Benjamin Sturgeon

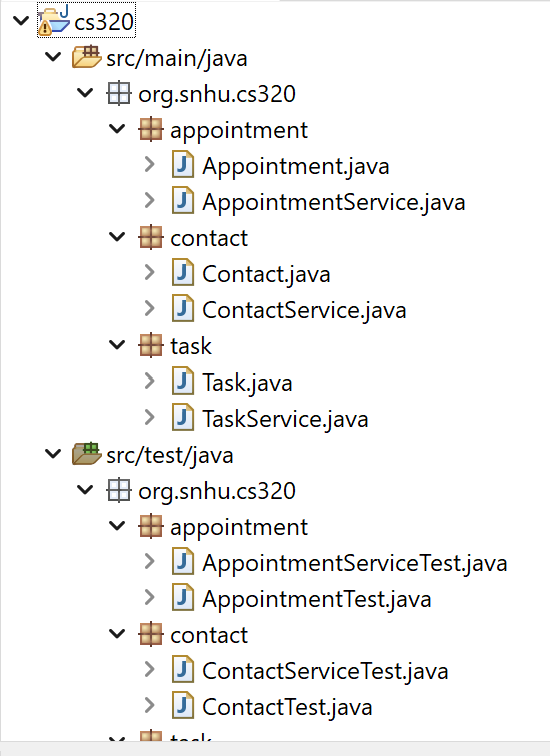
CS-320-13763-M01

October 20, 2024

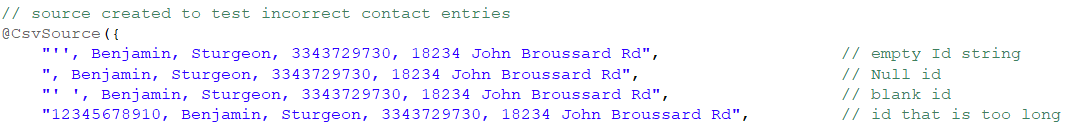
Project Two

**Summary**

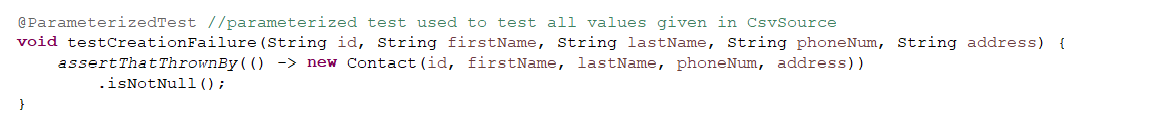
The completion of Project One provided excellent experience with Test-Driven development. I will first summarize the work that was done while completing Junit Tests and designing the application to support test analysis. Software requirements were successfully adhered to by modeling tests based on the guidelines provided by Grand Strand Systems. This began with designing a structure that would support tests. This is shown here:



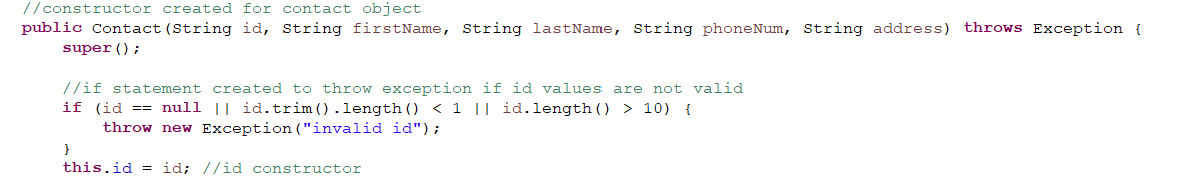
Following this structure allowed me to efficiently create tests that mirrored the desired functionality of the actual application classes. In addition to this, I adhered to the defined guidelines by creating fields that fulfilled specific requirements such as a unique contact ID string that “cannot be longer than 10 characters [and] not be null and shall not be updateable.” I tested this by first creating csv sources which contained invalid information such as this:



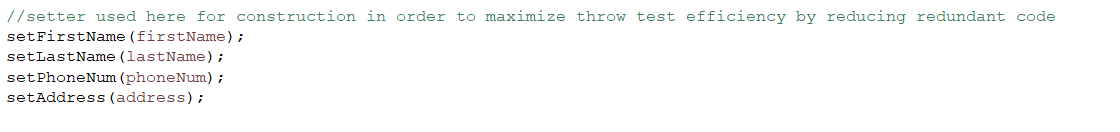
I then completed Junit parameterized tests to validate correct exceptions thrown by the invalid inputs:



