Project Two

Rachel Aldava

CS-320

Prof. Phillips

04/21/2024

**Project Two**

**Introduction**

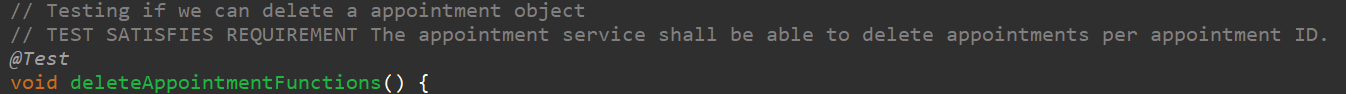
We were asked to develop several files which are intended to be a component of a mobile application. These files were developed in Java and had very specific requirements. As an example, here is a transcript of the Appointment and AppointmentService requirements:

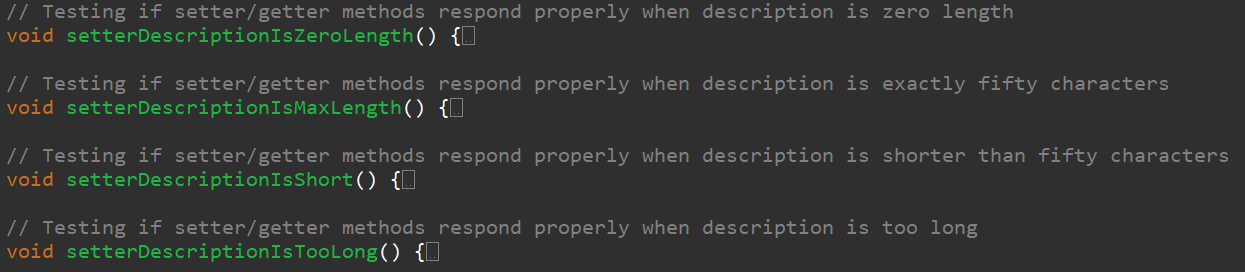
1. **Appointment Service**:
   1. Appointment Requirements
      1. The appointment object shall have a required unique appointment ID String that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable.
      2. The appointment object shall have a required appointment Date field. The appointmentDate field cannot be in the past. The appointmentDate field shall not be null. *Note: Use java.util.Date for the appointmentDate field and use before(new Date()) to check if the date is in the past.*
      3. The appointment object shall have a required description String field that cannot be longer than 50 characters. The description field shall not be null.
   2. Appointment Service Requirements
      1. The appointment service shall be able to add appointments with a unique appointmentId.
      2. The appointment service shall be able to delete appointments per appointmentId.

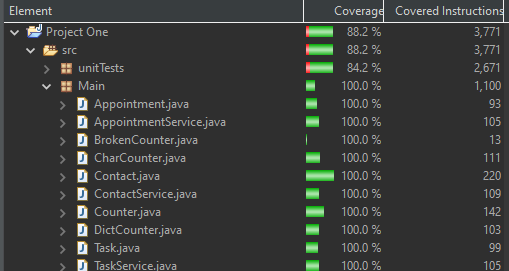
After developing these classes, along with some supporting classes, we developed a range of tests in order to ensure that our designs meet the specific requirements. This document will outline the methodologies used, discuss design decisions, and reflect on the project as a whole.

**Summary**

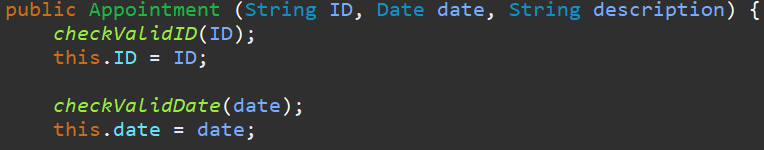
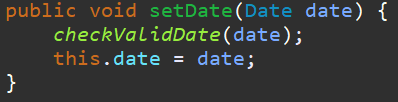
**The unit testing approach** we took was based primarily around the requirements. Because the requirements were so specific and left little room for interpretation, we were able to go over the requirements line-by-line and simply implement the requirement into a test. For example, a snippet from AppointmentServiceTest:

