A computer screen shot of text

AI-generated content may be incorrect.

Figure 1. Public constructor with control logic to check parsed string data

I reviewed the technical requirements for each class and developed the necessary attributes and methods to implement them. Some of these methods included control logic that checked the parsed data before storing it in private variables. In figure 1, the public constructor method for the Contact class has control logic to check to see if the parsed string data doesn’t meet the requirements. This approach aligned closely with the software requirements, ensuring data integrity and consistency by validating inputs according to the defined constraints. I incorporated conditional checks within the setter methods to prevent invalid data from being stored. In figure 2 below, this setter method checks to see if the phone number entered is null, empty or exceeds the character limits.

A computer code on a white background

AI-generated content may be incorrect.

Figure 2. Setter method with control logic

A screenshot of a graph

AI-generated content may be incorrect.

Figure 3. Test coverages for each class and overall project

The overall quality of my JUnit tests can be considered effective due to the coverage percentage and the thoroughness of the test cases I implemented. I checked the coverage tab at the bottom of the screen and confirmed that the project achieved an overall coverage of 80.4%. This indicates that a significant portion of the code was exercised by the tests.

To ensure robustness, I created unit tests that specifically addressed edge cases, such as null or empty inputs and inputs that exceeded character limits, in addition to standard successful cases. This approach helped verify that the control logic in the methods correctly handled both valid and invalid data. Although some individual classes had varying coverage percentages, the core functionality and critical paths of the application were well-tested. This gives me confidence that the system behaves as expected under a range of input conditions.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 4. Test case to confirm a successful date

To ensure the code was technically sound, I adopted a consistent and structured approach across all unit tests. This involved clear setup, focused method execution, and the use of precise assertions. For instance, in the date test in figure 4, I initialized a valid future date using a Calendar instance (lines 1–3), set it on the appointment (line 5), and confirmed that the object remained valid using assertNotNull (line 7). I then extracted the updated date and verified each field with assertEquals (lines 13–15) to ensure accurate data assignment. This same pattern of clear initialization, targeted updates, and validation through precise assertions was applied throughout the other test cases. This approach ensured that each method behaved as expected and maintained the technical soundness of the codebase.

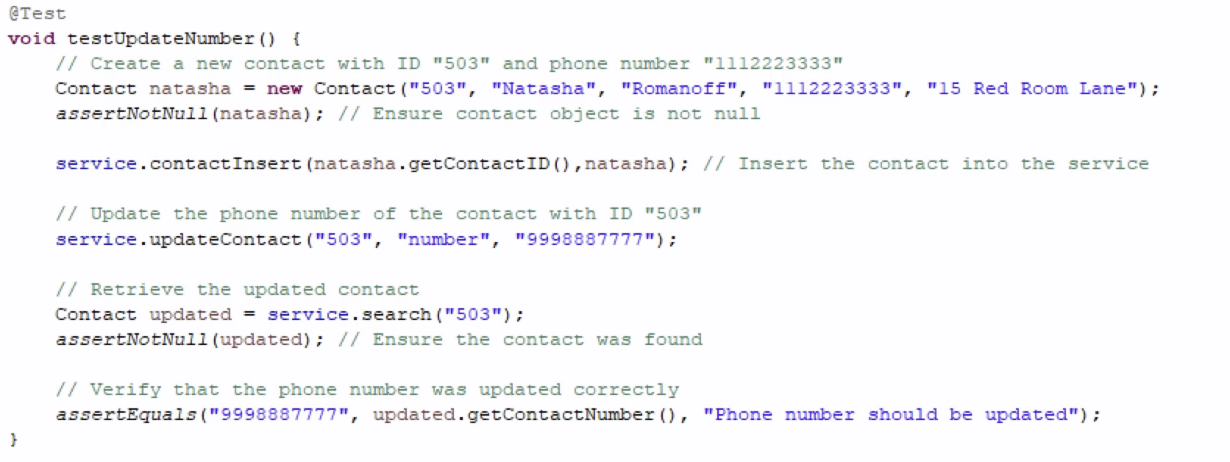


Figure 5. A test case that updates the phone number of a contact.

To ensure code efficiency, I structured the test to validate functionality with minimal operations. In the first two lines, I created a Contact object and confirmed its creation with assertNotNull, avoiding unnecessary setup. In the fourth line, I simplified the process by directly inserting the contact using service.contactInsert(...), a straightforward method call. I updated the contact's number in the sixth line with another direct method call. Retrieval and assertion in lines eight and nine were clear and focused on the relevant data. Finally, the update was confirmed with assertEquals in the eleventh line. This approach—minimal setup, direct invocation, and clear validation—is consistently applied in other unit tests for better performance and clarity.

Unit testing is a method used to test the smallest and isolated parts of a program, such as individual functions or processes, to ensure they perform as expected. It involves writing test cases that provide inputs to a specific unit and checking whether the output is correct according to the intended logic. Unit testing helps developers identify errors early in the development process, improves code quality, and supports modular development by allowing tests to run independently of external systems, such as databases or networks. Strategies such as logic checks, boundary testing, and error handling are employed to cover various scenarios (GeeksforGeeks, 2025). This approach ensures the reliability of each function and helps maintain clean, bug-free code throughout the development process. Overall, unit testing plays a vital role in enhancing the maintainability, accuracy, and efficiency of software.