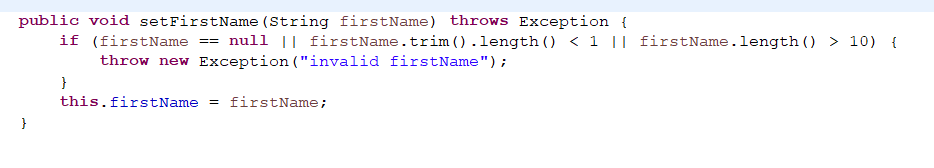
Exceptions were thrown based off of code which was set to throw an exception when invalid fields were attempted such as this example:



Tests were also completed to verify getters and setters were functioning correctly and only legal changes would be allowed. The same thrown exceptions and parameterized tests were used to validate each of these. Junit parameterized tests were preferrable to try catch clauses because they more accurately follow the test-driven approach and keep tests contained within the appropriate classes without any need for mixing with actual application classes. Therefore, my code remained both technically sound and efficient with little repetition. I also ensured sound code by using my setters within my constructors so that the constructors could be tested using the throw clauses of the setters. Examples are given here:



A setter containing the throw clause:



With these strategies, I am assured that the quality of my code for the contact, task, and appointment services is consistent and meets or exceeds the standards given by Grand Strand Systems. The consistency with which I address every line of code also ensures over 80 percent coverage with all defined requirements and testable quantities addressed including changes to fields and objects and their values.

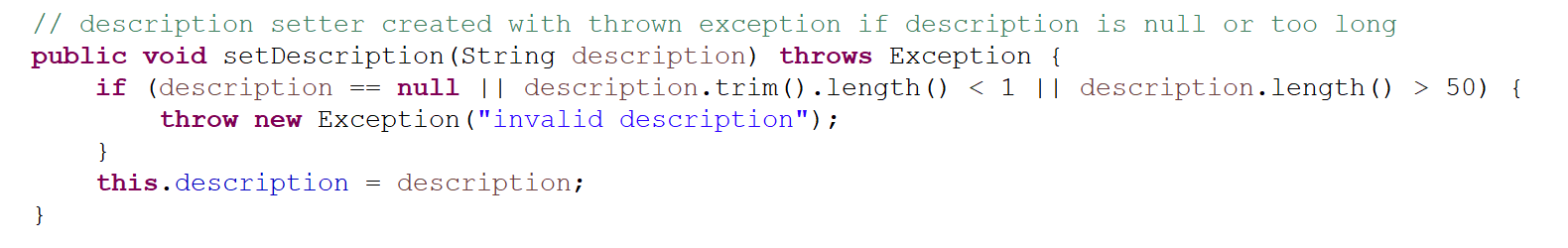
**Reflection**

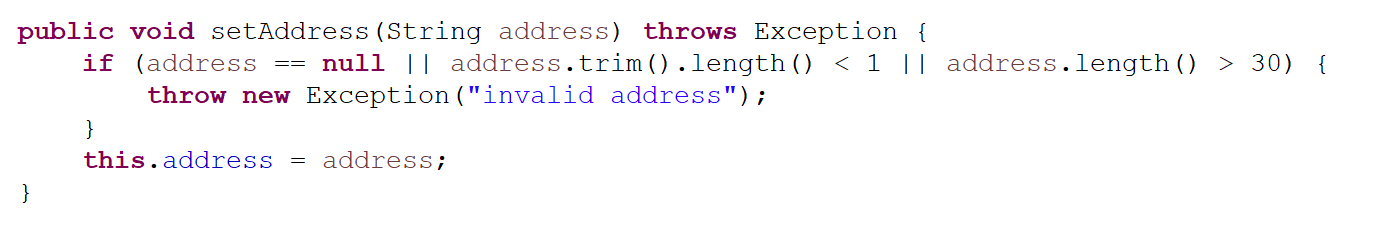
The techniques which were used in this project were defined first by a test-driven strategy, and then by object-oriented programming following the singleton pattern. The singleton pattern ensured that only one instance of each class could exist at a time. This was specifically implemented in the service classes so that validation could be completely controlled. Junit testing was the foundation of each test-driven approach with corresponding test classes for each application class. Each program class had a mirrored testing class in a separate testing package. This aligned with object-oriented programming by keeping test portable and encapsulated. Ways that the data could be made invalid were anticipated and tested to ensure that appropriate program responses occurred following Junit test annotation and parameterized tests to test all appropriate fail cases.

This project lacks techniques that align with automated testing or other high-level test such as End-To-End Testing because they exceed the necessary scope. This program is currently in its infant stages of development, for which unit testing is most appropriate. As development progresses and app complexity requires more advanced and integrated testing techniques, those would be implemented as needed.

