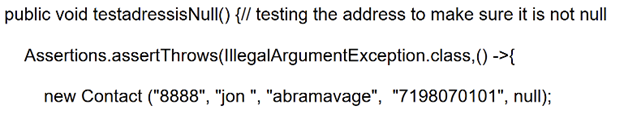
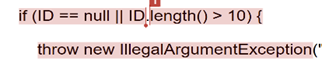
We used the Junit testing for all three assignments. ​When ​we​ look at the software we see they wanted to test certain variables for the contact test they wanted to test adding a contact. We also had to test for variables that ​were​ too long or if they had a null value.​ 

We used this code to test for that null value for the address field which was one of the software requirements. ​We also had lines of code that limited how long a field could be for example the phone number couldn’t be longer the 10 characters and had to be numbers which was another requirement.​ Other requirements we had were that the appointment id couldn't be updated and we used a final command to achieve that outcome. ​​Finally​​, we had requirements to add, delete, and change the fields in the service classes. The way we tested this was by trying to add duplicate contacts, tasks, or appointments and making an error appears. Then we had to delete one of those and try to find it. We broke these into individual test so we can easily test all the things one at a time. ​​When we look at how effective the test were we can see some were great others I missed the mark.​​ ​I feel into average on 55 percent coverage.​ When we ran the coverage report this showed what was tested and when we ran the JUnit test, we saw if the test we added passed or failed the test. ​We used this to make sure we had an effective test to little coverage and your codes not being tested.​ When we look at if the code was technically sound, we can check to see if the code we made adhered to what the requirements were. ​We also used an array list to keep track of the contacts, tasks, and the appointments this allowed for the adding and removing of these items in ​an easy​ to understand way.​ ​We also used assert equals and assert true to make the ​illegal​ argument exception this way you know what the program is looking for.​ This was the first step in making the code efficient. ​Other ways we made sure the code was efficient was using the best practice of coding this makes sure we had a reference guild that was understood in the industry that anyone could look up.​ I also used coding books for research on how lines of code should look and how to keep them short and precise. ​Another way I check for efficient was to make sure I didn’t have unused lines to keep things clean looking.​ I also had comments that explained what the code section was for. Finally, we labeled each variable before they were used so you know what it is and ​what​ it is for.

​​The experience I had with unit testing was a good learning experience.​ This style of testing allows us to run a ​coverage​ report. This shows what lines of code are being used and what ones are being tested by using color coded overlays. It also shows a percentage of how much of the code was used for testing. This was helpful as it showed what test we were missing or what ones were not working and needed improvements. ​When we look at the code below, we can see it is efficient and technically sound as ​it's​ easy to read.​ ​You can look at this and see if the ID equals null or is longer than 10 characters it will throw an illegal argument exception.​



​​We used Junit testing to create a test file using individual tests.​ We did this so we can check the output after every test. We do this by using illegal arguments and assertions that you see in the first picture. We also used static testing which is where you test the code without running the code. This looks like going line by line to make sure there are no errors and that we have all the requirements met. We also used functional testing. ​In this testing, to make sure that the code is able to run and that it meets the requirements needed.​ In this testing we also found where I had added things that were not needed. When we look at the testing’s that we didn’t ​use​ they are system testing, integration testing, and non-functional testing. ​Non-functional testing looks at the security how vulnerable it is to an outside attack.​ It also looks at how easy it is to use and how reliable the system will be over time. System testing looks at how your program affects the hardware its running on. ​It looks at will the system overheat or suffer a memory crash.​ Integration testing looks at how this program works with others and how they pass information to each other. When we look at Junit testing, we use this to find bugs early. This would be used in the early programing phase. When we look at static testing, we should use this all the time as its good practice and will reduce mistakes. ​With system testing the implications are will you need to upgrade your system to run this code this is also practical as this is a cost that need to be accounted for when looking at budgets.​ In integration testing this has a practical use to see if this program will cause your other programs to crash. ​Most companies imply this testing as they have multiple programs working together.​ ​Finally, we have non-functional and functional testing these have a practical use as they deal with what happens when the program is running.​ As ​these testing​ have a good implication as the non-functional deals with security and outside attacks and the functional testing deals with the program running without errors.

​​While acting as a tester I used caution to not make mistakes.​ It was important to look at the complexity as when I was working on the program I missed adding @Test and couldn’t get it working. It’s important to appreciate the interrelationship of the code as it helps me understand future projects as they had similarities when we moved on to the task and appointment parts. ​The ways I tried to limit my bias was by looking over the code at least one day later, so I took a fresh look and limit personal bias.​ When I asked for help, I had to get rid of the bias that I had that I’m right and they are wrong. ​When the code didn’t work, I had bias they it wasn’t my fault that I had to step away from it.​ ​Being disciplined in making quality code is fundamental to a programmer.​ As ​past​ programmers set the prescient for the way we code now, they make the early mistakes, so we run now with the rules abide by. ​Now that we are the programmers we will add our stamp to the rules of coding.​ This is why we don’t cut corners as that would set is back. ​It would also make the programmer look bad as your portfolio code that doesn’t work and doesn’t make sense to other programmers.​ It also shows poor work ethic and companies will look down on that. How I plan to avoid the debt is to take my time when making a program and getting help when I can’t manage the time. I will also make sure I and my team add extra time to project timelines so we can work efficiently. Finally, I will break down the code into chucks that I can work on.