Project Two

William Cocomise

SNHU

CS-320

Prof. Jonathan Norman

Summary

The testing strategy I used was closer to test-driven-development. Each technical requirement had an automated JUnit test. For example, both the Contact and Task services required that the object IDs be unique. So, both cases test whether two objects with the same ID could save to memory. In both cases, the tests passed and the objects with non-unique IDs were not saved.

There is one hundred percent code coverage for each model and service class. My goal for each service was to cover the requirements and not the entire application. So, each technical requirement and its related code got coverage. I learned that the setters in the model classes also needed validation code. So, the setters got tested and covered as well.

The assert statements in TaskTest.java are an example of technically-sound code. The goal of these "assert true" statements is to test that the task object initialized. This test calls the task-constructor and verifies the object values saved to memory.

The original service classes used an array list to store objects in memory. The updated version used hash maps. This is an example of ensuring the program was efficient. Each service accomplishes the same goals with these different collections. Task service adds tasks to a hash map in TaskService.java, lines 14-17. These collections have advantages over one another in different contexts. But for this specific set of requirements, a map was the more efficient and useful solution.

Reflection

The project one program uses automated white-box testing. I used unit testing for Java called JUnit. For each java class, an adjacent test class verifies the technical requirements. Take for example, the "Appointment” class and the “Appointment Test” class. First, I wrote the "Appointment" class based on the requirements. Then I wrote the "Appointment Test" class to verify efficacy. This method is white box testing because I know how the program operates.

The JUnit tests run against individual components of the application. One thing I have not done is functional testing of milestone applications. This would mean running the application as a user to verify functionality. To do this, I would still test the technical requirements. But this strategy could also test that the model and service classes work in unison.

Functional testing would be more critical if the application had a front-end component. The back-end logic aligns with automated unit testing. This is because the back-end logic can exist in a bubble. Whereas the front end might get distributed. So, it must exist in a variety of client-side environments. Functional testing in these different environments could catch the unknown unknowns.

Mindset

There is always some element of testing when developing software. In my coursework I have never adopted the mindset of "write the code and walk away". But this project was different from most. My mindset during this course has been to "begin with the end in mind". This means I have not been writing code from the bottom up. Instead, I am taking requirements and turning them into code. That includes the JUnit tests as well. In the past I would write code until the program met all the requirements. This time I was careful to only write exactly what the requirements called for. But there were instances in the model classes where I had written code that was too simplistic. For example, I needed “unmodifiable maps” to make the service more secure. These were not a technical requirement, but they were best practice.

Software developers can take defects personally. I have seen this in others at work and I have seen it in myself. It may even be beneficial for developers to have the mindset that they develop perfect code. Whether it is perfect or not, they are holding themselves to a high standard. This can result in higher quality code. But they would have to work hard against that bias to write effective tests. In testing my project two code, I had a surge of pride whenever a test passed on the first try. I would then feel ashamed if they did not. So, I can imagine learning to write tests in a way that increased their chances of passing. Rather than learning to write tests which verify the quality of the code.

Technical debt is another thing I have seen at work. So, I have some idea of how much it costs to fix. The best long-term strategy is to not take defects personally. Defects seem inevitable anyway. But it is not inevitable that the developer learns from them. The commitment to quality code makes the developer more valuable to the company. But the developer is less likely to learn from defects if they take them personally. Or at the very least, they will learn slower this way. Which makes them less valuable and less likely to have job security. So, it is a good plan to learn from defects and not cut corners in the future. Not only for the sake of the business, but for the sake of increased opportunity for the developer.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing : An istqb-bcs certified tester foundation guide - 4th edition. BCS Learning & Development Limited.