Summary and Reflections Report

The testing approach utilized for the software produced in this course was catered extensively to the software requirements listed. This is because a software’s functionality and usefulness to the buyer is entirely dependent on how well it meets the requirements they set out. If you were to deliver a mobile ordering app for a local restaurant, for example, and it lacked the ability to place orders from a smartphone, the app would be useless. It would fail to serve its main requirements and therefore be worthless to the company and result in a failed partnership. As such, the unit testing conducted here focused on crucial requirements to ensure a suitable app for the client. This can be seen through tests such as testAppointmentDateInPast() which checks for appointments set in the past. In this test, it attempts to set the appointment for 1995 and the program correctly denies this appointment as it is invalid. This helps fulfill the client requirement that all appointments are for future dates. While one single test cannot capture a program’s effectiveness in its entirety, a series of tests can help do this. This is referred to as a coverage percentage, a metric indicated the portion of lines of code tested through a series of unit tests. In the case of the AppointmentService and Appointment files they scored 80.4% and 91.3% coverage respectfully. Following the same conventions, the contact application scored 80.5% and 80.6% coverage and the task application scored 80.5% and 88.1% coverage. This leads to above 80% coverage for all tests and a high confidence in the effectiveness of the software as well as its ability to meet client requirements.

In addition to simply meeting requirements, I also made sure to check for technically sound and efficient code. This is because a lack of optimization and rigidity can lead to poor user experience and ruin an application’s success. If you were to take an application you use and enjoy, for example Microsoft Word, and introduce slowdowns and crashes there would be a major change in your outlook. You would trust the application less and move towards alternatives. One test to check for sound code was with testContactServiceUpdateFirstName(). This test first created a known functioning contact. Then it took the contact and updated the name. After this update it ensured that the updated name took effect and the new contact was correct. This helps to ensure proper functioning of sound software. In addition I was able to test both efficiency and soundness by testing invalid inputs. This involved tests like testTaskServiceDuplicateID(). This test created a functioning task and then attempted to create another task with a matching ID which is invalid. However, rather than just making sure it did not create another object, it tests to ensure the program threw an exception and stopped processing the request using Assertions.assertThrows(IllegalArgumentException.class, () -> {TaskService.createTask("123456789","Test Updates", "Please test all updates");. This helps with efficiency as the rest of the request was not unnecessarily processed as well as preventing unsanitary inputs from effecting the program.

The two main software testing techniques that were employed are unit testing and manual review. In unit testing, we create our program file such as TaskService and a second test file called TestTaskService. JUnit is then imported into the program as well. In the test file you can declare specific test cases you would like to execute. It is then run as a JUnit test and each of the test cases is executed. It then comes back with a report showing the successes and failures of the test to help pinpoint issues. However, a JUnit test result is only as effective as the test itself. If a test fails to have adequate coverage or depth, the results are meaningless. JUnit testing is so important because it provides a high level of confidence in a program as well as accessiblity to both small developers and large companies alike. It is a simple test that can be executed by as little as one developer and yet still provides value even to large companies. Another method used is manual review. In this method, a developer or tester reads over the code and checks for logical errors or common problems. This helps to find errors before furthering development and can help catch logical mistakes that other tests may not find. This method is important for many of the same reasons as unit testing in its accessibility and ability to spot errors that diminish a program. Spotting these errors early helps to minimize later costs in production due to faults and bugs. It can also scale very easily from an independent developer review to a large corporation with a dedicated team.

Two main techniques that were not used are performance testing and usability testing. Performance testing focuses on evaluating a program’s speed, efficiency, and other performance factors. This helps to ensure that the program is fast enough for the client and has no major performance hinderances such as a memory leak that can prove disastrous to a program’s usability. It can also be used to assess the type of hardware required for the program, also known as system requirements, based on the resources needed for ideal performance. The system requirements allow users to determine whether or not the application will successfully run on their device. Performance is useful to all programs, but some programs may value performance over others. Fast paced programs such as video games require super-fast responses and as such, performance testing should be integral to the development of a good game. Bad performance such as frame drops may result in fewer end users. However, a simple archiving tool isn’t as focused on performance. While it still would be nice to be faster and more efficient, a few seconds to minutes difference in execution time won’t be as impactful to the customer. Especially if items are only removed on an irregular basis. Therefore, a development team may choose to utilize these resources elsewhere in testing such as reliability and data integrity. Usability testing focuses on how usable the software is to the client and their users. This usually involves small batches of internal testing or alpha/beta testing that asks participants to rate different aspects of the program. These questions focus on usability such as “How easy was it to perform X task?” or “Are there any features you would like to see added?”. These tests help to make sure the program captures the true need of its audience and leads to a more satisfied user base. Again, these tests are useful in all situations, however, they are especially useful in live-service type programs. These are applications that are constantly being updated to improve customer experience as they usually have high competition. Some examples include Walmart, Target, and Amazon’s online retail applications. Testing what features users like and want allows these companies to adjust the software and make users happier which puts them ahead of competitors. For a less competitive product such as WinRAR, this style of testing is less useful. Especially after its initial release. Instead, these resources can be spent on ensuring that the program functions properly.

Maintaining an open, yet cautious mindset is important in software development and testing. This is especially important in self-published projects where you are reviewing your own code. Each aspect of a program is heavily entangled with other portions. A small update in one dependency can have a large rippling effect since other methods may make calls to the dependency. As such, when you update one area of the code, it is important to implore caution and check dependent sections of code to ensure function. For example, updating the Task object and allowing larger names would seem like a good way to expand the name field. However, failing to also check the addName() or updatedName() methods would result in several aspects of the program still adhering to the shorter name rule as they sanitize input separately. In order to limit bias in the review of my code I tried to do two main things. The first was to take a break between programming and testing. This helped to remove the code from recent memory and helped make sure that my brain wasn’t trying to edit the code in my head to make it look better. Another thing I did was to act as though I was reviewing someone else’s code for a critical project. This helped to alleviate the sense of perfection that I wanted to maintain for myself while also applying pressure as I want the critical project to succeed. This helped to create a highly meticulous code review with less bias towards myself.

Being disciplined in one’s commitment to quality and making sure to avoid corner cutting is paramount to being a good developer. Software is used in nearly every aspect of our lives and some of those aspects can be life or death such as autonomous driving, aircraft autopilot, or surgical robots. In each of these cases, a small bug or error in a program can cause the loss of many lives. One bad line of code could cause the car to turn into oncoming traffic, or the plane to nosedive to the ground, or the surgical knife to go a single millimeter to the side and slice an artery. In each case, we bear the burden of responsibility. We must ensure we do not falter, and we do not give into corporate corner cutting. We alone can prevent these needless deaths caused by poor quality code. And when death isn’t on the line, there is still other ways our lack of diligence can cost us. Cutting the corners at the base of a program can lead to massive amounts of reworking in the future due to dependencies as discussed previously. The time saved originally is lost when reworking and often significantly more time is lost fixing resultant mistakes. As such, our discipline can help to minimize these mistakes by doing things right the first time. Saving ourselves, the clients, and users’ valuable time. Discipline is important for everyone’s life; however, it bears greater importance when deciding the life of everyone. As such, we must uphold a promise to ourselves and humanity that we stay disciplined and diligent in our work to provide safe, quality software in all that we do.