Randy Ramos

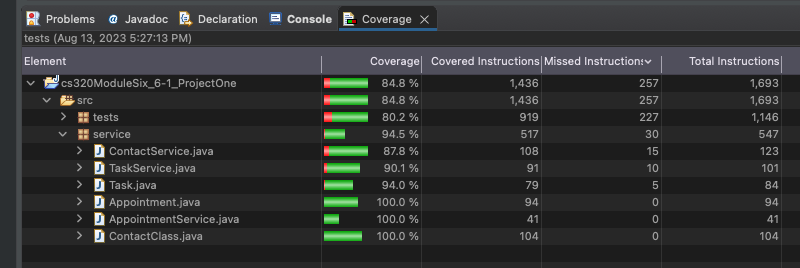
CS-320

7-2 Project Two: Summary and Reflection Report

1. **Summary**
2. **Describe your unit Testing Approach for each of the three features.**

The unit testing approach for the ContactService and TaskService has been well-aligned with the software requirements. Tests focus on specific length constraints and null checks for fields like firstName, lastName, phone, and address in the ContactService, while in the TaskService, the tests validate the length constraints for name and description fields to handle valid and invalid updates to tasks.

Coverage analysis was utilized to ensure a wide range of scenarios were covered, including edge cases and error conditions. Assertions were used to verify whether actual results match the expected results, ensuring the correctness of service methods. The TestAddTask() method in TaskServiceTest validates the successful addition of a task and its properties.



Both ContactService and TaskService classes have checks for null and length constraints on input parameters. The unit tests validate these checks and confirm that the appropriate IllegalArgumentExceptions are thrown when needed. The tests demonstrate good design and coverage of various scenarios to ensure the functionality correctness based on specified requirements.

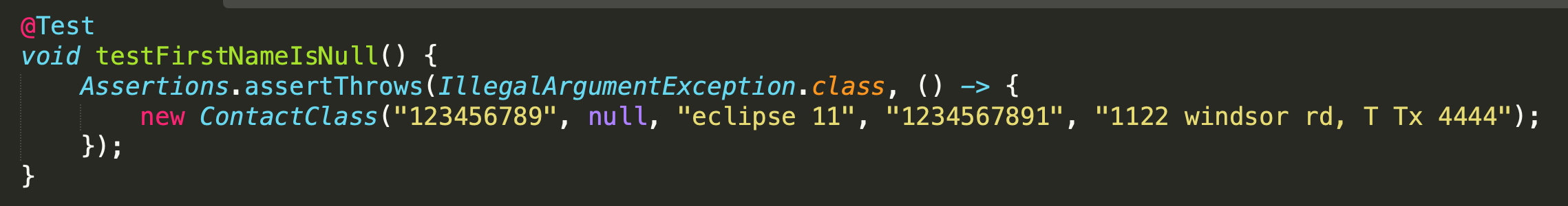
Although not directly addressed in the JUnit tests, efficiency is achieved by providing manageable, easy-to-read, and scalable classes. The JUnit tests focus primarily on functionality correctness. The following information supports the technical soundness of the code, with specific examples illustrating how the expected behavior is validated and how appropriate exceptions are thrown when invalid inputs are provided.

1. **Describe your experience writing the Junit Tests.**

These specific test cases ensure that the code correctly validates and handles invalid inputs per the ContactService class technical requirements.

**I. Ensuring Technical Soundness in ContactTest:**

* testFirstNameIsNull () checks that when the first name is set to null, an IllegalArgumentException is thrown:



