Southern New Hampshire University

Project 2: Summary and Reflections Report

Preston Burkhardt

Dr. Angel Cross, DIT, MMIS

CS-320-T4514 Software Test Automation & QA

April 16, 2022

Summary

My approach to writing unit tests for each of the three features was completely aligned with the software requirements. We were given specific requirements for each part of the classes to be built each week. To ensure that the classes functioned properly and were in line with the given requirements, I made tests that checked to make sure invalid input was handled properly and also tests to make sure valid input was handled correctly. For example, in the Task class, a task name couldn't be longer than 20 characters and it couldn't be null. I made a unit test that tested a task name that was more than 20 characters, then another one where the task name was null, and finally one where the task name was under 20 characters. I also made a test that checked for proper handling of a blank task name since all tasks should have at least a name.

I tried to ensure that my Junit tests were of a high quality. I did all of my coding in the IntelliJ IDEA integrated development environment (IDE) which allowed me to run my tests and generate a coverage report with class, method, and line metrics. All of my packages had 100% class and method coverage. For line coverage, all of them were greater than 90%. This means a vast majority of all of my classes were covered by my Junit tests. The class I had with the least amount of coverage was my Appointment Service class. For this class, I had 100% method coverage, but only 86% line coverage. When analyzing why this was, I found that the lines not covered were in a counting helper function used for testing purposes. Initially I had other classes that had below 80% line coverage. I refactored those classes so that they were easier test and added more tests to cover any lines that were missed. Overall, I feel that having 100% method coverage and more than 90% line coverage across all of my packages shows that my JUnit tests were effective in ensuring my program will function according to the requirements provided to us.

I ensured my code was technically sound by using concepts that I've learned from previous classes at SNHU and through some side projects I've done outside of school. One of these concepts is the use of private variables and public setter/getter methods to ensure that other parts of the program can't tamper with a class's variables unless explicitly calling a method to do so. This also allows us to implement logic for said variable to ensure it is set correctly. To ensure that these setter and getter methods were technically sound, I built tests that would feed them valid and invalid input, then see if they performed as they were expected to. An example of this is my setTaskNameCheck() test on lines 28-37 of my TaskTest.java Junit test file. This test was made to test the task name setter method, which in turn tests a helper function that takes care of all of the logic behind setting the name. The logic function, called nameLogic(), is used to ensure a task name can't be null and can't be more than 20 characters. The setTaskNameCheck() tests this by trying to make a new task with a null name, then ensuring that the taskName variable is not null. It then tries to set the task's name to one more than 20 characters and checks to see that the actual name is not more than 20 characters. Finally, it tries to set the task name to a valid name, then ensures that the valid name is returned when the getter method is called.

The above also covers how I made my code and tests efficient. To cut down on repeated code, I made private helper functions to handle things like the naming and description logic of the different classes. This made it so that if the requirements for a name ever gets changed (EX. maybe instead of 20-character limit a 10-character limit) it can be changed via updating only the code in the helper function versus updating the constructor and the setter functions. An example of this is in my Task class on lines 57-66, where you will find the function nameLogic(). Doing this also helped me be efficient with testing since I didn't have to make a separate test for the constructors and the setter functions. They both call the same helper functions, so testing the setter function tests the same helper function that the constructor calls. Also, since the tests often use the getter functions, they inherently test those as well. You can see an example of this in my setTaskNameCheck() on lines 28-37 of my TaskTest file.

Reflections - Techniques

During this class, I used various software testing techniques. The ones I used the most were specification-based techniques. This is when you develop tests based off of the specific requirements for the software you are writing. For example, we were told that the Appointment class had to have a string variable to hold the description of an appointment. This description couldn't be null and had to be less than 50 characters. When I made a test for the function that handles the logic for assigning the appointment's description, I used equivalence partitioning. Equivalence partitioning is where different inputs can be put into different partitions/groups. All the inputs in a said group are treated equally by the software program (Morgan, Samaroo, Thompson, & Williams, 2015). So, for the appointment description variable, this means a string that is 51 characters long would be treated the same as a string that is 100 characters long. Likewise, a 10-character string would be treated the same as a 40-character string. This makes it so that we don't have to build out a ton of tests testing every possible string, only one to test the valid number of characters and one to test the invalid number of characters. I also used structure-based techniques. I did this by building tests that went down each branch of it/else logic used in some of my functions. For example, my dateLogic() helper function on lines 93-110 of the Apointment.java file has one if statement and one if/else branch pair, so three total logic branches the program can take based on input. I built the setAppointmentDate() test in the AppointmentTest.java file to go down each of these branches, ensuring each one functions as intended.

