Summary & Reflections Report

**Summary**

My tests for Contact, Task, and Service objects went one character beyond the valid number of characters to test the accuracy when the input was barely out of the range of accepted characters(edge cases).

if (*Id* == null || *Id*.length() > 10) {

throw new IllegalArgumentException(

"Id cannot be longer than 5 characters"

);

}

I also tested each input with a null value.

if (*name* == null || *name*.length() > 20) {

throw new IllegalArgumentException(

"Value cannot be null"

);

}

My tests resulted in a test coverage percentage of 92-100%. While code coverage does not guarantee a code’s effectiveness, a test coverage above 85% would suggest:

* A significant portion of the code has been tested.
* Potential bugs and errors have been identified and fixed.
* The code is of higher quality

All edge cases were properly addressed for each object (Contact, Service, Task) .

The name is to be no more than 20 characters. The test below has 21 characters.

* + @org.junit.jupiter.api.Test

void testNameTooLong() {

*assertThrows*(

IllegalArgumentException.class,

() -> {

Task *task* = new Task(

"34527t5",

"GeoffreyInikoInityKik",

"A top notch student"

);

}

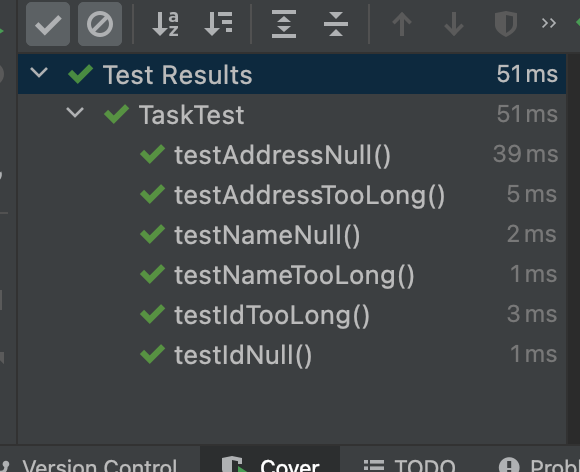
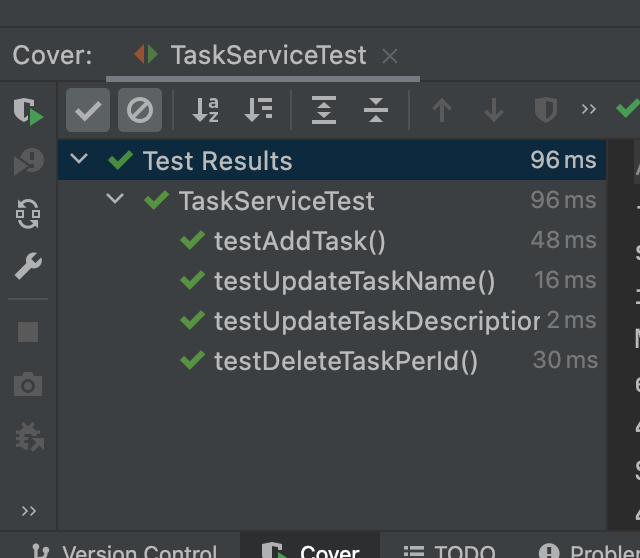
);

}

* All of the tests have the proper naming convention.
  + @org.junit.jupiter.api.Test

void testNameNull() {

The test results in execution time show the efficiency of the tests.



Tests are also readable and maintainable.

@Test

void testDeleteTaskPerId() {

TaskService *task* = new TaskService("345trf", "Iniko", "Cool Kid");

*task*.addTask("345trf", "Iniko", "Cool Kid");

*assertTrue*(*task*.deleteTasksPerId(*task*.getId()));

}

@Test

void testUpdateTaskName() {

TaskService *task* = new TaskService("43tfre21", "Inity", "A-Student");

*task*.addTask("43tfre21", "Inity", "A-Student");

*task*.updateNameById("43tfre21", "Geoff");

*assertTrue*(*task*.getTasksById("43tfre21").getName().equals("Geoff"));

}

**Reflection**

Testing Techniques

Milestones within modules three, four, and five involved utilizing JUnit 5 programming model ***Jupiter*** to conduct the software tests. Each module involved the following steps:

