# Summary

Describe your unit testing approach for each of the three features.

To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.

My approach was rooted in writing a set of class functions that carried out the desired task from the requirements document (rubric). This meant that my strategy was conceptually to make sure that the desired result was produced by the code, followed by reinforcing those tests to make sure that the code preformed internally as desired, even when that desire was implicit rather than explicit. By also making sure that I had thorough code commenting, I also increase the maintainability of the software.

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?

At one hundred percent, my unit tests covered every line of code, which means from a white box perspective that I have fully debugged in respect to its internal functionality. Since I combined the intent of the program with the internal function of the program, this means that these unit tests were also far more effective than just the coverage percentage. Furthermore, I made sure that the maintainability of the software was high from writing clean and concise tests that were segmented into proper units with explanations of what the purpose of each is, along with an elimination of magic numbers to leave no guesses on why any particular value was chosen.

Describe your experience writing the JUnit tests.

How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.

I kept my code technically sound by ensuring that edge cases not covered in the requirements documents were still covered in my code’s implementation. For example, in the Appointment class there is no minimum requirement on size, but I implemented one anyway:

/\*\*

\* @param string New description

\* @return N/A

\*/

public void setDescription(String newDescription) {

int maxLength = 50; // Upper bound from design doc

int minLength = 2; // "Go" is the shortest sentence

if(newDescription == null || newDescription.length() > maxLength ||

newDescription.length() < minLength) {

throw new IllegalArgumentException("Invalid Description");

}

description = newDescription;

}

This is also true of the Contact and Task services, which also had the same requirements issues with minimum length:

/\*\*

\* @param string New Address

\* @return N/A

\*/

public void setAddress(String newAddress) {

int maxLength = 30; // Upper bound from design doc

int minLength = 6; // "1 8 Av" is a real address

if(newAddress == null || newAddress.length() > maxLength

|| newAddress.length() < minLength) { // null OR too long OR too short

// Test this and change to add minLength || newAddress.length() < minLength

throw new IllegalArgumentException("Invalid Address");

}

address = newAddress;

}

These simple extra checks make sure that the function works as expected by a user, such that addresses and descriptions that are too short from being malformed are rejected. In the example of Contact.java, I made sure to research what the shortest possible addresses were in the United States to ensure that users would not be able to write addresses that were too short as well as those that may be too long.

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

To keep my code working efficiently, I made every If() statement not branch more than just once, keeping the time complexity to . I used the HashMap function for my Service documents (ex. ContactService.java) to obtain the best average performance possible for the use cases from a user’s perspective, as hashing runs at , keeping each function running at a blisteringly quick pace.

# Reflection

Testing Techniques

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

In this project I employed multiple major forms of testing, including boundary testing, equivalence partitioning, static testing, and so my unit tests contained a pervasive white box testing perspective. Static testing was used in evaluating the requirements set out for me, ensuring that nothing was majorly missed before any code was written. Boundary analysis testing and equivalence partitioning found their usage in testing minimum and maximum values of functions, ensuring their domain is covered in their use cases. Decision testing was used to construct the general code coverage functions, as they allowed me to understand the paths required to reach every line of logic.

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

I do not believe that I fully utilized the concepts behind black box testing. Since black box testing requires you to ignore the internals of the project, it was very difficult to divorce myself from my own knowledge of writing the program, making black box testing ineffective given my position. However, my code’s outputs were already correct and verified from white box testing.

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

In projects where testers are unaware or unfamiliar with the internals of the program, black box testing is incredibly powerful and can help keep costs down in multiple respects as training for writing black box tests is low and is summarized by the idea of checking if your output matches your given input. White box testing requires far more training and knowledge, as you must understand the code, but is far more powerful that black box testing alone as it shows errors in the logic of how a program runs, as well as proving when certain code will not or cannot be reached. Static testing is reinforced by making your foundation for your code significantly more study, as you tease errors out of your code before it is written, saving potential millions of dollars and countless hours of work. In fact, static testing is quickly becoming my primary interest, as it helps express that the power of coding exists outside of the code, and it is instead in the logical concepts that programmers use to create code.

