Summary & Reflection

**Summary**

Aside from classes dedicated to testing, six classes were created in this project, three to create custom objects and three service classes for interactions using the custom objects. The custom object classes were Contact, Task, and Appointment.

Each custom object’s attributes were required to be non-null and have length restrictions as specified in the requirements documentation. Each service class was also given a strict set of requirements via documentation. Services must be able to add, delete, and modify objects stored in memory. Since each object was slightly different, special care and planning was needed to execute effectively.

Since each service class would provide an interface for interaction with the base object classes, it made sense to restrict each service to a single running instance at any given time, so a singleton pattern was employed.

**Testing Approach & Reflection**

My unit testing approach for both object and service classes involved two primary objectives. First, validate functionality with acceptable inputs, then confirm appropriate errors were thrown when given unacceptable inputs. Before I began writing any test code, I created empty classes and used comments to outline each requirement in the documentation. When a compound requirement was given, I dissected it into two separate comments, iterating until each task was as simple as possible. For example, requirements state “The contact object shall have a required unique contact ID String that cannot be longer than 10 characters, shall not be null, and shall not be updatable.” Five segments were dissected from this requirement: contacts have a contact ID string, contact ID must be unique, contact ID cannot be longer than 10 characters, contact ID cannot be null, and contact ID cannot be updatable. Then, I grouped each of my comments into categories. This was approach allowed me to become more intimate with the requirements and have a higher chance of success when I started writing the tests.

My approach to testing Contact and Test object instantiation was to subcategorize the unacceptable inputs by constraint, such as when too long, or null. This was effective, because all parameters are validated for each constraint with identical code. For example, in Contact, the constructor checks one parameter at a time against all constraints with five structurally identical ‘if statements.’ When an ‘if statement’ is triggered, an illegal argument exception is thrown. Categorizing tests this way helps to identify similar possible failure points. If a point (such as checking for null) fails due to a logic error for any parameter, it can be assumed that the other parameters will also fail.

Displeased with the organization of the test code, I adjusted my approach for the Appointment class to test attribute first, instead of constraint first. Although the testcases would contain the same code, all constraints tested against a single parameter helps readability, maintainability, and increases effectiveness of manual testing techniques of the test class’s code and still provides the same high-quality results.

Since each of the service classes are structurally identical, each service test class is organized by method/task, such as adding or deleting an object. The four categories as testcases are: add objects, prevent duplicates, update objects, and delete objects. Within each category, all reasonable outcomes are tested. As a redundant check, tests are performed at least twice on separate objects. For example, the AppointmentService test class tests adding two unique appointment objects and confirms that each appointment is present in memory. It is important to perform these additional checks to weed out any unusual failures. Ideally, numerous unique objects should be tested against each constraint for added confidence. This is especially important as the underlying code becomes more complex.

**Effective Testing & Coverage**

Evaluating test quality based on coverage percentage should be used cautiously. Coverage checks test code for obvious outcomes. Even when coverage is reported as 100% , exhaustive testing can still identify problems. The coverage of my unit tests for all object and service classes is at 100%, but the underlying code being tested might be flawed, so both the testing approach and approach taken when writing the underlying code must be sound.

I wrote the object classes and the service classes with testing and reusability in mind. All methods return a value that can be tested against. In update methods, the object’s constructor is called to confirm the parameters entered would not violate one of the object’s parameter constraints. If this fails, the object’s constructor will throw an illegal argument exception. If it is successful, the update method will search for the object by its ID. If found, the object is updated in memory and a Boolean ‘True’ is returned. If unsuccessful, an exception with an identifiable message is thrown to make debugging and testing easier.

**Testing Techniques**

I used two primary testing techniques extensively in my code. Manual review/linter aided revision is often overlooked when considering self-tested code, because we edit and revise over iterations when writing code to ensure it performs as needed. A forgotten curly brace or semicolon can be found easily with the aid of an IDE’s built-in linter, but the linter does not catch everything. Sometimes, it identifies false positives. This is observed when I test for null values upon object instantiation. The linter identifies that the code always will fail when attempting to instantiate with null parameters. Since each parameter is not allowed to be null according to the requirements documentation, writing unit tests with null values remains important. We know an exception will be raised, but unit tests can increase confidence that no parameters can be null!

Unit tests were performed with JUnit 5. Instead of writing a user interface, where functionality is tested manually by running the code, unit tests can evaluate the functionality of each component. By writing unit tests that cover acceptable and unacceptable input, expected behaviors can be observed. The requirement documents was the principle document that determined what the tests needed to look for, so numerous unit tests were written based on if input lengths are acceptable, are non-null, and in the appropriate format.

Object Oriented Programming testing can be performed once more functionality is added and principal functionalities (add, update, and delete: Appointments, Contacts, and Tasks) are integrated. Upon integration, integration testing is used to evaluate if all modules are compatible with one another and unforeseen complications can be reduced or eliminated.

Realistically, at this time, the application is not designed for mobile use. Mobile application programming, requires additional libraries and special considerations far beyond the current state of the code. Android applications natively use Java, but many unique constructs are required to convert a desktop application to an Android application, even if written in Java. IOS applications are typically written in Swift, and have little tolerance for non-Swift applications to be ported. Once converted to a true application, non-technical users can interact with it, and in-memory containers are converted to either a database or can store reusable data can other types of testing, such as User testing, Acceptance testing, and System testing.

