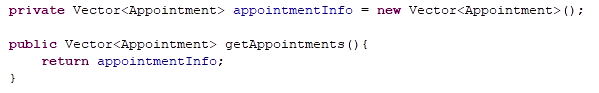
For this application, the client had very specific requirements for the variables. One example is that the IDs needed to be less than 10 characters long and unique from one another. For these requirements, I made sure that each variable was tested thoroughly by running wrong inputs and correct inputs. Also, to see if the IDs were unique, I made three different objects to test if any of their IDs were the same. Then, to verify the IDs length requirement, I made sure that an error was received if an 11-character long ID used. For the other variables’ length requirements, I used the same testing method as the ID, by inputting more than the allowed characters. I am confident that my JUnit tests are of good quality. The coverage percentages of the classes were high during the tests. I achieved these percentages by using statement coverage testing throughout building the class tests. Then, when running the JUnit tests, I made sure that every test passed without any failures or errors.

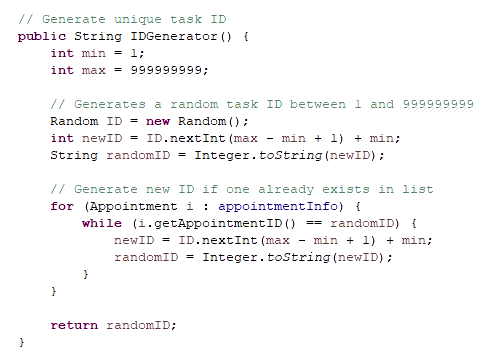
Next, looking at my code, I wanted to use a coding technique that would be easy for me to access data. So, I decided to store the objects in a vector. Then, I developed a method to store the vector in to easily access the information.



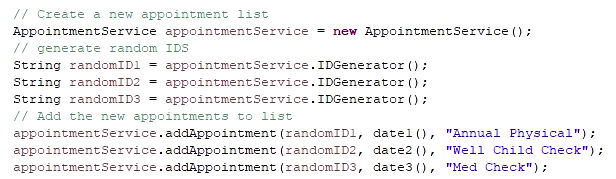
This method allowed me to pull any data that was needed for the tests by using getAppointments.elementAt()



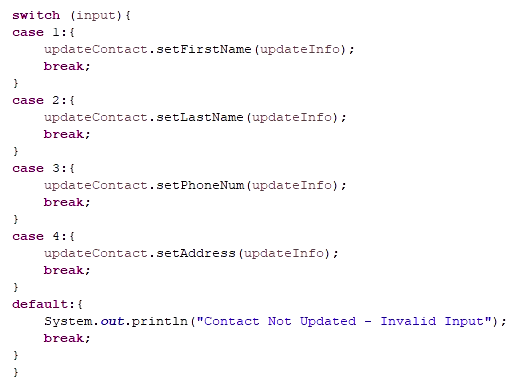
Furthermore, to develop an efficient code, I wanted to have methods that could get reused in all the classes. One of these methods was the IDGenerator. I reused this method for the Appointment Service, Contact Service, and the Task Service classes.



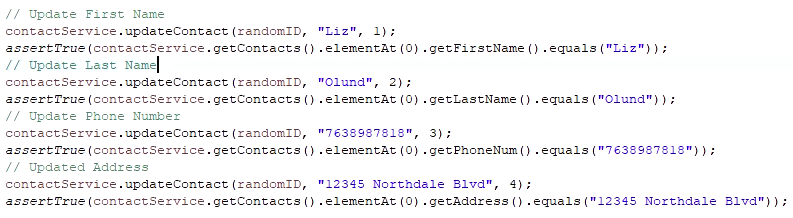
Then, this method allowed me to generate random unique IDs for objects during the tests.



Also, for the Task and Contact Service classes, I developed and reused a switch case. This function made it easy to test the updatable variable requirement.



Using the switch case to update the requirement, the method is called with the ID, updated data, and the switch case number.



