Matthew Muller

Dr. Bolton

CS 320

19 June 2022

Summary and Reflections Report

During my work developing and testing the contact, task, and appointment classes, as well as their service classes, my testing approach was directly aligned with the software requirements that were presented to me. For example, one of the requirements that was provided was a character limit on the string attributes that were present in each of the Task, Contact, and Appointment classes. I tested for this by having the setter methods throw an illegal argument exception, and then developing tests that asserted that this exception would be thrown after trying to set the strings with values that have more characters than allowed. I believe that my tests were effective in covering the requirements of the system but could have been developed in a way that ensures this through 100% coverage. For example, my TaskServiceTest has a coverage of 85% on the TaskService class and my TaskTest has a coverage of 75% on the Task class. However, all of my other JUnit tests have 100% coverage over the class that they are testing except for my ContactServiceTest which has 97.7% coverage over the ContactService class.

One way that I ensured that my code was technically sound was through the use of proper annotations in the tests. Each of my tests have the @Test annotation above them in order to declare the method as a test and allow me to provide assertions. An example is shown in the code below.

```
@Test
```

```
void testTask() {
        Task task = new Task("12345");
        assertTrue(task.getID().equals("12345"));
}
```

One way that I ensured that my code was efficient was through the use of conventional naming techniques and comments. This helps to promote efficiency especially when working with a team as it makes the code more readable and easy to jump into for someone less familiar with the system. For example, the code shown below shows my comments that help make the steps taken by this constructor method very clear.

```
//Create task with given ID
    public Task(String id) {
        //Validate provided ID
        if(id == null || id.length() > 10) {
            throw new IllegalArgumentException("Invalid task ID");
        }
        //Set taskID
        this.taskID = id;
    }
```

The main software testing technique that I employed for the three milestone assignments was white box testing. White box testing, also known as structure-based testing, focuses on ensuring that the data structures, internal design, and code structure of the software is working as intended. One of the important factors when it comes to white box testing is statement coverage.

When employing this technique, the aim is to traverse all statements in the SUT at least once in order to seek out any faulty code. This was an important goal during the milestone assignments as it helped to ensure that all of the requirements were being met and that the classes functioned as intended.

One testing technique that I did not end up employing while working on these milestone assignments was static testing. This is where the tester seeks to find defects in the code without actually running it. For example, data flow analysis was not used at all for these assignments. Here, the tester will follow the path that data takes as it is received and transformed into an output. This can help to detect code defects such as unused variables or unintended data flows.

Throughout the course of testing this system, it was important to employ caution when considering the relationships between the classes. Getting the Task, Contact, and Appointment classes to work properly with their service classes required attention to how the classes communicate with each other. This meant being cautious to have the correct scope for all variables and methods and making sure that all of my code was secure and technically sound.

For this project, limiting bias was not a very big concern since I was writing all of the code myself. However, limiting bias is a very important quality for a software developer to have, especially when working in teams. Sometimes, when reviewing code, it can be possible to allow bias towards your own code to enter your mind. This could result in you reviewing your code less thoroughly than the code written by others, which is likely to cause you to miss potential defects. This can be very detrimental to development, especially since it is such an avoidable issue. Being able to put aside any arrogance and review your own code in the same way you would review someone else's is a vital skill for any software developer.

The importance of being disciplined in your commitment to quality as a software engineer cannot be overstated. Cutting corners at any point in the development or testing phases can have a huge impact on the quality of your product. If you are not committed to the thorough testing of your software, it is almost guaranteed that you will let at least one defect slip through. This will lead to losses in time, money, and potentially business if you develop a reputation to be a careless programmer. This is why having discipline in your approach to coding and testing is so important for any software developer.

Resources

Software testing techniques. GeeksforGeeks. (2021, March 1). Retrieved June 19, 2022, from https://www.geeksforgeeks.org/software-testing-techniques/