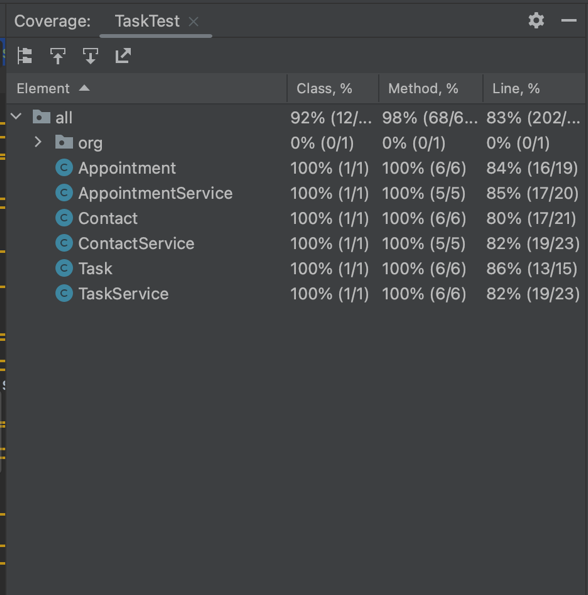
**Grand Strand Systems: Test Review**

Summary:

For this assignment, I was asked to create and test 6 different classes each needing to have details updated, deleted, edited, and searched. The first was the Contact class, complete with first and last name, address, phone number, and a unique ID as variables. This development wasn’t too complicated, as I knew I needed a data object to hold the contact classes that would be initialized. For testing, I knew that I would have to test for certain criteria that the requirements asked. I used Junit testing to test each class to ensure that the values were within limits and not null. Since there were so many variables, I tested each to ensure validity and consistency throughout the class. As you see below, I tested each to ensure the value inputted upon creation was properly saved.

@Test  
@DisplayName("Basic input test.")  
void testContactClass() {  
 Contact contact = new Contact("John", "Smith", "12345",   
 "0123456789", "123 Sesame St." );  
 Assertions.*assertTrue*(contact.getFirstName().equals("John"));  
 Assertions.*assertTrue*(contact.getLastName().equals("Smith"));  
 Assertions.*assertTrue*(contact.getPhone().equals("0123456789"));  
 Assertions.*assertTrue*(contact.getAddress().equals("123 Sesame St."));  
 Assertions.*assertTrue*(contact.getId().equals("12345"));  
}

For the Task class things were mostly the same except that there were fewer variables. Due to just having finished the Contact class, I knew exactly what to do and how to test it as well. I tested every class and method available in the classes. We can tell this by running the tests “with coverage” which tells us how much of the code has been tested.



As you can see when I run the tests with coverage it lets me know that 100% of both the class and methods have been tested. While the line percentage is not at 100%, this isn’t an issue as we don’t need to test every line of code. This ensures efficiency in my testing, and offers a great visual aid as well.

For the last class, Appointment, things were a little different. This class required a new type of data I hadn’t tested yet, Dates. When testing dates, we needed to check if it was null, if the data matched the inputted data, and if the date was in the past. Dates can be tricky because they require a specific kind of testing as shown below.

@Test  
@DisplayName("Testing for date in past")  
void pastTest() {  
 Assertions.*assertThrows*(IllegalArgumentException.class, () -> {  
 Appointment test = new Appointment("1234567890", inPast, "This is a description.");  
 });  
}

This was one of the Junit tests that checked if the data was in the past, and if it was it would throw an exception. For the above code, I intentionally put a date that was in the past when creating an appointment object, to make sure that it did correctly throw the error. Making sure that each test throughout the different classes gave us the expected result made sure that my code was both technically sound, and efficient. Each of the requirements explained what was expected, and I tested each class accordingly.

Reflection:

For this project, I employed Unit testing and acceptance testing. Unit testing is the act of testing each class separately for specific criteria. Acceptance testing is when you test for specific outputs to ensure the validity and structure are sound. For example, I tested each class separately, as well as tested them each for certain inputs/outputs that were expected. I did not use security testing because of the scope of the project and how there was no confidential data being accessed. While the address and phone number are considered personal, there was no need to test for security because there are no outside sources in my code. Also, the security will need to be tested when these classes are implemented into the application, and when that data is transferred. I also did not use performance testing, as the size of the classes built was not large, and that is usually saved for further in development process when a full product or prototype is produced.

Bias was limited while writing the code for this project because I did have someone reviewing it. Biases exist in programming when a programmer is responsible for testing their own code, and they treat it differently because of it. This can cause many issues in development with validity and with a developer's integrity. To offset this, I would spend one day programming a building the classes, and the next I would come back and write the tests. This gave me a fresh view of the program I had just written and kept me from having a bias. It is also important to have someone else review your code if possible as well.

All these techniques discussed are a result of my commitment to quality. When writing code, we not only have to consider our job and capabilities, but also the customer and client, and the data we are responsible for. The impact of poorly written code or unsecured programs can affect the public and individuals' livelihoods and should be taken seriously. It is important for a development team and companies to ensure that proper programming rules and development strategies are in place. Over the years we have learned that carelessness leads to disaster and have developed proper techniques and development lifecycles to ensure that the products we release are secure and efficient. I plan to uphold these principles throughout my career to ensure that any product I put out is safe, secure, and reflects my integrity and skills as a software developer.