* Setting up the class to be tested.
  + All data members and methods.
* Setting up validation parameters for the data members within the constructor.
* Creating a test class for each class.
* Importing all necessary ***Jupiter*** packages into the test class file.
* Declaring the object with parameters just outside the acceptable input for each test using
  + assertThrows
  + assertTrue
  + assertEquals
  + assertNull

Examples:

@Test

void testFirstNameLengthTooLong() {

Assertions.*assertThrows*(

IllegalArgumentException.class,

() -> {

new Contact(

"465464",

"JohnnyeRays",

"Green",

"7702367652",

"123 1st Street"

);

}

);

}

@Test

void testUpdateTaskName() {

TaskService *task* = new TaskService("43tfre21", "Inity", "A-Student");

*task*.addTask("43tfre21", "Inity", "A-Student");

*task*.updateNameById("43tfre21", "Geoff");

*assertTrue*(*task*.getTasksById("43tfre21").getName().equals("Geoff"));

}

I did not use the group assertions assertAll method, which allows for the grouping of different assertions at the same time. Other characteristics of the assertAll method include the following:

* All assertions are always executed, and any failures will be reported together.
* They provide better error messages than individual assertions.
* Can be used with other JUnit 5 features such as parameterized tests and test interfaces
* Especially useful for testing complex or interconnected functionality

Practical uses and implications for different software development projects and situations.

* Testing complex functionality
* Code integrity
* Code readability
* Reduce code duplication
* Increase code coverage

**Mindset**

Throughout the testing process, I always inwardly but consciously took into account the requirements of each object. Each line of code written has that in mind. This helped me to design code that would exhaustively examine the edge cases.

By having a solid understanding of the code and the requirements, I was able to write more comprehensive and effective tests, identify edge cases, scenarios, and dependencies that may impact the behavior of the code, and ensure tests cover all of these cases.

if (*Id* == null) {

throw new NullPointerException("Id cannot be null");

}

The above code has a requirement that the id field cannot accept a *null* value. The test below inserts a null value which should trigger the NullPointerException message if working properly.

@Test

void testIdNull() {

Assertions.*assertThrows*(

NullPointerException.class,

() -> {

Appointment *appointment* = new Appointment(

null,

"10-23-2022",

"Just a test"

);

}

);

}

I tried to limit bias when reviewing my code by following all the standards and procedures for JUnit testing.

Testing your own code has its advantages, as the builder of the code should know its limitations. Despite that, having another developer less or unfamiliar with the project will provide a closer to “*real world*” scenario test samples. The following steps will help minimize biases:

* Follow a checklist
* Work in interval with scheduled breaks
* Document the process

My understanding of the requirements gave me the knowledge to know exactly what should “*not*” work. As an example, I understood that the Appointment object took 3 fields and each field had specific parameters. So I knew exactly what should work and what should not. This was the foundation of the writing of my tests. This was useful, but, ultimately, it’s always ideal to have someone else review and test your code.

In order to produce high quality software, an engineer must be disciplined. Reasons discipline is essential:

* Consistency - makes certain you follow the same process every time
* Efficiency - Following processes saves time and effort which allows you to focus on other tasks
* Reliability - solid software is less likely to fail which builds a solid reputation with customers
* Security - less likely to contain vulnerabilities that could be exploited by hackers.

Discipline ensures best practices and established guidelines for software development are being followed at all times; resulting in high quality code free of errors and bugs.

Technical debt refers to the cost that arises when shortcuts are taken during software development, resulting in code that is difficult to maintain, update, and scale. Some effective ways to avoid technical debt are:

* Follow best practices
* Refactor regularly
* Keep code simple
* Manage dependencies
* Continuously monitor and improve code

When I first started testing the Appointment object I used the Java util.Date dependency to manage the dates and times in the application. However, this was causing accuracy issues and resulting in my tests being inaccurate. I ended up switching that dependency to the java.time package which provided improved functionality and allowed the tests to pass accurately.

Myers, R. (n.d.). *Avoiding technical debt with these "core Four" practices*. Scrum Alliance Resources. Retrieved April 15, 2023, from [Avoiding Technical Debt with These "Core Four" Practices](https://resources.scrumalliance.org/Article/avoiding-technical-debt-core-four-practices)