Anthony Wilkinson

CS320

10/13/25

Summary and Reflections

While working on the assignments, I developed a testing approach that closely aligned the software with its requirements. I began by breaking down each feature statement into its specific behavioral requirements. For example, the statement “*The task object shall have a required unique task ID String that cannot be longer than 10 characters. The task ID shall not be null and shall not be updatable.*” identifies four requirements: the ID must be unique, fewer than ten characters, not null, and immutable. I then wrote test conditions that enforced each of these rules. For instance, the requirements that the task ID must not be null and fewer than ten characters are validated by testNullTaskID() and testTaskIDTooLong() methods, respectively. This approach ensured that every requirement had at least one test, leaving all behaviors verified.

To ensure technical soundness, I structured my testing approach so that both valid and invalid inputs were verified for each requirement. For example, while invalid input for the “task id must be less than 10 characters” rule is tested with testTaskIDTooLong(), the valid input is covered in testValidTask(). I also used assertions such as Assertions.assertThrows to confirm that invalid inputs raised the correct exceptions, ensuring that error handling was tested as thoroughly as normal functionality. This approach allowed me to code efficiently by keeping tests limited to what was necessary to cover each requirement. I also used helper methods such as validateString to reduce repeated code, both increasing efficiency and reducing the risk of mistakes. Finally, with each test file achieving one-hundred percent coverage of the corresponding class, I am confident that my test suite is both effective and technically sound.

In developing this testing process, I most prominently used three testing techniques: unit testing, equivalence partitioning, and boundary testing. Unit testing confirms that the individual components of a program are working as intended. In my projects the units were classes like Appointment and AppointmentService. Testing units such as these is critical for all projects, as they are the building blocks of the program. When unit testing, I applied equivalence partitioning and boundary testing techniques to improve efficiency and reliability.

Equivalence partitioning groups input into categories based on the behavior they elicit from the program, allowing the developer to test just one input of each group. However, not all inputs within a category offer the same testing value. This is where boundary testing becomes important. Boundary testing takes advantage of the fact that most errors come from inputs at the edges of these categories. Using the two techniques together means selecting inputs on the boundary of each partition as the group's representative, which increases test reliability. I used this combined method by testing two inputs for each requirement: a valid input, and one just outside the valid range.

Some of the techniques I did not use include Integration testing, Regression testing, and User Acceptance testing. Integration testing verifies that different components interact correctly, ensuring the project works as a whole rather than just in parts. Regression testing checks for unintended side effects after changes to code, which is especially important in large projects. Finally, User Acceptance testing validates the program with real users to confirm that it fulfills its requirements in the right way.

When working on the project, I adopted a cautious and disciplined mindset while striving to limit bias. As a tester, I employed caution by carefully planning how to enforce requirements and conducting multiple static code reviews. This mindset required me to consider the complexity and interplay of the code, since these determine how requirements are fulfilled.

For example, the requirement that each contact ID must be unique cannot be enforced by the contact object itself, since it isn’t aware of other contact instances. Instead, the relationship between Contact and ContactService must be taken into account. Because ContactService manages all Contact objects, it is aware of every contact ID. Consequently, the uniqueness requirement must be enforced by ContactService. This example shows how understanding the interactions between classes is essential to maintaining correct functionality.

In addition to exercising caution, I also had to remain aware of bias, which is always a concern when one develops and tests the same code. For instance, it would be easy to assume I could implement the appointmentDate field the same way as any other field. Running the code, or even testing it, might confirm that it works. However, because the field is mutable, it must be treated differently. This would likely only be caught by someone thinking like a tester, rather than someone focused solely on completing a product. To minimize this bias, I adopted a test-driven development approach, planning the tests my unit would need to pass before I began writing the code.

This approach required a strong sense of discipline, a trait that is vital in software engineering. Discipline allows developers to stick to best practices even when doing so is tedious or inconvenient. For example, writing out the test cases for the contactTest class often felt repetitive, and if I hadn’t been disciplined, I might have rationalized writing fewer test cases. That would have left many object fields unchecked and reduced my overall coverage.

Discipline also extends to writing good code from the start. Instead of producing code that merely fulfills its task, I prioritized writing solutions that were simple, legible, modular, and secure. This mindset helps prevent the accumulation of technical debt. For instance, I used named constants to eliminate magic numbers and helper methods to reduce repeated code. I also ensured that I achieved 100% test coverage, reinforcing my commitment to code quality and long-term maintainability.