CS 320

Desmond Springer

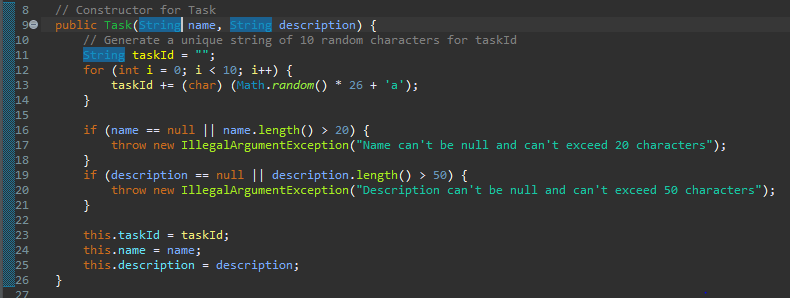
Charles Morris, Jr.

20 October 2024

Module Seven - Project Two

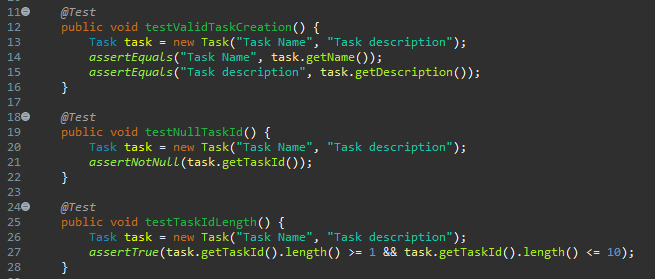
The client's requirements were to create a mobile application that contained contact, task, and appointment classes. Each class needed to be constructed with requirements such as IDs, names, phone numbers, addresses, descriptions, and dates. Each of these requirements had different restrictions. It was important to write tests that ensured all criteria were fulfilled correctly. For example, every ID needed a unique string no longer than ten characters, which could not be null or updated. To achieve this, I added code at the beginning of each constructor class to generate a unique ID of 10 characters long using a randomizer function built into Java. Not allowing the user to set or modify the ID would make it impossible for the ID not to meet the requirements.

My testing approach for the three features was to write validation code within each function of the contact, task, and appointment classes. If invalid input were entered, the code would throw an error. Then, I would write tests to ensure the validation code functioned adequately. The following code shows the constructor for the task class. Line 16 checks if the name is null or its length exceeds 20 characters when a task is created. If either of these checks is true, an exception will be thrown.



Writing JUnit tests generally involves creating a contact, task, or appointment object and using testing functions to ensure that data is stored correctly, and exceptions are not thrown. Each project contains a class file and an accompanying service class file, each with a separate test file. Each test file contains multiple tests to check every validation path within the classes. For example, in the above code, line 16 includes two checks: one to ensure that the name is not null and another to ensure that the name is no longer than 20 characters. Checking this code requires two tests. One test attempts to create a task with a null name, and another attempts to create a task with a name longer than 20 characters. Then, the test checks if the code throws an exception. By checking every path of validation, I was able to ensure 100% coverage of my code.

To ensure the code was technically sound and efficient, I wrote the object and service code to fulfill the requirements as simply as possible while enforcing all required restrictions. I also wrote each test to check only one validation at a time. This helped ensure that every validation was checked thoroughly. The following code shows three tests for checking the Task class. The first test creates a new task object and uses the assertEquals test to check that it was created properly with the specified name and description. Because the task constructor generates a unique ID at creation, the user can't make an object with the same ID twice.



The two testing techniques I used most were assertion and exception testing. I checked that contact, task, or appointment objects were created properly using assertEquals to verify that the arguments used to construct the object were the same as those retrieved from the object. I used assertThrows to ensure an exception was thrown when attempting to create an object with invalid parameters. Other test functions I used include assertNotNull, assertTrue, and assertFalse.

I also used a BeforeEach annotation in the appointment tests to generate a future date to test the appointment class. This was efficient because most appointment tests required a future date to be set up before running. BeforeEach allowed me to write this code once instead of having to place it at the beginning of every test.

I did not use dynamic testing in my tests because it was not necessary. Dynamic tests generate different parameters to test code with at run time instead of using fixed parameters like static tests. An example of dynamic testing is creating objects with multiple parameters from an array. This is useful when you have many input values with which to test functions.

It was essential to consider all requirements and exercise caution when writing the code and the tests. The main example of complexity and interrelationships in this project was that each object class had a corresponding service class. For example, the appointment class creates individual appointment objects, and the appointment service class allows the user to add and remove individual appointment objects to or from a list. Because the service class relies on the object class, it is necessary to ensure that each object class is written correctly; otherwise, every object added to the list by the service class could potentially contain faulty data.

Limiting bias was difficult because I was testing the code that I had written. Bias is a concern in this scenario because it is difficult for us to perceive our own biases. This is why it is better to have a third-party test code. However, I did my best to reduce bias by writing simple, efficient code that strictly adhered to the software requirements and restrictions. I also ran my tests using neutral input parameters yet capable of adequately checking the validations. An example of this is using parameters like "Task Name" or "Task description" when creating valid objects or "This name is too long to be valid" when making an invalid object that is intended to throw an exception.

Being disciplined and striving for quality is essential when writing tests because tests that are not thorough could result in unexpected errors, performance issues, and downtime for users, as well as extra work and profit loss for companies. As time goes on and the codebase grows, errors created long ago could be impossible to remove without ruining the integrity of the software. This would force developers to develop workarounds that would add unnecessary bloat to the code, causing technical debt. Writing quality code is crucial for creating error-free, efficient software that provides clients with a user-friendly experience and allows developers to continue their work unhindered by errors.