**Project Two**

Finnegan Thomas

Southern New Hampshire University

CS 320: Software Test, Automation QA

Prof. Gudivada, Akhil

June 18, 2025

**Summary**

Testing plays a fundamental role in building reliable software. It ensures that the application functions as intended while helping catch problems early in development, which reduces both cost and time investment. Unit testing is particularly useful here, as it focuses on validating individual parts of the code. In my assignment, I used the JUnit framework to create tests for each feature. These tests aimed to verify that each component behaves correctly, especially when dealing with invalid inputs or different logic paths. Looking over the tests I submitted, each one clearly targets a specific function or requirement laid out in the assignment.

I'm confident that my tests not only meet the defined requirements but also reflect what users would expect from the software. For instance, when working on the Task class, I created a test to confirm that the task ID is a unique string with a max length of 10 characters. My test used a valid input and passed successfully. I also included tests that deliberately triggered exceptions. For example, passing in an overly long ID or a null value to make sure those edge cases were handled as expected. To reinforce immutability, I marked the task ID as final, ensuring that it cannot be altered after creation.

Additionally, I made sure my unit tests followed recognized best practices in software testing. Each test was crafted to be clear, concise, and repeatable. In the ContactServiceTest class, for example, the addContact test (lines 22–26) demonstrates that a contact is properly added to the underlying data structure. This test was efficient, covering one specific scenario and executing quickly. I also tracked test coverage: my tests spanned 82.5% of the features showing that most of the logic was thoroughly tested proving an effective test suite. In the AppointmentServiceTest class, lines 17–23 and 25–31 show how I initialized data and confirmed that methods like addAppointment correctly added an appointment to the in-memory data structure. This validated that the method’s output aligned with the expected behavior and that the code was technically sound.

**Reflection**

**Testing Techniques**

There are several effective techniques that testers can use to build clear, reliable, and efficient test cases. Unit testing, a type of dynamic testing, involves actually running code to verify that outputs behave as expected. I made use of these unit tests to confirm that the actual results aligned with what was predicted. This method is especially helpful in revealing bugs or weaknesses within the code. Unit testing is also tied closely to white-box testing, which relies on an understanding of the software’s internal structure. In this context, knowing the logic behind the code is key, as the tests are meant to validate how the code handles different logical flows. Two common white-box strategies are statement testing, which checks that every line of code is executed at least once, and decision testing, which ensures every possible path from decision points in the code is tested. These approaches help identify code that may be redundant, untested, or vulnerable to edge-case failures and by reviewing coverage percentage, we can observe how much of the code was executed.

Of course, dynamic testing isn’t the only tool available. Static testing, which doesn’t require code execution, plays a major role in improving quality early in the development lifecycle. This kind of testing includes peer reviews, walkthroughs, and static analysis tools to examine code and documentation for clarity, consistency, and completeness. Going back to dynamic testing, it can be broken down further into techniques like black-box and experience-based testing. Black-box testing evaluates how the system behaves from the outside, focusing on inputs and outputs without looking at the internal code. Some notable forms of black-box testing include equivalence partitioning, which groups inputs into valid and invalid categories; boundary value analysis, which checks edge conditions; decision table testing, which assesses different logic combinations; state transition testing, which observes how the system changes in response to inputs; and use case testing, which aligns test cases with user scenarios. These methods help ensure the software is reliable, robust, and aligned with its intended purpose.

Experience-based testing adds another layer of insight, drawing on testers' intuition and previous experience. Techniques like error guessing can reveal less obvious flaws by anticipating where bugs might appear based on past experience. Exploratory testing, where the tester designs, runs, and evaluates test cases in real time, helps uncover unexpected behavior and deepen understanding of the system. Checklist-based testing ensures consistency and thoroughness by using predefined lists drawn from standards or previous testing experiences, which helps maintain high testing quality across different stages or team members.

