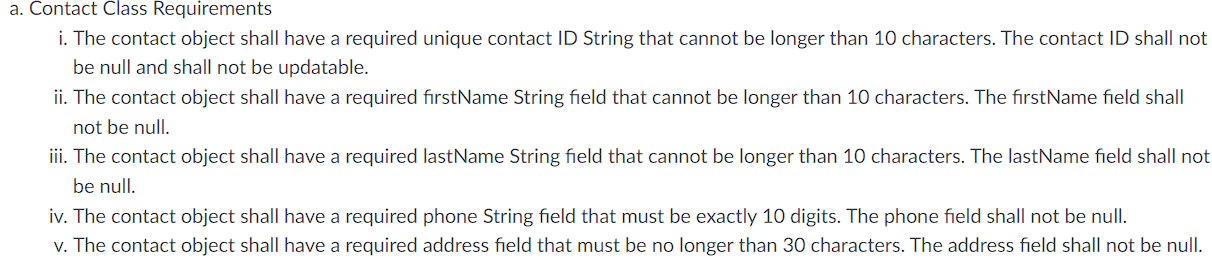
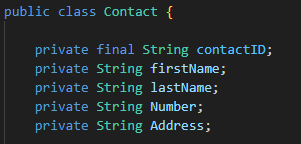
Nicholas Nevins  
CS-320

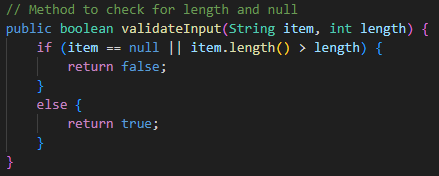
Project Two

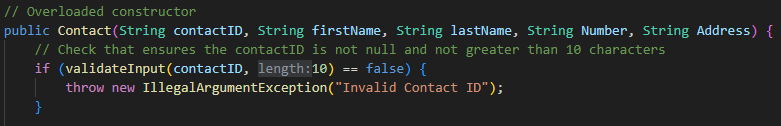
When approaching unit testing for this project, I started the same way for all three features. I began with an examination of the requirements are provided by the customer. That was always central to my design for both my code and my tests. With good, clear requirements, it was easy to build the classes needed to complete these projects. For instance, for the Contact Class, I was given the following requirements:



As you can see, we will have 5 class members. I created them as such:

Further, as I began building testing, I was sure to pay attention to the other requirements given to each class member. For instance I built validateInput to check a member’s size and if it was null:

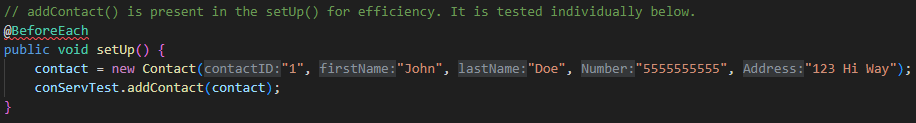


From there, I simply us that method to check length and null status like in the overloaded constructor:  


This is just one example of building to requirements. When building the actual test suite, I simply had to check to be sure that these errors I built in were throwing properly, and since I built those to the system requirements I knew that successful testing would mean that the classes were built to the customer requirements. Though I forgot to validate the constructor worked properly in my testing, everything else was fully covered. I feel strongly that my test is of good quality and fulfilled my assignment properly.

As I worked through this assignment, I came to be quite comfortable with JUnit testing. When approaching the technical side of testing, I was sure to use best practices, such as this set of tags: 

By utilizing the @Test and @Tag(“”) lines, this was a proper JUnit test by technical and best practices standards. Also from the ContactServiceTest class was this snippet:



The @BeforeEach line will save me having to make this object for every test, and will automatically take care of this step. This is one example how I ensured and focused on efficiency within my code. The aforementioned validateInput() method is another. At the suggestion of my instructor, I also changed the lists I had for each service class to a Map<>, which increases efficiency, especially as this code is inserted into more expansive projects.

I used, best as I can identify, three forms of testing; Unit testing, acceptance testing, and regression testing. Unit testing is evidenced by my tests in general, as I built tests to check that each individual unit functioned as anticipated and to requirement, which leads to acceptance testing. Since I built the tests to ensure that requirements were met by the code, this is an example of acceptance testing. Finally, regression testing occurred as I performed the others, running tests after I made changes and fixes to the code to ensure that everything I had already tested still functioned as anticipated. There were a few instances where this form of testing proved valuable as my changes broke once-working code.

I did not use integration testing, since I did not have an interface to fully perform these sorts of test. In the sense that I tested that the different units interacted properly, that is true, but I did not have an interface and so nothing was truly integrated. System testing is a near-final test to ensure that the product is deliverable. Since I did not have a full system to test, I could not run these tests. Performance testing, while import to ensure that efficient and speedy user experiences exist, were not run, since we are very early in the project and have no system together to test. Security is paramount, but I have no security features to test, so security tests were not run. Finally, compatibility testing would test that it ran across all needed operating systems and hardware configurations. This is meant for alter in the project than we are at now, and was beyond the scope of my work thus far.

It can be hard for me not to wax overly creative, even when coding. Approaching from a testing standpoint is a great exercise in learning and practicing caution. It becomes abundantly clear why streamlined, clean, and simple code is superior to anything more complex. When it was difficult to get something testing properly, it was often because I got overly complicated with it. Once I simplified, it became much easier to ensure that I had developed code that was not just testable, but that met the requirements exactly and could be insert into practically any system in order to function properly. Bias is present in any form of creation—I’ve learned that abundantly in my life—and coding is no exception. It is very easy to miss use cases that could break your code, or fatal flaws when you know what it’s supposed to accomplish and test to pass that code. I find that limiting bias is best accomplished for me by focusing closely on requirements, especially if they are provided by an outside client. If the code does only what it is required and nothing more, then focusing on those requirements should help to keep outside of yourself and avoid a bias allowing you to miss a vital flaw in your code.

Discipline, I have come to learn, is a non-negotiable trait in a developer. Code runs the world, from finance to war to the way we buy groceries and how we find romantic partners. It is vital that developers understand that testing is not an area to slack. Mistakes can happen, and there can be use cases that are impossible to foresee, but how can we sleep at night if we don’t do all we can to ensure that our code does not end a life or cause financial or physical damage? It’s important to remember that missed deadlines and exceeding a budget is far less important than keeping your product form negatively impacting a life. I intend to keep that closely in mind as I continue into my career.