**Project 2: Summary and Reflection**

In this case, the software requirements are outlined by the customer in the Appointment Services and the Contact Services. Classes are built to retain the respective properties and requirements. Some of the requirements are listed below.

* The object shall have a required unique appointment ID string that cannot be longer than 10 characters
* The appointment ID shall not be null and shall not be updatable.
* The appointment object shall have a required appointment Date field
* The appointment object shall have a required description String field that cannot be longer than 50 characters

JUnit tests are developed based on these requirements to see if the code meets the requirements.

The Junit tests provided are sufficient coverage of all tests required that is add, delete and update methods. The coverage percentage tool gave a 65% test coverage score initially. After reformatting the code and adding additional checks and requirements, the test gave about 80% coverage. The tests were based on the checklist created for the test, which focused on three main areas to test all which are covered. Some of the items on the checklist included,

* Testing for string length
* Checking for string ID
* Checking for string description

These are some of the items on the checklist when drafting a test and what needs to be tested and how.

Using spacing between lines that pose some trickiness is a good way to ensure that code is technically sound. Also, implementing unit tests for each test method that is add, delete and update as shown in the screenshot code below.

Graphical user interface, text, application, email

Description automatically generated

Efficiency of the code came from the use of libraries to aide in syntax comprehension. Some of the libraries used include import java.util.Date; import java.text.ParseException; import java.text.SimpleDateFormat; Using a loop, “line 14-21 AppoinmentServicesTest.java” seen below,

if(AppointmentService.addAppointment(A)) {

System.out.println("\nadded");

}else {

System.out.println("\nfailure to add appointment");

}

This tested the code for adding an Appointment and a loop in the event of an Appointment not being able to be added. Could be unavailable dates and so on.

In Modules 3 through 5 a series of Junit tests were run. The software testing technique used was unit testing. Unit testing is when pieces of the code are tested. The tests are to verify if the “design and implementation for that unit have been correctly implemented” (Garcia, 2017 p20). In Module 3 the pieces of the code tested were in the contact service class. They are:

* to add contacts with a unique ID.
* to delete contacts per contact ID.
* to update contact fields per contact ID.

Contact Service was verified through Junit tests with assertions added an example given below,

assertEquals(false, cs.deleteContact("name"));

Module 4 also required different JUnit tests to be run. The pieces of code tested in this section were related to Task Service and included checking string length and null. The required tests were checking for,

* to add tasks with a unique ID.
* to delete tasks per task ID.
* to update task fields per task ID.

Module 5 had Junit tests as well designed at checking the customer’s requirements with the mobile application. This section of code was to verify the appointment service and appointment class through Junit tests. Ultimately, the appointment service should be able to add unique appointment ID’s to new appointments as well as delete appointments based on new appointment ID’s. Unit testing is usually the first level of testing conducting during the development phase of the SDLC.

A software technique not used is integration testing. Usually, this is done after the units have been tested separately and are now ready for integration. The focus here is on creating and testing components (combined units), and interfaces used to facilitate the integration. The different strategies involved with integration testing will ultimately determine the order in which units are integrated.

The JUnit tests are implemented as a first level test that focuses on the functionality of the method and object. According to Garcia, “units are combined to create composite components. Integration testing should focus on testing components, interfaces” (2017 p20). Integration testing is always implemented after unit tests are complete. Both testing roles JUnit and Integration are usually taken by different software engineers and testers but ideally no one person is asked to take on both roles. Usually when unit and integration tests are complete the system is then tested, which basically a test of all the integrated components.

The general mindset when working on software testing is to be cautious throughout the entire process. Unit tests are among the first types of tests conducted during the SDLC and caution is advised because anything missed at this stage may be costly if spotted down the road of development. Understanding fully the requirements and the terminology used to setup tests was a key attribute in developing functional JUnit test. A great example is when the test was supposed to throw an Illegal Exception if the string input was too long a case in point below,

void testStringLength(){

Assertion.assertThrows(IllegalArgumentException,class() ->{

});

}

Trying to avoid being biased when testing code has always been the key to a good and accurate test. It poses issues when developers test their own code that is why most organizations prefer to keep the two separate. A more effective test-coding team can be derived from assign separate roles. All testing elements are focused on without any intentional or unintentional erring, as well as maintaining all testing parameters throughout the tests. An example, initially the Contact Service class was missing validation for null and lengths, tests for the Contact class, and extra delete field.

**References**

Garcia, B. (2017). *Mastering Software Testing with JUnit 5*. Packet Publishing.