Almera Wardle

Karl Lewis, Dr. Eng.

CS-320 Software Test, Automation QA

February 25, 2025

**Project Two**

**Summary**

In developing the mobile application for our customer, I employed a structured unit testing approach to ensure the reliability and correctness of the contact, task, and appointment services. Each feature was tested individually to confirm that it met the specified software requirements.

For example, in contact service, unit tests verified that contact creation adhered to constraints such as valid length and non-null names and phone numbers. In task service, tests ensured that tasks had valid names and descriptions, tasks were properly stored, and could be retrieved and updated correctly. The appointment service underwent validation for date constraints, ensuring appointments could not be set in the past and descriptions met length requirements.

My approach was aligned with the software requirements as each test case directly validated conditions specified in the project documentation. For instance, the requirement that a task name cannot be null or exceed a character limit was explicitly tested using assertions like:

@Test

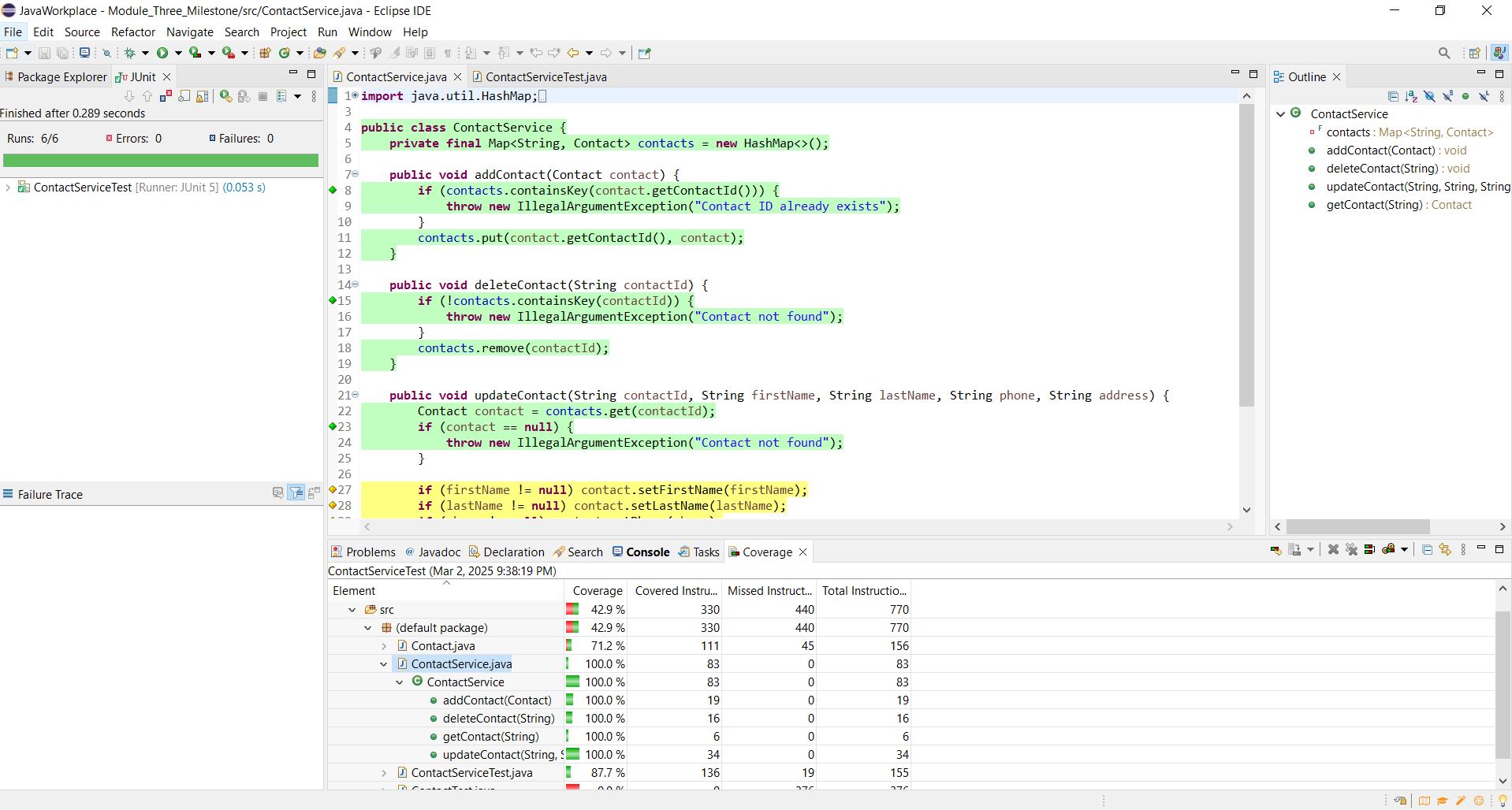
**public** **void** testInvalidName() {

*assertThrows*(IllegalArgumentException.**class**, () -> **new** Task("12345", **null**, "Description"));

*assertThrows*(IllegalArgumentException.**class**, () -> **new** Task("12345", "This name is way too long for validation", "Description"));

}

The overall quality of the JUnit tests was ensured by measuring coverage. By achieving a high test coverage percentage, I confirmed that critical components and logic pathways were tested. The effectiveness of my unit tests was demonstrated by their ability to catch errors early, preventing potential runtime issues.



Writing the JUnit tests involved identifying edge cases and ensuring robustness in the logic. I ensured technical soundness by validating both positive and negative test cases. For instance, the delete functionality in ContactService was tested not just for successful deletion but also for cases where an invalid ID was provided.

A successful deletion scenario was verified using:

@Test

**public** **void** testDeleteContact() {

Contact contact = **new** Contact("12345", "John", "Doe", "1234567890", "123 Main St");

service.addContact(contact);

service.deleteContact("12345");

*assertNull*(service.getContact("12345"));

}

Here, the test ensures that after deleting a contact with ID "12345", attempting to retrieve it should return null, confirming proper deletion.

To test invalid deletion attempts, I included:

@Test

