**7-2 Project Two Submission**

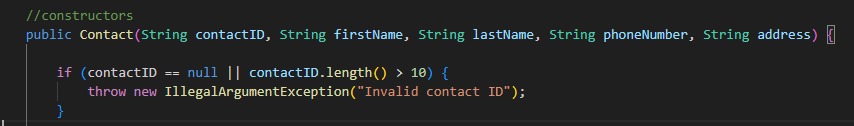
Emerald Tresch

Southern New Hampshire University

CS 320: Software Test, Automation

Professor Omar Toledo Lopez

February 23, 2025

In my Project One testing, I broke down the requirements into the most minor possible components and then wrote a single unit test for each specification. One Contact Service section requirement stated, “The contact object shall have a required unique contact ID string that cannot exceed 10 characters. The contact ID must not be null and shall not be updatable.” To address this, I implemented code to ensure that the contact ID cannot be null and to enforce the character limit. Additionally, I created a separate test to validate that the appropriate error is raised when attempting to create a task without a name. Next, I wrote another test to confirm that the task was created correctly when the requirements were fulfilled, and the value was stored as expected. I applied this same methodology to each requirement throughout the project. Both codes mentioned in this paragraph can be seen in the below pictures. 

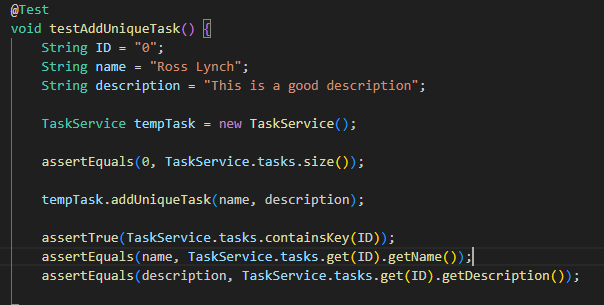
A screen shot of a computer code

AI-generated content may be incorrect.

The percentage of code coverage is a significant component of unit testing. Although it does not guarantee that all requirements are satisfied, it serves as an essential metric to ensure that the entire codebase is subjected to testing. I implemented the JUnit tests to prevent any errors or failures during execution. Upon conducting a comprehensive analysis of my unit tests, I found that the only area lacking coverage was the setters. In the Contact Service segment of Project One, these setters contained no additional logic beyond typical functionality. While many stakeholders may not consider testing critical, evaluating the additional logic I incorporated in the Task Service and Appointment Service segments of Project One is essential. All other aspects of the code were sufficiently covered.

To ensure the technical integrity of my code, I developed unit tests that thoroughly examined both the successful and unsuccessful paths of execution. I confirmed that the code functioned as intended with valid input while verifying that appropriate error messages were triggered in response to invalid input, effectively addressing potential issues. This is exemplified in the “testAddUniqueTask,” which validates the successful addition of tasks when the provided input is acceptable. Furthermore, I included a test that attempts to insert a task with a duplicate ID, affirming that the method correctly returns false as anticipated.

In addition, I prioritized the code's efficiency by ensuring it consistently handles invalid inputs and generates proper error messages. This focus is critical for maintaining efficiency, as inadequate error management can result in significant performance degradation or complete failure as the codebase expands. Two prominent examples of this focus in my tests include the “testLongName” and the evaluation of an “invalidContsructor”.



A computer screen shot of a black screen with colorful text

AI-generated content may be incorrect.

A black screen with text

AI-generated content may be incorrect.

In this project, I focused on the following software testing techniques:

* Unit Testing is a white box testing technique involving individual code units. It is the most fundamental level of testing and is essential for verifying that the code functions as expected.
* Boundary Value Analysis: A black box testing technique that tests the boundaries of input and output values, ensuring that the code handles extreme values correctly.
* Equivalence Partitioning: A black box testing method that divides the input space into equivalence classes, which helps confirm that the code manages all possible inputs appropriately.
* Decision Tables: A black box testing approach that uses tables to specify the combinations of inputs and outputs, ensuring that the code accurately addresses all potential input combinations.

For this project, I did not employ additional software testing techniques such as integration testing, system testing, and acceptance testing:

* Integration Testing is a black-box technique that tests the interactions between different code units. It is usually conducted after unit testing is completed.
* System Testing: A black-box method that tests the entire system, typically performed after integration testing.
* Acceptance Testing: This black-box approach involves users or customers testing the system, usually carried out after system testing.

Each testing technique discussed has practical applications depending on the specifics of different software development projects:

* Unit testing benefits any software development project, regardless of size or complexity.
* Boundary value analysis is particularly useful for projects with critical input or output values.
* Equivalence partitioning is ideal for projects featuring large or complex input spaces.
* Decision tables are practical for projects that require testing many potential combinations of inputs and outputs.

Throughout this project, I maintained a mindset of caution and precision. I recognized the complexity and interrelationships within the code I was testing, carefully evaluating all possible combinations of inputs and outputs. I also tried to limit bias in my review process by considering all scenarios rather than relying solely on my personal experience.

Bias can be an issue when developers are responsible for testing their code, as they may overlook errors. To mitigate bias, developers must maintain a critical perspective and seek assistance from colleagues when needed.

Finally, maintaining discipline and a commitment to quality as a software engineering professional is essential. This dedication means not cutting corners during code writing or testing and being willing to invest time and effort into enhancing the quality of your work. By prioritizing quality, you can help ensure your software is reliable and meets user needs.