Project Two

Ryan Sitner

**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.

My approach to designing tests for each of the three features was based 100% off of the software requirements. For example, in the Appointment Services feature, one of the requirements was, “the appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable”. From that sentence we can identify four requirements for the appointment object: a unique ID, no longer than 10 characters, cannot be null, and cannot be updated.

For the requirement “ID cannot be updated”, I simply did not code a function to allow for the ID to be updated, no tested needs to be done on this. However, the three other requirements can be tested. For the three remaining requirements I needed to create tests to make sure the system behaves properly when input is correct, and when input is invalid.

For a unique ID, I needed to create a test that confirmed we can add an appointment with a unique ID, and a test that confirmed the system wouldn’t allow an appointment to be added with an ID that already existed in the array. For the requirement that an ID can’t be longer than 10 characters, I created a test to confirm that the system would accept an appointment with a unique ID with 10 or fewer characters and would reject an ID greater than 10 characters. Lastly, I created a test to confirm that the system wouldn’t add an appointment whose ID was null.

When creating my tests I followed this same approach for every variable’s listed requirements across all three features (Contact, Task, & Appointment)

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?

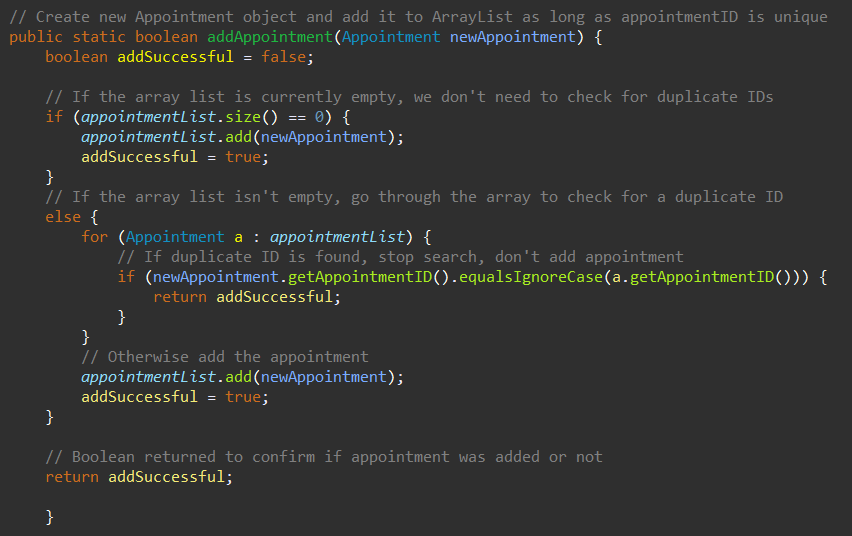
Following the aforementioned approach when creating my tests guaranteed that the coverage of the code that these tests provided would be high. This can be proven by running a coverage test on any of the six test classes that I created across the three main features.

When running a coverage test for my AppointmentTest.java class, the coverage of the Appointment.java class was 100%. Running a coverage test for the AppointmentServiceTest.java class, showed a coverage of 92.4% for the AppointmentService.java class. This trend continued across the four remaining test classes with the lowest coverage being at 82.3% for the Contact.java class (this was the very first class I built tests for and was still getting a feel for creating tests and creating strategies for creating tests). The other five test classes had a coverage rating of 90% or higher.

2) 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.

I made sure my code was technically sound by following industry best practices, making sure to efficiently comment in my code. Making sure it had enough detail to explain what the code does without flooding the code with comments on every other line.

A decent example of this is shown in the code below. The addAppointment function is a complex function that involves standard logic programming as well as nested logic programming. The comments in this function provide a clear understanding for what each part of the function does without flooding the code in comments.

Because many of the variables throughout the three projects had similar requirements, I was able to utilize code reuse, copying many of the tests and making the necessary edits.

I did my best to avoid complex code and keep redundant code to a minimum. But one place that could possibly use improvement would be avoiding code repetition. Specifically in the Appointment Service and Contact Service tests, I found myself for every test re-creating similar dates and re-creating similar contacts. I had tried to create a variable that could then be accessed by all of the following tests, but couldn’t get it to work properly and ended up moving on – having to copy/paste date and contact information and make changes where appropriate.

