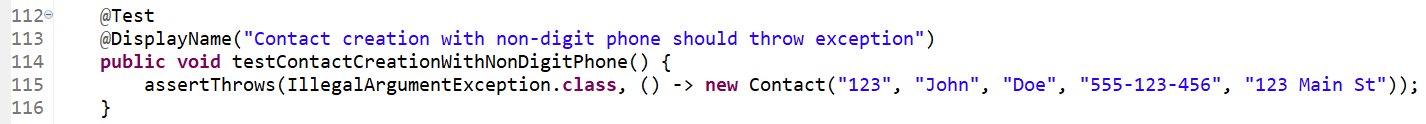
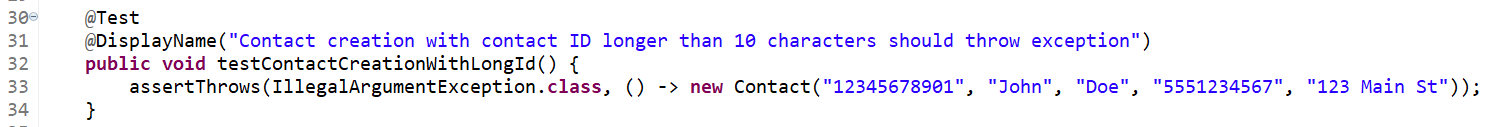
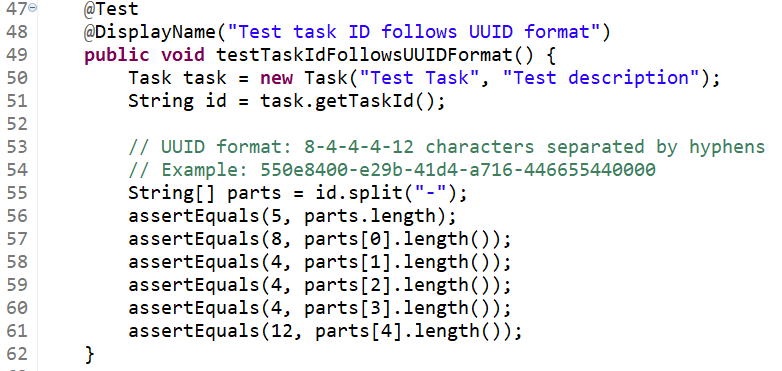
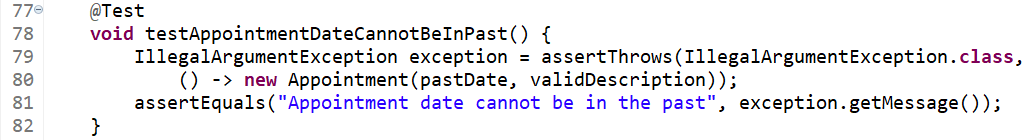
For each of the features, I utilized model testing and service testing. For the contact management system, the primary testing approaches were boundary value analysis, equivalence partitioning, and negative testing. For the task management system, I tested UUID generation, uniqueness verification, field validation with precise error message checking, service lifecycle testing, and concurrent operation simulation. For the appointment management system, I tested date validation, UUID format verification with regex-like validation, business rule enforcement, and data integrity across multiple appointments.

In the testing for the Contact class, I tested for the Contact ID and phone number requirements.

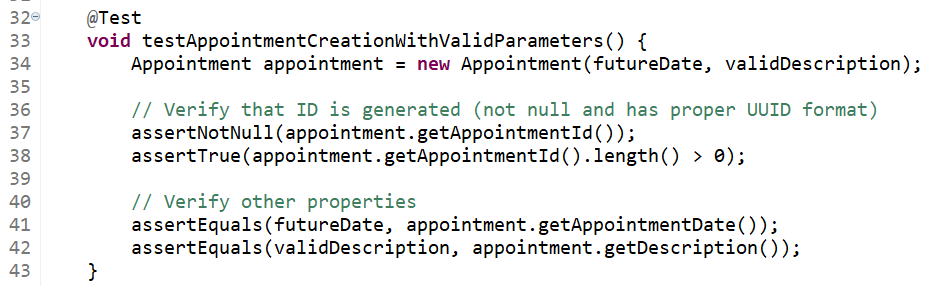
In the testing for the Task class, I tested to ensure that the UUID format was implemented correctly.

