**CS320 PROJECT TWO:SUMMARY AND REFLECTIONS REPORT**

**VICTOR NGETICH**

**SUMMARY**

For this project, I implemented and tested three service classes, **ContactService**, **TaskService**, and **AppointmentService,** which managed in-memory data structures for storing and manipulating object instances. My approach focused on creating **JUnit test classes** for each feature (e.g., ContactTest, TaskTest, and AppointmentTest), where I validated all core requirements using a combination of **positive tests** (valid data) and **negative tests** (invalid or edge-case data).

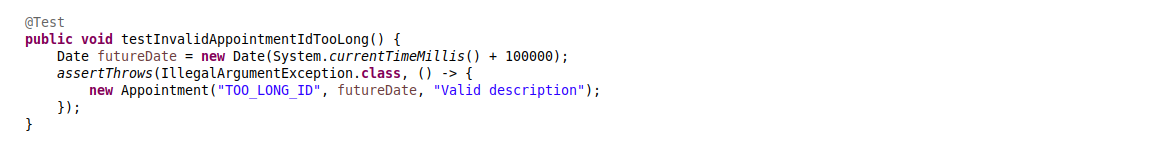
Every test method was clearly labeled and aligned with one project requirement:

* **ContactService**: Verified uniqueness of contactId, length limits, null checks, and proper updates to name, phone, and address.
* **TaskService**: Ensured taskId was unique, name and description limits were respected, and update methods worked as expected.
* **AppointmentService**: Checked that appointmentId was unique and non-null, the date was not in the past, and descriptions met requirements.

Alignment with requirements

For appointment object, the test:

* verifies that appointmentId cannot exceed 10 characters, directly supporting the requirement,The appointment ID shall not be null and shall not be updatable.



For ContactService;

this demonstrates enforcement of unique contact IDs, another core requirement.



### **JUnit Test Effectiveness**

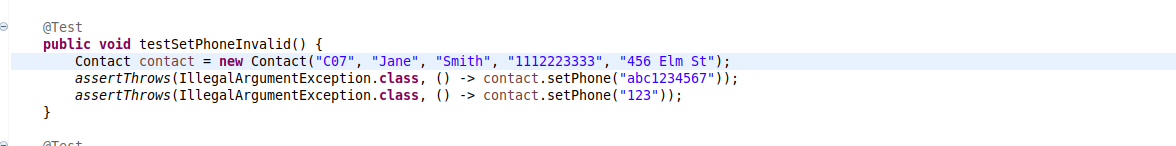
I ensured my tests achieved **at least 80% code coverage** by testing:

* Valid object creation
* Exception handling for invalid inputs
* Update and delete functionality
* Non-editability of unique IDs

Coverage tools in Eclipse IDE and thorough manual reviews confirmed that the tests covered all critical branches and logic paths.

### **Experience Writing JUnit Tests**

Writing JUnit tests was a structured and iterative process. I began with simple valid-case tests, then incrementally added boundary and negative tests. The use of assertThrows() was particularly helpful for validating exception conditions. For example:



It ensured only valid 10-digit phone numbers were accepted.

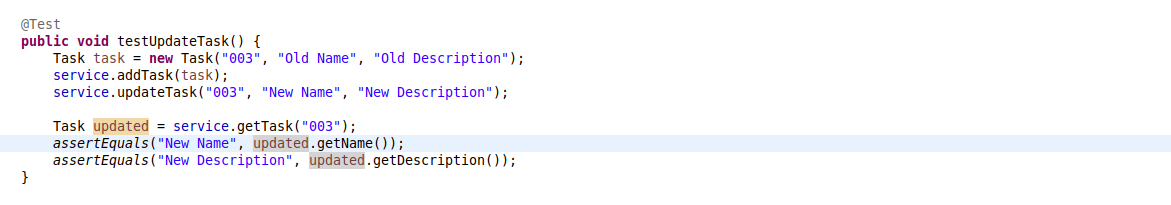
### **Technical Soundness**

I made sure my code remained technically sound by:

* Checking all test method names were meaningful and descriptive
* Testing expected exceptions using assertThrows()
* Avoiding magic strings or hard-coded test values where possible

### **Efficiency**

Efficiency was ensured through reusable helper methods and avoiding redundant tests. I also used small, focused test cases to isolate problems easily. For example:



**REFLECTION**

### **Testing Techniques Employed**

I primarily used **unit testing** with **white-box testing** techniques:

* I had full access to the class internals and wrote tests based on the known structure.
* I used **boundary testing** to check string lengths (e.g., exactly 10 characters vs. 11).
* I applied **exception testing** for invalid/null data inputs using assertThrows().

### **Other Testing Techniques Not Used**

I did not use:

* **Integration testing** – because the system only included isolated in-memory services.
* **System or end-to-end testing** – as no UI or database was involved.
* **Mock testing** – since no external services or dependencies were integrated.

### **Practical Uses of Techniques**

* **Unit testing** is ideal for validating individual classes before integration.
* **White-box testing** is especially practical during development for full control.
* **Boundary testing** is useful in user input validation or forms.
* **Exception testing** ensures application robustness under edge-case scenarios.

**MINDSET**

#### **Caution and Complexity**

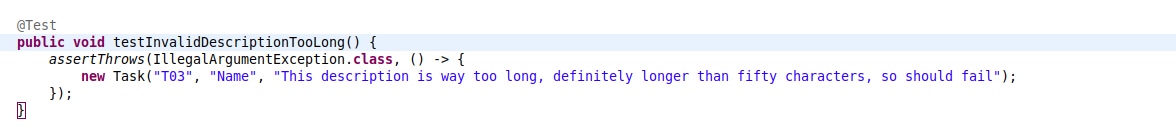
Throughout this project, I approached testing with caution, recognizing the **interdependency of service logic and validation constraints**. For instance, failure to correctly test the AppointmentService date logic could allow past-dated appointments. I paid close attention to constructor logic, data types, and exception triggers to prevent subtle bugs.

#### **Limiting Bias**

As both developer and tester, I was aware of the risk of **confirmation bias,** expecting the code to work as intended. To limit this:

* I tested invalid inputs even when I believed the logic was solid.
* I wrote tests that intentionally failed to confirm error handling.

For example, I tested:

to make sure null task ids were not silently accepted,

#### **Commitment to Quality**

Quality assurance is crucial in real-world software development. Cutting corners in writing or testing code can lead to **bugs, security flaws, and maintainability issues**. To avoid technical debt:

* I plan to **write tests alongside development** to prevent regressions.
* I will adopt **test-driven development (TDD)** when practical.
* I’ll consistently review and refactor code to ensure clarity and simplicity.

### **References**

Cunningham, W. (1992). The wycash portfolio management system. *Addendum to the proceedings of the OOPSLA '92 conference on object-oriented programming systems, languages, and applications*. ACM. <https://doi.org/10.1145/157709.157715>

Link, J., & Eigendorf, M. (2023). *JUnit 5 user guide*. JUnit Team. <https://junit.org/junit5/docs/current/user-guide/>

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons.

Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.