**Summary and Reflections Report**

*Nicole Hutto*  
Southern New Hampshire University  
CS 320: Software Testing, Automation, and Quality Assurance

**Summary**

**Unit Testing Approach**

Project One, I executed three fundamental elements—Contact, Task, and Appointment services—that were each created to fulfill specific software need statements. My unit testing strategy encompassed validating both positive instances (conforming to the requirements) and negative instances (not conforming to the requirements) to ensure each service was handling its data correctly.

The Contact Service underwent testing to ensure the integrity of the contact ID. It was verified that the ID could not exceed 10 characters, could not be null or empty, and could not be changed after the contact was created. These limitations on the contact ID were accomplished by not providing a way to set it after the contact was created but by only allowing a way to validate it before the contact was created.

With the Task Service, tests confirmed that the task ID met the same 10-character limit and null checks. Both valid task names/descriptions and deliberately invalid inputs were tested to ensure proper exceptions were thrown.

For the Appointment Service, I checked that appointments could be made and removed. I ensured that the tests allowed for valid date formats and that exceptions were triggered for past dates or malformed inputs. I also checked that appointments could be made for any number of future dates.

**Alignment to Requirements**

Every test case was driven by a requirement and mapped directly to an acceptance criterion from the project documentation. This traceability ensured that no requirement was overlooked and allowed rapid verification of test coverage against client expectations.

**Quality of JUnit Tests**

I achieved at least 80% coverage with my JUnit tests for every feature area, and I think that qualifies as pretty good verification and validation for the code paths that cover the happy path and the sad path as well. For instance, the tests I wrote for the Contact class didn't just verify that it could be created with good data; they also verified that it would throw exceptions when created with bad data.

**Experience Writing JUnit Tests**

I ensured that my tests were technically sound, and I did this by using descriptive comments, a consistent format, and even naming conventions. To give you an example:

The help system provides immediate access to a large body of information and can be an essential tool when conducting a static review of the system. However, because the material is so extensive, readability is a major factor in the effectiveness of the tool. Therefore, the documentation was rewritten and restructured to allow the user to review the content without having to search too deeply into the pages for the pertinent information. The content itself was not altered significantly.

To guarantee the optimization of the program, I merged a number of checks together into single conditional statements where it made sense to do so.

if (taskID == null || taskID.length() > 10) {

throw new IllegalArgumentException("Invalid ID");

}

This reduced redundancy while maintaining clarity.

**Reflection**

**Testing Techniques Used**

There were two main techniques used for testing the software:

* Static Testing – Manually reviewing the code, line by line, to identify errors in syntax, logic, and structure before it is executed. (Myers et al., 2011).
* JUnit Testing – Automated unit testing to validate outputs against expected results, covering both valid and invalid input scenarios (Beck, 2004).

**Testing Techniques Not Used**

An approach that this project did not apply is security testing. This testing is used to find vulnerabilities that could be exploited by an attacker and that the system designers didn't account for. Some of the things this project could have tested for are such vulnerabilities as injection attacks, insecure dependencies, and weak authentication. For instance, an OWASP Dependency Check could be used to find outdated libraries that have known flaws or that used to have known flaws but have since been fixed.

**Practical Uses**: Static testing is most effective early in development; JUnit testing is vital for regression prevention; security testing is essential in systems handling sensitive data.

**Mindset**

I took a careful and detailed approach to testing, knowing that missed validations could lead to defects and security vulnerabilities. As I worked through the code, I generated coverage reports to see how well my tests exercised the code. When I noticed that some lines weren't tested (like some unused setter calls that were okay because they were behind a try/catch), I had to make some decisions. Should I just test those lines? Or, since I knew the reports were probably just giving me some false positives, should I refactor the code to make it less nutty?

In order to decrease bias, I interspersed breaks between the activities of coding and testing. Doing this enabled me to regard my code as though it was the work of another that I was reviewing, and that made me more objective.

**Commitment to Quality**

Testing is heavily relied upon to assure the quality of both hardware and software systems. When this vital function is shortchanged, the entire project is put at risk and the unfulfilled promise of quality assurance is abandoned. Why, then, would anyone shortchange testing? The obvious answer is that they are trying to save time and/or money. But testing is sometimes thought of as an area that can be economized upon. And, from my experience, that is not a justified way of thinking.

• Costs are up for remedying defects found in the late stages of development.

• Possible flaws in the security system.

• Harm to the reputation of the developer and the company (Kaner et al., 2008).

To evade technical debt, I intend to:

• Keep a traceability matrix of requirements to test cases.

• Regularly refactor code for clarity and performance.

• In all development phases, employ automated testing.

**References**

 Beck, K. (2004). *Test-driven development: By example*. Addison-Wesley. <https://books.google.com/books/about/Test_driven_Development.html?id=gFgnde_vwMAC>

 Kaner, C., Bach, J., & Pettichord, B. (2008). *Lessons learned in software testing: A context-driven approach*. Wiley. <https://www.wiley.com/en-us/Lessons+Learned+in+Software+Testing%3A+A+Context+Driven+Approach-p-9780471081128>

 Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). Wiley. <https://books.google.com/books?vid=ISBN9781118031964>