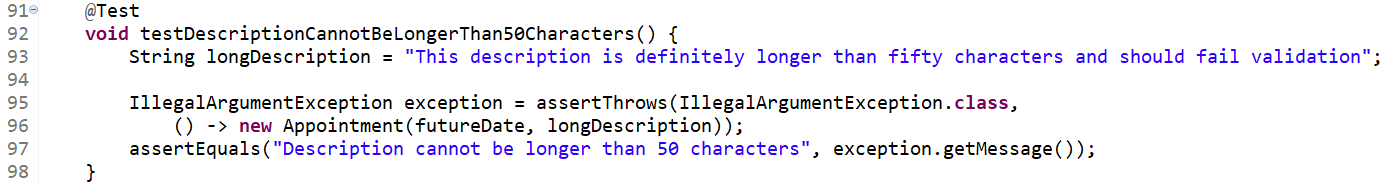


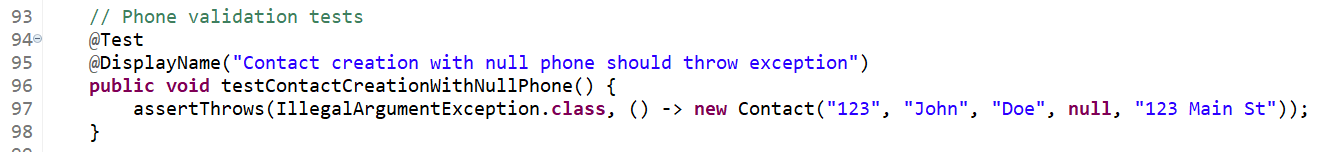
When testing the Appointment class, I tested the date constraint requirements to ensure that appointments could not be created for past dates.

Based on potential test scenarios, I covered roughly 90% of comprehensive test scenarios across the three classes. I also had high coverage through positive/negative test pairs. In terms of methods, I tested all public methods across the three classes. Finally, I covered all validation scenarios for exception path coverage.

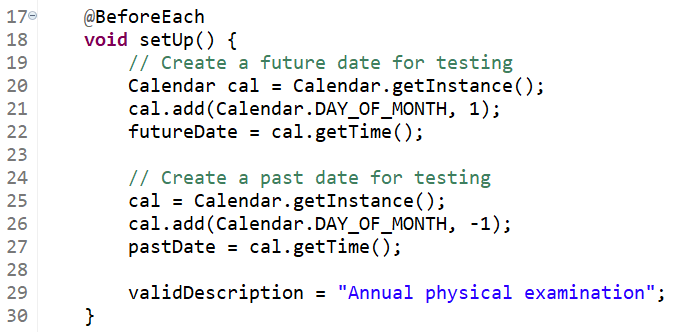
There were a couple of things I considered to ensure that my code was technically sound. I made sure that my tests were isolated and focused on single functionality.

This method in AppointmentTest.java solely tests the constructor method for the Appointment class and runs assertions for each of its attributes. The tests were also written to have deterministic dependencies by not utilizing random inputs in any of them.

Each assertion in the test classes has clear specific expected versus actual comparisons.

For example, this test method specifies which argument is being tested, the phone number for the Contact, as well as the expected result, a thrown exception.

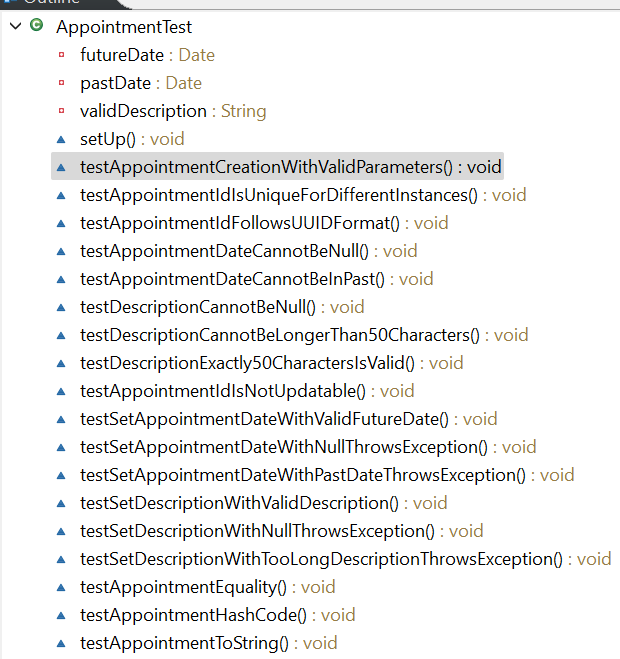
I utilized the @BeforeEach tag along with a setUp() method for the AppointmentTest class to minimize the boilerplate code for testing different aspects and methods for the Appointment class.

