# Project Two: Summary and Reflections Report

Riley Eichenour

Southern New Hampshire University

CS-320: Software Test, Automation QA

Doctor Robert Nardelli

August 16, 2025

## Project Two: Summary and Reflections Report

### Summary

A screenshot of a computer program

AI-generated content may be incorrect.While working on each of the three features, for the application the software requirements were directly converted to code and accompanying tests. Along side the tests, the code was written as a direct translation of the requirements. An example from the Contact requirements was “The firstName field shall not be null”. The tests verify creation and updates are functional with field with descriptions that were tied to the requirements. The code validation included the Java equivalent of the requirements as a direct translation.

A screen shot of a computer

AI-generated content may be incorrect.The tests were shown effective through the previously mentioned direct mapping of requirements in plain English to code and JUnit, which is an abstract verification the tests were effective. A concrete example of how the tests were effective was the code coverage. A coverage report of my tests showed 100% instruction coverage for each class. Additionally, the test suite focused on expected functionality was verified and the conditions that should throw errors did. The Appointment tests for the description had multiple outlets to be tested through creation with valid input, tests with null description input, and tests with description input that exceeds the length. The tests followed a logical structure with the test setup, test action, and finally test assertions.

A screen shot of a computer program

AI-generated content may be incorrect.I ensured my test code was efficient using the Nested Annotation, BeforeEach Annotation, RepeatedTest Annotation, and a utility function for generating random strings. The ContactTest class was constructed to include the WhenCreated and WhenUpdated nested classes to provide a clean structure and ensure that only the necessary applicable methods will be tested. Specifically in Contact WhenUpdated utilized the BeforeEach to setup the arbitrary test Contact to prevent code duplication. The RepeatedTest annotation on the updateContactWithValidInput in combination with the generateRandomString function ensures the test efficiently covers different cases, without having to rely on a programmer deciding a set of sufficient strings to verify functionality.

### Reflection

Unit testing based on customer requirements was the only technique utilized. Unit tests focus on testing atomic units for expected behavior. Some examples of expected behavior could be successful object creation/updates, attributes are the expected, and error conditions are triggered based on the requirements. This technique provides a very specific test in isolation to determine the expected behavior adheres to application requirements. This testing can be used to verify the pieces of the system explicitly meet the requirements, independent of other components.

Two additional important techniques for effective testing would be integration tests and static testing. Integration testing can be considered the next layer on top of the unit testing, which verifies the functionality between components of the system. This testing can verify the interactions between components meet the requirements, and can prevent regressions when making changes to individual components continue to function for the interactions with other components. This type of testing can verify that isolated portions of the system work together in a way that is expected and can make sure changes to one component do not break the interactions between components, such as method validations, data structures, and timing. Static testing commonly utilizes tools in a CI/CD pipeline to perform tests on the code, without the code running. This could involve code formatters to verify the codebase has a consistent style between developers, linters to verify the logic in the code meets the code base standards, and dependency check tools to verify the dependencies in the project are not vulnerable.

Unit testing serves to prove that code works in a very small, isolated environment, especially through the lifetime of a project. Individual pieces can prove to be functional, without having a full application to utilize the component. As a project evolves, unit tests can serve to prevent regressions by providing a consistent, automated testing method without user error or omission. Integration testing serves to prove that code works when interacting with other components. Alongside unit testing, these tests can prove a step in the project process is functional, despite not having the component of the system fully implemented. Like unit tests, these tests can serve to prevent regressions of the system. Static testing serves multiple purposes, including, but not limited to, verifying code is consistently formatted, standards of the codebase are followed, and dependencies are up to date/secure. Through the entire lifetime of an application static testing can make sure that developers adhere to similar styles or approaches to problems, even as the individual developers change. Employing tools like a linter can prevent bad coding practices from entering the code base. Automated dependency check tools can verify that the dependencies don’t get too out of date or have critical security vulnerabilities.

My caution was exercised through careful analysis of the requirements and clear communication. “[A]n empty string would be a valid value for all fields that cannot be null and have no length minimum?” (Eichenour, 2025) was a question I would have posed to the stakeholders to make sure this was a valid case. The requirements explicitly stated a string couldn’t be null and a maximum length, which implicitly an empty string would satisfy; however, this could be an edge case where the requirements missed crucial information. During Module 5 I assumed that appointments were immutable for all fields, since the requirements did not state they needed to be updated like other components. In my submission I explicitly stated this assumption was made to provide record of the assumption.

Bias was limited during code review by writing tests that adhered to the requirements as opposed to adhering to the code implementation. Decoupling these two aspects meant the code was not assumed right and instead was proven right. For example, tests to prove error conditions were written and showed I had some logic inverted on one of the validations. Had I adhered to the code written, then the tests would’ve been a false positive representation of the code functionality. Assumptions that the software is right means the required attention to detail may be missed ensuring the code is safe, functional, maintainable, and solves the right problem.

Being a disciplined software engineer is important to limit the potential for bugs to negatively impact users, maintain availability of the application, promoting application maintainability, and avoid introducing technical debt. It’s important to avoid cutting corners to prevent the negative impact bad software can have on users. In crucial applications, such as the healthcare field, a bug can cost the lives of people through negligence to thoroughly test and write secure code. Peer reviews of code, like GitHub pull requests, is one way to avoid technical debt by reducing the bias a developer may have in relation to the code they wrote. Gradually refactoring code to reduce maintenance cost or increase testability is one way to prevent technical debt over time and establish better codebase standards.

## References

Eichenour, R. (2025, July 15). Module3, Contact Requirements Inquiry [Review of *Module3, Contact Requirements Inquiry*]. *General Discussion*.