The techniques used in this project represent the basis of practicality in code development because they are the foundation of testing in the early development process. Test-driven development guarantees that most errors that could happen within a program are caught and anticipated from the very beginning to the end of development. This reduces the chance that expensive errors or omissions will occur within the program, potentially interfering with the development process. Test-Driven development also provides an opportunity to test programs and ensure that they actually meet the client’s needs. In addition to this, Object-oriented programming benefit the practicality of the development process by making code more structured and portable. This is why it was easy for me to implement the same Junit testing techniques across each service, because the consistency and structure meant that as few changes as possible were necessary when reimplementing code to match the new classes’ needs. Both of the techniques are part of the popular Agile methodology, which encourages the incremental style of programming that this project followed.

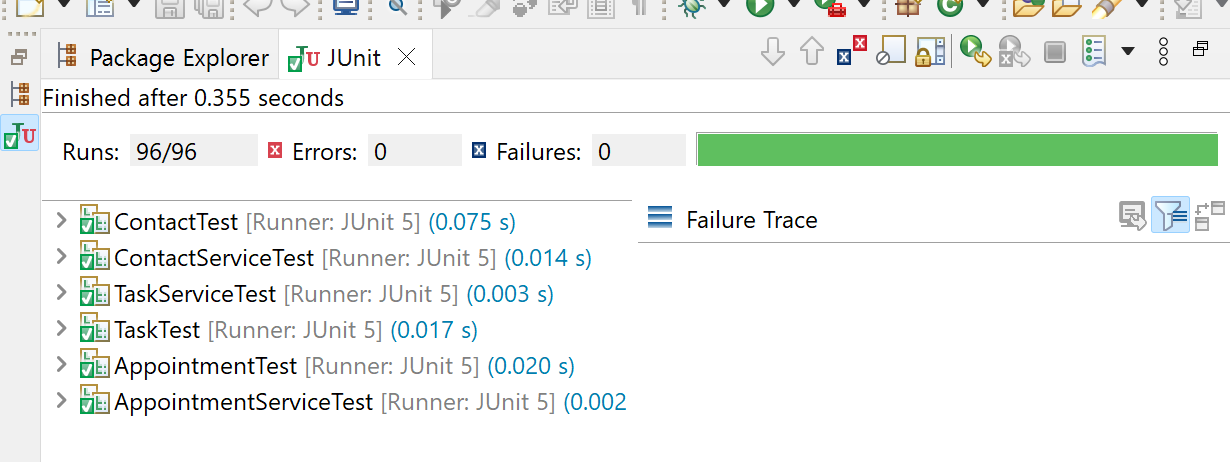
The primary mindset that was adopted in the development of this project was that of consistency and caution. Once a successful testing strategy was employed, this strategy was mirrored in each of the subsequent classes and services as additional features were added in the app. For example, it can be seen here that the same strategy was used to validate a setter in both the contact and appointment classes:





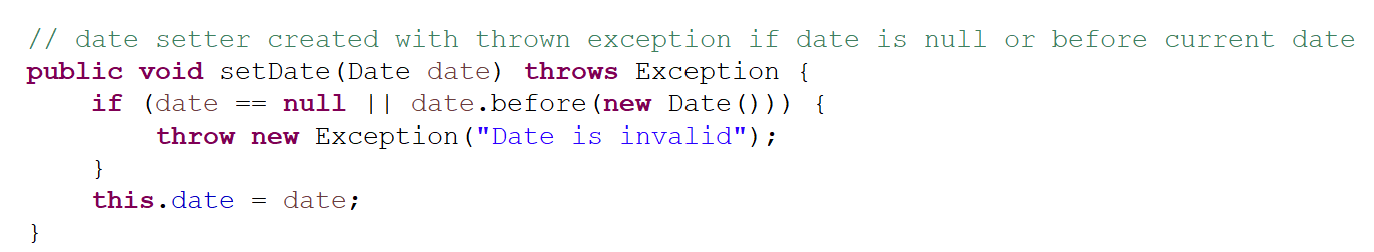
It can be clearly seen that the code for each class is very similar. This consistency is how testing and validation can be maximized in terms of efficiency and meeting the needs of the program. Attention to detail was given to ensure that every Junit test followed the same strategy and standards throughout all of the code of the project.