 One testing technique that I employed in this project was boundary value analysis. This technique tests values at the boundaries of input domains and focuses on edge cases where defects commonly occur. It also validates both valid boundary values and invalid extremes, such as requirements regarding name or phone number lengths. Another testing technique that I utilized was equivalence partitioning. This technique divides input domain into equivalence classes, and tests one representative from each class. It assumes that all values in a class behave similarly. Some examples of this testing technique are valid phone numbers which include all 10-digit numeric strings, invalid names which all exceed character limits, and valid dates for appointments which are all in the future. I also implemented state-based testing throughout the various classes. This technique tests object behavior across different states and verifies state transitions and invariants. It also validates that state changes do not affect other properties. For contact updates, I verified that the ID remained unchanged after field modifications. For the relevant classes, I also confirmed that generated UUIDs never changed. The last testing technique I utilized was negative testing. Negative testing tests system behavior with invalid inputs, verifies proper error handling and exception throwing, and ensures graceful failure modes. All constructors and setters were tested in this matter with null parameter validation, invalid data format testing (non-numeric phone numbers), and non-existent ID operations (delete and update).

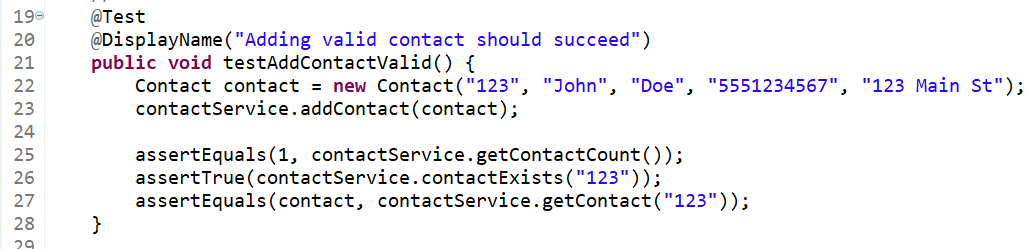
There are some software testing techniques that were not used for this project. The first of which was performance testing. Performance testing measures system response times and throughput, tests under various load conditions, and identifies performance bottlenecks and scalability limits. Another testing technique that went unused was integration testing. Integration testing tests interaction between different system components. It does so by verifying data flow and interface contracts and identifying integration-point failures. Each of the components of the project operates independently with minimal inter-component dependencies. Property-based testing was not used either. This technique generates random test inputs based on defined properties, tests invariants that should hold for all inputs, and discovers edge cases through automated exploration. The validation rules for the classes involved in the project are deterministic and well-defined, making traditional example-based testing more appropriate. Another unused testing technique was mutation testing. This introduces small code changes to test suite effectiveness. This measures how many mutations are caught by tests and identifies weak spots in test coverage. The project scope focused on comprehensive functional testing rather than meta-testing approaches.

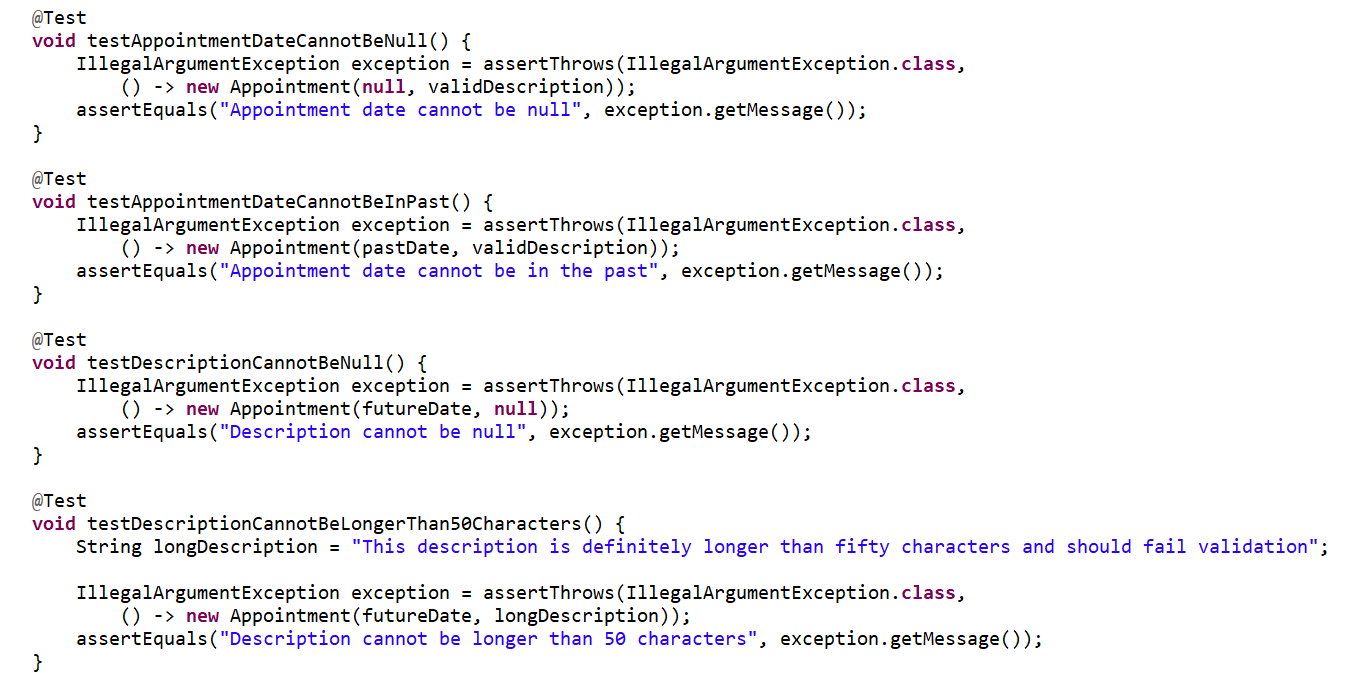
Boundary value analysis is used in e-commerce systems for price validation, financial software for transaction limits and account balance boundaries, and form validation for character limits in user input fields. Equivalence partitioning is used in user authentication for valid and invalid credential combinations, file format validation, and valid versus malformed request structures in API testing. State-based testing is utilized in state machines (order statuses), game development (player state management), and workflow systems (document approval processes). Negative testing is used in security testing to account for SQL injection and XSS attack prevention. In mobile apps, it is used for network failure and offline mode handling. In payment systems, it takes care of invalid card number and expiration date handling. Performance testing is used for load testing web applications for Black Friday traffic spikes. It is also used for database query optimization and indexing effectiveness. For mobile apps, it looks at battery usage and memory consumption analysis. Integration testing is used for microservices to validate API communication between services. It also tests database integration with regards to ORM mapping and transaction management. Another application is third-party integrations for payment gateways and authentication providers. Property-based testing is used for mathematical functions to test commutativity and associativity properties. It is also used for parsing libraries and handling round-trip parsing consistency. It also tests compression algorithms, validating that decompression yields the original data. Mutation testing is used in critical systems, such as medical device software validation. It is also utilized in financial algorithms to ensure trading system robustness, as well as safety systems for automotive and aerospace software.

