Liam Breslin

CS-320

2/17/2025

**Project Two Summary and Reflections Report**

Each unit test that was created was made with the intention to directly test each functionality that is required of the codebase. For example in Appointment.java, one of the requirements was for there to be a unique appointment ID string that cannot be longer than 10 characters, and it cannot be null. In the corresponding JUnit tests, these two are the unit tests that were ran to test for the ID being null or over 10 characters:

**@Test**

**void testInvalidAppointmentId\_Null() {**

**Exception exception = assertThrows(IllegalArgumentException.class, () ->**

**new Appointment(null, getFutureDate(), "Software Testing")**

**);**

**assertEquals("Invalid appointment ID", exception.getMessage());**

**}**

**@Test**

**void testInvalidAppointmentId\_TooLong() {**

**Exception exception = assertThrows(IllegalArgumentException.class, () ->**

**new Appointment("CS1234567890", getFutureDate(), "Software Testing")**

**);**

**assertEquals("Invalid appointment ID", exception.getMessage());**

**}**

Throughout all of the tests in the Project the total test coverage was 82% which is a good mark for general coverage, as aiming for too high of can lead to unnecessary time wasted. Throughout the entire project proper naming conventions, clear commenting, good encapsulation, and other standard best practices were applied. Additionally the assertAll function was used in cases where multiple assertions needed to be made at the same time to allow for accurate test results. The following is a good example of this and industry standard code formatting :

**@Test**

**void testValidAppointment() {**

**Date futureDate = getFutureDate();**

**Appointment appointment = new Appointment("CS320", futureDate, "Software Testing");**

**assertAll(**

**() -> assertEquals("CS320", appointment.getAppointmentId()),**

**() -> assertEquals(futureDate, appointment.getAppointmentDate()),**

**() -> assertEquals("Software Testing", appointment.getDescription())**

**);**

**}**

Throughout each of the programs the most common type of software testing technique utilized was Unit Testing. In each program and their corresponding test programs, the tests were broken down into smaller more digestible bits that could focus on testing specific functionalities of the code; isolated from each other. In these different unit tests further software testing techniques were utilized. For example boundary testing was utilized in all of the non Service files when testing for proper length of the ID, description, or date validity. In these examples some negative testing was utilized. A separate example of negative testing of the ones already mentioned is testing for duplicate IDs. When testing for the duplicate IDs, an invalid input is provided to check how the system handles the error. In each ServiceTest files integration testing was utilized making sure that TaskService manages Task objects, or ContactService manages Contact objects correctly. Finally through the tests exception testing is used. Some examples being testInvalidAppointmentDate\_Null() or testINvalidDescription\_Null(). Both of these check whether an IllegalArgumentException is thrown when the data is provided.

Though a handful of testing techniques were used throughout each milestone, many common types weren’t. First there was no performance testing in my programs, only testing for functionality. An example of where having performance testing could be beneficial would be to test how AppointmentService handles adding a thousand appointments to the system. On the same side of this Security testing was also non done. None of the tests utilized were geared towards breaking or exposing it with malicious inputs. This could’ve been done with SQL injections to spot out any vulnerabilities in our code that could be taken advantage of. The final software testing technique that I find of value that wasn’t implemented is mutation testing. In my code all of the tests check for expected behavior. With mutation testing it would put intentional bugs into the code to assess how effective the test cases are, to measure whether the test can detect the intentional changes.

Starting with unit testing, it is widely used across many projects with frequent changes. It’s generally strong in maintaining high code quality, helping developers catch bugs early but does require additional integration testing. Boundary testing is good for handling user input and enforcing the constraints put on the users input. Negative testing is good to see how systems respond to invalid input, like incorrect login credentials. Sometimes boundary and negative testing can have overlap. Integration testing ensures smooth interaction and reduces the risk of system failure due to miscommunication between those components. Additionally it helps spot out compatibility issues earlier in the SDLC. This can take a longer time to do properly. Exception testing is good at preventing software crashing and unhandled exceptions. Because of this proper exception handling also helps increase security due to these errors not exposing the system. This leads us into Security testing where this will enhance the users trust in our program. It will stop us from having costly data breaches and cyber attacks that can lead to exposure in our system, costing us money and the simultaneous loss of trust with our consumers. In order to have good security leveraging ethical hacking techniques like OWASP is great. Now with performance testing it is important to ensure proper scalability for our system. Without good performance testing it can lead to memory leaks, real-time response issues, and overall poor stability. This then leads to larger downtime of the application which is the last thing that we want post launch. This is then why mutation testing can benefit nearly all of what was previously mentioned in software testing techniques. Mutation testing helps us create more accurate and thorough tests that will help push the system into the best possible state it can reach.

When coding and then testing a code base, it’s imperative to review it with a detailed oriented focus. Though this code in particular wasn’t very complex. It was serviceable enough to display the importance of focus when specifically dissecting one's own code. Specifically, the testing part of the development cycle is crucial to identify and fix bugs as early as possible so a high quality product can be released at launch. Without this step done properly, the software can among other issues be exploited, run incorrectly, fail to meet general requirements, and will ultimately deliver a poor experience for the users. When making good tests it was then specifically that one as a coder has to truly know their code. It forces you to think from another lens that you don’t typically view your code from. Strengthening you as a whole in the process.

When I first made my tests via my code they seemed to be well made to me. But that of course is a red herring. I went towards several online sources to see reference JUnit tests to learn from others and I could immediately see the inefficiencies in my code; that were later changed. In coding specifically while there are proper ways to do things, such as popular industry naming conventions. Coding can be a very personal form of art to many. An art form that is oftentimes easy to oneself but not always to other people. Being able to take note of that bias and share your work with other developers who haven’t worked on the code with you is an excellent idea. Having a fresh pair of eyes that aren’t jaded from the hours of coding done on to it oftentimes leads to new enhancements and tweaks to the code that otherwise would likely be missed from a developer who’s worked on it themselves. Finally, being disciplined as a software engineer is crucial. Being a paid professional means that you will be working with people all the time. A show of how good a developer is at working with others is oftentimes shown in their coding habits. So by staying on top of your work, so that it is easily understood and malleable by others is key. More specifically cutting corners when it comes to testing code can be even worse. Writing poor tests can lead to issues with the software and its functionality as previously mentioned. But even worse it can lead to large delays, financial costs, and overall unnecessary technical debts throughout the entire development life cycle among every party.