Carolyn M. Ríos Borges

CS-320

Module Seven: Project Two

Instructor: Jeff Phillips

**Project Two**

**Follow-Up Summary and Reflections Report**

Describe your unit testing approach for each of the three features.

- To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

- 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?

Describe your experience writing the JUnit tests.

- How did you ensure that your code was technically sound and efficient?

To ensure my unit testing approach aligned with the software requirements for Customer Service, Task Service and Appointment Service, I adhered to each one of the requirements and resources provided. To support my claims, as well as the efficiency of the code, I have provided portions of code, from each milestone, that cover the unique ID requirements for Customer Service, Task Service, and Appointment Service:

**Customer Service:**

// Required contact ID. Cannot be longer than 10 characters, cannot be null, cannot be updated.

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

throw new IllegalArgumentException("Invalid contact ID.");

}

**Task Service:**

// Required task ID. Cannot be longer than 10 characters, cannot be null, cannot be updated.

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

throw new IllegalArgumentException("Invalid task ID.");

}

**Appointment Service:**

// Required appointment ID. Cannot be longer than 10 characters, cannot be null, cannot be updated.

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

throw new IllegalArgumentException("Invalid appointment ID.");

}

Each JUnit test was effective (coverage percentage wise) and ran successfully. Purposefully creating errors, such as adding more characters than required (adding 11 or 12 characters instead of 10), and later recreating them in Task and Appointment services milestones, ensured that each part of the code was working as intended. Once each portion of the code was carefully tested for errors, I reverted the characters back to their original numbers.

Reflecting on the software testing techniques used and unused, and their practical uses and implications:

The two software testing techniques that were used for this project were functional testing and unit testing. As previously mentioned in Module Five Journal, unit testing allowed me to test code units separately and reuse code, while functional testing allowed me to verify the code’s functionality and effectiveness. Implications for both software testing techniques are similar in nature: if a bug in the code is not caught on time, it could delay the project and incur more costs. Software testing techniques, such as Static Application Security Testing (SAST) were not used in this project. These software testing techniques are related to security and vulnerabilities, and help identify, for example, vulnerabilities that would allow unauthorized access. An implication for utilizing this test would be false positives.

To employ caution, I ensured code robustness by implementing JUnit tests that performed negative testing. That is, to say, I fed my program incorrect input and verified that it handled it correctly. To add, it is important to also appreciate the complexity and interrelationships of the code: the more complex the code is, the more complex the interactions can be, which means bugs and errors are more likely to occur. As an example, we can refer to page two of this project, where we can see the interrelationships at work between Customer Service, Task Service, and Appointment Service.

To finalize, developers would almost always introduce bias to their test cases. Try as we might, sometimes we only code for the happy paths and tend to miss the unhappy ones. Or we focus too heavily on the unhappy paths and tend to forget or miss the obvious happy ones. More importantly, though, because we wrote the code and understand how it works, we tend to write tests based on that knowledge and forget or miss tests without that knowledge. Lastly, depending on the field we find ourselves working in, the software we write may have high consequences for failure (this is why we must be careful with bias, and be disciplined in our commitment to writing quality code). For example, in the medical field, people’s lives can be lost if we’re not careful. A perfect example of this is our Module Seven Discussion Post, where we discuss how a computer controlled radiation therapy machine, known as “Therac-25”, affected the lives of 6 people, 4 of which unfortunately passed away. This specific example demonstrates what poor software design and engineering, no timing analyses and no unit testing can cause.

**Reference**

Leveson, N. (n.d.). *Medical Devices: The Therac-25\**. Retrieved June 19, 2025, from <http://sunnyday.mit.edu/papers/therac.pdf>