My testing approach was very aligned, and specifically tailored, to the software requirements. My strategy for designing my tests was to basically go through each requirement one-by-one and create a test that would verify the requirement in question was implemented properly. For example, one of the requirements for the Contact Service was that the Contact Service should have the ability to maintain a collection of contacts that it can add and remove from. In order to test this, I created functions named testAddContact and testRemoveContact that would attempt to, respectively, add and remove a contact from the collection and then afterwards would make sure that the resulting collection was one contact larger, or smaller, than it was prior to the function call.

The Junit tests that I created for the Contact Service and Task Service were of high quality. I believe that the coverage tests support my claim. For example, the tests for the Contact Service had a coverage score of at least 76% for each of the files that contained code that the test suite was designed to test. The only files that fell below this score, having a score of zero, were the files that had no code that the test suite intended to test.

As I mentioned earlier, my goals for writing the tests were that every requirement that was specified would be covered and verified by at least one test. This strategy is also the way I ensured that the code was technically sound. For each of the technical requirements, I made sure that there was a corresponding test that verified the expected behavior was present in that aspect of the software. For example, it is specified that the Last Name, among other fields, must be less than a certain number of characters. This is the exact technical requirement that I test in the function testSetLastName2. I create a contact, then I try to update that contact with a last name that contains an invalid number of characters and then I verify that the name has not been updated, i.e. the system rejected the update due to the invalid name length.

When creating this software, I made sure both my code design and my programming strategy were efficient as possible. In regards to the code design, I tried to make this efficient by making sure the program didn’t need to recompute calculations/information and would instead keep track of it. For example, the requirements specify that each ContactID in the Contact Service’s collection of contacts should be unique. One way to accomplish this would be count how many contacts already exist in the collection before adding a new one, this way we would know what the next ID should be. This, however, is not the most efficient way of implementing this requirement because of the fact that all of the contacts must be counted on every addition. A much more efficient way to do this, which is the strategy I ended up going with, is to simply have a variable that keeps track of the next ID to be used upon the addition of a contact. This variable will simply be incremented after each addition, which is a much less processor-intensive task than counting all contacts in the collection. One way I made sure my coding strategy was efficient was by reusing my functions as often as possible, rather than manually coding up a certain routine every time the routine was needed. For example, setting a first name is something that can happen in at least two different scenarios. The first scenario this may need to happen is when the Contact object is created. The second time the may need to happen is when the system is attempting to edit the first name of a Contact object. Since both of these scenarios use the same routine, I created a function that will handle this routine and will be called in both scenarios, this improves programming efficiency in that if I ever wanted to make a change to the routine, I only need to update one location in the code rather than updating several.

The most important software testing technique I utilized for the milestones was the practice of making sure each and every project requirement had a corresponding test to verify the behavior of the program is as expected. When employing this technique, one simply has to write a unit test for each desired feature. A unit test is a small function whose sole purpose for existing is to demonstrate the behavior of one and only one (hence the name “unit”) key feature of the project. These unit tests can be combined with a framework called Junit which facilitates both the execution and analysis of these unit tests. I also employed the use of this framework in my testing process for the milestones. For example, one of the requirements specified that the description field of the Appointment object should not exceed 50 characters. In order to verify the correct behavior of the program in this regard, I created a function that created an Appointment object and attempted to set the description to a value that exceeded 50 characters. Afterwards, I confirmed the update was rejected by retrieving the current description and checking if it was still equal to the value it held at the beginning of the function.

The most important/useful technique that I did not use for the milestones is the debugger. The debugger is an incredibly powerful tool and arguably the most useful tool a programmer has at their disposal. Basically, it allows one to run their program execution as normal, but with the added option to pause the execution at any point during the program’s run and inspect the value of variables and data structures. For example, I could have used the debugger to step through the constructors of the Contact, Task, and Appointment classes to observe the control flow moving in the correct directions and that the correct decisions are being made at each conditional statement.

Unit testing is a good strategy because of the fact that trying to test too much all at once can be very cumbersome. The goal of unit testing is to breakdown and organize testing development into small and easily manageable cases that only deal with a single aspect of the program at a time. The combination of the unit testing strategy with the framework Junit allows the developer/tester conducting the tests to easily keep track of which features are working and which features are not. Junit will even provide a “score” or percentage that will, based on the coverage of the tests, indicate to the programmer how much of their code is working as expected. The increased efficiency and organization gained for using this testing technique will transfer to other areas of the development process like quality, forward compatibility, and meeting deadlines.

Using the debugger is also a very use strategy because it allows one to slow down the execution of the program to a level on par with human comprehension. This is a game-changer in regards to the way a programmer can interact with their programs and especially their tests. The rationale behind this statement is that speed is the biggest double-edged sword of the field of programming. Things happen very quickly; and this is a good thing when things are working as desired, however it can be very frustrating and difficult to find the problem when things aren’t. With the help of the debugger, one can pause execution at the area of interest and take a closer inspection to the innerworkings of the program during the specified use case. This ability further builds upon the unit testing strategy’s implications of better project development and overall improved business.

When working with a large system, components will inevitably get interconnected. This is the main reason why it is important to employ caution when testing and fixing code. The solution to one error can be the genesis of another error. I employed great caution to avoid this common issue happening. Another reason it is important to be cautions when testing is bias. When the tester is the original developer of the code, they can fall into a bias trap due to the fact they know what the code is *supposed* to do, as a result they subconsciously assume that a portion of code is correct rather than taking the time to comprehensively verify the truth of the matter. The importance of being disciplined in creating quality software cannot be overstated. It can be frustratingly tedious to create and carry out comprehensively exhaustive tests, especially when testing a big system. This, however, it preferable to the inevitable consequences of cutting corners: bad code. Bad code leads to bad software which leads to unhappy clients and a ruined business reputation. In order to avoid technical debt, I will maintain a checklist of aspects of the program that need to be tested; this list will take the form of Junit tests. I’ll run the tests after any major changes to make sure that my new additions don’t mess up old code. This is how I plant to prevent old mistakes from building up as they will just have to be fixed eventually. The issue is further exacerbated by the fact that the fix would most likely happen during a time in which the context of the code segment in question is not as fresh in my head as it was during the genesis of the bug.