**Project Two: Summary and Reflections Report**

Gracie Ream

Southern New Hampshire University

CS-320

August 19,2024

The unit testing that was applied to each of my assignments was a form of white box testing. This is where the source code is put through a specific design test to prove that the software is correctly going to do what it is told to. Specifically, I tried to align these to the software requirements by following the guidelines for each assignment module. One of them had to include an ID that could not be longer than ten characters. Creating a unit test to test this portion would tell whether the code that was written correctly incorporates a situation where the user cannot put in an ID that is longer than ten characters. A unit test was written for each of the assignment module requirements, and that was a teller on whether I was accurate on the software requirements. The overall quality of the Junit tests was average, the first assignment came out 5/5 testing, however the next two assignments took longer to complete it. The coverage was only at around 17% the first time I did the tests, where the next assignment it came out to around 48% which is a little better.

To ensure that my code was technically sound, I realized my mistakes through the previous assignments and worked on those to ensure better code tests for the next assignment. If something worked for me in the previous assignment, I would use those in the next one as well in a different situation. For example, in the Task Service assignment, the ID could not be longer than ten characters long. The test code for this is:

@Test

@DisplayName("Task Id Cannot have more than 10 characters")

void testTaskIDWithMoreThanTenCharacters() {

Task task = new Task("Name", "Description");

if (task.getTaskID().length() > 10) {

fail("Task ID has more than 10 characters");

}

The Appointment Service assignment also had the requirement of not having an ID of more than ten characters. My code for this test was:

@Test

@DisplayName("Appointment ID cannot have more than 10 characters")

void testAppointmentIDWithMoreThanTenCharacters() {

Appointment appointment = new Appointment(Date(2025, Calendar.JANUARY, 1), "Description");

if (appointment.getAppointmentID().length() > 10) {

fail("Appointment ID has more than 10 characters");

}

These portions from different assignments are similar but built onto with the appointment service assignment. This ensured that my code was technically sound by not only testing the code to ensure that the requirement was met, but also continuing what worked for me and expanded it.

To ensure that my code was efficient, I used the same unit tests for similar requirements just with different objects. For example, in appointment service test, I had to ensure that the task description was not null as well as the task date. For these two requirements, I used the same unit tests.

@Test

@DisplayName("Task Date is not to be null")

void testAppointmentDateNotNull() {

Appointment appointment = new Appointment(null, "Description");

assertNotNull(appointment.getAppointmentDate(), "Appointment Date is null");

}

@Test

@DisplayName("Task Description is not to be null")

void testAppointmentDescNotNull() {

Appointment task = new Appointment(Date(2025, Calendar.JANUARY, 1), null);

assertNotNull(task.getAppointmentDesc(), "Appointment Description is null");

}

}

Both of these worked for me and ensured that my code was efficient by creating comparable and reliable tests for both requirements.

For the testing techniques for the assignments and project one, I wrote Junit test cases, and ran the coverage for each one. These both provided me with insight on what part of my code was working and running accurately and what needed updated. There are various other software testing techniques that I did not use in these assignments. Static testing is one way of testing the software. Static testing takes place early on and the requirement specifications are reviewed through different methods. Static testing is most practical when the budget is lower, because this method will reduce costs by reviewing the code earlier on. It is also more practical when trying to reduce time and working dependently on a team. The testing technique that I used, Junit testing, is more practical when the development team or developer is testing out specific units of code and ensuring that they meet the specific requirements accurately.

The mindset that I developed for the project was to ensure that each part of code met the requirements of all three assignments. I really tried to employ caution when writing the test units because I wanted the most coverage possible. Ways to reduce bias is to ensure that you are testing each unit of code and not just assume that your code is already correct and running efficiently. This I tried to do, and fixed even though my code had no errors in some spots, some of the tests were not executing correctly so I knew that I had to go back and fix what wasn’t correlating to the test. It is very important to be disciplined as a software engineer to avoid mistakes and possibly major released defects in the future. If a software engineer does not extensively test and deploy their software, it is possible that a released escapement could happen, which is a flaw in the code that can lead to major mishaps or disasters to happen in the real world. Extensive testing would also save potential debt in the field because fixing major issues sooner rather than later when it is released could save a great amount of money.