CS-320 Project 2

Summary and Reflections

Nicholas D. Nealeigh

Southern New Hampshire University

CS-320-J7223

Professor Joseph Rangitsch

17 DEC 2022

### 

### 

### 

### 

### 

Contents

[1. Summary 3](#_Toc122282064)

[2. Describe your Unit testing approach for each of the three features 3](#_Toc122282065)

[3. Describe your experience writing Junit tests 3](#_Toc122282066)

[4. Reflection 4](#_Toc122282067)

[5. Testing Techniques 4](#_Toc122282068)

[6. Mindset 5](#_Toc122282069)

[References 7](#_Toc122282070)

# Summary

## Describe your Unit testing approach for each of the three features

In regard to my unit testing approach being aligned to the software requirements, I accomplished this by ensuring that I read the assignment guidelines for each module and ensured that the code I made followed the client’s strict requirements as far as what could be mutated, and the size of various object parameters. This can be seen in my Contact.test file from lines 30-73. These are the set/get methods. The client requested that contact ID cannot be changed after it is set, so I ensured that there would be no set function for it. Each of the other set functions have an “if then statement” that ensures that their input cannot be null or greater than whatever limit the client provided. This code could have been improved had I been able to implement this restriction into the constructor as well, however I was unable to overcome this challenge. Whichever function that will process input and pass it to the constructor should handle these limitations instead.

The overall quality of my Junit tests ended up being pretty good based on coverage percentage. I ended up with well above 80%, ensuring that most pieces of code that actually “do something” were tested. This coverage percent indicates which “statements were executed in at least one test case and which ones weren’t.” (Testing) by this statement, it’s clear that most of my code ran through tests to ensure that it is viable and functional.

## Describe your experience writing Junit tests

Writing Junit tests was a new experience for me. It included many hours of researching, learning how to use “@TEST” as you can see in each of the test.java files, as well as learing how to utilize @BeforeEach to improve efficiency later on. This was a change that I first implemented in module 4’s tasktest class and it saved time and space not needing to initialize the different variables in each “@TEST” function. This can be observed in Line 11 of TaskTest.java, as well as in module 5’s AppointmentServiceTest.java and AppointmentTest.java files line 14 in both.

I made sure that my code was technically sound by running it through a number of tests, ensuring that the constructors and get functions worked properly. These tests all all observable in the classTest.java files and ensure that the program functions properly with appropriate input. I ensured that my code was technically sound by following the standard practices that I’ve learned when programming in java throughout my education. I ensured that the class’s variables were set to private. This can be seen in lines 6-10 of contact.java, while I also included default constructors, parameterized constructors (lines 13-28) and set/get functions to ensure that the variables could be accessed(lines 31-73). As each of the classes in this project were similar, appointment.java and task.java bear similar features. For the service files, I ensured that there were available functions for adding, deleting, and updating the tasks as requested by the client. This can be seen in lines 13-61 of the contactservice.java file.

Efficiency is a little more difficult for me to show. We weren’t working with large data sets, where I would consider stuff like BigO efficiency. Most of the add/delete/update functions just feature sequential searches(line 15, 32, and 42 of contactservice.java show this), had I implemented an ordered list, I could have utilized binary searches instead and cut down on some time. Without knowing the size of the data set however, it didn’t really make sense to try to implement that due to my unfamiliarity.

# Reflection

## Testing Techniques

The software testing techniques that we went over in this course included static and dynamic testing. I think that I utilized both of these types of tests for project one. One component of static testing involves a visual overview of the code. I did this by reading through the code, looking for mistakes before running it. I don’t have access to an automatic analysis tool, unless eclipse’s in-built syntax checks count, so I was unable to accomplish that portion of the static testing process. For dynamic testing, I believe that unit testing through Junit tests counts. Junit is a separate module which we utilize and run the code through to check for coverage rate, failures and errors. As the code is being run, I believe that this counts as dynamic testing.

Some other testing techniques that I didn’t implement include black box, white box, regression, and usability testing. (Boog, 2022) The black box method involves checking the code from the perspective of a user, this wasn’t possible due to the lack of a driver class for the project, with no way for a user to interact with it. Likewise, White box testing was not possible, as I lack the experience needed to run these sorts of tests. Regression testing is a topic that we didn’t really cover in the course, and seems to involve using tools to check for inconsistencies in new builds, something that also didn’t apply to the project. Due to the lack of a gui, usability testing also didn’t feel applicable.

## Mindset

My mindset in the project was very much as an inexperienced beginner. I was careful in every step, ensuring that I didn’t make mistakes or deviate from the module’s instructions too much. An example of caution employed in this project was for each of the classes, (contact, task, appointment), while they bore obvious similarities, I ensured that their code wasn’t simply copy pasted with the variable/function names changed. An example of this is within the setdescription function that all of these classes have, the length is also different for them. This can be seen if we look at line 52 of contact.java versus line 49 of appointment.java. Being careful when implementing similar classes can help prevent mistakes. It would have been really easy for me to accidentally use the same length for description in each class.

I tried to limit bias in my code by just going over each class after I was finished with them, even if I was already confident that they were free of errors. I think that an overconfident developer can easily make a few mistakes here or there and allow those mistakes to get into a final build. It’s good to be confident in your work, but no to the point that you feel review is unnecessary.

As provides of goods and services, which software developers often are, we should always strive to provide a perfected end product. The closest career field for this analogy would probably be a carpenter or a welder. They also produce goods and services, though those are physical rather than virtual, the idea is still there. You wouldn’t want to buy a bed from a carpenter with “carpenter debt” where the carpenter will need to come back and fix things down the line as he cut corners on your project. In the same vein, you wouldn’t commission a welder to weld a pipeline or bridge with “welder debt” impying the chance of failure due to cur corners down the line. It’s normal for these tradesmen to provide perfected finished products without an expectation of refactoring. It should be normal to expect the same from software developers.

# References

*Testing code coverage in Eclipse - Cornell University*. Cornell.edu. (n.d.). Retrieved December 19, 2022, from https://www.cs.cornell.edu/courses/JavaAndDS/files/codeCoverage.pdf

Boog, J. (2022, December 10). *9 types of software testing in software engineering*. The QA Lead. Retrieved December 18, 2022, from https://theqalead.com/test-management/types-of-software-testing/