Bias in the review of code was relatively easy to prevent, because Junit tests have one of two outcomes: pass or fail. To prevent any bias on the coverage of my code, I tested every line of code in the classes, and only accepted the results once all test passed, such as what is shown here where I run a test on every class in the project and all tests pass:

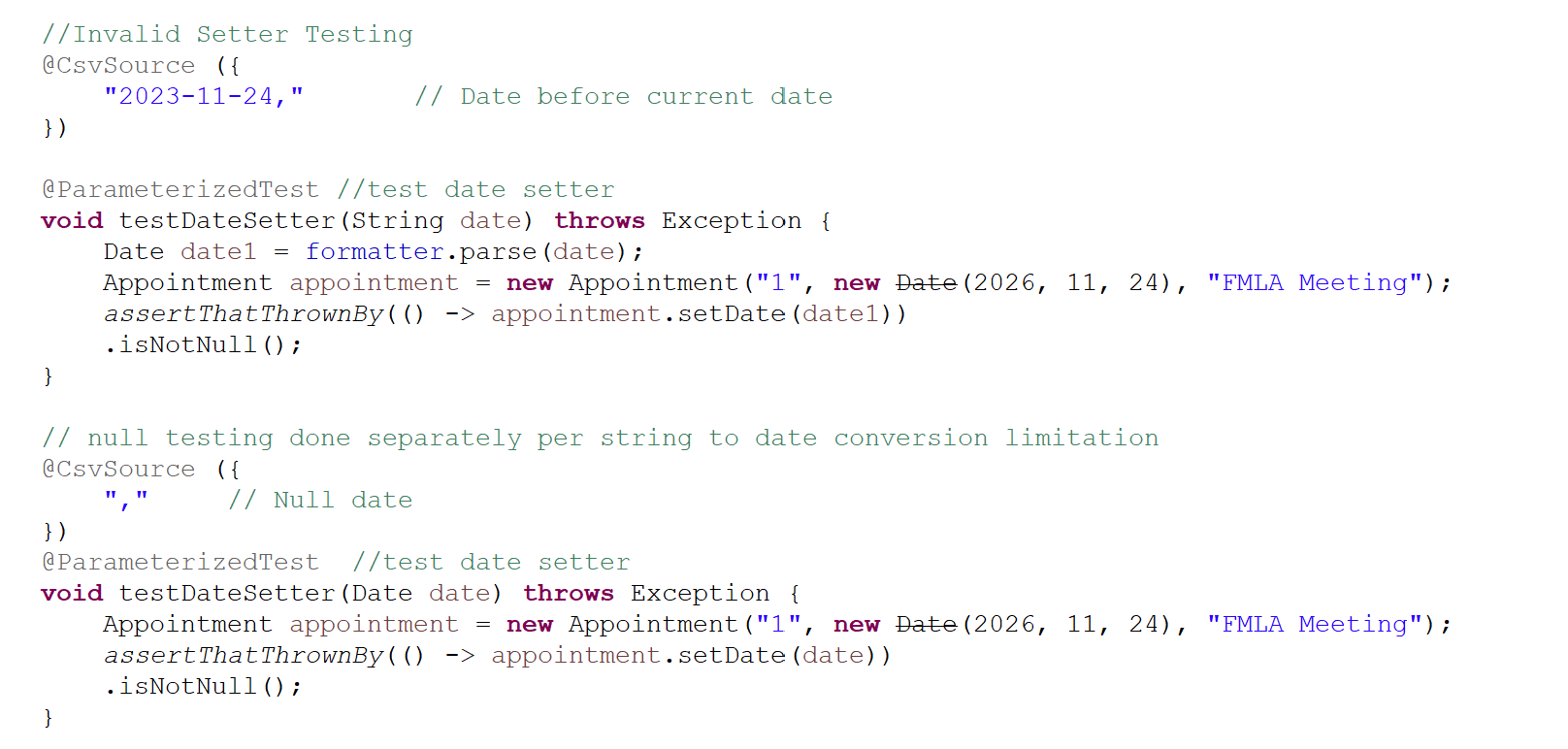


By adhering to the strict results of testing, I did not have to interpret results with how I felt about them. A failed test simply meant that additional corrections needed to be given, and I could ensure that I did not fail to meet the needs of the program.

This strategy also ties into the discipline that was adhered to in this project. As a software engineering professional, it is my responsibility to ensure that all requirements are met, and all foreseeable failure cases are accounted for. Rather than cutting corners in conducting tests, I made sure to adapt code to ensure that it tested every field. A good example was the testing of the date field in the appointment class. This required a unique approach to complete successfully Junit tests that were more separated than that of the other classes and required more attention to detail. This is shown here:



Each test had to completed separately:



I could have cut corners and failed to test the data setter thoroughly, but that would have left a major potential gap in the test coverage of the project. Cutting corners like that is a good way for programmers to ensure the failure of their program, and that is not acceptable. Thank you for choosing me to participate in the analysis and testing for your program.

Resources:

GeeksforGeeks. (2024, October 4). *Abstraction in java*. https://www.geeksforgeeks.org/abstraction-in-java-2/

Brian Hambling, Peter Morgan, Angelina Samaroo, Geoff Thompson, & Peter Williams. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide - 4th edition*. British Computer Society.