Final Project Summary and Reflection

My unit testing approach for this project was very effective at testing my code to see if it met the customers demands. I built unit tests to target any logic I employed, and in many cases tested to see if an incorrect value would be accepted by the program. For example, on line twelve of my AppointmentServiceTest class I built a test that specifically tries to create an appointment with a date in the past and an appointment with too long of a description. I also sent through an appointment with a date in the future which should be accepted by the program, and I assert that the size of my map, which I had been using to store data, is one. I then create a second valid appointment and assert that the second appointment exists in the map before iterating through the map to check that the program I wrote to randomize appointment ids was working correctly. Finally, I targeted the last function in my class and deleted one of the appointments, leaving the appointment map with just one appointment. By writing this code for testing I was able to cover all the logic in my AppointmentService class and ensure a code coverage level of one hundred percent for this class. This level of coding helps to ensure my code is technically sound and will work as intended every time the code is utilized.

The testing method I utilized for this project was all unit testing which is testing to ensure the code is functional and works as it was designed to. This type of testing helps ensure the program hits all the customer requirements because it is easy to build tests specifically for those requirements. Unit testing is typically the first level of testing and is done by the development team before the program is sent to the testing team to perform other tests. If done correctly this can help the development team ensure the program they are sending is at least functional, because nothing is more embarrassing than sending over a program that is supposed to do something and having someone send it back because it does not even meet the required functionality.

I only used unit testing but there are quite a few more testing techniques. Developers and those close to the development team that have some knowledge of the underlying code perform white box testing methods which include unit testing, integration testing, system testing, and acceptance testing. Unit testing is testing done on small programs individually before they are combined into a larger program, which then becomes integration testing. Integration testing is similar, but it ensures a group of smaller programs work correctly together to ensure the entire section behaves the way it is supposed to. System testing is another step up, when all the integration testing is done system testing starts which is typically white box testing of the entire system to see if the program performs the way it is supposed to with each piece working in tandem. A simple developer way to explain this is that unit testing is testing of a class, integration testing is the testing of a module, and system testing is the testing of the entire program. While these versions of testing can be black box it has been my personal experience that this testing is usually done white box, although I could see some larger companies separating the programming of tests to programmers who did not write the initial code.

Black box testing is when the testing moves out of the hands of the people in or close to the development team and into the hands of people who do not understand or know the code. This is a very important phase of testing because, as any developer will tell you, users do not always use the program the way the developers designed it and that includes the testers. For a bit of anecdotal experience, I just recently had a project manager testing a function that downloads data from an Android analyzer to a desktop program consistently use the download to new directory option rather than the download to existing directory option because he thought if there was new data he was downloading he should use the new option. In this case it did not matter, the new function and the existing function were both checking if the directory existed and confirming that the user wished to overwrite the existing data so the only difference was the existing function came up with the name of all existing directories and allowed him to choose, the new function just had a default name. This example shows that sometimes you must program for the unexpected because users can do unexpected things. In my experience acceptance testing usually falls in this category and is almost always black box, done by someone who understands the product but has little to no idea how it is coded. Testing beyond this includes performance testing, security testing, usability testing, and compatibility testing which all test how software might work on different environments. These tests are usually all done by those designated to work with pre-release versions of the code and are typically a pain in the butt if any flaws are found at this stage because the development team must figure out how to fix it without creating additional issues.

Since this was a very small program I just simply tested for the logic in my program and left my testing and programming as simple as possible. For low level functions this is preferable, it is best of functions stick to just doing one thing to make it easy for development to build on top of them. However, in my own professional experience I usually try to write my code to do both what I need the code to do and what I do not want the program to do because odds are good someone is going to try to get the program to do something I did not expect when writing the code. I have found that sometimes code can be structurally sound in unit testing but can be broken when another piece of the program is added to the equation. For instance, I recently added a warning popup to let a user know if something went wrong when a calculation was being done on a piece of the program we use and how to fix it. On the other side of the screen, we had a spinner attached to a change event that sent a call to a listener which ran this code and the spinner, if the mouse was pressed, could call this piece of code hundreds of times a second. At that speed all limits I put into my program for this popup to function would break and the popup would continue to show up even when it was not supposed to, eventually freezing the application. Therefor it is very important to understand how different parts of the program interact and what might happen when the user starts messing with the program.

Limiting bias while writing test cases is difficult if you also wrote the code. It is important to try to think about how others might view your program and what they might try using it for or how they are used to the program functioning and might try to utilize in the same way in the future. It is nearly impossible to capture all possible scenarios, but it is important to test for as many as a user possibly can. This also runs into a developers discipline while programming, well disciplined programmers will program for the end user and attempt to capture all possible scenarios while poorly disciplined programmers sometimes do not even test their code for the required functionality, assuming it to just work correctly based on what they have written. I have also found that it is good practice to code things as simple as possible. The less code a developer writes to fix an issue the less potential bugs that programmer might have to deal with. I do not mean using techniques like a ternary instead of if statements or lambda instead of writing out an action listener is the way to go because they are less code, functionally they are the same. I mean if one if statement with one expression will do, do not write two.