The testing approach I employed for Project One was aligned to the software requirements because the Junit tests properly threw the expected exceptions when improper arguments were entered while the tests otherwise did not throw any exceptions. I started by creating a test contact that met all the requirements that were set forth in the assignment. I then conducted tests to make sure that the code I had written, in fact, recognized the test contact as having met all of the requirements. After that, individual tests were run for each parameter individually to make sure that the code would recognize when one of them did not fit the requirements. For example, when the phone number parameter was entered as only having seven digits instead of the required ten digits, the test would correctly throw the expected exception.

The best way I have found to be technically sound, in my limited experience with writing code, is to be very careful about syntax and punctuation and to be specific without over-complicating the problem. Trying to do too many things with one command can make code look messy and can sometimes be difficult to keep straight. However, always using a different command for each individual task can make code much longer and more arduous to read than it needs to be. For example, in Contact.java, the lines

if (address == null || address.length() > 30) {

throw new IllegalArgumentException("Invalid address");

}

lay out two requirements for one parameter, but there are four other parameters that are written in a similar way. Each parameter has its own lines for exception handling, but they each check more than one thing at a time, such as the parameter length and making sure the parameter is not null.

Writing the code in this way made sure that the code was technically sound by keeping each line easy to understand while simultaneously being efficient with the code by not using significantly more words or space than were required to accomplish the task.

Several different types of testing techniques were utilized during the module three, four, and five milestones. One example of such a testing technique is the black-box technique of equivalence partitioning. Black-box testing is when the code is tested for desired outcomes without having to worry about how the code arrived at those outcomes. The modules all had classes that assigned string values to different variables that all needed to be within a specified range of characters. Equivalence partitioning is useful in testing to make sure the program responds properly if the variable is assigned a string that is over the maximum character allowance. Boundary value analysis could also be utilized to make sure that strings that are exactly the maximum length allowed are still considered acceptable by the program.

Project One did not really require utilizing white-box testing, such as statement testing or decision testing. White-box testing is when the code is tested to make sure that specific parts of the code execute in the desired fashion. I never had to utilize any logic, such as if statements or while loops, while writing the code for these modules. Therefore, test cases did not need to be constructed to check that each possible outcome of any logic statements were executed as expected. Ideally, each branch of an If statement would be tested to make sure that it would execute with the proper input.

The different testing techniques have their places within different projects, with some projects leaning more heavily on certain techniques and utilizing others less frequently. A project, such as Project One, that deals with a lot of input would need a lot of equivalence partitioning and boundary value analysis, but it may not require statement testing because it doesn’t involve logic statements. A different project may rely heavily on logic statements, and therefore require a great deal of statement and decision testing to ensure that the logic statements all use the correct branches. The important thing as a developer is to know the different types of testing techniques, how to apply each of them, and to recognize when they should be utilized on any given project.

The mindset I tried to adopt when I set out to write this program and the testing associated with it was simply to write the best code that I am capable of producing. Having said that, I am still a novice at writing code, so the best I am capable of is not as high quality as someone with years of experience. I always employ a sense of caution while coding in order to make sure that I don’t make careless mistakes. The two most common mistakes I make are syntax errors and not using the correct variable names within functions, and these errors are both very avoidable if I slow down and pay more attention to what I am doing.

I can imagine that for a seasoned veteran developer it could be upsetting or annoying to have someone else look through your code for a review and point out all of the things they feel you did incorrectly. With experience comes a familiarity with performing specific tasks a certain way that feels comfortable to you, and for someone else to tell you that your way of doing it is not the best way could feel like they are saying you have been doing it wrong for years. However, as I noted earlier, I am very inexperienced with writing code, so I am very appreciative of any constructive feedback I get regarding how I could improve my code. Therefore, I do not feel like I have any bias when reviewing code beyond that of my inexperience limiting the number of ways I can come up with to write a specific piece of code.

It is incredibly important to remain disciplined while developing software and maintain a certain level of quality because it is the morally correct thing to do. Producing a shoddy product is to shortchange and deceive those who invested in its production or who will purchase the final product. Most people have had an experience purchasing a product only to realize later on that it had some kind of problem or quickly became obsolete. It is important to keep these experiences in mind during development to ensure that you aren’t the reason someone else has that same unpleasant experience with the product you were in part responsible for creating.

References:

1. Hambling, Brian, et al. *Software Testing : An ISTQB-BCS Certified Tester Foundation guide - 4th edition*, edited by Brian Hambling, BCS Learning & Development Limited, 2019. *ProQuest Ebook Central*, https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074.