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

One way that I worked to make sure my code was efficient was by trying my best to make methods for things that were frequently repeated. For example, when adding appointments/contacts/tasks to their appropriate lists, I created a single function that checked to make sure all variable requirements were met, if so it would add them to the list, otherwise it wouldn’t, and then returned a boolean to signify whether or not the item A screen shot of a computer program

AI-generated content may be incorrect.was added to the list.

That being said, efficient code is probably the category that I failed the most in within these assignments. Because I’m still getting used to creating JUnit tests properly, my focus was on making sure test coverage was high, and that everything was working properly. I wasn’t thinking about if there was a more effective way of creating the code. Using a Map instead of a List to utilize the Map’s containsKey() method would help make the code more efficient and reduce the need of for loops when checking if an ID already exists. And as mentioned in the previous question, finding a way to properly create a variable that every test in a specific java file could use, would have saved a lot of time having to create a new date for each Appointment Service test.

**Reflection**

1) Testing Techniques

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

During the last three modules, the primary software testing technique that I used was Unit Testing. Unit Testing is a key practice in software development where individual components, or "units" of a program are tested in isolation to make sure they work as intended. It focuses on testing a specific section of code, typically a single function or method, by isolating it from the rest of the system to verify its behavior independently. These tests are often automated, meaning once the tests are written, they can be run repeatedly to ensure the code continues to function correctly as changes are made to the code. Unit tests are defined by clear inputs and expected outputs. The test sets the input conditions and checks if the results match the expected output. A core part of unit testing is using assertions to compare the actual output with the expected one, determining whether the test passes or fails. Common assertions include verifying if values are equal, if an exception is thrown, or if a certain condition is met.

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

There are countless numbers of other testing methods out there that were not used during these assignments. Some of these include integration testing, system testing and performance testing. Integration testing, system testing, and performance testing are distinct but complementary types of software testing, each with its unique focus and purpose. Integration testing ensures that different modules or components within an application work together correctly. It verifies that data is passed correctly between components and that they interact as expected. This type of testing typically occurs after unit testing and helps identify issues with interfaces or communication between modules.

System testing evaluates the entire software system as a whole. It checks whether the application meets functional and non-functional requirements, including usability, security, and compatibility. This testing occurs after integration testing and ensures that all parts of the system work together seamlessly, simulating real-world user scenarios.

As for performance testing, these tests focus on assessing the application's speed, responsiveness, and scalability. It helps ensure that the system can handle the expected traffic, respond quickly under stress, and be able to scale as needed. Each type of testing plays a critical role in delivering a reliable, user-friendly, and efficient software product.

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

Unit testing, integration testing, system testing, and performance testing each serve unique roles in software development, with practical applications depending on the project's requirements. Unit testing is essential for early bug detection and ensuring individual components function correctly, especially in agile environments with rapidly changing code. It allows for quick feedback during development and is crucial for code reliability. Integration testing focuses on verifying the interaction between different modules or external systems, making it vital for projects involving complex dependencies or third-party services. It helps identify issues in data exchange and communication between integrated components. System testing validates the entire application’s functionality, ensuring all components work together and meet user requirements. It's essential for large projects or those with strict compliance needs, offering confidence that the system functions as expected before release. And performance testing is critical for applications that need to handle high traffic or large data volumes, ensuring the software performs well under stress and can scale as needed. It helps optimize user experience by identifying bottlenecks and preventing failures during times of high traffic. Each type of testing addresses different project needs, from code reliability and interaction to overall system behavior and performance under load.

2) 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.

As a software tester, I approached each process with caution, carefully identifying potential bugs and errors in the code. I frequently switched between the mindset of a developer and a tester, considering all scenarios an end user might encounter. I made sure to test every outcome of each method, both valid and invalid. In my coding I remained aware of the code's complexity and the interrelationships between its components. It’s crucial to always consider how what’s being tested might impact other parts of the code or systems, and how a change in one location may cause code that was previously successful, to fail. As a tester, I understand that the code being tested is often interdependent, meaning changes in one section can influence the behavior of other areas within the application.

