Paulina Weaver

Professor Luo

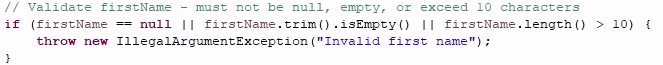
CS 320

02/23/2025

7-2 Project Two

During the development of the Contact Service, Task Service and Appointment Service, my testing approach was closely aligned with the software requirements. I designed tests to validate each requirement, ensuring that all functionalities operated as intended. For the Contact, Task and Appointment classes, I tested various inputs to ensure they met constraints, rejecting any that were out of bounds or invalid. For example, the Contact class has a requirement that the first name must be a non-null string with a maximum length of 10 characters. To verify this, I created tests that confirmed valid inputs worked as expected while invalid inputs, such as names exceeding 10 characters, null values, or names consisting only of spaces, were correctly rejected.

**Contact.java**



**ContactTest.java**

A screenshot of a computer program

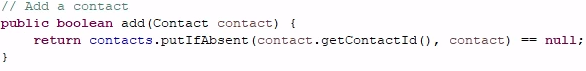
AI-generated content may be incorrect. A computer code with many text

AI-generated content may be incorrect.

A computer screen shot of a code

AI-generated content may be incorrect.

For the Contact Service and Task Service classes, I tested core functionalities such as adding, deleting, and updating records. One key requirement is that the service must allow adding contacts and tasks with unique IDs. To validate this, I implemented the testAddNewContact\_ContactAddedSuccessfully() method, which creates a new contact or task, adds it to the service, and then verifies its successful addition. Additionally, I included a second test, to test adding an already existing contact or task, to ensure that duplicate contacts or tasks with the same ID cannot be added:  
**ContactService.java**