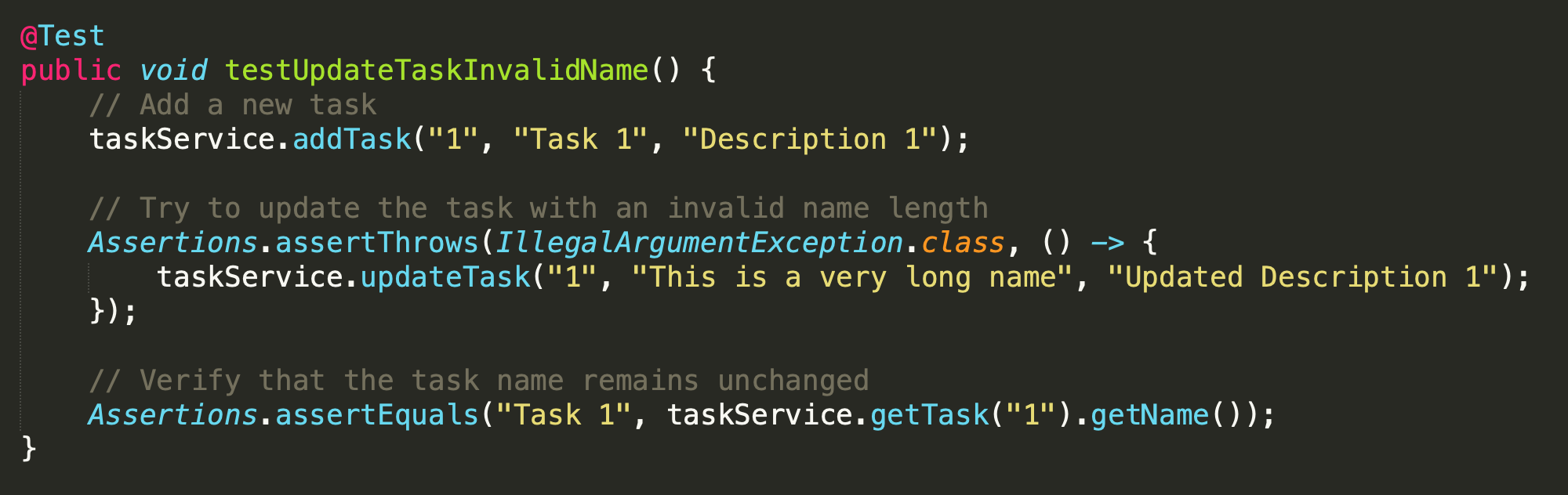
* testPhoneToLong() checks that when the phone number length is greater than 10 digits, an IllegalArgumentException is thrown:



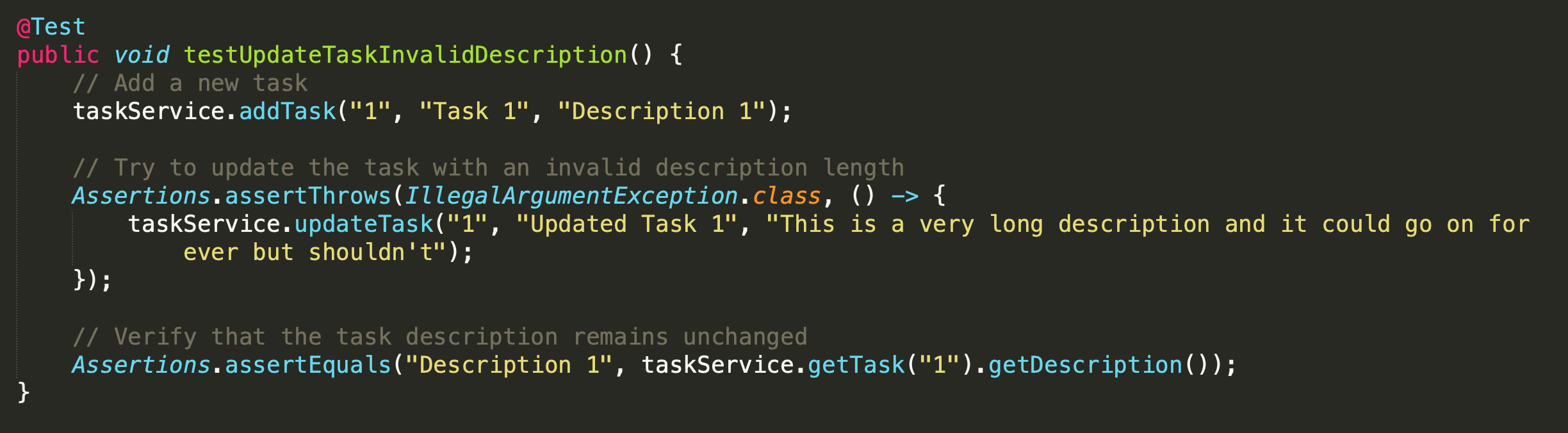
These specific test cases ensure that the code properly enforces length constraints on task names and descriptions, validating that the TaskService correctly handles such cases.

**II. Ensuring code was efficient in TaskServiceTest:**

* testUpdateTaskInvalidName() checks that when the new name length is greater than 20 characters, an ‘IllegalArgumentException’ is thrown:



* testUpdateTaskInvalidDescription() checks that when the new description length is greater than 50 characters, an IllegalArgumentException is thrown:



1. **Reflection**

In software development, rigorous testing is vital to ensure software applications' reliability, functionality, and security. This is especially true as the complexity of software systems continues to grow. Effective software testing techniques play a crucial role in identifying defects, validating correct behaviors, and enhancing the overall quality of software products. This reflection briefly explores the software testing techniques used in project one. These explanations come from the following classes; ContactClass, TaskService, and Appointment. By analyzing the applied testing techniques and their implications, I gained insights into how testing practices contribute to the creation of robust and dependable software solutions.

