CS320 Summary and Reflections Report

Tristin T. Watson

Southern New Hampshire University

Table of Contents

[Table of Contents 2](#_Toc117338013)

[1. Summary 3](#_Toc117338014)

[1a. Describe your unit testing approach for each of the three features. 3](#_Toc117338015)

[1b. Describe your experience writing the JUnit tests. 4](#_Toc117338016)

[2. Reflection 4](#_Toc117338017)

[2a. Testing Techniques 4](#_Toc117338018)

[2b. Mindset 5](#_Toc117338019)

[References 6](#_Toc117338020)

CS320 Summary and Reflections Report

# 1. Summary

Unit testing was quite a challenge. Testing basic getters/setters and constructors was no problem and felt simple. The more complicated methods really started going over my head though. This basically meant that the contact class, appointment class, and task class were tested to a decent mark while the contract service class, appointment service class, and task service class were left feeling a little lackluster. To provide an example of what I mean, the contact class, appointment class, and task class all had 100% successful unit testing coverage. The task class unit testing is shown in the picture below.



Although the three simple classes had good test coverage, the three more difficult contract service class, appointment service class, and task service class all had partial coverage and failing tests as shown in the picture below.



Text

Description automatically generated

I think part of the reason for this was a lack of understanding of unit testing itself. Unit tests are meant to test the methods I wrote but at the same time, you need to think up the code to test your code. I found that rather daunting, to say the least. Although writing code to test your code seems simple, I spent several hours a week trying out different options but never found anything truly solid and noteworthy. This is clearly just my lack of understanding and not that there is not a solution. With a little more effort in searching for a solution to my problems and I might have discovered it.

To ensure that my code was technically sound, I wrote code based on the software requirements requests. An example would be that if the requirements asked for a method that lets you add tasks to a list by id, then you make just that. A good way I found to perform a requirement like this was to auto-assign an id to the task when generated. That makes the id nonupdatable and not null to start with. This is also efficient since the only time an id is referenced is when a method needs to be used. There is no way to change the id, and it is unique to the specific set task.

The snippet of code below generates a new id by adding 1 to the last id number and then changes the id to a string to use in the task constructor. Lastly, the constructed object is stored in a HashMap by the id string. The reason I chose to store the objects in a HashMap is to save memory and make the code more efficient when looking up the task by id again. If the object by id was stored in an ArrayList, then each time you need to grab a specific id, you would need to iterate through the entire ArrayList until you found the id. HashMap’s do not have this problem since they just use the specific id as the reference to the stored object.

Text

Description automatically generated

# 2. Reflection

The specific testing techniques that I employed during this project were testing using JUnit, Whitebox testing, and static testing. Unit testing tests the logic of my code. Using Junit specifically lets me understand which of my objects and methods are working correctly. You can check many different things with unit tests such as null, true, and error throws. Whitebox testing lets you test the inner workings of your software. The way I used Whitebox testing was to help ensure that the logic of my program was working smoothly. Lastly, is static testing. Static testing lets you check for defects in your software without executing code. Static testing helps find defects in code and can find problems in the code throughout development.

The software testing techniques that I did not use during my milestone work are manual testing and dynamic testing. Dynamic testing is the testing done when code is run. I never ran my code, so I do not know for sure what defects may come up when the program is used. Manual testing is when a user tries different interactions with the system manually. Since the code was never run, manual tests were not used to find defects.

Unit testing allows you to test code in units. To elaborate, if you have five different classes, then you can test each class individually as units. Unit testing is a simple way to determine if your program's logic is functioning correctly. Whitebox testing lets you try input and output cases to check the logic of your code. You can ensure that your program’s coverage is thoroughly done through Whitebox testing. Static testing allows you to test your software without running code. It allows you to check the flow of your program and can give you a clear view of where potential defects in your program could pop up. Overall, these three testing procedures can allow you to make a seamless program that has proper logic and flow.

## 2b. Mindset

Caution when testing and coding is a must. While not all code should make a programmer weary, if you are working with any foreign/new software packages caution is always necessary. Although we never specifically encountered it in this course, in the software security course, we covered vulnerabilities from different packages and frameworks used in projects. An example is that one specific defect was found in Apache Sling JCR Base. This vulnerability could let attackers perform injections to remotely access data when running older JDK versions. Although a little off-topic, what I am getting at is that testing code is very important. Interrelationships in code can help test successfully. Most methods have many connected actions that are taken to execute them. The best way I found to test the updateTask method in my TaskService class was to first use the addTask method. The reason for this was that if there were no task objects stored in the HashMap to start, then the easiest way to fix that is to add one. The test failed but still explains the thought process that I was working on.

Text

Description automatically generated

Limiting bias was rather simple in reviewing my code. The way I managed it was by looking at the code objectively. Does the code solve the software requirements and testing requirements. The contact class, appointment class, and task class passed both the software requirements and testing requirements. The contract service class, appointment service class, and task service class on the other hand passed the software requirements but not the full testing requirements. The reason for the failed testing requirements are from the failing unit tests and a lack of full unit test coverage on the classes themselves.

Discipline is a must in every profession, let alone software engineering. A successful software engineer might not always find the best solution but can give their best to the solution they found and keep striving to improve. Cutting corners while testing code can create greater flaws later down the line of production. That being the case, it is a must to properly test code during and after you create it. Avoiding technical debt is simple in words but sometimes difficult in action. The best way to avoid technical debt is the test code during and after you produce it and follow software requirements to the T. An example of avoiding technical debt is in my task class, I followed the software requirements exactly when developing the getting and setters. To elaborate, the description variable is supposed to be less than 50 characters and not null. To solve this problem, I set the parameters for the description variable to check for null and if the variable was attempting to be set to greater than 50 characters.

Text

Description automatically generated

References

gfg. (2022, December 20). Software testing: Static testing. GeeksforGeeks. Retrieved February 5, 2023, from <https://www.geeksforgeeks.org/software-testing-static-testing/>

imperva. (2020, September 24). What is white box testing: Types &amp; Techniques for code coverage: Imperva. Learning Center. Retrieved February 5, 2023, from <https://www.imperva.com/learn/application-security/white-box-testing/#:~:text=White%20box%20testing%20is%20an,and%20integrations%20with%20external%20systems>

NVD. (n.d.). *NVD - Vulnerability Database*. NVD. Retrieved February 16, 2023, from <https://nvd.nist.gov/vuln/search>

Thanh, V. (2020, May 22). *15 testing methods all developers should know*. CircleCI. Retrieved February 5, 2023, from <https://circleci.com/blog/testing-methods-all-developers-should-know/?utm_source=google&utm_medium=sem&utm_campaign=sem-google-dg--uscan-en-dsa-maxConv-auth-nb&utm_term=g_-_c__dsa_&utm_content=>