**ContactServiceTest.java**

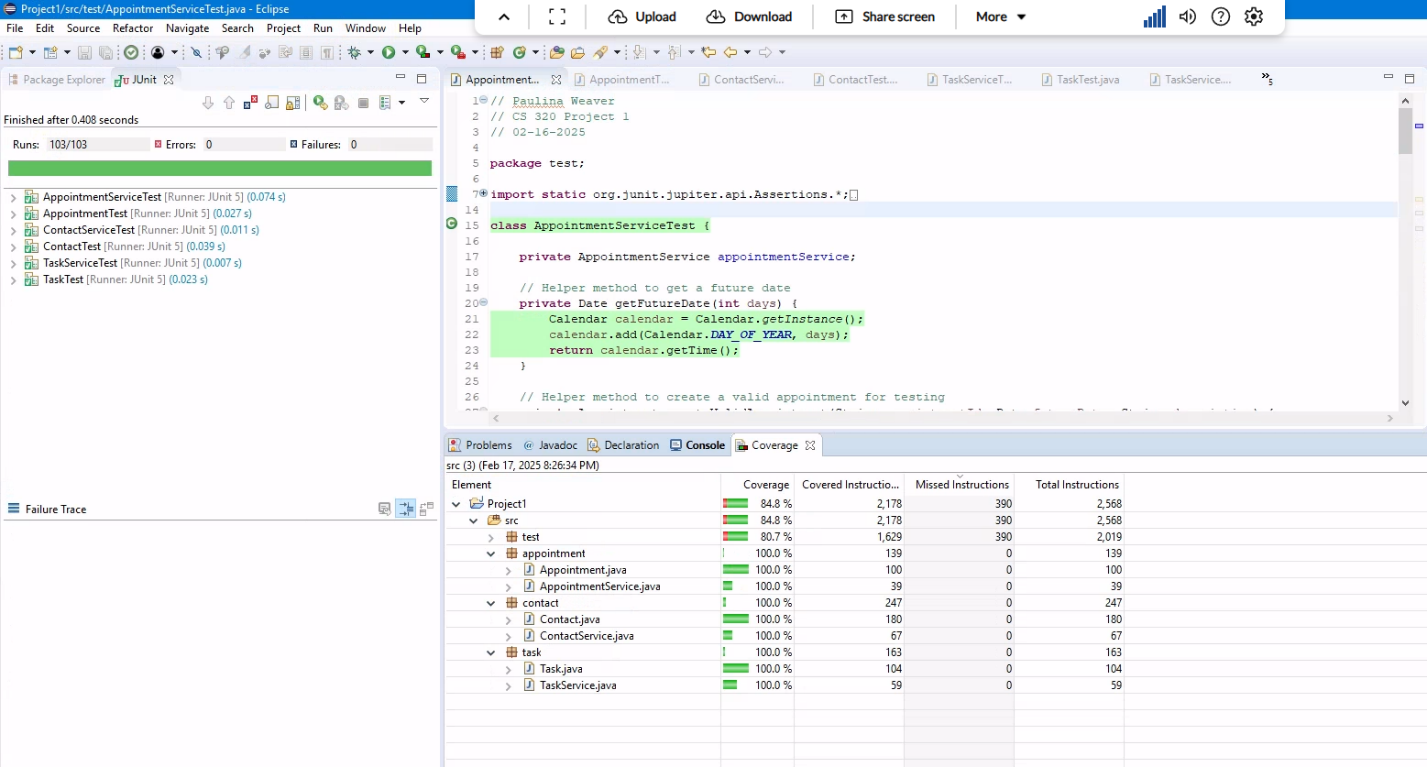
A screenshot of a computer code

AI-generated content may be incorrect. A screenshot of a computer code

AI-generated content may be incorrect.

This ensured that the system properly enforced unique identifiers and handled duplicate entries correctly.

I ensured that my JUnit tests covered 100% of my implemented code by testing all possible scenarios, including valid inputs, boundary cases, and invalid inputs.



By using assertions such as assertThrow(), assertTrue(), assertFalse(), assertNotNull(), and assertEquals(), I was able to verify that each method in my Contact Service and Task Service functioned correctly. For instance, my deletion tests covered both successful and unsuccessful scenarios. In testDeleteTask\_TaskSuccessfullyRemovedFromList(), I verified that a task could be added and then successfully removed from the service. The use of assertTrue(isDeleted) ensured that the method returned the expected result, while assertNull(taskService.getTasks().get("123")) confirmed that the task was completely removed from the task list. Lastly, I would verify that the size of the list was updated correctly.

**TaskServiceTest.java**

A screenshot of a computer program

AI-generated content may be incorrect.

Similarly, testDeleteNonExistentTask\_NoTasksDeleted() tested the system’s behavior when attempting to delete a task that does not exist. By asserting assertFalse(result), I confirmed that the delete operation correctly returned false when the task ID was not found. Additionally, assertTrue(taskService.getTasks().containsKey("123")) verified that existing tasks were not mistakenly removed. Lastly, I checked that the list size didn’t change.

**TaskServiceTest.java**

**A computer code with text

AI-generated content may be incorrect.**

These tests both validated that the deletion functionality worked accurately and confirmed that the task service handled edge cases properly, ensuring that only valid deletions were processed. By covering both successful and failed delete attempts, my JUnit tests provided strong coverage and reliability for this functionality.

To ensure my code was technically sound, I always aimed to follow best practices such as writing organized and clear test cases with effective assertions. I also made sure to use appropriate setup methods to maintain test independence. In the ContactServiceTest, TaskServiceTest, and AppointmentServiceTest classes, I implemented a function to clear all existing records before each test. This prevented data from previous tests from interfering with new ones, ensuring that each test ran independently.

**ContactServiceTest.java**

A close-up of a computer screen

AI-generated content may be incorrect.

When testing task addition, I created a new Task object and verified that it was correctly added to the service. By using assertions like assertTrue(isAdded), assertNotNull(taskService.getTasks().get("123")), and assertEquals(task, taskService.getTasks().get("123")), I confirmed that the task was successfully stored and accessible.

**TaskServiceTest.java** A computer code with text

AI-generated content may be incorrect.

To make my code efficient, I focused on reducing redundancy. One key approach was implementing the @BeforeEach method, which clears any prior data before each test. This eliminated the need to repeat the setup in each individual test, making the code cleaner and more efficient.

**ContactServiceTest.java**

A close-up of a computer screen

AI-generated content may be incorrect.

I also added several helper methods to increase readability and reduce code duplication. For example, in AppointmentSerivceTest.java, I added helper methods to get a future date, create a valid appointment for testing, and to get the current size of the appointment list for testing.

A computer screen shot of a program code

AI-generated content may be incorrect.

I also wanted my test to be quick yet thorough. I refrained from testing too many functionalities in a single test. Instead, I focused on testing specific functionality to make it easier to pinpoint the exact cause of a failure. For example, in the testSuccessfulContactDeletion\_ContactSuccessfullyRemovedFromList() method, I focused only on verifying the delete operation, rather than testing multiple features at once, such as deleting a contact that doesn't exist. I created a separate test for that scenario.

**ContactServiceTest.java**

**A screenshot of a computer program

AI-generated content may be incorrect.**

A screenshot of a computer code

AI-generated content may be incorrect.

By isolating the delete functionality into specific tests, I was able to quickly identify any issues and avoid failures from unrelated parts of the system.

The software testing techniques used in the Contact Service, Task Service and Appointment Service were unit testing and boundary value testing. I created JUnit tests to verify the functionality of each method created, ensuring that they met the provided software requirements. These tests included checking valid inputs, enforcing constraints, such as maximum character limits and unique task IDs, and handling edge cases like null values or excessively long input strings. Unit testing ensured that individual methods functioned correctly in isolation. For example, I wrote tests for the add, delete, and update methods, verifying that tasks and contacts could be added, modified, and removed. Assertions like assertTrue(), assertFalse(), and assertThrows() helped confirm whether the methods returned the expected results under different conditions. I used boundary value testing for validating input constraints. For example, the Contact Service was tested to accept names up to and including the 10-character limit, while rejecting names longer than that. For the Appointment Service, I tested scheduling logic by verifying that valid appointment dates were accepted while invalid ones, like past dates, were correctly rejected. These tests ensured that the application consistently handled inputs at the boundaries of its defined constraints, preventing unexpected behavior or data inconsistencies.

Some other software testing techniques that were not used are integration testing, system testing, regression testing, and maintenance testing. Integration testing typically occurs after unit testing, verifying that all components work together as expected (GeeksforGeeks, 2024). This would be useful in a larger application where the Contact Service, Task Service, and Appointment Service interact with one another or with an external database. System testing evaluates the entire system to ensure it meets overall requirements. Since my testing focused on individual components, system testing would be more useful in a fully deployed environment to ensure the entire application functions as expected. Regression testing ensures that changes to the code do not introduce new defects. This would be useful when updating or adding new features to the Contact, Task, or Appointment Service, ensuring that existing functionality remains intact. Maintenance testing is performed after the software has been deployed and ensures that updates, bug fixes, or modifications do not negatively impact the system. If this application were in production, maintenance testing would help verify that patches or enhancements continue to meet requirements without causing unintended issues.

Unit testing helps catch errors early by verifying individual components in isolation, preventing small issues from escalating. Boundary value testing ensures inputs follow set limits, like making sure a name isn’t too long, or an appointment date isn’t in the past. For larger projects, integration testing is important to make sure different parts of the system work together properly. System testing goes a step further by checking the entire application to confirm it meets all requirements. Regression testing helps prevent new updates from breaking existing features, while maintenance testing ensures the software remains reliable after deployment. While unit and boundary value testing were enough for my project, more advanced testing methods become necessary as applications grow and become more complex.

As a software tester, I approached my work with caution, diligence, and thoroughness. I tested various components of the software carefully. I understood that software systems often have many interconnected parts, so a change or bug in one area could accidentally affect others. For instance, when testing the testDeleteTask\_TaskSuccessfullyRemovedFromList() function, I made sure that deleting one task didn’t impact the other tasks. I did this by checking the size of the task list before and after the deletion to confirm that only the intended task was removed. This careful analysis of how components interact was essential to ensure the software worked correctly.

To minimize bias in my code review, I stayed impartial and focused solely on following the technical requirements, ensuring all required functionality was present. I approached the review process with the mindset of a third-party tester rather than the developer, which helped me avoid assumptions about how the code should work versus how it actually performed. One way I achieved this was by writing tests before reviewing my implementation, ensuring that my expectations were clearly defined based on the requirements rather than my own familiarity with the code. On the software developer side, bias can be a significant concern when testing one’s own code. As developers, we tend to assume that our logic is correct, which can lead to overlooking edge cases or failing to test for unexpected inputs. For example, I might assume that an appointment deletion function will always work correctly because I wrote the logic myself. However, without thorough testing, I might miss edge cases such as attempting to delete an appointment that doesn’t exist or handling database connection failures.

To mitigate this, I relied on writing both positive and negative test cases. For instance, in my test for updating an appointment, I included a case where the appointment ID did not exist. This forced me to consider scenarios beyond the expected flow. Additionally, I reviewed my tests by stepping away from the code and revisiting it later with a fresh perspective, helping me catch any overlooked issues.

Maintaining discipline in writing and testing code is crucial to building reliable and maintainable software. Cutting corners might save time at first, but it often leads to bigger issues like bugs, security risks, and technical debt. If problems aren’t caught early, fixing them later can be costly and time-consuming. For example, skipping edge case testing in an appointment scheduling system could result in users booking overlapping appointments or missing confirmations. Catching these issues early with thorough testing prevents major headaches down the line. To avoid technical debt, I focus on best practices like writing clear, targeted unit tests, following coding standards, and keeping detailed documentation. By staying disciplined and following proper testing and development processes, I can help create software that is stable, efficient, and easier to maintain in the long run.

References

GeeksforGeeks. (2024, October 22). *Types of software testing*. https://www.geeksforgeeks.org/types-software-testing/