As I worked on this project, I used a few testing techniques. With the needed requirements, the program had a lot of if-statements. When developing tests for these statements, I used decision tables. With the if-statements, it is important to test as many arguments as possible. For example, I will use the ID requirement if-statement if(ID == NULL || ID.length() < 10). For this line of code, four tests should run to thoroughly test this argument. These tests would be if the ID is null and the length is more than 10, if the ID is not null and the length is more than 10, if the ID is null and the length is less than 10, and if the ID is not null and the ID is less than 10. This is where decision tables were useful. They allowed me to organize what tests I should run using TRUE and FALSE statements for each argument. Also, as stated, I did a lot of statement coverage testing. This technique is a dynamic test that figures out if any lines of code were missed in testing. Furthermore, I did mostly exploratory testing for the milestones. Exploratory testing is a technique where the test cases are prepared during the testing. So, there is a limited amount of planning when doing exploratory testing. One static testing technique that I did not use was the peer-review technique. This technique allows developers and testers to look at the program and collaborate on its test cases. Unfortunately, I do not know anyone personally that could provide feedback. Another testing technique that I didn’t use is called error guessing. Error guessing is when an experienced tester can guess what errors may arise by examining the program. As much as I would love to have this ability, I do not have the programming or testing experience to do this yet.

All these testing techniques are common in professional development. First, exploratory testing is performed when the requirement documents are not fully available (“Popular Software Testing Techniques with Examples,” 2022). Without a requirement document, it is hard to plan what will need to get tested in the program. Thus, the test cases get made later, after the program's development is finished. Also, exploratory testing may get used when the project is smaller. Next, decision tables are used when the program has specific inputs that need to get verified. For these inputs, the program must act accordingly. When using decision tables, they aid in visualizing exactly what inputs should be tested. Also, statement coverage testing is an essential part of testing. It makes sure the program is fully tested. Without running statement coverage tests, lines of code with errors could get missed. Then, Error guessing is an experience-based technique, but it can prove very useful during projects. Error guessing can save time by knowing what issues to look for in a program. Finally, peer reviews are helpful in team projects. This technique allows the developers and testers to collaborate on the program's testing.

While working on this project, my mindset was to focus on having all the requirements finished and working properly. So, while testing I used caution to make sure every aspect of the requirements was tested thoroughly. Since there were two files for each piece of the project, any error in the code would have thrown off the functionality of the application. By testing the code, it was a lot easier to find any issues and it gave me an idea of where to look. So, I appreciated how testing allowed me to be confident that my application files worked properly with one another. Also, by using JUnit tests, they allowed me to develop my programs without bias. Testing limits bias by telling the developer what is wrong with their code. With testing, the developer cannot assume their application is perfect, just because it’s their code. I judged my code based on what the tests told me. Though, I can see where bias could become a concern when a developer tests their own code. For example, they could see an error in the test, but write it off as nothing. Then, that developer’s error could turn into a major issue within the application. For this reason, it is very beneficial to work with a team. Working in a team allows a developer to collaborate with others and have testers look over their code. These communications help developers to not have any bias towards their own code. Finally, as a software engineering professional, it is very important to be disciplined in providing your best quality work. By not taking short cuts in the code testing, a lot of time will be saved by not having to fix errors in the future. Also, taking the time to make sure the code works properly will provide a more secure and reliable application for the users. If anything gets missed during the testing process, an error could get released, which could cause lawsuit issues for the client and company. Also, cutting corners in development will most likely cause technical debt. To avoid any technical debt, I plan to use multiple testing techniques to make sure the application functions properly and has no errors. Also, when working on future programs, improving my ability to create quality code will aid me in avoiding any technical debt.

References:

Software testing techniques. (2021). *Geeks for Geeks*. Retrieved from [Software Testing Techniques - GeeksforGeeks](https://www.geeksforgeeks.org/software-testing-techniques/)

Popular software testing techniques with examples. (2022). *Software Testing Help*. Retrieved from [Popular Software Testing Techniques With Examples (softwaretestinghelp.com)](https://www.softwaretestinghelp.com/software-testing-techniques-2/)