Project Two: Summary and Reflections Report

Lyric Hart

Southern New Hampshire University

CS-320-19695 Software Test, Automation QA

Dr. Karl Lewis

February 22, 2026

Abstract

The purpose of this report is to summarize the unit testing strategies applied to each of the software milestones, that being contact, task, and appointment services. This is because of the development of a mobile application project for consumers to delegate their daily schedules. It entails software requirements by means of showcasing JUnit effectiveness through the coverage and validation of valid and invalid inputs and also expresses the technical decisions made throughout. This report also includes the testing techniques that were necessary, contrasts them to ones that were not utilized, and assesses their usage among broader applications. Lastly, it reflects on the developer mindset that is needed for mastering the techniques and disciplines in the workplace.

Alignment to Requirements

When it came to my unit testing approach for the contact, task, and appointment services, I made sure that they were aligned with the project requirements in terms of validation and functionality. For each class, there were rules and criteria when it came to field lengths, immutability, and null values so that my tests could verify that they were enforced. One good instance is for my Appointment class; I had to make sure per requirements that the appointment date was not in the past. What I have down below gives a clear picture:

```
// Check to see if the apt. date is in the past on system records
@Test
public void pastDate() {
    Date past = new Date(System.currentTimeMillis() - 1000);
    assertThrows(IllegalArgumentException.class, () -> {
        new Appointment("125", past, "Checkup");
    });
}
```

To further explain, it creates a date object that exists in the past, then subtracts it from 1000 milliseconds, which equates to one second, and then makes a new date timestamp. This will allow the appointment constructor to detect a date that is in the past; therefore, it rejects it. This is also similar to the Task class, where it needs to deny task names that are greater than 20 characters. Or, as well as the Contact class that requires phone numbers to be exactly 10 digits. All of these validation tests helped ensure my JUnit tests were in line with software requirements.

Effective Tests

In terms of the effectiveness of my JUnit tests, this was proven through the data of my coverage percentages for each of the milestone services. This is for both scenarios that are valid and invalid. Each individual class that I wrote that had the correct behavior was due to a condition check. For example, in my appointment service, the prevention of duplicate appointment IDs was essential to avoid an overlap. If you look down below at the code screenshot:

```
// Test to be able to detect duplicate appointments
@Test
public void duplicateAppointment() {
AppointmentService service = new AppointmentService();
Appointment a1 = new Appointment("1", futureDate(), "Optometrist");
Appointment a2 = new Appointment("1", futureDate(), "Orthodontist");
service.addAppointment(a1);
assertThrows(IllegalArgumentException.class, () -> service.addAppointment(a2));
}
```

It shows the ability to enforce and declare restraints on customer ID uniqueness so that there is no confusion when it comes to tracking appointments. That also includes my tests covering the getter methods, constructor behaviors, and other operations such as adding, retrieving, or even deleting objects within the appointment service milestone.

In previous discussions, I brought up screenshots of my coverage percentages and how they mattered to the quality of each of my milestones. By the time I came running JUnit tests for each one, they all were roughly 80% or higher. Some think that having exactly 100% coverage is necessary to ensure that a program is sufficient for final phase operations. However, that is not the case when it comes to quality. Eric Boersma is an experienced software developer and manager of development and makes one good point in his post:

"This is the biggest flaw with test coverage as a metric. Inexperienced technologists believe that the right target number for test coverage is 100%. They believe that because tests are important, your tests should cover every line of executable code. This view is short-sighted. If you think back to our example function earlier, you could write a test that executes all three branches of your function, and you'd have 100% test coverage. But if your tests don't actually test any assertions, your 100% test coverage metric is useless." (Boersma, 2025).

Technically Sound Code

Which is why I want to move onto how my code was technical and sound in its efficiency. The code in each milestone needed to be robust, manageable, and execute the intended functions needed to make it all work as one. Ayesha Mahmood, a writer for medium.com, makes an excellent case for this point: "You're not just coding for the machine—you're coding for the human who will read, debug, or extend your work months (or years) later. That mindset changes how you write code." (Mahmood, 2025). She then gives three points on how and why code needs to be professional and industrial for the purposes of collaboration and development in the workplace.

In the case of my code, I made sure that my code was well explained via comments. Especially for me, when I code, I prefer leaving documentation as a method to keep track of what needs to be worked on and what could be improved. Let us take a look at one example:

```
        assertThrows(IllegalArgumentException.class, () -> {
            service.updateName("1", null);
        });

        assertThrows(IllegalArgumentException.class, () -> {
                service.updateName("1", "ThisNameIsWayTooLongToBeValid");
        });
    }
```

The screenshot above is from my ‘TaskServiceTest.java’ class, and this is for testing to see if any updated task names are invalid if they attempt to bypass the constraints already written into the program. The constraints were that if task names were greater than 20 characters, then they would be deemed invalid for storing in the database. The ‘assertThrows’ methods were helpful in managing error handling so that the exceptions would be caught.

