Module 7 Journal

For the three features of my project, the AppointmentService, the TaskService, and the ContactService, my unit testing approach was largely the same. I first ensured that the code threw the exceptions it needed to when it encounters out of bounds and null inputs to ensure the software met the requirements. For instance, I created a test that ensured any null input was rejected with an IllegalArgumentException.

@Test

void testAppointmentNotNull() {

//test constructor throws null exception

assertThrows(IllegalArgumentException.class, () ->

{Appointment appointment = new Appointment(null, null, null);

});

}

This ensures that an appointment can not be null.

To ensure that my tests were effective, I ensured that the coverage percentage of my tests was greater than eighty percent. This means that the tests I constructed covered eighty percent of the code that needed to be tested. A high coverage percentage means ensures that your whole program is being tested, reducing the risk of issues down the line if the program is changed or altered.

Working with Junit tests was a little frustrating for me. I received feedback that each test should assert only one thing, but, for instance, if I were to test the appointment constructor creates a correct object, I must establish each field is correct:

@Test

void TestAppointmentConstructor() {

Appointment appointment = new Appointment("1234567890", new GregorianCalendar(2033,Calendar.JANUARY,1).getTime(),"Description");

assertTrue(appointment.getAppointmentId().equals("1234567890"));

assertTrue(appointment.getAppointmentDate().equals(new GregorianCalendar(2033,Calendar.JANUARY,1).getTime()));

assertTrue(appointment.getAppointmentDescription().equals("Description"));

}

To increase efficiency I could have separated this into several tests that each test one field of the constructor.

In this project, I used automated unit testing by creating Junit tests to test each module of the code. Unit testing is a testing technique where each module, class, or function of software is tested independently. This ensures that each unit of the software operates as required, limiting the problems that could come from them later on down the line. I also performed functional testing, which tests the functional requirements are met, in this case, that the AppointmentService, ContactService, and TaskService portions each have the ability to add and delete objects.

In this project I did not perform any integration testing, or ensuring the software integrates properly into a larger project. Acceptance testing was also beyond the scope of this project. Acceptance testing ensures the software meets the end-user’s expectations. Performance testing was also not part of this project. Performance testing tests the software’s speed, or stability.

The mindset I adopted while creating the tests had what was probably an overabundance of caution. I found integrating one method particularly frustrating. In the Appointment class, we were required to use the Date class, which was difficult to use in the AppointmentService class. The Date class does not create dates in an intuitive way, so setting a specific date is difficult and frustrating. I had to call on a separate class, the Calendar class, to set a date, then send that date to the Date class. In the end, It looked like this:  
 new GregorianCalendar(1900,Calendar.JANUARY,1).getTime()

Creating a new GregoriandCalendar object, then sending that object’s exact time to the Date field, which takes a Date object.

To eliminate bias in my testing method, I tried to take some time between creating the software and creating the code. Software developers can get into a thought process of thinking their code can’t possibly be wrong as they were the ones who created it. This can lead to limiting the tests created, or eliminating tests completely due to thinking, for example, “I wrote in input validation, I don’t need to test that it works.”

It is important to be disciplined in your commitment to quality as habits are built over time. If we begin to cut corners when we feel like we don’t need to test for everything, it becomes that much easier to skip testing all together. This is a dangerous mindset to enter, as it will inevitably lead to mistakes. Technical Debt, or the cost of future work due to shortcuts taken at the beginning of a project, can be a problem. To avoid this, we must take the time to test our code thoroughly. The old adage comes into mind “An ounce of prevention is worth e pound of cure.” This is true in software testing as well. If one creates good, thorough tests in the beginning, they will continue to work even if the code changes in the future.