# Name: Andre Rowe

Date: 02/22*/25*

## **Summary and Reflections Report**

#### **Unit Testing Approach**

In developing the mobile application’s contact, task, and appointment features, I employed a structured and methodical approach to unit testing, ensuring that each service met the defined software requirements. For each service:

**Contact Service**: I created tests to validate the creation of a contact, ensuring maximum length constraints for the ID and phone number, and that first and last names followed correct formatting. I also tested update and deletion functionality under various edge cases.

**Task Service**: Tests verified that all task creation constraints were satisfied (e.g., description length and due date). I also tested how the service handled task updates and deletions, ensuring it adhered to the functional requirements regarding deadlines and potential conflicts.

**Appointment Service**: My tests focused on the creation and management of appointment entries, checking for scheduling conflicts, handling invalid dates, and properly storing appointment details.

Overall, the approach was closely aligned with software requirements by focusing on correct behaviors (e.g., correct date formats and handling null values) and explicitly testing documented constraints. For instance, in the Appointment Service requirements, it was specified that appointments could not overlap; I created a dedicated test to ensure that two appointments with the same time would throw an exception.

#### **Effectiveness of JUnit Tests**

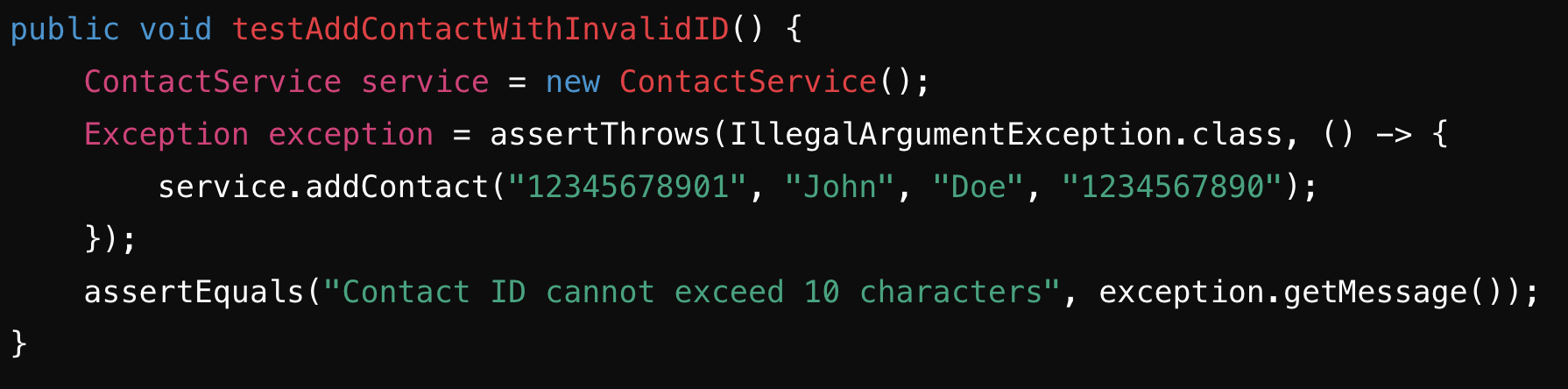
To assess the effectiveness of my test suite, I measured code coverage with a focus on branch and statement coverage. With more than 85% coverage, the JUnit tests adequately covered both typical and edge-case scenarios, such as:

* **Boundary Conditions**: Testing maximum input lengths (e.g., a 10-character contact ID) and invalid inputs (e.g., an empty string for the contact name).
* **Exception Handling**: Checking that appropriate exceptions were thrown when constraints were violated, such as for tasks with past deadlines or appointments overlapping in time.
* **Multiple Paths**: Validating that each service method functioned correctly when passed different input types and sequences, ensuring robust coverage.

A coverage above 85% indicates that a substantial portion of the code was executed and validated, leaving minimal untested paths. Though coverage alone is not the sole indicator of test quality, it supports the claim that the tests extensively examined the project’s functionality.

