The unit testing approach I used for all three features included white-box testing. It worked because I had to rewrite the majority of my assignments in the order that I had to rewrite the majority of my assignments in the order that it worked. I conducted some research to help me write the service classes and tests. The three regular classes are also undergoing changes. I made certain that I duplicated what was asked of me based on the software requirements. For instance, when developing the Appointment class, The appointment ID couldn't be more than 10 characters long and couldn't be null or updatable, according to what I was told. I created private variables, setters, and getters, as well as a function Object() [native code] to connect everything. I included an if statement in the setters that says if the input is null, it should display an error message or go past a certain number of characters, but if it isn't null, it should add the requirement. This also applied to all of the other classes I'd created.

The overall quality of the JUnit tests is probably 50%. I had not realized the importance of testing for failed tests; I had only tested the project when it worked This round of Junit test writing was a little more technically sound; I figured out how to write a slightly better test that met the requirements. I believe I could only improve on only things I could improve on testing for a failed scenario. I ensured that the code was written efficiently by writing the services differently. I had written the service as a hashmap, which I believe would have worked in this case, but I couldn't get it to work. I ended up looking at various examples of how other people wrote code, and once I had my version, writing Junit tests became much easier. For the tests, I wrote the following:

**void** testdeleteTest()

{

ContactService service = **new** ContactService(); service.newContact("1234", "Adam", "Sanderson", "123-123-4321", "8932

Bellway Dr");

service.newContact("1334", "Amy", "Millie", "122-123-4321", "8932

Anderson Dr");

service.newContact("1244", "Krystal", "Amvbers", "153-123-4321", "8932

New York Dr");

*assertEquals*(3, service.getContactList().size());

service.deleteContact("1234");

*assertEquals*(2, service.getContactList().size());

service.deleteContact("1334");

*assertEquals*(1, service.getContactList().size());

service.deleteContact("1244");

*assertEquals*(0, service.getContactList().size());

}

This test to makes sure that the contact is deleted; which works in this scenario.

**void** testNewContact()

{

Bellway Dr");

ContactService service = **new** ContactService(); service.newContact("1234", "Adam", "Sanderson", "123-123-4321", "8932

Contact contact = service.getContactList().get(0);

*assertEquals*("1234", contact.getContactID()); *assertEquals*("Adam", contact.getFirstname()); *assertEquals*("Sanderson", contact.getLastName()); *assertEquals*("123-123-4321", contact.getPhoneNum()); *assertEquals*("8932 Bellway Dr", contact.getAddress());

}

This ensures that the contact is created, which in this case is correct.

As previously stated, I wrote the tests intending to implement the intention of implementing white-box testing for correctly programming the user and vice versa. I was aware of the project's specifications and kept them in mind while developing the software. With this in mind, I tried to think about potential scenarios when developing testing. However, this was one area in which I needed to do more research.

While working on this project, I had to adopt a specific mindset. It was essentially a shift in mindset that enabled me to run tests and apply them to the code I'd written. This way of thinking was unfamiliar to me, as it is with anything new, but I eventually got the hang of it. I also realized that I worked on this project with little to no caution. Instead of making global variables public, I considered making them private. I also did not take into account the various possible values for testing. I thought about whether the objects I made passed, but I didn't look to see if they failed. I also realized that I had created extraneous objects that were not required in some of the services I had written.

"Have I written this code cleanly and without ambiguity, and have I included unnecessary things?" I believe I would be concerned with it. This has been a source of frustration for me as I've progressed in programming. I would take the long route when there was a better and more concise way to write code there was a better and more concise way to write code, I would take the long route.

Not cutting corners when writing or testing code is critically critical not to cut corners when writing or testing code because you are learning how to write it for future employers. A lack of care in programming reveals the developer's personality and work ethic. Which of these, whether intentional or unintentional, hurts a negative impact on them? I believe that practicing is the best way to avoid becoming a practitioner in the field best way to avoid becoming a practitioner in the field is to practice.