**Writing Technically Sound & Efficient Code**

To ensure my test code was technically sound, I often referenced JUnit and open-source documentation to confirm correct and complete use of JUnit’s methods. Performing a manual review following the completion of each testcase and running the test code frequently can greatly reduce time spent debugging later.

Prior to the start of this course, I was unfamiliar with the @BeforeEach annotation available in JUnit. I used the singleton pattern in all three service classes, so to allow tests to run independently and in any order, I wrote a method in the service classes to clear objects in memory and called it before each test using this annotation. Left alone, tests could fail due to left over data from the randomly selected test/s already performed.

Performing unit testing in multiple programming languages and having already been familiar with JUnit and object-oriented programming techniques, I was extremely comfortable writing unit tests. However, repetition and experimentation are extremely beneficial. As programmers, we are always sharpening our skills and many paths meet at the same destination, so I regularly reflected on my code to see how it could be improved. For example, I felt I could use more practice using lambda expressions, so I found ways to incorporate them into some of the test code after confirming a non-lambda variant of the same test performed as expected. Lambda expressions can be technically challenging, but many agree that they make code more human-readable than longer alternatives. My use of lambda expressions can be observed in all test classes anywhere an assertion expects an error to be thrown.

As a codebase grows, efficient source code and test code exponentially becomes more desirable. This is equally true when a process needs to run repeatedly. In a mobile application, hardware resources are at a premium, making efficient use of resources and adaptation for performance a must no matter how simple an application may seem. Even though test code will not be part of the final product, efficient test code can reduce time needed to test larger projects. As more code is added, an inefficient method can drastically slow progress.

In my test code, I could have written a single test case that covered all boundaries and requirements. Likewise, I could have written short but complex code that is less readable. Unintuitively, shortening code does not increase performance or efficiency. I chose to segment and organize my testcases into categories not just for added readability, but to allow me (or other developers) to turn on or off testcases for more dynamic testing processes.

Attempting to algorithmically change my code for better efficiency would be unrealistic, since most (if not all) of my code was implemented with well known, tried and true techniques, so I focused on writing tests that were efficient and technically sound with respect to human readability.

**Exercising Caution & Realizing Bias**

Considering my familiarity and comfort using JUnit and programming in Java, I did not need to exercise as much caution as I would have when these concepts were still new to me. Fortunately, this gave me an opportunity to explore the different ways to organize my tests and gain additional experience with the Eclipse IDE. (I typically write Java in IntelliJ.)

Even in situations where code is extremely familiar and proven, erring on the side of caution and performing comprehensive testing early and often and still writing unit tests for basic functionality can save loads of time and frustration when tracking down bugs. For example, I know that the technique I used to create the three objects is sound, as it is essentially boilerplate Java class construction. I could have submitted it to the company’s repository without testing, but if something goes awry, my tests will fail and show exactly where and what problem occurred.

A common pitfall for developers can be observed when hubris and ego prevent from cooperating in a team environment due to stubbornly refusing to see other developers’ points of view. Even when a task such as creating simple objects and service classes like this can be done in numerous ways. Sure, some ways may be more organized, effective, and/or efficient, but being too confident in one’s own code hinders improvement.

Larry Wall, the creator of Perl, outlines three (nefarious) virtues of a great programmer, Laziness, Impatience, and Hubris. According to Austin Pocus of Hackernoon (2020), “Hubris is the worst of the three… Have you ever had someone defend their code like it was their child, refusing to make changes or improvements?” This mentality rings true to hubris and laziness for all code, in an extreme case, a developer may refuse to write tests, but even the simplest code can have faults. Identifying weaknesses early through unit tests and possessing the humility to subject one’s own code to testing reduces time spent later tracking down a problem.

Authoring code can be viewed as an artform and a science, so developers often become very intimate with their own code and become blind or ignorant to defects. This results in a narrow scope within their own testing approach. Observing others, accepting advice and evaluating one’s own code without the hinderance of bias and pride is critically important for all stages of a software developer’s career.

**Discipline**

Even when a project can be completed in a shorter timespan than allotted, starting the project early, seeking different ways to do things, and experimenting with both existing proven methods and new ones is massively beneficial to writing quality code. This can be observed in my decision to adjust my organizational method when testing code. Although the output is the same, it can be argued that my second organizational iteration makes the code easier to read and thus more maintainable and reusable.

A constant quest to try new things and learn new material, especially in a field that evolves at an incredibly fast pace, keeps concepts fresh and encourages developers to keep an open mind. Having an unsatiable desire to learn and improve is a characteristic that I hope I never lose. At times of frustration or dissatisfaction, it can be important to take a step back and reevaluate or recharge. Although it seems counterproductive, taking regular breaks and returning to code, especially when stuck, can be an incredible way to plow through roadblocks that are otherwise impassable. I often struggle taking breaks from my work, because I have a tendency to get carried away with it. As a reminder to take breaks, I often employ the Pomodoro technique, where I focus without distraction for 30-minute intervals followed by a 5–10-minute break.

Works Cited

Pocus, A. (2020, March 23). *Larry Wall’s “Three Virtues of a Programmer.”* Hackernoon. Retrieved August 12, 2022, from https://hackernoon.com/larry-walls-three-virtues-of-a-programmer-are-utter-bullshit-fykp32ck