all possible path in a flowgraph

3-identify and list all possible path from the flowgraph, 4- write test cases to cover every single path; 5- execute, rinse and repeat

The practical use of the black-box technique can also be explained as follow:

1-identify requirements or specification; 2- choose inputs for the positive and negative scenario;

3-determines expected outcomes for all the inputs; 4-build and execute test cases with the inputs

5- compare actual input with expected, detect defects, fix them and retest.

Software testing techniques improve software quality by addressing specifications and non-functional requirements. They identify bugs, defects, and errors early in the code. However, the choice of software testing technique depends on multiple factors such as the level of risk, test objectives, the type of risk, the regulation, time and budget, and the development life cycle.

**b-i:** I think I was focus on testing all the requirements. I always thinking about the success and the failure cases. I asked myself if there are other ways to check each statement to make sure decision controls are running as expected. The complexity and interrelationship of the code were important to appreciate because they impact how we design tests case and how we use valid and invalid inputs. Suppose that the id and the firstname attribute of the contact class should not be greater than 10 characters. We test each attribute separately, considering the other attributes have valid inputs. Thus, the test result is clear; otherwise, the result will be unclear if we test the same two attributes.

**b-ii:** As a developer, I will try to limit bias, but I will not be as efficient as another tester checking my work. I believe bias will be a concern for developers. The reason is simple, from one project to another, developers get used to code in some way, which will not reinforce their tester skill. For example, I might code a for loop that forgets to consider a value that is supposed to be in the range(boundary value analysis). Also, time pressure could be a handicap for a developer to focus on testing his work correctly.

**b-iii**: Software engineers must be disciplined while coding or testing to reduce bugs, errors, defects and improve software quality. People lost their life; business lost their notoriety and credibility because of bugs, defects, and errors. In my career, I think teamwork will improve software quality and my testing skill. I will test my code and let my co-worker test my code and vice-versa.

Reference:

TOGBE, Z (2021) TaskServiceTest.java [mobile app source code]

Hambling, B. (2015). *Software testing: An ISTQB-BCS certified tester foundation guide*. London: BCS.