My testing approach demonstrated high caution through comprehensive validation testing and edge case coverage.



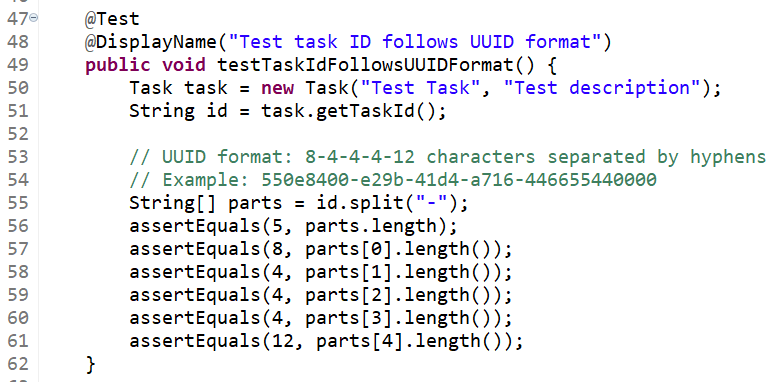
I utilized maximum length boundary testing for all string fields as well as UUID format validation with specific pattern matching. Understanding code interrelationships was critical because of service-model dependencies, state management, and exception propagation. One example was in the testing of the updateFirstName() method in the ContactService class. Because the method calls the setFirstName() method from the Contact class, both layers must be tested to ensure the code is functioning correctly.

 Bias is always a primary concern when it comes to a developer testing the code they have written. Many developers fall into the trap of thinking “I wrote this code, therefore there is nothing wrong with it and will function as I intended it to.” There were a couple of ways that I attempted to address bias when it comes to test implementation. The first was systematic test coverage. By using DisplayName annotations ot ensure comprehensive scenario coverage, I made sure to cover every aspect of each of the software requirements in my tests even after writing the code for them. I also extensively tested failure scenarios, not just happy paths. This meant causing parts of the code to fail purposefully and to check that it did fail when expected. Finally, I utilized boundary exploration to test maximum values, not just typical use cases.



Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Code quality is paramount when it comes to software engineering. Even though after many years of experience it can be easy to go through the motions rather than writing code purposefully, that is how unnecessary boilerplate code can end up in programs where it has no place. The same goes for writing tests for code as well. Any corner cut when testing code leaves the potential for at best, a minor bug that does not impact the end user. At worst, it could result in security vulnerabilities or render an application unusable. As a practitioner in the field, I plan on following continuous integration practices and automate test execution for every code change. I also will make sure to have my code reviewed by peers or managers to ensure that I am following best practices with my software.

 The testing for the UUID for the Task class is a demonstration of my testing discipline, by testing the exact UUID format rather than simply checking for non-null values.