Efficient Code

When it came to making my code readable and efficient for review and full functionality, I made sure that it was not only organized but also created methods to create redundancy logic. If you take a look at what I have below, I’ll explain what the code does:

```
// Test to be able to detect duplicate appointments
@Test
public void duplicateAppointment() {
AppointmentService service = new AppointmentService();
Appointment a1 = new Appointment("1", futureDate(), "Optometrist");
Appointment a2 = new Appointment("1", futureDate(), "Orthodontist");
service.addAppointment(a1);
assertThrows(IllegalArgumentException.class, () -> service.addAppointment(a2));
}
```

The code block above allows for the detection of duplicate appointments. Clearly, customers cannot get their appointments mixed up; otherwise, there is a fault in running daily errands. To prevent this, I created this countermeasure to reject any appointments that may come up as a copy due to the appointment ID. It is always crucial for a developer to watch over their code carefully so as to avoid legible errors. Tom Colvin is the CTO of Conseal Security and makes a solid point on this:

“The same of course is true for your own audits of your code. The more straightforwardly written it is, the more chance you have of spotting your own security flaws — or better, not putting them in in the first place.” (Colvin, 2023).

Other Techniques

In terms of techniques, these were primarily done for testing objectives. From what I remember studying, there are many other methods used for software testing. Those being integration, functional, static, dynamic testing, and more to name a few. Unit testing was utilized for the majority of my milestones. But to highlight, I also had to first incorporate boundary testing, which is verifying that the maximum constraints and length requirements are met. Secondly, there is exception-based testing to accomplish the proper handling of errors when the boundaries are crossed by user input.

Uses and Implications of Techniques

The techniques that I listed above all serve distinct goals but all meet for the common good of a program. Unit testing is ideal for early development and mitigating logic errors quickly. Boundary testing is useful for validation rules that need to be strictly invoked, such as character lengths or date requirements. Exception-based testing acts as a failsafe for system readiness once invalid data is found. There is also system testing, which is to check that the entirety of a system is cleared for deployment.

Caution

However, throughout my whole project, I wanted to adopt the mindset of getting to know what my code does when it is executed in Eclipse. That is one of the reasons why I advocate for documentation in a program, because it helps me to comprehend what I am writing so that other developers may see what I intended. I have an interest in understanding the intricate design of certain systems. In terms of demonstration, I had to think about the possible idea that someone's appointment ID could be lost. I had to implement precautionary measures to ensure the appointment service would retrieve a lost ID. The screenshot below gives a clearer picture:

```
// This tests for missing appointment IDs
@Test
public void testGetMissingAppointmentIDs() {
    AppointmentService service = new AppointmentService();
    assertNull(service.getAppointment("849"));
}
```

Bias

Someone would probably ask if I had any bias when it came to writing my milestones. I wanted to avoid this in order to not have inconsistency with real-life scenarios. So, instead I wrote generic real-world examples to get an idea of how the code would act when run in Eclipse. I included more evidence down below to give a brief explanation of what I did to get the accurate results of my coverage tests:

```
// Test to be able to add appointments to the database
@Test
public void testAddAppointment() {
AppointmentService service = new AppointmentService();
Appointment a = new Appointment("1", futureDate(), "Optometrist");
service.addAppointment(a);
assertEquals(a, service.getAppointment("1"));
}
```

From what I have above, I created a method for adding a new appointment to the hash map system for safekeeping. It has all the necessary elements, those being the ID, future appointment date, and what type of appointment the user wants to book. It creates consistency to allow for how the appointment service system will work in unison.

Discipline and Conclusion

In conclusion, I enjoyed not only the challenge of getting to know how unit testing operates but also how software developers involve other methods of testing to make their projects functional and adaptable to user inputs and how they can serve communities for the benefit of others. The discipline is what taught me to not rely merely on resources to figure out what I should do but also to experiment with what is possible. That is why even when I wrote C++ for one class, I took time to write a beta version of a global currency converter, which is on my Github account to this day. It is how we as engineers are trained to solve problems: by means of knowledge, collaboration, and human morals.

References

- Colvin, T. (2023, February 2). *Code Readability > Efficiency: Here's Why - CodeX - Medium*. Medium; CodeX. https://medium.com/codex/code-readability-efficiency-heres-why-725f017cfee9
- Eric Boersma. (2025, December 22). *Software Test Coverage: Everything You Need to Know - Tricentis*. Tricentis. https://www.tricentis.com/learn/software-test-coverage-what-you-need-to-know
- Mahmood, A. (2025, September 25). *How to Become Technically Sound Beyond Just Coding*. Medium. https://medium.com/@ayesham/how-to-become-technically-sound-beyond-just-coding-c1fb1d0e57ae