Summary and Reflections Report

Sarah Deleppo

Southern New Hampshire University

CS-320

Professor Moscon

20 February 2021

**Summary**

**Unit Testing Approach**

To ensure that my unit tests covered all the software requirements laid out in each portion of the project, I approached each assignment as I would any project in a professional setting. By taking a Software Development Lifecycle-based approach, I viewed each set of class requirements as acceptance criteria for completing the assignment. As such, I wrote each JUnit test as a form of acceptance testing, the purpose of which is “to evaluate the system’s compliance with the business requirements and assess whether it is acceptable for delivery” (STF, 2020). Additionally, when developing the mobile application itself, I ensured that my code was as concise as possible while meeting the software requirements. Due to this, nearly every method written in each class was executed by my JUnit tests, as each line of code was required for the functionality of the system as a whole. This ensured my JUnit tests had a high coverage percentage and evaluated each class based on the requirements for each.

**Writing Junit Tests**

To ensure my code was technically sound, I first wrote each method in each of my class objects to accomplish the software requirements laid out in the project. Next, I wrote test cases to make certain all code written functioned as intended. For example, the following snippet of code illustrates a JUnit test which utilizes the Appointment Service methods “addApt" and “deleteApt”. The test case below ensures the following functionality: first a new appointment is added to the Appointment Service HashMap, then said appointment is deleted from the HashMap by citing that appointment’s ID as its key. Finally, the assertNull method is used to ensure that said appointment is null and no longer in the HashMap.

*@Test //tests if appointment was successfully deleted void testDelete() { Appointment apt = AppointmentService.addApt("111", new Date(), "descript"); AppointmentService.deleteApt("111"); assertNull(AppointmentService.getApt("111")); }*

By running this test case as a JUnit test within my IDE Eclipse, I can see that the test passes, which means both the Appointment Service class and the test case functions as intended and therefore is technically sound.

When initially writing my JUnit tests, I was calling fail when the code I was testing did not pass the acceptance criteria, however, this was inefficient. I ended up writing more code than necessary for my first two milestones which slowed down my tests overall. To make this more efficient for my final project, instead of calling fail when a test did not pass, I utilized the assertion class within Java. The following snippet of code reflects these changes, this code tests whether a new Contact phone number allows only digits, not letters. Rather than calling fail if the phone number contained letters, I used assertEquals to check if the phone number was empty and did not allow letters. This allowed me to ensure my code executed faster, was more concise and efficient overall.

*@Test*

*//test if phone number allows letters*

*void testAllowsLetters() {*

*Contact con = new Contact("test", "test",*

*"test", "tttttttttt", "test 1029475");*

*assertEquals("", con.getPhone());*

*}*

**Reflection**

**Testing Techniques**

Employing techniques such as regression testing allowed me to work cautiously while changing my code to ensure the changes, I make to my program do not negatively affect the interrelationships and complexity of the program as a whole. Regression testing is re-running tests to ensure the previously tested software preforms after making changes to the code. Regression testing is vital in nearly all software development projects due to the fact that programs are constantly being changed, optimized, and edited in order to make certain the code is secure and as efficient as possible. Re-running tests allows for developers to make big changes to code while testing to see how relationships between objects are affected.

For my project, I mainly utilized functional testing as I was writing JUnit tests, and did not use many non-functional testing techniques. Non-functional tests are tests that do not involve running code, and generally check non-functional aspects of software such as usability, and performance. As I took on the role of a developer for this assignment, I mainly tested my code using functional tests which ran portions of my code to check the functionality. However, in a situation where there is a dedicated tester and software that is to be tested outside of the source code, non-functional testing is vital to the Software Development Lifecycle. This ensures the software runs smoothly, performs well, and reliably.

**Mindset**

As I worked on this project, I adapted the mindset that it is better to take the time ensuring my code runs efficiently and smoothly, rather than take shortcuts. As a result, there were several instances in which I had to go back and change existing code to make it more efficient. However, due to the fact that the code was complex, and several of the methods relied on one another to function, I had to employ my fair share of regression testing to ensure that my code was still functioning as intended. Regression tests are used when you make changes to the original code after creating unit tests, therefore you must re-run or edit as needed your unit tests to ensure that the changes you’ve made introduced no new bugs. I found myself continuously re-running tests to ensure that any changes I made did not break my program in order to be as cautious as possible while developing. For example, early on when I was initially writing the Contact Service class, I had attempted to store Contacts within an array, I ended up changing this later to a HashMap as I felt using contact IDs as keys within a map would be more efficient. When I made these changes, I ended up having to completely rewrite my JUnit tests to reflect them, otherwise the tests would not function appropriately.

When assessing code as a developer, it can be all too easy to overlook mistakes or inefficient code, as while the program makes sense in one developer’s mind, it may not to another. To prevent this bias from happening with this project, I employed the help of a fellow software developer to review my code. I told them of the acceptance criteria, I.E. what the program is attempting to achieve, and asked if they felt my code accurately meant those requirements. Additionally, this allowed me to see my code from a different, fresh perspective, which helped make my code more efficient and more readable to other developers. In a professional context, code review is invaluable as often there are multiple developers working together, so having readable code is of the utmost importance to working harmoniously.

Cutting corners can lead to errors released in the product a software engineer is designing, inefficient messy code, or even a product that is not functional at all from a product requirements standpoint. Not only does this reflect poorly on the software engineer themselves, but it also reflects poorly on the company as a whole. As a software engineer, specifically a tester, we act as gatekeepers for the company’s products. If software is released with bugs, defects, sloppy code, or simply did not function properly, the company as a whole is the face of this failure. Not only this, but dugs and defects, depending on the software one is writing, could have potentially deadly results. For example, a company by the name of CareFusion has had several emergency recalls involving their medical equipment as the defects in their software caused patients to miss vital doses of medicine (Garde, 2014). These released bugs could have cost people their lives, and while not all lack of testing can have such detrimental effects, it is important to act accordingly to prevent the release of faulty code for the sake of the product and company. As a software engineer, it is my responsibility to provide clear time estimates of my tasks to my peers when coding. This will help to avoid technical debt. In worst case scenarios where a time limit has been reached, it is of the utmost importance to communicate the situation to my team and discuss a plan where correct efficient code is released. In the end, releasing bugs and defects will cost the company more time and money, so these steps can be utilized to avoid this.

References

Garde, D. (2014, May 21). CareFusion runs into more 'DEADLY' trouble With infusion pump.

Retrieved February 17, 2021, from https://www.fiercebiotech.com/medical-

devices/carefusion-runs-into-more-deadly-trouble-infusion-pump

Stf. (2020, September 13). Acceptance testing. Retrieved February 19, 2021, from https://softwaretestingfundamentals.com/acceptance-testing/