My approach was aligned to the software requirements by me following what the guidelines were. Through each of my projects I kept the guidelines up on a second monitor and when I felt finished with one I moved on to the next. I believe I showed this by first having all 38 tests, even if one was failing ( which I missed this myself, which is a mistake on my part). Following turning in the project I did go an fix this to help discipline my self further into making sure my code is released to the customer in a complete state. I think the first week showed this best for me. I struggled to turn in some code that was even functioning. For the final week I turned in my full project that had the week one section completely recoded to make sure that it was functioning. For example, I had issues entirely with ContactTest.java, the update fields were not working at first as my code was a mess at first. I restructured my code and ended up with this: **void** testUpdateValidFields() {

Contact contact = **new** Contact("12345", "Bruce", "Wayne", "1234567890", "Wayne Manor");

contact.setFirstName("Clark");

contact.setLastName("Kent");

contact.setPhone("0987654321");

contact.setAddress("Fortress of Solitude");

*assertEquals*("Clark", contact.getFirstName());

*assertEquals*("Kent", contact.getLastName());

*assertEquals*("0987654321", contact.getPhone());

*assertEquals*("Fortress of Solitude", contact.getAddress());

}

This ended up working for me and created my UpdateValidFields portion of the code.

As I stated above my Junit tests were overall effective, I had a 37 out of 38 when the project was turned in. This results in a 97% overall cover percentage.

My Junit tests were technically sound by focusing on one before moving onto the other. I also started adding some of the comments between code to let me know what I was working on (referring to the //). This helped create code that was not too complicated like:

// test to make sure you dont add the same task twice

@Test

**public** **void** testAddDuplicateTaskId() {

Task task1 = **new** Task("901", "Task NineZeroOne", "Task 901");

Task task2 = **new** Task("901", "Task FourFiveSix", "Task 456");

taskService.addTask(task1);

*assertThrows*(IllegalArgumentException.**class**, () -> taskService.addTask(task2));

}

My code was efficient by ensuring that nothing was duplicated and focusing on the guidelines I was given. I essentially created a checklist and when I was done with one I would move on to the next Junit. I would use the @beforeEach to help me with this, creating what is a new appointment service and move on. This would help me not only test individually but also run all the tests with Junit and not have them overlap.

When I was working on the appointmentServiceTest I would do AddValidAppointment first, This would be then coded as the following:

@Test

**public** **void** testAddValidAppointment() {

Calendar calendar = Calendar.*getInstance*();

calendar.add(Calendar.***DATE***, 1);

Date futureDate = calendar.getTime();

Appointment appointment = **new** Appointment("123", futureDate, "Checkup");

service.addAppointment(appointment);

*assertEquals*(appointment, service.getAppointment("123"));

}

Before the next test the code :

@BeforeEach

**public** **void** setUp() {

service = **new** AppointmentService();

}

Would delete and then set up a new appointment so then my next set of code:

@Test

**public** **void** testAddDuplicateAppointment() {

Calendar calendar = Calendar.*getInstance*();

calendar.add(Calendar.***DATE***, 1);

Date futureDate = calendar.getTime();

Appointment appointment1 = **new** Appointment("1234", futureDate, "Checkup");

Appointment appointment2 = **new** Appointment("1234", futureDate, "Checkup 2");

service.addAppointment(appointment1);

*assertThrows*(IllegalArgumentException.**class**, () -> service.addAppointment(appointment2));

}

Would then work.

I mostly used manual testing and automated testing for all of this. The project is on the simpler side so I could rely on those two techniques for all of this. For example, when I was using manual testing, I had some errors get thrown in my Contact and contact service. My mistake was that contact and contact service were not in the same package, and the automated tools would not tell me that this was issue. The red underline would essentially underline entire blocks of code and say “ can not find the ID variable” or some other issue. I do not remember exactly what was said by the system. Obviously Unit testing was used for the entire project, as we used Junit testing through eclipse.

I did not use something like regression testing. This ensures that when you change the code that it did not introduce any new issues. I did not use performance testing either as this can focus on something like the speed of the program, as we were not really focused on the speed of the program, I never gave it any attention.

For manual and automated testing, its practical uses are really always there. You can not really do any other testing with out using these. Junit tests are already used from a tool, which is software that runs on your system. Through checking for bugs you are already looking at the code and fixing issues: which is a type of manual testing. Junit tests are extremely helpful for the end goal of the project. It’ll take information feed into the program and automatically test it to whatever code you have written. This is like testing for a null contact ID and giving the system and ID to test, which would not throw an error. Regression testing is also more practical with manual testing due to the fact that once you write new code, you are going to test and check it afterwards. Performance testing is practical in any program that needs to get results quick. This could be for something like a math related program. When fed information and you need that information as quick as possible, maybe something like calculating the trajectory of a rocket or something in space. Very practical in these situations but most well written code should be pretty quick if it is efficient.

The mindset I adopted was “throw myself at it till it works but do not be afraid to take time to ensure good code”. I try to rely on myself and what I can find out there for existing information (such as how I found out about the @beforeEach for my code). It was important to appreciate the complexity of the code because once you find the “flow” of information for the program to flow through, it kind of helps write the code itself. What I am thinking of is once you wrote one Junit test, it helped me flow into the next one by more or less testing the same exact thing but just change the variable in what I was testing (like contact id to first name to last name to number to address). It was fairly easy to test whether the fields were null or too long once I knew how to code it.

I tried to limit Bias in my review of code by reading my code as if it was not mine. As I still am very new to coding, I struggle with it a lot. It was easy to separate myself from my own code. I think if you do not limit bias it can create huge issues, where maybe you think you are better than you actually are. I primarily have been learning python, so when I would work on the website im learning python on then come to java, I would look at my code from a new stand point. It could make someone more experienced not notice small mistakes on the logic of “I would not make a mistake that small so I am not going to check my variables to make sure they all are the same FirstName” and you could realize that one was firstName or FirstName due to holding the caps key too long (this happened to me).

Being disciplined in your commitment to quality is extremely important as a software engineer because if you do something like cut corners, it could result in bad code. I live my the logic of “ old habits die hard” so creating a habit of cutting corners is not going to help me in the real world. I plan to avoid technical debt by taking additional time before I get my job to build up a resume of projects I worked on in my free time to show that my code is quality. My plan is to take an additional year or two after college to get some certifications in different coding languages on top of my degree.