Furthermore, from the research of my peers as well as myself, there are practical examples of when different forms of testing would have saved lives or hundreds of millions of dollars. Therac-25 was demonstrated to be a white box testing issue stemming from an integer overflow (Schwarz, 2015 as cited in Andre, 2024). CrowdStrike was shown to be an issue of both acceptance testing and black box testing (CrowdStrike, 2024 as cited in Francisco, 2024). There is also the Mars Climate Orbiter, which had an issue that stemmed from static testing not being thorough enough (Liley, 2012 as cited in Sculthorpe, 2024).

Mindset

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.

When working with task management systems as well as contact services, there is a surprisingly high requirement for security. When security is breached on these applications, someone’s entire history of movement can be revealed, and attackers can use this to rob or defraud the individual at key moments of the day where their odds are maximized. For some individuals, their identity is largely anonymous, such as those who produce videos for YouTube or Twitch, where a location leak requires them to move. These facts show that simple services can be quite important, so erroneous calls to an API or exploits such as buffer overflows and other brute force methods would be potentially catastrophic. This means that boundary analysis on these relevant values is incredibly important, but we must also trust that the application has input sanitization in its methods for extracting values.

Assess 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.

When working on your own code, black box testing becomes exceedingly difficult to make work for you other than in the simplest of terms of measuring outcomes. Furthermore, as the person who wrote the logic of the program, I may have already fallen prey to logic errors that I may not immediately recognize. In order to try and eliminate bias, I have to think about how I feel about my code I wrote eight or more years ago, and what kinds of documentation I needed to understand it and what other problems arose in them, then apply that perspective to my current code.

Finally, evaluate the importance of being disciplined in your 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?

It is incredibly tempting to cut corners and simply jump to coding rather than gathering requirements, diagramming, pseudocode, and testing it all before coding. I very well understand this temptation, as I have been writing scripts in Lua and AutoHotkey for over ten years. The common thought among most new junior programmers is to simply throw up their hands and ask why they need soft skills, and why they need to diagram before coding. I certainly used to be that person, but as my knowledge progresses, I constantly return to the thought of how foolish those thoughts truly are. Software scales past the human imagination, as a 4/100¢ change in value of a McDonald’s cup may seem negligible, but with 100 sold per hour at every location, 41,822 locations, this is $4,014.92 per day. If the bug lasts for a single month with zero downtime, it may very well eclipse your pay even at FAANG companies as a junior. These thousands a day are the kind of error one can expect from a naive programmer using a floating-point integer for measuring cash instead of the money data type. In contrast, an issue that causes downtime is instead measured in thousands of dollars per second.

So, to then turn to my own code, it is necessary as a professional to conceptualize, gather requirements, design the system, then finally code the system, while performing testing at every single phase as seen in the V-Model (Hambling et al., 20 pp. 48-49). I am currently writing a missile guidance system and linear code communication program, and the urge to simply start coding instead of diagramming is incredibly strong. However, for the health of my own profession as well as my own sanity, I will continue to resist this temptation. I know what my code looked like eight years ago, and I will feel the same about this code too in eight more years. But, if I insist on acting as a professional first and always, I will secure my own future and allow myself to maintain my own projects.

References

Andre, J. (2024, August 13). *7-1 Discussion: Released Escapements and Defects* [Discussion post]. Southern New Hampshire University Brightspace. <https://learn.snhu.edu/d2l/le/content/1644140/viewContent/33581609/View>

Francisco, M. (2024, August 12). *7-1 Discussion: CrowdStrike* [Discussion Post]. Southern New Hampshire University Brightspace. <https://learn.snhu.edu/d2l/le/content/1644140/viewContent/33581609/View>

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS Certified Tester Foundation Guide* (B. Hambling, Ed.; 4th ed.). BCS Learning and Development Ltd.

Sculthorpe, S. (2024, August 15). *7-1 Discussion - Connor Sculthorpe* [Discussion Post]. Southern New Hampshire University Brightspace. <https://learn.snhu.edu/d2l/le/content/1644140/viewContent/33581609/View>