**Summary**

When creating the JUnit tests for the Contact Service, Task Service, and Appointment Service programs, I tried to strictly align myself with the software requirements. I checked for invalid parameters such as ‘null’ and values that were over the maximum allowable characters for each different variable, using the ‘assertThrows’ method. I also made sure that the normal creation of objects was allowable and executed without error, using the ‘assertDoesNotThrow’ JUnit method. Beyond this, I made sure that trying to create an object using an existing ID would result in an error. The same goes for trying to update or delete via an ID that doesn’t exist. Creating tests based on the requirements was the most efficient way to make sure all restraints were enforced, and time was not wasted developing code that was unnecessary. Well-designed unit tests should cover at least 80% of code, and the JUnit tests of contactservice, taskservice, and appointmentservice covered 98%, 97%, and 98% of instructions (including 93%, 91%, and 95% of branches) respectively!

To ensure that my code was technically sound, I implemented various methods throughout TaskTest.java (which tests Task.java) and TaskServiceTest.java (which tests TaskService.java), as well as ContactTest.java (which tests Contact.java) and ContactServiceTest.java (which tests ContactService.java), and AppointmentTest (which tests Appointment.java) and AppointmentServiceTest.java (which tests AppointmentServiceTest.java):

**ContactTest.java**

tested all getters with ‘assertTrue’ (starting on line 19)

tested parameterized constructor (line 31), including normal case (line 34) and invalid parameters (staring on line 39)

**TaskTest.java**

tested all getters with ‘assertTrue’ (starting on line 25)

tested parameterized constructor (line 36), including normal case (line 39) and invalid parameters (starting on line 44)

**AppointmentTest.java**

* tested all getters with ‘assertTrue’ (starting on line 98)
* tested parameterized constructor, including normal case and invalid parameters (starting on line 44)

**ContactServiceTest.java**

implemented setUp method to create a new ContactService object before each test (line 19)

tested each method in ContactService.java (line 27, 43, 62)

tested create normal case (line 30, 48, 67)

tested invalid parameters for each relevant parameter in each method

**TaskServiceTest.java**

implemented setUp method to create a new ContactService object before each test (line 25)

tested each method in TaskService.java (line 33, 48, 66)

tested create normal case (line 36, 53, 71)

tested invalid parameters for each relevant parameter in each method

**AppointmentServiceTest.java**

* implemented setUp method to create a new AppointmentService object before each test (line 43)
* tested each method in AppointmentService.java (line 67, 92, 110)
* tested create normal case (e.g. line 71, 76)
* tested invalid parameters for each relevant parameter in each method

Efficient unit tests should typically take less than a second to complete. The terminal output for contactservice shows that tests completed in 0.094 and 0.025 seconds – a total of 0.119 seconds. Tests for taskservice completed in 0.089 and 0.016 seconds – a total of 0.105 seconds. Finally, tests for appointmentservice completed in 0.100 and 0.026 seconds – a total of 0.126 seconds. Each of these is around a tenth of a second total, which is super-fast and efficient! I made sure to take advantage of the @BeforeEach JUnit annotation to efficiently take care of setup tasks that needed to be done for each test (as seen on line 19 of ContactService.java, line 25 of TaskServiceTest.java, and line 42 of AppointmentServiceTest.java). The Maven project was built successfully, and all tests completed without error, verifying the requirements have been met!