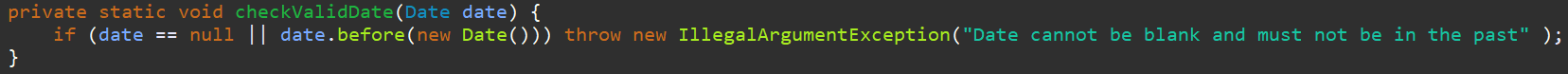
 Once we established the basic functionality unit tests, we proceeded to develop tests for each public method within the class. In this way, we were able to **align to the software requirements** by specifically developing tests from those requirements, and then identifying edge cases. For example, for most string inputs, we tested null values, zero length values, short values, max character length values, and values which exceeded the character length:



There is a certain difficulty in measuring the **overall quality of the JUnit** tests in terms of how effective they are, since there could be an aspect of the project which was overlooked by both the developers and testers. One measurement for quality is **coverage percentage**, but this has limitations. For example, if one were to design code which multiplied two numbers together, and the values a unit test supplied were (null, null),(0,0), and (2,2), then the code return a + b; Would cause the unit tests to report that there were no issues with the code. Such a situation might achieve 100% coverage and yet still be insufficient. For all three features, and various supporting files, **we achieved 100% JUnit test coverage** which means that our tests covered every instruction of the project. This alone is not sufficient to conclude that there are no issues with the code. We are confident that we covered the obvious and relevant edge cases, but because this code relies on libraries which are external to this project, we cannot ensure that there is not an undiscovered vulnerability which could allow for unexpected behavior.

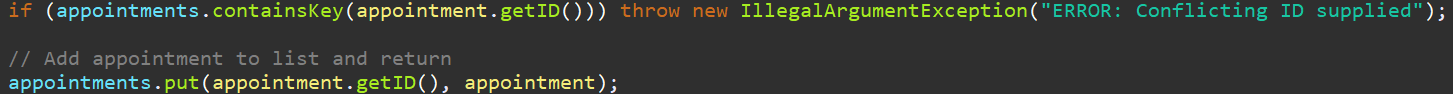
One of the ways we ensured that our code was **technically sound** was adhering to software development best practices including Object Oriented Programming (OOP) principles. These principles are encapsulation, abstraction, inheritance, and polymorphism (Chandel, 2021). We practiced encapsulation and abstraction by ensuring that most of the internal logic, variables, and methods were private and only accessible or modifiable through controlled “getters” and “setters”. We applied inheritance and polymorphism in our different implementations of ID generation which featured a parent “Counter” abstract class and two children counter classes which followed different strategies for generating ID strings. Another best practice we applied was ensuring that a given method did one and only one thing; We did not construct one all-encompassing input validation method, but rather, created individual validation methods and linked to them during object creation. In so doing, we were also able to use these same input validation methods whenever a particular field needed to be updated. This not only reduced the number of tests we needed to write, but also ensured that there would not be two different areas of code determining whether a particular field value was valid.



In terms of **efficiency**, the code did not require any complicated algorithms. There were three significant logical checks which were employed in these three features; One was to check the length of a string, to check if an id string matches another id string, and to check whether a time was before or after the present time. We trust in the accumulated expertise involved in creating java libraries and we utilized the built-in string checking algorithms. In order to store ID strings, and also to store their associated objects, we utilized standard Java HashMaps to store key:value pairs, where the key was the ID and the value was the object, then used the standard methods to check whether the HashMaps contained a given string.

