Johnathan Dumke

06-16-2025

CS-320

CS 320 Project Two

**Summary:**

1. **Describe your unit testing approach for each of the three features.**
   1. **To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**
   2. **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?**

For the projects Task Service, Contact Service, and Appointment Service I used a systematic unit testing strategy using JUnit 5. Unit testing is very important and in the article by Dziak, Mark he says this about unit testing “These tests allow for a step-by-step development of a finished, and ideally error-free, piece of software.” In my projects I validated through functionality and enforcing data constraints through negative test cases. I also wrote test cases to validate all constructors, setters, and service methods. One example in Task Service is that I used testSetNameTooLong() to confirm that names exceeding 20 characters correctly triggered an exception. The code for this was:

assertThrows(IllegalArgumentException.class, () -> task.setName("This name is far too long for the field limit"));

An example of Contact Service is that I validated name, phone, and address rules. I used testPhoneNotTenDigits() ensured the phone number met the 10-digit requirement. The code was:

assertThrows(IllegalArgumentException.class, () -> new Contact("003", "John", "Doe", "12345", "123 Main St"));

And in Appointment Service I have a date validation which is testPastDate() which ensured past dates were rejected:

Date pastDate = new Date(System.currentTimeMillis() - 10000);

assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", pastDate, "Past appointment"));

For all three projects I got 100% on my Junit tests. I confirmed this by all setters and constructors were tested for valid and invalid values. Each path leading to an exception was triggered and all expected values were asserted for correctness using assertEquals() and assertThrows(). Using Junit tests helped determine bugs early on. This is one example of code that I used to help find issues:

assertThrows(IllegalArgumentException.class, () -> new Contact(null, "John", "Doe", "1234567890", "123 Main St"));

I realized how important null checks are as well.

1. **Describe your experience writing the JUnit tests.**
   1. **How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**
   2. **How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**

To make sure that my code was technically sound in Appointment Service I validated using this line of code below as an example,

Date futureDate = new Date(System.currentTimeMillis() + 10000);

Appointment appt = new Appointment("123456", futureDate, "Dentist Appointment");

And in Contact Service I used field setters and their validations. Example of code below:

assertThrows(IllegalArgumentException.class, () -> contact.setFirst\_Name(null));

And to make sure my code was efficient I used HashMap lookups for constant-time performance in all service classes. For example:

appointments.containsKey(appointment.getAppointmentId())

I also reused objects like Date futureDate = new Date(System.currentTimeMillis() + 10000);

**Reflection:**

1. **Testing Techniques**
   1. **What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.**
   2. **What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.**
   3. **For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations**.

Some of the software techniques that I used were unit testing. This testing was isolated testing of individual methods and classes. Another kind of testing was boundary testing and this testing tested upper/lower limits of input. I also did negative testing which was testing that ensured invalid inputs caused the appropriate exceptions. An example of this would be null, overly long IDS. The last kind of testing was exception testing. It explicitly tested that illegal arguments threw exceptions using assertThrows. Some techniques that I did not use was Integration testing and this kind of testing validates how multiple components interact. Another one that I did not use was Mocking and stubbing. Since all classes were self-contained and didn’t depend on external systems, there was no need to simulate behavior.

The practical usage for unit testing is used in all projects to validate each method and determine it is doing exactly what it is suppose to do. And boundary testing it is important if there is user input. For negative testing it ensures users get meaningful errors instead of system failures. And exception testing makes sure errors are handled gracefully.

The extent I deployed caution is that I adopted a cautious and skeptical mindset. I wrote tests for edge cases, for example AppointmentTests I tested for a null Date, even though it seemed unlikely.

assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", null, "Missing date"));

It was important to appreciate the complexity and interrelationships of the code you were testing because you need to realize that if you allowed bad input it would compromise the entire system.

The best way to avoid bias is that I made sure the main logic was complete and saved the writing tests until after. An example of this would be that I decided early on that a phone number will always be provided, but testing helped catch that assumption. In ContactTest, I wrote:

assertThrows(IllegalArgumentException.class, () -> new Contact("008", "John", "Doe", null, "123 Main St"));

**Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.**

It is important to not cut corners because it can create bugs, poor maintainability, and instability. An example of this would be if I had skipped appointment date validation, users could schedule appointments in the past, creating downstream logic errors. To avoid this, I wrote

assertThrows(IllegalArgumentException.class, () -> new Appointment("12345", new Date(System.currentTimeMillis() - 10000), "Past appointment"));

So going forward I maintained test driven practices, reviewed all validation paths, and ensured 100% test coverage for all business-critical classes.

References

Dziak, M. (2025). *EBSCOhost Research Platform: EBSCO*. EBSCO Information Services, Inc. | www.ebsco.com. https://www.ebsco.com/products/ebscohost-research-platform