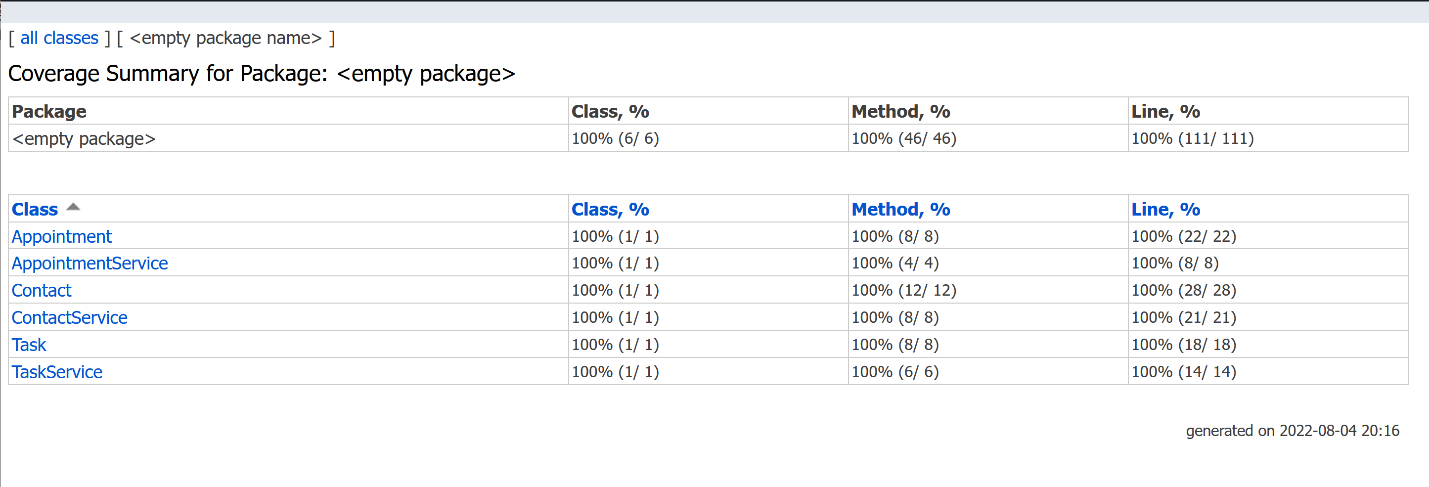
For this project we built and tested simple CRUD services and their associated domain models. Each of these services and domain models were tested using jUnit unit test to verify that the code adhered to the requirements.

**Testing Approach Summary:**

The approach to unit testing these domain models and their services was strait forward. For each feature, the requirements were broken down into one or more tests that ensured the code raised exceptions when expected and provided idempotent results when the inputs were valid. For example, the requirement “The task object shall have a required name String field that cannot be longer than 20 characters. The name field shall not be null.” Was broken down in tests that ensured a) the name field was never null and b) the name field was never longer than 20 characters.

The above describe approach to testing aligned with the provide software requirements. I can make this assertion because care was taken to test each of the provided constraints. One to show this is the unique, non-null and character length constrains for each of the domain models. For all of the domain models the associated unit tests verified those constraints. To see this in action review the method “testContactID” in ContactTest.java along with the “createTask” in TestTask.java. Furthermore, all of the service tests execute test that validate the service constraints like “add new thing” and “update existing thing” as constrained by their respective requirements.

Given the constraints and testing requirements the quality of the tests was more than sufficient. Not only was each constraint tested but they were testing in different scenarios i.e. constructing and object vs updating it via a setter. Additionally, the test coverage for every feature was 100%. This means that every line of code in the domain models and service code was tested. See the following screenshot of the test coverage report.

****

The approach taken to ensure that the code was technically sound and efficient was to doe the following:

1. Minimize repeated code.
2. Select appropriate data structures.
3. Provide addition functionality that would aide in testing beyond the basic functional requirements.

Repeated / copy and pasted code was mitigated by refactoring common logic. For example, string constraints for each of the domain models was performed by a single method. Line 30 to 37 in Task.java illustrates this approach.

Saving and accessing domain objects in the services was constrained to happen in-memory. To avoid the inefficiencies of searching a list like data structure a HashMap was selected for this purpose. Line 16 in TaskService,java shows where and how the hashmap is used.

In TaskService.java line 73 show how additional methods / functionality was added to improve the testability and useability of these features. These extra methods enable verifying that the correct number of objects were stored in memory. This is useful as a smoke or sanity test to ensure that things like delete operation remove the objects rather than concealing a logical error.

**Reflection:**

As stated above, the software the testing technique employed for the project was unit testing combined with test coverage. Unit testing is a dynamic testing method where small unites of source are executed with known inputs. The output of the test is monitored to determine if the unit of code under test returns expected results (Hambling et al, 2015). If the results are as expected the test passes otherwise it is considered a failed test. Test coverage is the process of examining how much of the source code is actually being exercised by unit tests.

The other testing techniques that can be employed to test these objects and services or are functional, integration and, static testing.

Like unit testing, functional and integration testing are dynamic testing approaches. Functional is concerned with testing that the application functionality performs per specifications (Hambling et al, 2015). Integration testing on the other hand tests how larger and more complex software behaves when it systems or components interact with external systems or with each other (Hambling et al, 2015).

Static testing, unlike dynamic testing, does not exercise source code. Static testing aims to uncover defects, security issues and compliance issues before code is ever executes. This is done via code reviews and other software that performs analysis on the submitted code (Hambling et al, 2015).

When building and testing code it is very important to maintain the correct mindset. When I was testing this project, I tried to adopt the attitude that all the code was incorrect until the unit tests proved otherwise. It was important that I understood how a change in something as simple as a getter or setter method could break the expected behavior of the services. So, I employed caution when making changes until I was sure that greater than 85% of the code for both the services and domain objects were tested.

Bias in software testing can lead one to make poor decisions when reviewing your own or others code. In particular, one can easily overlook a logic, performance or, security issue when thinking only about specifications. A great way to overcome bias in reviewing submitted code is to have multiple reviewers, static testing and a uniform policy on test coverage. For this project I limited my personal bias by writing the unit tests first and then implementing the projects second. I think this limited my personal bias well.

Finally, it is incredibly important to disciplined in you approach to building software if you want to deliver high quality products. Not consistently following established best practices, code reviews and static code analysis can lead to defects and a tarnished reputation. To avoid skipping some of these steps one should automate as much of this as possible including selecting people to review your code. Having a CI/CD system that constantly enforces your code quality and code coverage requirements helps avoid holes in your approach a makes things like technical debt less of an issue. This is because you can be reasonably confident that your tests will capture any major issues during a refactoring.

**References**:

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition). BCS The Chartered Institute for IT. Retrieved from

https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb