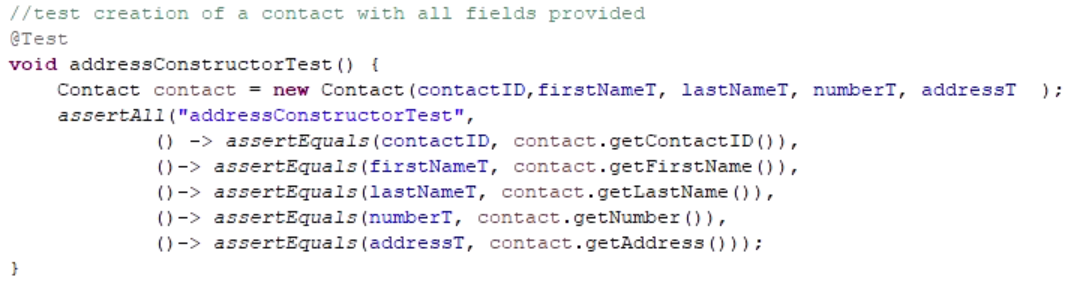
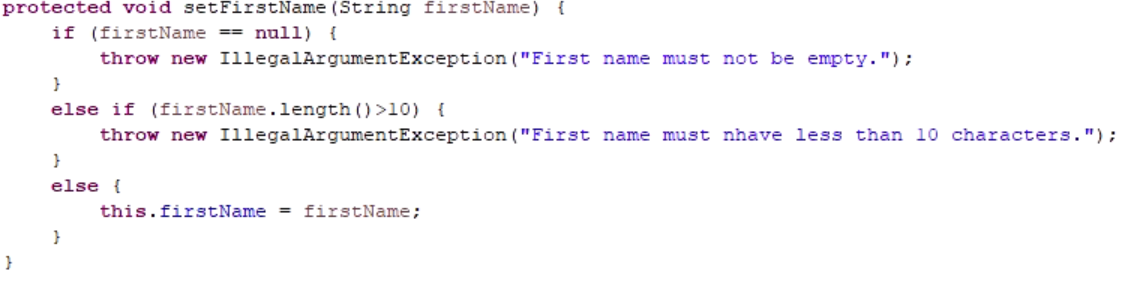
Identifying the requirements of the software was the first step in my approach to both creating the necessary classes for the assignment and creating the test. Once I had identified those requirements, I created the Contact and Contact Service classes with each class following the assignment's requirements.

After identifying the requirements, I used those requirements as the basis of both my classes and test classes. By following concepts, I read about in the class reading, and some additional reading and videos online, I believe that my test coverage is very high. While I intend to continue studying these concepts and read more about testing outside of this class, I am confident in my understanding of these concepts for the context of the class.

The following code illustrates a constructor class that tests a contact with all the contact variables filled in. This constructor test is the final of several tests, each one checking the constructor with different variables filled in. This test shows the culmination of all these tests and illustrates the soundness of my code. The use of similar tests in the project illustrates the coverage of my tests.

  
 The following code shows a setter with multiple exceptions rather than using multiple distinct functions. I chose to use a single setter with exceptions. I used this strategy with all the variable setters, having one function covering all possibilities within the requirements.



I primarily employed static testing, testing done without executing code (Hambling, et. all, 2015), to reach the milestones goals. Primarily I relied on requirements analysis and review, code reviews, as well as unit tests, tests focused on the smallest possible code segments (Hambling, et. all, 2015), and integration tests, tests focused on how code components interact (Hambling, et. all, 2015), to ensure I was meeting the goals. Early on I did a lot of secondary research to figure out exactly how to write JUnit tests. As I created the code being tested, technically I was engaging in white-box testing, testing where the tester is familiar with the code. (Hambling, et. all, 2015) I did some dynamic testing, testing where code is being executed (Hambling, et. all, 2015), via the use of unit and integration tests, making sure the code ran, but my goal was to create code that ran and pass my tests the first time. I did run more dynamic testing once I followed up on my code for Project One to ensure every test was succeeding its intended goal. I was most successful once I figured out how to write JUnit tests.

I did not employ black-box testing, testing where the tester is not familiar with the code (Hambling, et. all, 2015), as that would have been impossible. I also did not do any system testing, tests focused on the entire software application (Hambling, et. all, 2015), or acceptance testing, testing from the perspective of the end user. (Hambling, et. all, 2015)

For an actual software application, each type of testing has its place and goals. Static tests can help catch errors and bugs before any failed dynamic testing. Code review and requirements review can help guide a developer in creating the correct code components and ensure they are writing quality code. Dynamic testing helps ensure that each level of code is functioning as well interacting code components are fulfilling their purpose both individually, through unit testing, and together, through integration testing. White box testing is fundamental to software development, as a developer should be able to create quality code and be able to prove that quality through tests. Black-box testing helps ensure that the software and its code is performing as expected by the end user in addition to the intentions of the software developer. System and acceptance testing both help ensure that the completed product is performing correctly as a total package.

In the project, I focused on what I assumed would be the same mindset as a professional programmer and then as a tester, follow the requirements. After examining the requirements, I built the code all around those requirements. When writing tests, I used that same list of requirements, while also further brainstorming on what would meet or fail those requirements. Outside of the guidelines of the requirements, I do not believe I followed much caution. I did a lot of research and tried several different tests when I began writing tests before coming to the strategy I followed for the rest of my tests.

Avoiding bias is an absolute must when approaching any project. But I must acknowledge that bias, especially implicit bias, is challenging to avoid. And, once again, I returned to my analysis of the requirements to avoid bias. Every line of code and every function was examined with the idea of, is this following the requirements, and what does this add to the project in following requirements? My checklist of requirements and my notes on those requirements guided everything.

Cutting corners means that requirements are not being met. Smart programming means learning to identify what code should use new unique code or when to implement the concept of polymorphism. Cutting corners can be a result of poor work ethics and/or poor focus. Or it can also be from improper resources, including and especially time, and poor support from an employer or team. Technical debt itself is a concept I have some understanding of the concept, but honestly unsure of practical experience or practical understanding of it. It is one of the concepts on my list to further research and explore.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition). BCS, The Chartered Institute for IT.