Eric Schooler-Davison

Omar Toledo Lopez

CS-320

August 17th, 2025

Project Two

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements?** Support your claims with specific evidence.

In order to best support the software requirements I designed my JUnit tests around the requirements. They were designed to test that the requirements were met, for instance I used add task/contact/appointment success to check that the requirements had been met as long as the data met the expectations.

* + 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

Coverage percentage is used to indicate how thoroughly you have tested your code. A high percentage shows a comprehensive test of each class, line, and branch of code. My coverage percentage was above 80 percent across the board, showing an effective coverage of code by parsing each line at least once.

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound?** Cite specific lines of code from your tests to illustrate.

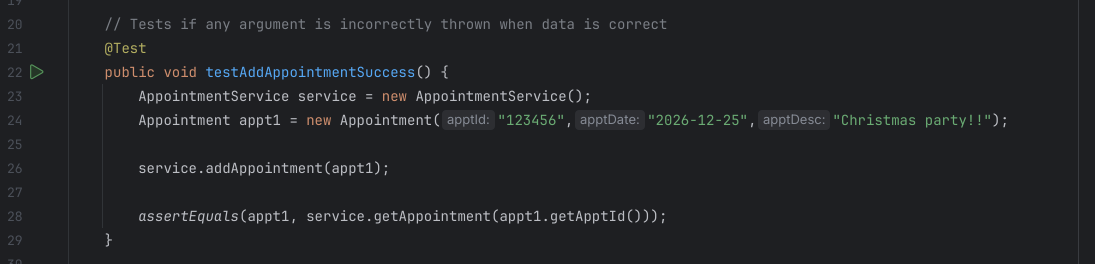
In order to ensure that my code was technically sound, I had to create JUnit tests that checked for proper execution of code. This is again where coverage is incredibly important. Its also important to tailor your tests to test for the functionality required. For instance, the Contact and ContactService class both required that addresses, names, and phone numbers were not too long. To test this I used the Contact class to check for failures in each.



The screenshot above is an example of contact phone number. The phone number had to be exactly 10 digits, no more or less. Because of this I made a JUnit test for both too short and too long. All tests passed so we know that the method properly throws an exception when it should.

* + 1. How did you ensure that your code was **efficient?** Cite specific lines of code from your tests to illustrate.

One thing to keep in mind when writing code is never to write the same thing twice and always make good use of white space and comments. I will also mention the JUnit tests, where a fail case needs to be specific to the exception being thrown, a success just needs to make sure the code executes correctly. While one could write a JUnit success test for each method (and their might be some edge case where you would), you can also just make a success JUnit test that checks that the whole program works as intended when given the data it expects. I got better and better at doing this and my last one (appointmentService) checked for accuracy by testing a appointment against one that had been stored, see below.



1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

JUnit tests were the main mode of program testing done in this course, as evident by its name, unit testing was the testing technique employed. Of course, unit tests are only unit tests if they are used on a unit of the code, but we also performed functionality tests when we ran the whole program, testing that it functioned properly with success tests. It is also worth mentioning that the testing preformed was a kind of white box testing, where we verified functionality, parsing every line of code for maximum coverage.

* + 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.

Black box testing was not performed during this course. The program we had did not have any output or accept any input. Black box testing is where the program is a mystery box and you only test what goes in vs what goes out. We also did not complete any non-functionality tests where the priority is optimization, security, and usability. The program was not in a usable state. No integration testing was done as there was no preexisting code to integrate the new code with. All code was made on its own, and not part of a complete whole.

* + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

Black box testing involves only testing inputs and outputs, but the best example I think of is beta testing for video games. The player is not aware of the code behind the game, the only thing the beta tester sees is how the game behaves when they play it. This is very useful for detecting errors that were not considered by developers, or for QOL enhancements. The end user is the goal after all, so testing that it behaves properly for the end user is important.

Non-functionality tests are not testing if the program functions, and instead test if it functions WELL. This is were security and performance come into play. As we have already discussed video games I will continue to use them as an example. For a video game, non-functionality testing would look for performance such as FPS. If the game works that’s fantastic, but if you get 5 FPS on a video game, the experience would be very poor. Technically the program works, but if it does not perform as expected by the end user, can you really call it a success?

Last thing I mentioned was integration testing, where preexisting code is taken and tested to ensure that the two parts work as a whole. This is usually done continuously in between sprints, adding newly made code into the code base and testing proper functionality between each part. We don’t want to get to the end, and put maybe 20 classes that were made in a vacuum, each part is of the same whole, and it should act like it.

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution?** Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

Code to me feels like a kind of art, you have to understand what the big picture is supposed to be while making the code. This is why people employ pseudo code and flow charts to plan the program at a higher level. I felt as if I was a member of a team, creating code parts to integrate into a larger whole that I did not see. With that in mind, I focused on testing for maximum coverage, and revisiting my code to correct issues or mistakes I made. In fact, after getting our feedback for the contact class in an earlier module, I actually fixed and applied feedback right away, even though I did not realize at the time that it would be part of the final project. Coding is complex, it takes many people and many moving parts. It’s not like a language that we speak, the code I make has to have good comments and thoughtful design. Spaghetti code is impossible to read and thereby impossible to understand. Even worse is the fact that its impossible to implement code with other code if its incomprehensible. This is why best practices exist, because every line you write is but one small part of a much larger whole.

* + 1. Asses the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

Bias is prevalent among coders, and exists in many forms. I am not yet confident in my own abilities to believe that my code lacks flaws. But, it is easy to believe that you fully understand the requirements of functionality, and are applying them properly. Also, the coder may not consider how the user might actually use the application. For instance, the contact class restricts the phone number to be exactly 10 digits long. However, if we are not only discussing the US, then a country code would be required, which would not be possible to enter. You also become more and more proud of what you’ve created, making it difficult to be critical. If a coder was to make something completely on their own, they could easily not understand the use of the program, or believe that “Hey, class x is perfect, it passes all JUnit test!” and thus not look there.

* + 1. Finally, evaluate the importance of being **disciplined** in you commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Every mistake you make will come back to bite you or the team sooner or later. You also can never completely avoid making mistakes. So why would you ever choose to make more work for yourself and your team CONSIOUSLY, when it could be done right the first time. Considering the cost, which rises exponentially the later a mistake is found, I am committed to create code where I am unconscious of intentional error. To do it right the first time and not compromise mine and the teams success. Even as a lone coder, imagine how overwhelming it would be at the end of the SDLC to get a program that does not function and not even knowing where to start.