Ryan Grunest

August 17th, 2025

CS-320 Software Test, Automation QA

Robert Nardelli

Project Two

Based on the requirements outlined in Project 1, the tests for both the task service and the task itself fully test the requirements for each value contained within the class. In the Task class and associated tests, there are requirements that:

* The task id cannot be longer than 10 characters
* The task id cannot be null and cannot be updated
* The name is required, cannot be longer than 20 characters, and cannot be null
* The description cannot be longer than 50 characters, and cannot be null

For each of these requirements, there is an associated test to ensure that the test either succeeds or fails and that an assertion error is thrown on failure. An example is the “testTaskIdNull” test, which verifies that an assertion is thrown if a task is created with a null task id.

The tests are generally of high quality. While I’m not certain of the exact coverage percentage without a coverage tool (in node an example would be cobertura), the tests include all lines and conditionals within both the task and task service. For each public method, there is a test that checks the return value. There are tests to verify the constructor and validation methods.

I used unit tests to verify that my code was technically sound. An example would be within TaskTest.java, from lines 6 - 12. This test (testValidationTaskCreation) verifies that a user is able to successfully create a new task when entering in all required fields. It also ensures that the associated getters return the expected value that was passed into the constructor.

To ensure that my code was efficient, I only wrote assertions on values that needed to be tested within the testing methods. An example would be in TaskTest.java, from lines 14-20. The name of the test (testTaskIdNull) insinuates that the test is checking the requirement that a Task cannot be created without a null task id. The first 3 lines of the test attempts to create a task without a task id. The assertion within the test verifies that there was an exception that was thrown. There are no other assertions that need to be tested within that test, as other tests will cover other requirements and edge cases.

Some of the software testing techniques I used when testing the objects created in modules 3, 4, and 5 include state based testing, equivalence partitioning, test fixtures, boundary value analysis, and positive and negative testing, and exception testing.

Some software testing techniques I didn’t use for these milestones include mocking, stubbing, test driven development, and various coverage techniques.

State based testing is practical when you are trying to verify the object’s state after doing something. This type of testing assumes that when you do something to an object, there is an expected change.

Equivalence partitioning is a testing technique that splits testing input data into groups. These groups of data are used to represent a certain type of data. An example would be positive and negative testing scenarios. This can be useful to test different scenarios without having to run tests with large datasets.

Test fixtures are a way of setting up your tests to ensure that your testing code is DRY. This helps with creating a testing environment, and is useful in cases where you need to set up a similar environment for a set of tests.

Boundary value analysis is a type of testing to ensure that edge cases are or are not met. An example would be testing a string within an object that has a requirement of being no longer than 10 characters. You could write boundary tests to ensure that there is a failure if an object was instantiated with a string longer than the requirement, and is successful when it meets that requirement.

Positive and negative testing are types of tests that verify positive and negative scenarios. These are useful when you want to check the output value of a set of inputs. If you are expecting that the returned value is “positive” (true or equal to the expected outcome), then you would write a positive test scenario. If you are expecting that the returned value is “negative” (true or not equal to the expected outcome), then you would write a negative test scenario.

Exception testing is a type of test that verifies an exception is thrown when given certain inputs. This is useful if you want to verify that your code throws expected exceptions, and can be helpful for testing validation logic.

Mocking and stubbing are ways to isolate your code from the rest of the system. Oftentimes, you are trying to write tests for a specific section of the codebase, but might have dependencies on other parts of the codebase. Mocking allows you to create fake objects that replicate objects in real world scenarios. Stubbing allows you to create fake methods or functions with predefined return values, to see how your tested code reacts to those function / method outputs. This is useful when testing complex systems with lots of dependencies.

Test driven development is a style of software development where you write tests first to define the requirements of the code, and then write code in order to meet those requirements. It is useful when trying to write code where high code coverage is required, and is expected to meet a clearly defined set of requirements.

Code coverage techniques, such as statement or branch coverage, are ways to verify the coverage of the software by tests. Statement coverage displays the amount of lines executed and which lines were not executed. Branch coverage verifies the conditionals / decision logic within the code that was run. These are helpful when a developer is trying to maintain a high level of code coverage of the codebase, or if there is a team requirement to meet a certain percentage of code coverage.

I adopted a cautious mindset when writing tests for this project. For me, the goal of writing tests was to test for both successful scenarios and failed scenarios. I also understood the complexity of the service classes and their close relationship with the associated non-service classes. For example, deleting an appointment removes it from the map, which impacts subsequent retrievals.

To limit biased review of my code, I wrote for both valid and invalid cases. In the appointment service tests, I verified that adding an appointment works as intended, and trying to add a duplicated appointment results in an error. Testing for negative scenarios ensures that not only what is expected of the software works, but also tests for edge cases that are expected to fail.

It is important to be disciplined in your commitment to quality as a software engineer. Doing so has a direct impact on the quality and longevity of your code. When you fail to be committed, you end up creating defects that can be costly for the company, and potentially have lasting effects on you and your surroundings.

In the future, I plan to avoid technical debt by writing tests that encompass not just expected positive scenarios, but negative test cases as well. This will reduce the likelihood of creating hidden bugs when releasing to production.