Each testing method serves a valuable function in a well-rounded quality assurance process. Static testing is especially helpful early on, allowing teams to catch issues before the code even runs. By analyzing requirements, design documents, and code structure, teams can catch potential flaws in logic, miscommunications, or security risks. This proactive approach can cut down on long-term costs by reducing technical debt and shaping the test planning process. It helps establish clear testing goals, allocate resources wisely, and organize schedules effectively. In agile workflows, where constant iteration is key, static testing boosts feedback quality and speeds up delivery. On the other hand, dynamic testing is crucial for validating the behavior of the system during runtime. Techniques like white-box, black-box, and experience-based testing ensure not only that the code works, but that it works well across different conditions and user expectations. In models like the V-model, where development stages are directly linked to test phases, this testing helps confirm the integrity of everything from small components to full system interactions, including the user interface, providing a comprehensive check on software quality.

**Mindset**

Phillip Armour (2005) states, “the challenge in testing systems is that testers are trying to develop a way to find out if they don’t know that they don’t know something” (p.15). It’s impossible to exhaustively test the list of possible inputs, we therefore have to use samples of inputs through boundary value analysis and equivalence partitioning to assist. This lack of knowledge is why it was crucial for me to employ high caution while testing the application. For example, I was uncertain how multiple null value parameters would affect the output of methods or how multiple valid inputs I did not test would affect incorrect methods. This is part of the testing process and assessing the level of risk associated with a project. Fortunately, this project was low risk, however, I still had to ensure the system acted in accordance with expected outputs and requirements. Understanding how classes interact with each other and manipulate data belonging to one another was essential in creating effective tests. Another example is where the contact service test class needed to have a setup method before each test case. This set up method instantiated a contact object. That means that the contact service test class depended on the contact class to uphold its end of the bargain and deliver a valid contact object. This contact class also needed to be tested and therefore the contact service class also had to have faith that the contact test class provided a sound, effective, and efficient test suite for the contact class. This example demonstrates the importance of understanding system components and their relationships in regard to testing.

In addition, I attempted my best to limit bias on testing my own code by curiously and ceaselessly attempting to find errors and defects. Glen Myers (2012) explains that programmers know what the code is designed to accomplish, therefore they may not recognize when it does otherwise. This is an important area of interest especially when designing unit tests for tests I wrote. I had to shift my perspective of the application and attempt to think like a tester whose job is to uncover errors to ensure a safe and error-reduced application. One way I did this was by pretending I was an end user who wanted to troll the application and find input values that I knew may leak through error handling logic to find further errors and defects. This worked well for me as I thought of new input values/combinations that could potentially break the logic. Strategies like the mindset shift are paramount to reducing bias. It’s easy for a developer to believe they created solid, effective, and efficient code which could easily devalue their unit tests since they may have not thoroughly tested their methods and classes.

Lastly, adhering to standard software engineering practices and design can save developers, teams, businesses, and end-users time and headache. There have been numerous examples of how lousy or insufficient testing of applications has led to released defects and escapements that have less than desirable consequences. For example, the Therac-25 machine was a cancer treatment machine whose developers, testers, and engineers had over relied on the software without properly analyzing its critical safety instructions which lead to 6 cases of massive overdosages of radiation. As a developer, I’ll ensure I follow guidance from my peers, organization, and standard safety practices. This not only protects and secures the application but end-users and businesses alike. In addition to adhering to well-known standards and practices, I can employ correct, simple, and testable code from the start of development which will reduce the amount of technical debt.

**References**

Armour, P. (2005, January). *(PDF) the unconscious art of software testing - the subtle psychology of testing.* Research Gate. https://www.researchgate.net/publication/278230907\_The\_unconscious\_art\_of\_software\_testing\_-\_The\_subtle\_psychology\_of\_testing

Myers, G. (2012). *The art of software testing*. Purdue. https://www.cs.purdue.edu/homes/bxd/307/MyersChap02.html