CS 320

Module 7

Project Two

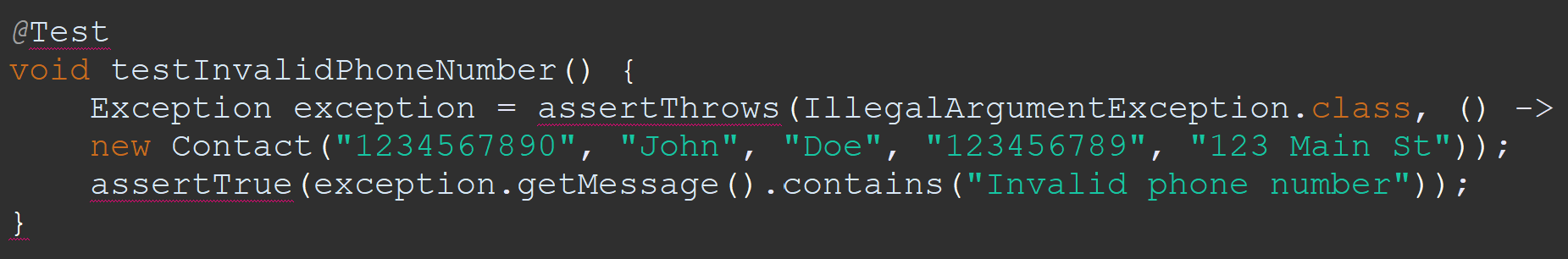
10/12/24

Brandon Gerbasi

**Summary**

In developing the ContactService, TaskService, and AppointmentService, my approach to unit testing was focused on ensuring that each service met the specific requirements outlined in Project One. Each test was designed to verify that the respective service functions correctly under various conditions, including both valid and invalid inputs.

For the **ContactService**, the tests validated that each contact object met the constraints for unique ID, name length, phone number format, and address length. I used boundary testing to check edge cases, such as ensuring that the ID was not longer than 10 characters, and that invalid inputs, like a phone number with fewer than 10 digits, threw appropriate exceptions. For example, in the ContactServiceTest file, I tested how invalid phone numbers were handled:



Similarly, in **TaskService**, I verified that task objects were correctly created and updated, ensuring that the task ID adhered to the specified length constraint and that descriptions were handled appropriately. My tests for this service included checking that tasks with invalid data (like overly long IDs) were properly rejected. For example, in TaskServiceTest, I ensured that the task description did not exceed the 50-character limit:

A screen shot of a computer code

Description automatically generated

For the **AppointmentService**, I focused on ensuring that appointment dates were not in the past and that descriptions adhered to the 50-character limit. I also tested the addition and removal of appointments to verify that the service maintained integrity when multiple appointments were handled simultaneously. In the AppointmentServiceTest, I checked that invalid appointment dates (i.e., dates set in the past) were properly rejected:

A screen shot of a computer code

Description automatically generated

To ensure alignment with the software requirements, each test corresponded directly to a functional requirement. For example, I made sure that invalid inputs, like null values or excessively long strings, were rejected in line with the specifications provided. My goal was to achieve complete test coverage, and I am confident that my approach succeeded in identifying potential issues early in development.

**Reflection**

**Testing Techniques**

The main technique I employed was **unit testing**, where each method in isolation was tested to ensure that it worked as expected. This allowed me to verify that methods performed correctly when given both valid and invalid inputs. For example, I wrote tests to ensure that contact IDs were unique, and that invalid task descriptions triggered exceptions. Unit testing ensured that each service functioned properly in isolation before integration into the overall application.

I also employed **boundary testing** to ensure that the services handled edge cases, such as input that was just at or beyond the limits of what was allowed. For example, for **ContactService**, I tested both valid and invalid lengths for contact names, ensuring that a name exactly 10 characters long was accepted, while a longer name was rejected:

A computer code on a black background

Description automatically generated

Some testing techniques I did not use included **integration testing** and **regression testing**. These techniques are more suited to larger-scale applications or systems where multiple components interact. Since I was focused on ensuring the correctness of individual units, these methods were not directly applicable to this project.

**Mindset**

Throughout the project, I maintained a mindset of **caution** and **thoroughness**. I was deliberate in writing tests for both expected inputs and potential edge cases, as I wanted to be sure the application could handle unexpected situations. For example, I made sure to test how the system responded to invalid appointment dates, such as dates set in the past. These tests were critical in ensuring that the application performed as expected:

A screen shot of a computer code

Description automatically generated

To avoid **bias** while testing my own code, I took breaks between coding and testing. This allowed me to approach my code from a fresh perspective, reducing the likelihood of overlooking errors. By doing so, I was able to test the code more objectively, as if I were reviewing someone else’s work.

**Discipline** was crucial in ensuring high-quality results. Instead of cutting corners or skipping tests when the program seemed to work fine, I made sure to thoroughly test every scenario, including those that might seem unlikely in real-world use. This rigorous approach gave me confidence that the application would function correctly under all conditions.

**References**

* GeeksforGeeks. (2021, November 10). Unit testing - software testing. GeeksforGeeks. https://www.geeksforgeeks.org/unit-testing-software-testing/
* Vogella, L. (2021, July 9). How to write tests in Java using JUnit. https://www.vogella.com/tutorials/JUnit/article.html
* Messen, J. (2023). What is boundary testing? TestArchitect. https://testarchitect.com/what-is-boundary-testing
* DeVry, R. (2020). Software testing principles and practices. https://devry.edu/software-testing-principles