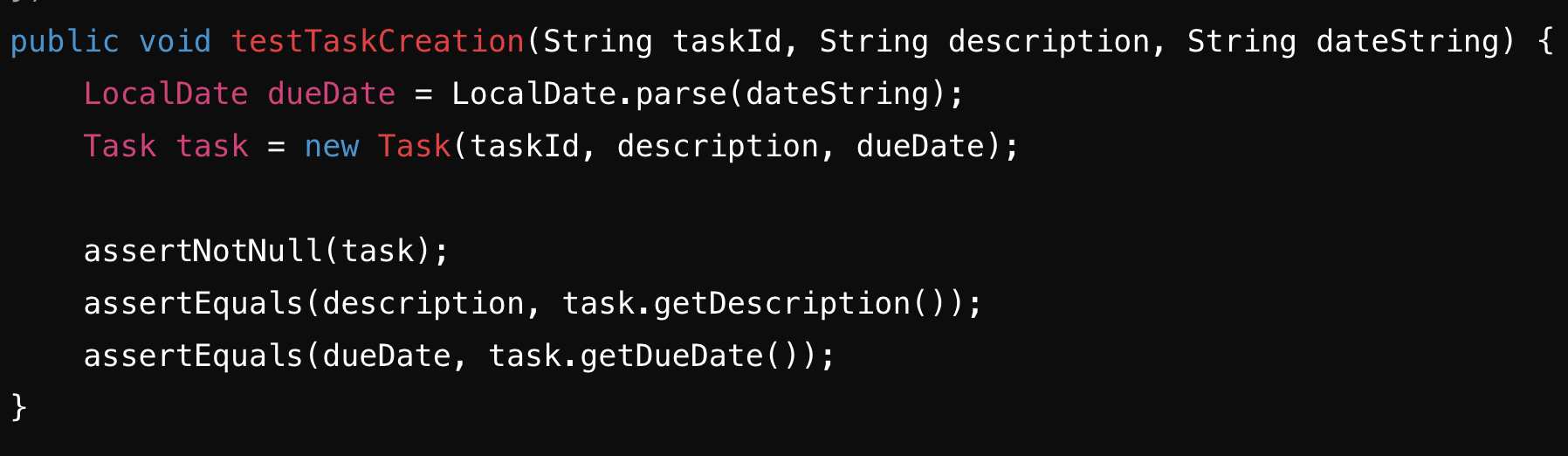
#### **Experience Writing JUnit Tests**

**Ensuring Code Was Technically Sound**  
My primary tactic involved focusing on specific assertions that tested each requirement thoroughly. For example:



In this snippet, the test checks the ID length requirement and ensures that an IllegalArgumentException is thrown when the ID is too long. By comparing the exception message, I verified the correct error was triggered. This approach not only validated business logic but also confirmed that descriptive error messages were relayed to the user.

**Ensuring Code Was Efficient**  
For efficiency, I utilized parameterized tests to reduce redundancy and systematically cover multiple scenarios without writing separate methods. For instance, in the Task Service tests, a parameterized test might look like this:



Here, multiple test cases run with minimal boilerplate code, ensuring fewer lines to maintain and quicker test execution. Likewise, these tests remain easy to expand if new valid or invalid inputs need to be tested.

### **Reflection**

#### **Testing Techniques Employed**

1. **Black-Box Testing**: Focused on input-output validation, particularly crucial for validating constraints like contact name lengths and task due dates.
2. **White-Box (Glass-Box) Testing**: Ensured coverage across various logical branches within each service class. This included testing exception blocks and conditionals related to input validation.
3. **Boundary Value Analysis**: Checked edge cases (e.g., maximum length strings, earliest and latest possible dates) to ensure the system behaved correctly at the limits.

#### **Other Testing Techniques**

1. **Integration Testing**: While I did not explicitly integrate multiple modules in a single environment, an integration testing phase could be beneficial for verifying the interplay between the contact, task, and appointment services under realistic workflows.
2. **Mutation Testing**: Involves tweaking source code (mutating it) to confirm tests catch these deliberate alterations. This rigorous method was not used but could add depth by exposing weaknesses in the test suite.

#### **Uses and Implications for Different Projects**

* **Black-Box Testing** is highly suitable for testing user-facing functionalities and external APIs, where correctness is gauged by outcomes rather than internal structures.
* **White-Box Testing** can be employed in contexts requiring high reliability and performance, such as safety-critical systems, ensuring all paths and conditions are exercised.
* **Boundary Value Analysis** is universally applicable, as nearly every software system has limit-based validations (password length, transaction amount ranges, etc.).
* **Integration Testing** is vital when distinct modules communicate frequently or depend heavily on each other’s data.
* **Mutation Testing** best serves environments requiring extreme reliability (e.g., financial or medical applications) by validating the resilience of the test suite.

#### **Mindset**

**Caution and Complexity Appreciation**  
Throughout testing, I remained cautious by acknowledging that any single change could have cascading effects. For instance, a modification in the Appointment Service’s date-checking logic risked invalidating older appointments or the logic ensuring no scheduling conflicts. Maintaining a careful mindset was key to catching subtle errors.

**Limiting Bias**  
Performing self-reviews and seeking peer feedback helped mitigate bias when testing code I had written. It’s all too easy to overlook bugs in one’s own work due to assumptions or overconfidence. Incorporating test reviews from team members (or from a fresh environment) helped uncover edge cases, such as null-check scenarios that I missed initially.

**Discipline and Commitment to Quality**  
Discipline ensures minimal technical debt. Every untested path or rushed feature can compound, leading to larger issues later. By consistently enforcing thorough testing, I reduced the likelihood of regressions. In practice, this means:

* Continuously writing tests as part of the development workflow, not as an afterthought.
* Using coverage tools and code reviews to hold myself accountable for test completeness.
* Documenting both test procedures and known limitations to maintain clarity.

In the future, I plan to continue employing high standards, including code reviews, automated testing, and continuous integration pipelines that enforce coverage thresholds. This disciplined approach will help avoid technical debt and ensure the software remains maintainable and robust.

**References**

* Fowler, M. (2018). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.