An example of this comes from when I was in the process of creating tests for Task Services. Within the TaskServicesTest.java file, I had created five tests for it to run. But when running the JUnit test, for some reason it was only running four tests. It wasn’t showing the fifth test, which was to confirm a task couldn’t be created with a duplicate ID. When running the JUnit test, the results weren’t showing that the fifth test was skipped, or had failed, it looked like the fifth test was just missed entirely.

I spent a long time going over those handful of lines of code. With no red or yellow lines highlighting the code, there was no indication that what I had written was wrong. I checked for syntax errors, logic errors, and anything else I could think of. I eventually found that the reason the test was being skipped was that I had created the testNoDuplicateID function as a *static* void function. In that one moment of first declaring the function, because I failed to reflect on the interrelationships between the components of the code and classes, it caused me a solid 30 minutes of troubleshooting. That’s why it’s critical to always take your time and use caution while coding. Reflect on the complexity and interrelationships of the code while you’re coding.

A computer screen shot of a program code

AI-generated content may be incorrect.

1. 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.

As a developer, the main bias I worked to avoid was the assumption that "I know I wrote the code correctly." To counter this, I focused on writing tests that checked for the specific exception errors that should be thrown, rather than just confirming that any exception was thrown. This approach ensured that the code behaved as expected and helped prevent overlooking potential issues.

A black background with colorful text

AI-generated content may be incorrect.

1. 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? Provide specific examples to illustrate your claims.

As a software engineering professional, maintaining a disciplined commitment to quality is critical for the success of any given project. Writing and testing high-quality code ensures that the software is reliable, efficient, and maintainable, which in turn creates trust with stakeholders and end users. Cutting corners, especially when writing or testing code, can lead to numerous problems down the road, such as bugs, performance issues, and an overall lack of scalability. Inadequate testing can result in undetected issues that could cause failures in production environments, negatively impacting the user experience and, ultimately, the reputation of the software. An example of this would be Apple’s original release of Apple Maps. Apple cut corners by deciding to forgo the crucial hours spent testing the software before releasing it to the public, resulting in end users being the ones finding the multitude of errors and overall inaccuracies of the application. Mistakes such as train stations, mountains, lakes and bridges disappeared or other structures showing as being situated in the middle of the ocean (Academy, 2016).

Cutting corners might seem like a quick fix to meet tight deadlines or reduce initial development costs, but it often leads to technical debt, which can accumulate and compound over time. As technical debt increases, the cost of making changes or adding features grows, as developers are forced to spend more time addressing legacy issues instead of innovating or enhancing the software. This cycle of quick fixes without proper testing or refinement can make the codebase increasingly difficult to manage, increasing the likelihood of critical errors. It also slows down future development, as new code must work around the accumulated technical debt (VanderMey, 2024).

To avoid technical debt as a software engineer, I can follow best practices such as writing comprehensive unit and integration tests, adhering to clean code principles, and refactoring code regularly to improve readability and performance. Additionally, staying proactive in code reviews, collaborating with team members, and prioritizing long-term goals over short-term gains helps ensure that the software remains robust and maintainable. Regularly refactoring code to address known inefficiencies or complexities can prevent these issues from becoming entrenched. By staying disciplined in their commitment to quality, I can create software that not only meets immediate requirements but also stands the test of time, reducing maintenance costs and ensuring long-term project success (Codacy, 2024).

Sources:

Academy, C. (2016, May 24). *When Coding Goes Wrong*. Medium. <https://medium.com/@coderacademy/when-coding-goes-wrong-e46d84c6565f>

Codacy. (2024, March 25). *Avoiding Technical Debt: How to Measure, Manage, and Tackle Technical Debt*. Codacy.com; Qamine Portugal S.A. <https://blog.codacy.com/avoiding-technical-debt#bodyContent>

VanderMey, J. (2024, June 22). *The True Cost of Technical Debt*. Vervint. <https://vervint.com/article/true-cost-of-technical-debt/>