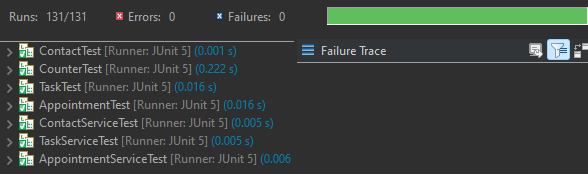




Checking whether a given time was before or after the present time was achieved by utilizing the Date data type, as specified within the requirements. While researching the best way to implement the logic, we discovered that there are some non-intuitive pitfalls involved with relying on helper utilities which manage the Date (Coding with John, 2020). Because of this, we decided to avoid integrating any type of translation from a string to a Date or a Date to a string and instead simply compare the two Dates in a direct manner:



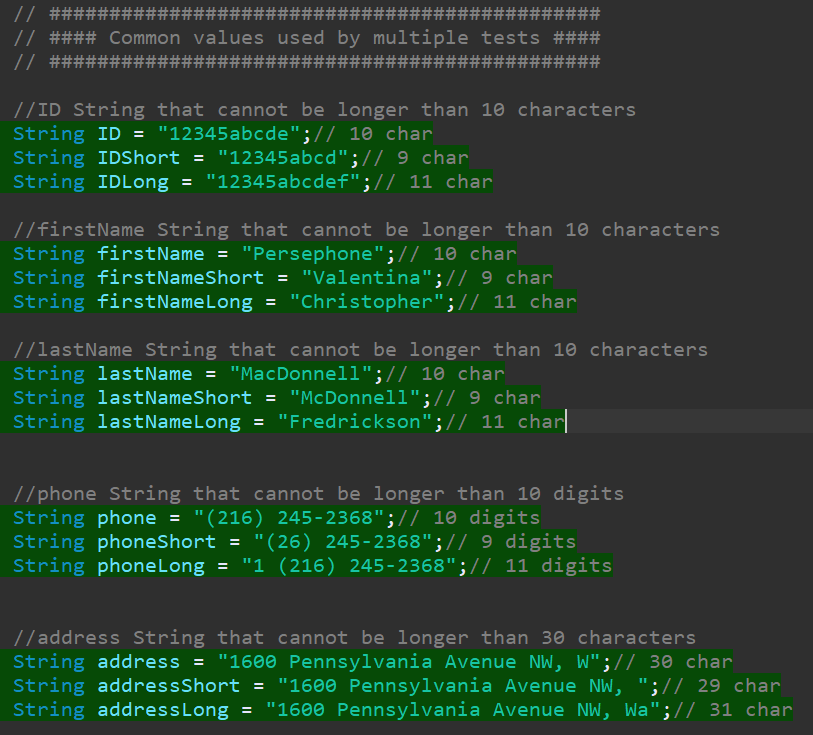
**Reflection**

The **software testing techniques that we employed in this project** were mostly under the umbrella of white box testing. We wrote unit a total of 131 unit tests across the various class files. Task and TaskService , Contact and ContactService, and Appointment and AppointmentService represent pairs of code on which we could perform integration testing. These interactions are included as JUnit tests. Throughout the testing process we employed manual code review in order to ensure that the code functioned correctly and efficiently. We performed a small amount of functional testing to our ID generation algorithms in order to ensure that they both produced results in a timely manner. We believe that both have an O(*n*) run time complexity where *n* is the number of characters in an ID string; this effect is mostly (but not entirely) due to strings being immutable in Java, so each time the counters increase they must build a new string. Within the 10 character limit, depending on the ID generation strategy chosen, there can be up to 65535^10 – 1 unique ID values, although many of these will not be readable to a human. The most restrictive strategy is simply a numerical counter which will have 10^10 - 1 possible ID values.

Due to the nature of this project, there were other **software testing techniques which we did not use for this project**. Black box testing, for instance, is a process which involves a tester using a complete system without external visibility to the code (McKeever, 2023). Because we were developing components of a larger system rather than a complete and final system, we were not able to perform black box testing. Similarly, because this was not a complete system, we were unable to perform acceptance testing, system testing or some aspects of integration testing. There are many aspects of functional testing which we were unable to perform because, again, this was not a complete system.

