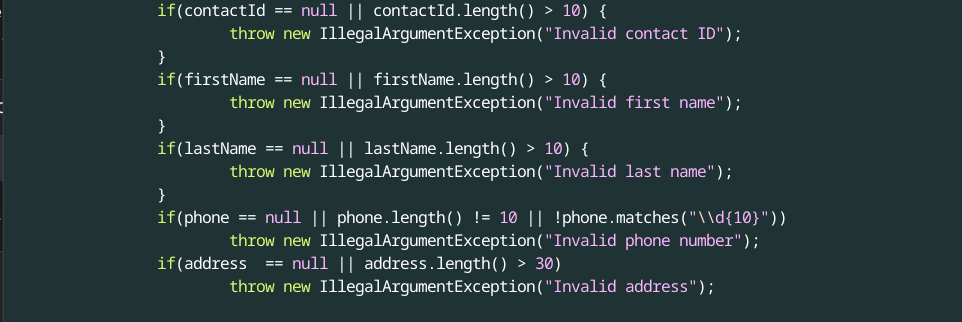
Project one, focused on three primary features including the ContactService, TaskService, and the AppointmentService. Each of these objects managed different sub classes that included the Contact, Task, and Appointment. Validation and constraint checking was implemented and tested with the use of the JUnit testing suite using various testing techniques (Junit 5, 2023).

Each of the objects required their own testing and validation of requirements. The ContactService needed the ability to add, update, and delete contacts with the input being verified that it was not null and had a 10 digit phone number.

Each test was closely aligned with the functional and validation requirements specified in the class constructors. For example, Contact required a valid 10-character phone number as well as character length, validated with the following logic:



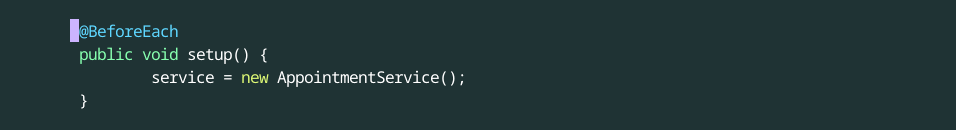
The same pattern was followed for the other classes for the user input validation with variation happening depending on the requirements of the project.

The effectiveness was measured by how well the tests covered the different possible user inputs. The achievement of complete or near complete coverage was accomplished through positive testing, negative testing, and boundary testing with validation limits

JUnit tests were implemented to assure that the programs properly handled the conceivable user inputs, both positive and negative. This included best case scenarios where the user input was proper as well improper input, and edge cases. The writing of these tests followed a similar pattern and was an iterative process that also helped ensure the technical soundness by testing the constraints defined in constructors of the class.

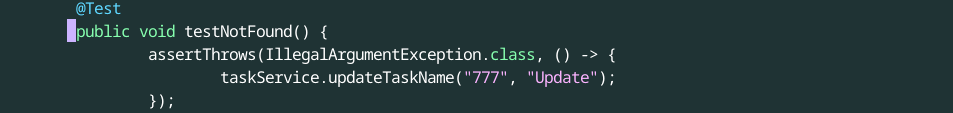


Due to the iterative nature of the tests, reducing redundancy became a necessity. This was accomplished through the use of @BeforeEach methods in the JUnit testing library.



The primary testing strategy involved unit testing or the testing of individual units of code thereby isolating the parts and checking for errors (Jorgensen, 2022). Boundary testing was also implemented for field limits as well as exception testing which focused on any null values in required fields. Due to the stage of development that the project is in, other testing techniques were not applicable such as full system testing or in depth integration testing. This makes sense as unit testing is an excellent choice for checking and enforcing the hard set rules during development.

A defensive mindset was a must during the testing of the system as it helps in excellent testing coverage. As such it is the mindset that I focused on, assuming the many different inputs could break the program. As an example, in the TaskService, an update was attempted on a task that doesn’t exist in order to test the exception trigger.



To limit bias, the tests were written in a manner that tried to break the code. This is in direct contrast to writing tests that just confirm what outcome is to be expected. I designed the tests with the assumption that my implementation may have flaws in it.

Quality in one's finished product is always something of importance and software engineering is no exception. The improper implementation of testing due to time constraints or laziness cas introduce bugs that are much harder to fix later in the development cycle. In order to avoid this I commit to TDD and the implementation of DRY coding techniques which can make maintenance easier as well as increase the readability of the code. Regular code reviews will also be implemented.

References

Jorgensen, P. C. (2022). \*Software testing: A craftsman’s approach\* (5th ed.). CRC Press.  
JUnit 5. (2023). \*JUnit 5 User Guide\*. https://junit.org/junit5/docs/current/user-guide/