**Reflection**

This project consisted of three services: Contact, Task, and Appointment. Each of these were used to interact with the corresponding objects (classes). Unit (component) testing was implemented using JUnit 5 (dynamic testing framework) with Oracle’s Java programming language inside of Apache NetBeans IDE (Integrated Development Environment), after statically testing the code’s syntax with the compiler/IDE. The three objects and corresponding services were quite similar in structure and behavior; thus, each subsequent class could use the previously created class as a template. For example, each of these class “pairs” used data validation for almost identical parameter restraints, such as ID (or id) and DESCRIPTION (or description), both of which had a maximum number of allowable characters and could not be null. The ID attribute could not be updated in either of these classes as well. Although, with each subsequently created class, I attempted to refactor to further optimize and simplify the code. I believe the last classes I created (AppointmentTest and AppointmentServiceTest) ended up with the “cleanest” code. For JUnit testing, the @Test annotation was used throughout the test classes (which used the format <ClassNameTest>) to indicate which methods were test methods (named with the format <testMethodName>). I also integrated the @DisplayName annotation to use a common name for the test methods, although not really necessary here. Each of my test classes also utilized the @BeforeEach JUnit annotation to run certain initializations required for each test method in the test classes. This block of code runs setup tasks before each test method, such as creating a new instance of the class to be tested and assigning intended valid/invalid parameter values and testing that these values were not null (using assertNotNull). The JUnit assertDoesNotThrow method was used to test valid parameter value assignments (positive testing) and other acceptable and predictable behaviors of the system. The assertThrows method was used to test for parameter values that are not acceptable (negative testing) and to ensure that such values are caught by the correct error handling services. For example, the AppointmentServiceTest class makes sure that dates scheduled in the past are not allowed and throw an IllegalArgumentException. Constructors (including input validation) were thoroughly tested using assertThrows and assertDoesNotThrow (for valid normal case inputs), as well as testing (limited) boundary/edge cases and other invalid inputs. Further boundary testing could have been done, using equivalence (class) partitioning and/or boundary analysis techniques, but this was not required in this situation. All getter (accessor) methods were tested using assertTrue, and all methods (Create, Update, Delete) were tested in each of the three service classes.

I have primarily used black-box testing when unit testing these classes, as most of these tests are based on business requirements and functionality (behavior) of the software and don’t really require knowledge of the internal structure of the classes. Black-box testing was utilized because this technique covered all of the listed business requirements for the software, which was primarily concerned with the inputs and outputs of the software. Non-functional testing would typically be implemented; however, these requirements were not known. Non-functional testing includes testing usability, performance and efficiency (load, speed, scalability), security (dependency checks, etc.), and other measurable attributes. White-box testing could be used to test that the internal structure and functionality of the code itself worked as intended. This includes testing conditional branches, loops, and exception handling of the code. This is a recommended supplement to the black-box testing already performed, in order to verify the program is logically handling exceptions and expectations in the correct manner. Although white-box testing was not intentionally implemented, JaCoCo shows that between 83% (TaskService.java) and 93% (ContactService.java) of all statements/branches were covered by the unit tests, with acceptable completion criteria typically around 80%! Other types of testing techniques should be integrated as the levels of testing progress from unit testing to integration testing, to system testing, and finally to acceptance testing, in which end users and stakeholders agree that the system conforms to the requirements. Retesting, regression testing, alpha and beta testing may be necessary, as well as ongoing maintenance testing after deployment. Not every testing technique should be implemented into every type of software project that is created. Unit testing, however, should be used with just about every project, since a unit can be a component or module or other type of class, which is typically created by a developer for any project. White-box testing is frequently performed during unit testing (and other test levels) of most software projects to verify the internal code structure and data flow are as intended. This should be supplemented with black-box testing in most situations to test the functionality and behavior of the software using various inputs and their expected outputs, using positive/negative testing, as well as boundary/edge testing techniques. Black-box testing of non-functional requirements are often required to test performance and efficiency with software requiring high performance (like gaming or real-time correctional systems) or those requiring robust security (such as those that manage sensitive data). I think it is important that any software planning on production deployment should incorporate alpha testing (within the dev/test environment), as well as beta testing (by external users in production environments). Retesting and regression testing may also be needed if test cases fail or changes are made to the system (respectively). It is also wise for any officially deployed software product to incorporate maintenance testing for security vulnerabilities and any new changes to the system.

Regarding this project, test coverage sufficiently met all software requirements that were defined, and so technically is could be released into production. However, in the real-world, there would be many more requirements to be defined, such as those for nonfunctional aspects of the system, like security and performance. As a software tester, caution should be applied to reach the goal of comprehensive but efficient testing. Time, money, and quality are three of the most critical constraints in software development and project management, and testing is the costliest part of the software development process. A tester should understand the code’s complexity and interrelationships to accurately test interactions between different components. Bias must also be kept in mind when testing software, as it is a common mistake to assume certain aspects of your system will work as intended and to bypass testing code thoroughly. Lack of disciplined Quality Assurance (QA) practices can lead to system errors or data corruption and continued accumulation of technical debt, leading to more challenging and time-consuming future development and a potentially damaged reputation. Frequent code refactorization, review, and testing should be implemented using a Test-Driven Development (TDD) agile approach or other similar methodology. Finding defects/failures early on will provide the most efficient process, leading to less money and time spent and the highest quality end-product to be released!

**References**:

Garcia, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing.