**7-2 Project Two**

**Nick Tilton**

**Southern New Hampshire University**

**CS-320 Software Test, Automation QA**

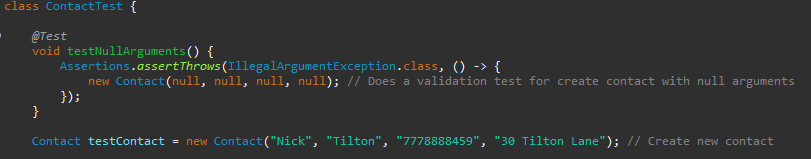
**Prof. Springer**

**June 21st, 2025**

**Summary**

**Unit testing approach for each of the three features:**

For testing the contact, task, and appointment services, I mainly focused on checking each field and making sure they behaved as expected based on the project requirements. For the Contact class, I wrote tests to confirm that the ID was unique and no longer than 10 characters max, along with that the first and last names were under 10 characters max aswell and that the phone number inputted was exactly 10 digits, and the address was less than 30 characters total. I used JUnit’s assertThrows() to catch invalid inputs and assertTrue() to verify the valid ones. This approach helped me to ensure that each part met the specifications given in the project rubric. For example, in ContactTest.java, I set up test cases like this using JUnit:

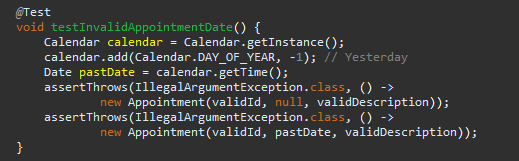


**Defend the overall quality of your JUnit tests:**

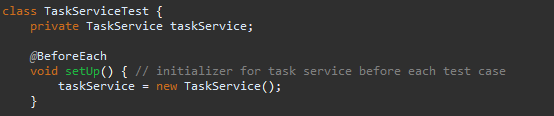
This made sure that the given input IDs longer than 10 characters were rejected by the program instead of being taken as a valid input ID. I feel good about the quality of my JUnit tests because I achieved roughly over 90% coverage across all the classes after some fine tweaking. That means I tested nearly every possible path, condition, and exception not just the straightforward obvious test cases. (Ma’ayan, D. D. 2018) points out that when tests cover a large portion of the code, it usually means the code has been checked carefully, catching both obvious mistakes and more trickier edge cases that appear randomly. Of course, having high test coverage doesn't guarantee everything is flawless as anything can happen, but it does suggest the code has been well tested and that most major issues probably would have been caught along the way and do not exist anymore.

**Describe your experience writing the JUnit tests:**

Writing the JUnit tests really helped me to verify that my code was solid by paying close attention to validation and potential flaws within the code base. I didn’t just test whether the methods worked under normal circumstances, I also made sure to cover edge cases that may appear randomly along with invalid inputs. For instance, in AppointmentTest.java, I checked that trying to set dates in the past would appropriately trigger a null case causing a false input:



This line verified that the necessary appointment date code was functioning properly. I also focused on making my run tests efficiently, rather than writing repetitive and or redundant code, I reused viable setup methods and structured my tests logically using @BeforeEach. This approach kept the code tidy and minimized unnecessary code duplication. By preparing shared objects and data before each test, I avoided repeating the same initialization code across multiple test methods. This made the tests easier to understand and maintain, and it also streamlined any future updates if the setup needed to be changed at any point. Here's an example from TaskServiceTest.java:



As said above, this ensured that every test began in a clean and consistent test environment.

**Reflection**

**Testing Techniques:**

All in all, I used a variety of testing methods to ensure my code was working correctly and reliably without failure. For unit testing, I examined individual pieces of logic on their own to verify that each part functioned as expected. This approach helped me identify and fix errors early on. I also carried out boundary testing by checking values at the extreme ends of acceptable input ranges, which is especially useful for catching issues that might arise at those critical points with certain user inputs. I also incorporated assertions to verify how each method handled both valid and invalid inputs. This helped ensure that the methods responded correctly and consistently across various given user inputs, providing an extra layer of confidence in their reliability. Overall, these techniques gave me confidence that the code was solid and prepared for the next stages of integration whatever they may be. Within my project tests I did not include integration testing, system testing and or static testing within my process. Integration testing verifies how modules work together, especially when services depend on each other. System testing tests the entire application end to end before deployment looking for vulnerabilities or fail points. Static testing, like code reviews, can catch simple mistakes early within the development process (Kanai, S. 2022, May 25). While unit testing is suitable for small, isolated components, larger projects benefit from integration and system testing to ensure all parts coincide together smoothly without failure, these more advances methods of testing also aid in catching additional issues within the codebase.

**Mindset:**

While working on this project, I stayed cautious and thorough, testing even simple methods to prevent unexpected failures. For example, I tested edge cases like null values, incorrect lengths, and non-numeric characters in the phone validation, recognizing that small errors could impact later updates or deletions. Understanding the code's complexity was key, as silent failures could cause bigger issues in the ContactService. By carefully examining each method and its connections, I caught problems early and avoided small issues turning into larger ones.

Limit Bias was another vital part of this project. It’s common to believe your code works simply because you created it, so I made a point of including negative tests with invalid inputs to challenge the codes capabilities ensure once this hits a userbase there are limited to no errors with use. Rather than only testing for basic success scenarios. Focusing only on ideal cases could have led me to overlook crucial edge cases. When testing your own work, familiarity can hide mistakes, making bias a real challenge. To mitigate this within a professional environment, I would usually utilize peer reviews and have others review my tests in a to ensure thoroughness and objectivity for the project.

I strongly believe that quality should always come first. Cutting corners with testing can result in serious bugs slipping into the live environment while users are interfacing with the program, damaging user confidence and company reputation. To avoid this, I intend to develop modular and easily testable code so anything can be adapted easily, while also incorporating comprehensive test suites, and utilizing automated testing through continuous integration systems. For example, I will implement branch protection policies that mandate all tests pass before any code is merged, helping to keep the software dependable, resilient, and straightforward to maintain in the long run with little maintenance and upkeep to do.

***References:***

Dor Ma'ayan. (2018, March 19). *The Quality of Junit Tests*. ResearchGate; unknown. <https://www.researchgate.net/publication/323847238_The_Quality_of_Junit_Tests>

Kanai, S. (2022, May 25). *JUnit Testing Framework - A Comprehensive Guide | HeadSpin*. [Www.headspin.io](https://Www.headspin.io). <https://www.headspin.io/blog/junit-a-complete-guide>

‌

***‌***