The **practical uses and implications for** functional testing are to ensure that the program meets the requirements outlined. Unit testing constructs a series of tests which can set the standard for how a particular section of code operates. Any modifications or optimizations to the internal logic of the code can re-use the original unit tests to ensure that it still behaves in a similar and predicable way. Integration testing helps to ensure that the pieces of code can properly interact with each other. System testing ensures that the system can function properly as a single unit, and acceptance testing ensures that the system meets the end user’s expectations. Code reviews helps to reduce errors which could cause unexpected behavior or reduce performance.

The **practical uses and implications for** nonfunctional testing are that they help identify issues with how the system operates. One of these nonfunctional tests might be measuring the response time of the system; a web page which takes five seconds to load is not, generally speaking, part of a good system. Other aspects of nonfunctional testing might include other aspects of how usable the system is, whether the system scales well, and how reliable the system is over time.

While **acting as software testers, we strove to employ caution**. Mentioned previously in this document was the fact that simply because the unit tests have wide coverage does not mean that the code is functioning as intended. We are, ultimately, limited by our imagination. While constructing the unit tests and testing the features we were mindful of this fact. Again, we attempted to account for every possible input value to the functions, but there may be something we overlooked. While writing the tests we attempted to isolate dependent variables by defining input values which the unit tests shared. In this way, it avoids false negatives or false positives which can arise from constructing new input values in each test.

Some **amount of bias** in this project was unavoidable. Because our development and testing teams were combined, and because the combined headcount for both of our teams was one person, the same person who wrote the code was also tasked with writing the tests. The incentive structure for this individual was to produce a product which met the requirements and contained no errors. The consequence of producing a product which contained errors or did not meet requirements was that they would receive a poor evaluation. Thus, the incentives for both the development team and the testing teamed were aligned toward receiving a good evaluation. This incentive alignment may help with some aspects of bias, but it does not eliminate it. Because the code was developed and tested by the same person there is a weakness of the same interpretations and thought patterns being used; The tester remembers what the original code was supposed to do rather than what it actually does. We partially controlled for this by scheduling development and testing on different days. We also introduced a common structure which was used across all three features. These steps helped to reduce the effects of bias, but did not eliminate them.

Regarding **commitment to quality and avoiding technical debt,** there was a lot of investment put into the first feature. If we were to chart time spent on the first feature, alongside developing unit tests for the first feature, it would show a very heavy up-front time investment and relatively little time spent on the second two features. During our second feature we slightly changed some internal logic, but for the most part it took the form of the first feature. The third feature implementation, while it did feature new logic regarding time, had the shortest development time. We feel the reason why the second two features were easier to create was because we were working off of a solid foundation established during the first feature. Time and discipline invested in the first week paid dividends later by avoiding technical debt. Because all three features were so similar we did consider consolidating the three features into a parent:child relationship, but it seemed as if the concepts which the classes represented “appointments”, “people”, and “tasks” were too dissimilar to combine, even if their implementations within the code were similar. Finally, we mentioned before that there were known issues with the common Date helper methods which could produce errors in multi-threaded applications. We intentionally avoided using these easier ways to handle the Date datastructure and instead sought out solutions by working with the built-in methods. For example, we manually calculated what internal Long value would represent the year 3000, which was perhaps a comically absurd choice for a future date which also happened to test for a y3k bug. Testing for a y3k bug could count as avoiding technical debt, right?

Right?

**Works Cited:**

Chandel, M. (2021, November 4). What are four basic principles of Object Oriented Programming? Medium. <https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727>

Coding with John. (2020, October 23). Java’s SimpleDateFormat is a Disaster Waiting to Happen [Video]. YouTube. <https://www.youtube.com/watch?v=JdNQoTJmzis>

McKeever, G. (2023, December 21). What is Black Box Testing | Techniques & Examples | Imperva. Learning Center. <https://www.imperva.com/learn/application-security/black-box-testing/>