Static testing is a method of examining code and related documents without executing the program. It aims to identify defects early by reviewing source code, design documents, and requirements, either manually or with automated tools. This process helps catch errors, such as syntax mistakes and unused variables, thereby improving code quality and reducing the costs associated with bug fixes (GeeksforGeeks, 2025). As the sole developer, I conducted manual code reviews and static analysis to find issues early, ensuring the final code met quality standards and functioned as intended, ultimately enhancing software reliability and maintainability.

In this project, I did not employ system testing or integration testing, both of which are essential in the software development life cycle. System testing examines the entire software from the user's perspective to ensure it meets all requirements, including both functional and non-functional aspects such as performance and security, without requiring knowledge of the internal code. In contrast, integration testing focuses on how different modules or components interact, identifying defects at their interfaces. This type of testing can be performed using various approaches, including top-down and bottom-up methods. While unit testing evaluated individual functions for correctness, implementing these additional testing techniques would have enhanced the verification of the overall interaction and real-world functionality of the application (PractiTest, 2024). Incorporating such tests in future projects would improve the software's reliability and increase user satisfaction.

Unit testing and static testing are crucial components in various stages of software development projects. Unit testing is particularly effective in projects that prioritize code reliability and modularity, such as those involving agile development or complex algorithms. It enables the verification of individual components early and frequently, allowing for the identification and prompt addressing of defects.

Static testing, on the other hand, involves reviewing code and documentation without executing them. This method is especially useful in the early development phases, as it helps identify errors, enforce coding standards, and enhance maintainability before any dynamic tests are conducted.

Together with integration testing and system testing, these techniques create a comprehensive testing strategy. While unit and static testing primarily focus on early defect detection and code quality, integration testing ensures that components communicate smoothly, and system testing validates the software from the user's perspective. By applying the appropriate mix of these methods based on project complexity and requirements, teams can optimize both quality assurance and resource utilization.

I approached unit testing with a methodical mindset, recognizing the importance of verifying the reliability of the contact, task, and appointment services. I focused on the complexity of the code, particularly the control logic that validated parsed data before storing it. For instance, I implemented checks on appointment dates and task descriptions to ensure they met specific criteria, such as string length and valid date ranges. Each JUnit test was designed to isolate edge cases, such as the maximum length for a contact name, while also testing successful inputs. This careful approach not only helped catch subtle issues early but also ensured robust data validation, instilling confidence in the effectiveness of the testing process. Writing the tests was challenging, so I included in-line comments to track coverage and clarify my thought process. Overall, the JUnit tests balanced functional success paths and boundary testing to handle both valid and invalid inputs reliably.

To minimize bias in my code review and testing, I approached the code as if I hadn’t written it, consciously refraining from any prior knowledge. I questioned every assumption and tested each function with the expectation that it might fail. Bias is a genuine concern when developers test their work because it’s easy to focus only on what you anticipate will work, rather than actively seeking out flaws.

To address this, I intentionally created test cases that pushed the limits of input—such as strings that exceeded the maximum allowed length or dates in incorrect formats—to ensure the validation logic triggered exceptions correctly. For instance, when testing the appointment class, I didn’t only check valid dates; I also tested invalid formats and past dates to confirm proper error handling. This approach helped me uncover edge cases that I hadn’t initially considered. By treating the code with skepticism and concentrating on failure scenarios, I was able to reduce personal bias and develop more thorough, objective tests.

Being disciplined in maintaining a commitment to quality is essential in software engineering. Even small shortcuts can lead to significant issues later, affecting both users and future development. While cutting corners in writing or testing code may save time initially, it often results in bugs, security vulnerabilities, and maintenance problems that are much more costly to fix down the line. For instance, skipping thorough boundary testing for input validation could allow invalid data to corrupt a database or cause application crashes.

To avoid technical debt, I plan to adhere to best practices, such as writing comprehensive unit tests, documenting my code clearly, conducting regular code reviews, and refactoring when necessary. During Project One, I ensured that each test case addressed both successful outcomes and edge cases, even when it required additional logic or extra time for debugging. This discipline guarantees that the code is not only functional today but also reliable, scalable, and maintainable in the future.

**Citations**

GeeksforGeeks. (2025, July 22). *Unit Testing - Software Testing*. GeeksforGeeks.https://www.geeksforgeeks.org/software-testing/unit-testing-software-testing/#

GeeksforGeeks. (2025, July 11). *Static Testing - Software Testing.* GeeksforGeeks. https://www.geeksforgeeks.org/software-engineering/software-testing-static-testing/

PractiTest Team. (2024, April 17). *System Testing vs Integration Testing.* PractiTest. https://www.practitest.com/resource-center/article/system-testing-vs-integration-testing/