The software requirements of my program were met to their fullest extent. For each class, I went over the functionality of my code and compared it to the requirements laid out for my program to ensure all expectations were met. I reviewed every variable from every class and created a function to check the length and null requirements for each one. I made sure there was no way for an incorrect input to make it through the variable check function I had created and be assigned. I did this by including the variable check functions in each update function and within each method of adding a new object. Then, I ensured each method worked accordingly within my JUnit tests.

By the time I finished writing my code, the coverage percentage of my JUnit tests was around 70%, all put together. I wanted to increase the coverage, and when reviewing my tests, I noticed that the exception throw assertions I had integrated were yellow, meaning the code was reached but there was no output from it. It was like dead code. To increase the coverage of my tests, I added an “assertTrue” statement, so that when the compiler reached my exception throw assertions, I could verify that they were working properly by validating the statement output from the exception. For example, in my variable length assertions, the exception would output a statement that says “Maximum of X characters.”, with the value of “X” depending on the specific requirement for that variable. After I added an assert true statement following each exception throw, it not only verified that the correct output was displayed, it verified that the exception was indeed thrown. The inclusion of these statements also brought my JUnit test coverage up to a little over the target coverage of 80%.

In order to ensure my code was technically sound using the JUnit tests, I called the different functions I had created and then included assertion statements to verify that the value I had assigned to a particular object’s variable matched what was assigned. For example, I included a call to the addContact() function and included an assert true statement to ensure the values I assigned to the variables are accurate. I also verified that any incorrect values would be asserted as false. Then, for functions that would update a variable, I included assertion statements to prove that the variable was updated properly. For example, in the Contact Service class, there is an updateFirstName() function. In my JUnit tests, I called the function and then asserted that the updated variable was true to the assigned variable.

I ensured that my code was efficient by avoiding dead code and unnecessary repetition. For example, in the set accessor functions for object variables, I included the variable check functions I had created. This meant it was unnecessary for me to include the variable checks within the update functions throughout my program. An example of this is the setFirstName() function in the Contact class. This function includes the firstNameCheck() function, which is also in the Contact class. In the Contact Service class, when the updateFirstName() function is called, since the setFirstName() function is called, there is no need for me to include an additional call to the firstNameCheck() function. I also omitted any set and get accessors that would never be used from the program, since their inclusion wasn’t wholly necessary.

I employed a few software testing techniques, consisting of both static and dynamic methods. The static technique I used involved looking over the code manually to make sure that my program would function logically and there was no redundant or dead code. I did not need to run the code with this method, and dynamic testing would not have made these issues apparent in the same way. The dynamic methods I used included JUnit tests and testing using Main(). The JUnit tests were helpful for ensuring the methods worked as intended, and that exceptions were thrown when they needed to be. Testing using Main() helped me troubleshoot the in-memory data structure I implemented and get it fully functional. These techniques required me to run the code, and the output from these tests helped me determine what changes needed to be made. I did not, however, use any review testing when creating my code. Review testing is a type of static testing technique where the author(s) seek formal or informal peer review to look over their code and find possible defects. Instead, I stuck to other static and dynamic testing techniques.

Static testing is useful for finding defects in the code before it is complete. When creating a program, it cannot always be executed at regular intervals to ensure functionality because it is not finished yet. Static testing solves this problem by reviewing the code and following the logic process that has been implemented to find any present defects. In my case, I looked over my code to find what code I could cut out and how I could make it more efficient. This is an issue that cannot be solved through dynamic testing alone, since this kind of flaw wouldn’t create a compiler issue. Dynamic testing is the opposite, and involves executing the code in order to find faults. This can be done earlier, depending on the complexity of the program, or at the end to ensure everything works as intended. Dynamic testing includes the utilization of tools such as JUnit tests to confirm that the code not only works, but that all possible flaws are covered in testing. I used JUnit tests in my testing process as well. Review testing is a form of static testing, but it involves finding a peer to either formally or informally review your code and designs for any defects. This can be done towards the beginning of the SDLC to make sure that the code design is sound and meets the requirements. This can save time later on in the process.

When testing my software, I wanted to ensure that there were no holes or vulnerabilities as far as incorrect input goes. I made sure that the requirements for the variables within the different objects would be preserved regardless of whether an object was created through a constructor or through a method, or if an object’s variable was updated. This is why I included the field checks within the set accessors, the constructor, the addObject() functions and the update functions. The oversight to ensure that the requirements are preserved from all aspects of the program is important, especially because it can not only help your program be complete but thorough testing can be conducted from those angles.

Bias can be a factor when testing your own code, especially if you want to avoid tweaking a weaker aspect of the code or if you are too confident in your code. I avoided bias by facing the issues my code had and fixing any issues that were brought to light by the JUnit tests. For example, I had a lot of trouble implementing the in-memory data structure. Instead of giving up on it, scrapping it and leaving the program without a way to store the objects for comparison, I buckled down and messed with it until it finally worked.

Being disciplined as a software engineer is an important quality to have. Cutting corners and being lazy with your practice will negatively affect your code and reflect badly on you as a professional. If your code is sloppy, it may affect your ability to keep your job or find another in the future. I intend to avoid this by keeping myself up-to-date in the latest techniques in the field and by employing best practices when creating and testing my code. This includes avoiding redundant and dead code, making the program efficient, and testing the code from as many angles as possible.