Project Two

Southern New Hampshire University

Aug 20th, 2022

Ryan A. Manchanthasouk

**Follow-up Summary**

The testing approach was directly aligned with software requirements as can be seen in the Task class for example. In the Task class there are if statements that check for character length which was required for all the variables. The TaskService class is also aligned with requirements as can be seen in the methods. A unique ID is required, the task service is able ot delete tasks based on task ID, and the task service is able to update task fields based on name and description.

The TaskTest and TaskServiceTest tests were quality tests based on an 83.3% coverage from TaskTest and 100% coverage from TaskServiceTest. More tests could have been written to cover 100% of the Task class, but the test written were sufficient enough to cover software requirements.

The code is technically sound because I followed modern-day coding standards in terms of name-conventions, white-space, and there were no errors. To start off, I followed the camel Case naming convention for variables and methods (private String taskId;, private void checkTaskId). All constructors and classes followed the pascal case naming convention, where each word starts with a capital letter (public void setName, public class Task).

I made sure my code was efficient by not repeating myself, keeping it simple, and having a separation of concerns. I made sure to not repeat myself when creating variables, methods, or classes when writing code. I kept thing simple by following simple naming conventions and keeping the code straightforward (private String taskId;). I maintained a separation of concerns by writing methods that addressed the concerns of the specific classes. An example of this is where methods for the Task class only address variables in the Task class, such as

( private void checkTaskId(String taskId) {

if (taskId == null || taskId.length() > 10) {

throw new IllegalArgumentException(

"Error: The task ID was null or longer than 10 characters");

} else {

this.taskId = taskId;

}

}).

**Reflection – Testing Techniques**

I followed the principles of testing regarding each milestone. I used both static and dynamic testing techniques. In terms of static testing, I had technical review from my professor, inspection, and static analysis. As for dynamic testing I utilized statement coverage and conditional statement coverage.

I was able to use technical review from my professor by the end of each milestone. Professor Bolton ran my code and let me know what worked correctly or what didn’t work correctly; the feedback I received on Milestone Three gave me the correct feedback I needed to succeed in the rest of the modules.

In terms of inspection, I thoroughly went through my code after Milestone Three to ensure that my code met customer requirements and correctly addressed the rubric prompts. I was able to meet milestone requirements on each module because of this.

Static analysis was utilized in each module. I had to ensure that the program was directly aligned with customer requirements and ensured that tests also followed suit, properly testing code based off the requirements. I ensured that the code generally followed the same standard for Java throughout each class.

As for dynamic testing I used statement coverage and conditional co-jointly. Through statement coverage, I was able to identify that I was not properly testing each segment of code, which led me to overall including conditional testing. These testing techniques led to properly testing specific segments of my code.

**Testing Techniques Not Used**

I did not use specification-based techniques such as boundary value analysis or decision tables. Boundary value analysis would have been extremely useful in terms of testing the boundaries of the range because boundaries are generally not handled gracefully. Decision tables, although very useful, were not used because of the small scope of this project. Possibly, on a larger project, decision tables would be extremely useful because they would allow developers to create a visual representation for conditional statements before writing the program. Some benefits to testers are how they will help with design techniques, search the effects of combinations, and model complicated logic (Geeks4Geeks, 2021, Sep 2).

**Reflection – Mindset**

Acting as a software tester I was very cautious while writing tests. This is reflective by providing over 80% coverage in the application. I ensured to test all methods and anything with variables that could cause an issue to the system. The percent coverage gave me the confidence in the unit tests that I wrote.

It is important to appreciate the complexity and interrelationships of the code that I was testing because of how everything worked together in the greater scheme of things. An example of this would be how an appointment would be assigned an ID, which would be how the system identifies appointments. If an appointment ID were to be somehow invalid and fail, this would fall back on the developer and tester, who’s jobs were to ensure that functions had try-catch blocks and that all variables with specific requirements were tested correctly (i.e. appointment IDs should not have less than 10 characters and no more than 10 characters). The same thing applied to all the classes across the board; if there was something to test, it should’ve been tested to give the confidence that the program is meeting requirements.

**Mindset – Bias**

I tried my best to limit bias when writing code in terms of ensuring the app met software requirements. Bias would be a huge concern for developers that are responsible for testing their own code, in the sense that a developer may write code that doesn’t necessarily pass requirements but will past tests. This bias can cause a huge negative impact on the relationship between the customer and the developers.

As an example, in my case, I really wanted to ensure that my tests covered at least 80% of the code that I wrote. At first, my only goal was to write software that was easy to test, without completely reaching requirements. This caused me to sacrifice well-written complex code. The lesson I learned was to write good code, complex if it needs to be, but to also write even better tests that accurately test for software requirements set by the customer.

**Mindset – Discipline**

As a software engineering professional, discipline is extremely important. One should never cut corners when writing or testing code because it could cause issues further down the line as the software grows. An example is how in continuous integration and continuous deployment a best practice is to test at every stage – this ensures that the program continuously works and meets software requirements. Without testing at every stage, this could cause a lot of technical debt that could cost a fortune in the long run, as seen in companies that have previously used the waterfall method.

Waterfall is a software development lifecycle methodology approach that is linear in progression, meaning that there is little room for changes once developers reach a certain point in development. Testing isn’t started until the second part of a project which means new bugs are going to be found – there is even a chance that the development team will skip the testing phase overall to meet deadlines.

I plan to avoid technical debt as a practitioner in the field by following best practices, which in this case is to test frequently and test often. Testing at every stage may seem like a burden, but after completing the mobile application, I’ve found that it is necessary to meet software requirements and avoid technical debt.

**References**

Geeks4Geeks (2021, September 2). *Software engineering: The make-buy decision or decision table*. GeeksforGeeks. Retrieved August 4, 2022, from https://www.geeksforgeeks.org/software-engineering-decision-table/