I didn't use any experience-based techniques during this class since I had no experience of building tests or testing software. The more experience I gain, the more these techniques will become available to me. Also, I did not use static testing. Static testing involves analyzing the code without running it. For this class most of the tests were in the form of JUnit tests, which I built and ran to ensure that they worked while also providing sufficient coverage of all classes, methods, and lines of code.

Each set of techniques has its practical uses as well as implications. For specification-based techniques, their practicality lies in the fact that they test the specifications given for the project/program. Most, if not all, software will have some kind of requirement(s). Sometimes these requirements are given by a customer or governing body. Specification-based testing techniques will ensure that these requirements are met and the software meets the needs of the customer who commissioned it. Structure-based techniques are practical in that they can test the not so obvious requirements. Often times a customer doesn't give exact requirements like we received for our assignments. A customer requirement may be "we need something that a user can use to update their password" and it's up to the development team to build come up with all of the different functions to make this happen. Structure-based techniques will be used as different parts of this feature are built and tested. Static testing techniques are practical because the code doesn't have to be run to perform these types of tests and analysis. This may come in the form of code reviews or pair programing. Static testing is helpful in that it gets another set of eyes on the code. We all make mistakes and we've all missed a semicolon or perhaps built a function that would later not work. Sometimes these mistakes are easily caught by someone just going over the code before it makes it to the test environment to inevitably fail. Finally, experience-based techniques are practical in that they come from past experiences. Perhaps a developer has built something similar in the past and they remember all the edge cases that almost broke their production build. Obviously, they don't want to do that again and this is where experience-based techniques are utilized.

Reflections – Mindset

With being a software tester, it is helpful to be of a cautious mindset. You want to ensure that you are slow and steady with you testing so that you maximize your coverage, ensuring as much of each class, function, line, etc. is tested and tested correctly. When you are in a development role, you can sometimes get into a “groove” where you build out a class quickly without really thinking too much about it. This “groove” isn’t beneficial when testing because it may cause you to miss different things or maybe even build a test that creates a false pass. You also need to analyze the complexity and interrelationships of the code you are testing. It’s helpful to map out what variables are in a class, what functions set or mutate which variables, which functions call other functions, and what variables these functions pass or receive. Taking the time to do this will save you time when building tests as you will have an idea of how to possibly test multiple lines/functions with a single test. It also helps in having at least a basic understanding of the code you are testing and what it’s supposed to do. Imagine trying to test a car but not knowing what a car does. Knowing that the car is used to travel from point A to point B and the gas pedal is how it does it would be extremely beneficial with the whole testing process. The same goes for testing software.

Of course, when testing your own code or perhaps the code of your company, there is some inherent biases you will have. It is extremely important that you limit these biases as much as possible. Said biases may cause you to miss some important tests or “take it easy” when inspecting your code. You don’t want this to be the reason your code or the product your company built fails after release. This could cost you your job, you/your company’s reputation, and possibly even lead to litigation/legal action. In order to limit my own bias, I try to test to the specifications given. I also like to take breaks in order to come back to the code with a fresh mindset. Finally, to mitigate my own biases, I try to imagine all the ways that the code I'm making or testing may fail and then build tests around these scenarios. It can't be stressed enough how important it is that your testing of your own code or your company's is not compromised by your biases.

Finally, it's important to be disciplined when it comes to having a quality mindset. When you build your programs, think ahead. Think about what it will be like if you have to come back to implement a change to your code in the future. Think about how you will test the code that you just wrote. Writing code that is A) easy to implement changes to/reuse and B) easy to test, will make your job and the jobs of others a lot easier now and in the future. Technical debt is an action where quality is sacrificed for time. Corners might get cut so that a piece of software is built/shipped quicker. While the payoff is usually in the near term, the issues it causes later on down the road often surpass the initial benefits. When writing code or testing code, if you cut corners, it only hurts yourself or your company when it comes time to inevitably fix/build the corners that were cut. As a practitioner in the software engineering field, I will always try to voice my concerns on possible outcomes if shortcuts are taken. A customer may be near-sighted when pushing for deadlines for the building of their product. It's important that the development and/or testing team bring forward the issues that the customer may face later as they may not even be aware of said issues.

References

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide* (3rd ed.). (B. Hambling, Ed.) BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb