Project 2

Ryan Laird

6/18/2022

Southern New Hampshire University

CS-320

**Project 2**

The testing approach within all three classes began in the class itself. The requirements were coded into each field to ensure that an exception would be thrown if any one of the constraints were broken. The tests were then designed to purposefully break one of these constraints at a time and check for the specific exception. This design checks the code to make sure that the code detects the proper constraints and provides feedback on the status of each field in relation to the requirements. The approach for the service classes was slightly different. Since the field requirements such as not null and length were already covered within the class testing, the service testing was meant to test the functionality of the operations on that class. The only exception that gets thrown by the service classes are when a presented ID is not found. In the service tests that exception is checked for when a non-existent uniqueid is passed to any of the functions. Apart from the unique ID case, every service test operates by creating a new object, running the tested function on that object and checking the result for the expected result. This ensures that the functions provide the expected output and therefore whether they are operating as they should be.

Using these tests, the coverage through the entire package was over 80%, but upon further inspection the only missing coverage according to eclipse was in the tests themselves. Eclipse stated the coverage in every class and service file as 100%. While I have no doubts the tests are effective based on this coverage, my key area of concern is with the unique id function. There is no explicit test for this, and I worry that there is a chance this code my break with more rigorous testing. However, by utilizing a specific test for every requirement in the services and classes I ensured that the code was technically sound according to the provided requirements. For example, the task service test contains



. The efficiency of the code is also tested by checking the functionality of the various operations, mainly in the service tests.



In conjunction with the rest of the tests in the class test file, every case is accounted for.

White box testing was used for all tests throughout this project. The tests were written by accounting for the structure of the code and how the code checks for errors. A key example of this is when testing for fields to fit within the constraints, those constraints are purposefully broken and the console is then checked for the correct exception to be thrown. Without knowing the type of exception thrown and the coded constraints, these tests would not be able to be implemented. Below is the setter for the appointment description showing the exception being thrown when the length is above 50 characters or the string is null.  


This function is then tested with 

The test attempts to set the description to 51-character string. It then concludes the test as passing if the program throws an IllegalArgumentException as a result.

Black Box testing wasn’t utilized within my test cases. My test cases all required at least some knowledge of how the code underneath was functioning. Black box testing is the process of testing software without access to the code itself. This is useful for checking functionality from the perspective of an end-user. For this project, white-box testing was requested as it required Junit tests to be written to check against provided requirements within the structure of the code.

Testing against requirements has a few practical applications. Things such as an unaccounted-for null field has the potential to crash software or create unexpected bugs. In addition, ensuring the software actually meets the requirements set by the client is crucial to creating a deliverable product. Finally, clearly defining a fields constraints early in development, and ensuring the software enforces those constraints is important to working with those same fields later in development. A field with a known length and contents assists with processing the data in those fields or even simply displaying that data in a user interface.

When I began writing the tests I took on a thorough mindset. I wanted to ensure that my tests achieved sufficient coverage and that not only every requirement was tested, but everywhere the requirement could be broken was tested. This required me to understand the interconnectivity of the code as a fields requirements could be constrained correctly through one method of setting, but that does not mean they would be constrained everywhere. A prime example of this is with the setters within the classes. For instance, the name field within Contact was properly constrained during the construction of the class, but those constraints needed to be tested in both the initialization of the class as well as the setter. This insured that the constraints could not be broken when modifying an existing field and resulted in two very similar tests.  


During the testing process for each class I tried to remove the inherit bias that comes from owning the code you are testing. It is very easy to excuse issues that you created yourself, and these excuses are detrimental to properly testing the code. One misstep that can be made is assuming that a certain function works properly because you created it and therefore doesn’t need to be tested. I ran into this with the unique IDs in all of the classes. I wrote a function that utilized Java’s UUID class to generate a random string of 10 characters and utilized this string as the unique ID for each class. That being said, I am still new to Junit testing and couldn’t come up with an efficient or effective way to test whether or not the unique Ids were genuinely unique or not and so I made the assumption that if no errors were popping in the tests that relied on this function then there were no issues with the function itself. This is a mistake with my testing and one that came from my bias.

Being disciplined and committed to quality is crucial for a software engineer. Writing and testing code is a balance of meeting delivery times as well as quality standards. One issue that may come with non-comprehensive testing or cutting corners is unexpected errors and bugs. These errors could lead to financial loss in order to fix the error, or more serious consequences depending on what the software is used for. In addition to software errors there is also the issue of maintainability. If code is rushed out the door, the likelihood is high that sections of the code are sloppy and unmanageable. This makes maintaining the code down the line take much longer and cost more than it would have if it was done right from the beginning. These consequences are known as technical debt, and as a developer it’s important to be aware of this issue and be thorough with testing and to not cut corners throughout development.