Christopher Manley

CS 320

Project 2

My approach to unit testing began with the software requirements. Having a list of requirements gave me a starting point of knowing exactly what needed to be tested. For example, one common requirement was that a name variable needed a length limit and needed to not be null. Having this requirement let me know that I needed to test that the name be less than the limit and not null. Similarly, I developed unit tests for any other requirements that were provided.

Evaluating the overall quality of JUnit tests is relatively simple thanks to coverage percentage. However, due to the nature of the tests, simply checking the coverage percentage wasn’t enough to know that my tests were effective. For example, in the contact section, there was a JUnit test file for contact.java along with one for ContactService.java. These two files were closely linked, with multiple mutators and accessors only being utilized by ContactService. Therefore, I went through the specific coverage of each test to ensure that the tests properly covered everything they needed to.

One way I ensured my code was technically sound was by deliberately testing every method in each file. Because certain methods (namely the mutators) aren’t ever directly accessed, I made sure to check the test coverage to verify that I tested every method. For example, in the TaskService section, I tested the updateName method. This function searched for a particular task by id, then it called the setName function in the task.java file to change the name. By testing the updateName method, I was able to confirm that both that method and the setName method functioned correctly.

To make both my main code and test code efficient, I primarily used two techniques. The biggest way I simplified my test code was through the use of @BeforeEach when testing the service files. This eliminated the need to fill the lists with test samples in each test. As for the main code, I utilized coverage percentages once again. Because I was careful with testing every method, I was able to see if there were any unnecessary redundancies or unused code. If I found any, I was able to simplify the code.

The main software testing technique I used for this project was JUnit testing. JUnit testing allowed me to test each specific part of the code to ensure it functioned properly and produced the expected output. If any errors were present, the tests would reveal exactly what method was causing problems. For example, in ContactService.java, I had a typo in one of my functions that was causing it to not work. The unit tests singled out that function as a failure and I was able to manually review it and find the typo.

One technique I did not use for this project is a team inspection. In a more official setting, it can be helpful to have someone who did not write the code inspect it. This can help reduce the bias that can happen when reviewing your own work. It can be particularly helpful when searching for ways to improve the code, as hopefully the original developer already did what they thought was best.

Employing caution as a software tester is important because it helps ensure that the tests are comprehensive and test correctly. For example, in the project, I employed caution when ensuring that I tested every function. Because files worked so closely together, I had to trace the path of the program as it ran the tests. It is important to understand these complexities to make sure that everything is being put through tests and not left out.

Whenever I develop a program, I attempt to minimize my own bias towards my work. As a developer, I am naturally inclined to take pride in my work, and it can sometimes be hard to admit that something has been done sub-optimally. To reduce this bias, I treat the code as if someone else had written it when I am reviewing. While I think bias is less important with mistakes (as mistakes are generally objectively incorrect), efficiency and safety are less clear and therefore bias has a greater impact on reviews.

While I would argue discipline is important in every aspect of life, I believe it has particular importance when developing and testing software. The first thing to consider is the possible repercussions of a lack of discipline. Depending on the specific software, many people can be negatively affected by a lack of discipline. For example, improper safety techniques can lead to a loss of consumers’ private information. A disciplined approach to writing code is very helpful when testing because proper techniques will lend themselves to more efficient tests. Disciplined testing is equally important. By ensuring that testing is comprehensive, possible weaknesses or mistakes in the program can be eliminated. Additionally, weak writing and testing can result in increased maintenance costs to fix things later. To avoid this technical debt, I plan on utilizing various testing techniques and taking a dedicated approach to testing.