Testing Mindset and Strategy Review

Matt Krueger
December 10, 2023

In my approach to testing, each software requirement was matched with a specific test case. I made sure that every test focused on just one requirement, following the single responsibility principle. For example, to test the requirement "task ID should not be null," I wrote a test named TaskTest.testNullId. In this test, I created a Task with a null ID and then checked if an InvalidArgumentException was correctly thrown. This way, each test directly relates to and checks a single requirement.

I am confident about the quality of my JUnit tests for the contact and task services. I had code coverage over 80%. The omitted code was “getter functions” on the base classes and “delete functions” in the services, which I didn't feel needed testing since they were either generated by the IDE or simply involved calling a delete on a HashMap. The tests are good quality because they test all of the different partitions of inputs without repeated tests for any one partition. Test data was selected to be at the boundary of each partition. For example, in testing the Task service, I used a Task name with 20 characters (valid), an empty string (valid), one with 21 characters (invalid), and a null value (invalid).

I made sure my code was technically sound by using getter and setter functions such as Task.getName on the base Task and Contact classes, ensuring single responsibility. The constructers and the Service class methods reuse these function so there is no repeated code. Invalid conditions are appropriately handled by throwing illegal argument exceptions with a descriptive method. This ensures that the error stack trace is concise and helpful. The code was made efficient by using a HashMap data structure to store Contact and Task instances in the Service layer. This ensures we can do our CRUD operations central to the requirements in the fastest O(1) time.

For each milestone in the project, first I used static testing techniques. This is examining the specification and requirements of the code without the need to write or execute any tests. This was to ensure there were no contradictions or ambiguities. This technique is even more crucial in professional projects as it follows the best practice of early testing, significantly reducing the potential costs associated with errors in projects. During implementation, I shifted to dynamic testing techniques. This involved writing unit tests for each component to execute the code with various conditions and inputs. This is a white box testing technique and it is essential for validating the operations of individual components. Some integration testing was involved when testing service classes since they use our custom base classes. These techniques are requisite in any software project for verifying and maintaining the software's correctness.

Several techniques were not used when developing the milestones. First, I didn't apply black box testing, which focuses on the macro-level functionality of a component by examining its inputs and outputs without delving into the internal logic. Known also as functional testing, it's typically used to verify a project's overall functionality, not for individual components. This approach is more suited for QA acceptance tests in projects with a standard Software Development Life Cycle, as it helps ensure that the developer's work aligns with the specified requirements. Additionally, regression testing was not employed. This technique involves retesting previously functional features that might be affected by new development. It wasn't needed in my case since there was no iteration on the tested code. However, in projects with ongoing development and updates to existing code, regression testing is crucial to identify errors introduced by new changes. Lastly, I didn't use non-functional testing, which assesses aspects like security, latency, and scalability—attributes that are not directly related to specific functions

of the software. I omitted this because the modules at this stage did not have set non-functional requirements needing verification. In more advanced projects, especially those transitioning to a production environment, defining and testing non-functional requirements is essential to meet user expectations.

Caution, bias, and discipline were import concepts to keep in mind when evaluating my testing mindset. I identified my bias as being overconfident in the code that I had written. Since I carefully reviewed my code and didn't see any issues, it was difficult to imagine that it contained bugs. When I wrote tests and ran them for the first time it always felt a little surprising when I found an issue as this code was pretty simple by my own standards. This could be eliminated by having another person test my own code. In my case it was the simple addition of testing for null on a class being invalid that ended up failing a test because I forgot to write that error case into the implementation. I had to be disciplined to avoid looking at my source code when designing test cases to make sure my test cases are being written purely off the requirements. Otherwise if you are looking at your code to write tests, a missed error case like mine in the implementation stage would likely also get missed in the testing stage. It was also important to be disciplined to not take simple code for granted and be caution enough to test your own assumptions.