**public** **void** testDeleteNonExistentContact() {

*assertThrows*(IllegalArgumentException.**class**, () -> service.deleteContact("99999"));

}

This ensures that trying to delete a non-existent contact (ID "99999") throws an IllegalArgumentException, preventing unintended removals and enforcing error handling. By testing both valid and invalid scenarios, I ensured that the deleteContact method functioned as expected while handling edge cases gracefully.

I improved code efficiency through thoughtful data structure selection. For example, in the TaskService, I used a HashMap instead of a List for storing tasks:

**public** **class** TaskService {

**private** **final** Map<String, Task> taskMap = **new** HashMap<>();

**public** **void** addTask(Task task) {

**if** (taskMap.containsKey(task.getTaskId())) {

**throw** **new** IllegalArgumentException("Task ID must be unique.");

}

taskMap.put(task.getTaskId(), task);

}

This approach eliminated the need for iteration when accessing elements by ID, significantly improving performance for larger collections.

**Reflection**

I employed unit testing and static testing, both of which fall under white-box testing. Unit testing involved testing individual methods to confirm expected behavior, while static testing involved analyzing the code to identify logical errors before execution.

Other testing techniques not used include integration testing, which verifies interactions between modules, and automated regression testing, which ensures changes do not break existing functionality. While not used in this project, integration testing is crucial for complex systems where multiple components interact. Automated testing allows for consistent validation of the codebase, while ensuring time and cost efficiency.

As a software tester, I adopted a critical mindset by questioning whether my implementation truly met the requirements. I intentionally sought edge cases to break my code and expose vulnerabilities. For instance, I tested whether an appointment could be created with an invalid date, ensuring input validation was enforced.

To limit bias in code review, I tested against specifications rather than assumptions and verified not only that those tests succeeded but also that they had the intended effects. As a developer testing my own code, bias is a significant concern. It is easy to test only the paths I expect to be taken and overlook edge cases.

Maintaining discipline in software quality is essential to avoid technical debt. Cutting corners in testing can lead to unstable software, increasing maintenance costs over time. To prevent this, I plan to implement continuous testing practices and enforce code reviews to maintain high standards.