**a. Testing Techniques**

1. **Software testing techniques.**

For this project, multiple software testing techniques were applied to ensure the reliability and correctness of different classes: the ContactClass, TaskService, and the newly introduced Appointment class. In the ContactClass, Positive Testing was employed to validate that the constructor accurately sets contact attributes with valid arguments, while Negative Testing covered scenarios where attributes exceeded length constraints or were null, triggering the appropriate IllegalArgumentException. TaskService underwent Positive Testing, ensuring the addTask() method functions correctly and task details can be retrieved as expected. Negative Testing was implemented to handle error conditions when attempting to manipulate non-existent tasks, and Error Condition Testing verified that the updateTask() method reacts appropriately to boundary cases. For the Appointment class, Positive Testing verified that the constructor sets attributes properly, while Negative Testing addressed potential issues such as null or excessively long input, and Error Condition Testing confirmed that the class handles situations where appointment dates are set in the past. These testing techniques collectively bolster the robustness of these classes, enhancing their ability to manage contacts, tasks, and appointments effectively while gracefully handling various potential issues.

1. **Other software testing techniques.**

Several key software testing techniques were utilized in the project's evaluation, including Positive, Negative, Boundary, Null, and Error Condition Testing. These methods were essential in assessing the system’s functionality, robustness, and error handling. However, it's important to note that there were several other valuable testing techniques that were not explicitly employed. One such technique is Black Box Testing, where the system is tested based solely on its specifications, treating it as a "black box" without knowledge of its internal workings. Grey Box Testing, a combination of Black Box and White Box Testing, was not used but offers the advantage of partially knowing the internal structure while testing from an external user perspective. Dynamic Testing, involving the runtime execution of code to observe actual behavior, and Load Testing, which simulates real-life workloads, were not applied in this project, but they provide valuable insights into system performance. Lastly, Stress Testing, which examines the system's behavior under extreme workloads, was also not explicitly employed. By considering these additional techniques in the future, software developers and testers can further enhance defect detection, expand test coverage, and ensure the software aligns comprehensively with its intended requirements, leading to a more robust and reliable end product.

1. **The practical uses and implications.**

Positive testing ensures that code behaves as expected with valid inputs, which is essential for basic functionality and critical system components. Negative testing tests software with invalid inputs, identifying vulnerabilities and mitigating risks from user-provided incorrect data.

Boundary testing identifies issues near system limits, preventing errors like buffer overflows, and is useful for input validation and edge conditions. Null testing validates code handling of 'null' values, improving stability and reliability in software systems.

Error condition testing ensures graceful handling of exceptional cases, uncovering security vulnerabilities and enhancing software resilience. These techniques combined in development projects guarantee thorough testing, reliability, security, and stability in software solutions, meeting user expectations.

1. **Mindset**

**I. Caution.**

In this project, I approached my role as a software tester with a strong focus on caution and meticulous testing. My primary goal was to mitigate risks by carefully considering boundary conditions and exploring various error scenarios. This approach minimized the chances of significant defects in the final software product.

My commitment to a cautious mindset was crucial in ensuring the software's reliability. I understood the complexity of the code being tested and identified potential areas of weakness and unexpected interactions. This allowed me to optimize the testing process and cover various code paths, leading to a more comprehensive evaluation.

I applied specific testing techniques in different modules. For example, in the 'ContactClass,' I used Boundary Testing to handle input length limits, and in the 'TaskService,' I utilized Negative Testing to ensure robustness. I also employed Null Testing in the 'Appointment' module to manage null values effectively.

Overall, my cautious approach aimed to uncover issues, validate assumptions, and contribute to creating high-quality software products. This approach was vital in producing reliable contact, task, and appointment service applications, enhancing user satisfaction and trust in the final product.

**II. Bias**

In my code review process, I employed various strategies to reduce bias and maintain objectivity. I focused on meeting specified requirements, checking for edge cases, and ensuring consistency. I also considered the user's perspective to improve software quality.

However, as a software developer, there are biases and challenges to testing my own code. Confirmation bias may lead me to overlook issues, while assumption bias might cause me to miss critical user scenarios. Familiarity with the code can lead to oversight, and self-validation could result in less rigorous testing.

To counter these biases, involving independent reviewers and testers is essential. This multi-perspective approach enhances software quality by minimizing potential biases and ensuring comprehensive testing.

**III. Discipline**

Discipline is crucial in software engineering for delivering high-quality software. It involves adhering to best coding practices, rigorous testing, and avoiding shortcuts. Disciplined development ensures reliability and maintainability and reduces technical debt. For example, code reviews, automated testing, and consistent adherence to coding standards help prevent technical debt. Refactoring, like in the 'ContactClass' and 'TaskService', ensures data integrity and minimizes issues. Disciplined practices lead to a robust and maintainable software system for Grand Strand Systems.

Resources:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

Boni Garcia. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.