Sarah Beaty

CS 320: Software Test Automation and QA

Southern New Hampshire University

April 15th, 2023

My testing approach was well aligned with the software requirements for all three sections of the project. For each section, I ensured that each of the requirements was checked by creating exceptions that would be thrown if any of the fields were outside of the specified range given in the requirements, such as null fields are fields that were too long. Each service is capable of creating and deleting the specified object, as well as being updatable for the Task and Contact services.

For the project, I made sure to run the JUnit tests frequently to ensure that they were working as intended. I also made sure to run my JUnit tests with coverage to see how much coverage I was getting overall. By running these coverage tests, I could easily see what percentage of the code I had written was being tested. This made it easy to check what I still needed to work on, and what was covered already. To ensure my quality, I tried to have my coverage over the ninety percent mark for every test I ran.

To ensure most of my code was technically sound, I was using exception throws. I was checking that these exceptions were being thrown when the criteria was met for the specific requirements, such as the contactID being null or too long. I did this for all of the required fields in each service. The tests in my ContactTest starting at line 22 and TaskTest starting at line 35 reflect this use of exception throws. I was also running through the logic out loud while I was testing my requirements. This helped me to understand the logical flow of my tests and ensured that each of my test results were what they needed to be.

To ensure that my code was efficient, I was using the BeforeEach and AfterEach set ups and tear downs. These helped my code to be easier to read and less redundant by setting up some test vectors and objects to change and delete during the tests. These can be found in ContactServicesTest starting line 15, TaskTest starting line 14, TaskServicesTest starting line 15, and each of the other Junit test classes.

For each section of my project submission, I used very similar testing techniques. One testing technique I used was a black-box technique called equivalence partitioning. This was used in each of my sections as one of the first tests. I used equivalent string inputs to check to see if the objects would be created and that they had the proper information stored in them. Because the requirements had very specific boundaries for the data strings, it would work for any value I put in that was within those boundaries. Another technique I used was a white-box technique called decision testing. This technique was used in each of my sections as I was inputting fields with the intent of trying to get an exception to be thrown, to check that it was working. I did this to check for null fields and fields that were too long. I was also testing for coverage, being mindful of the percentage of coverage that my tests were doing.

There were a few black-box testing techniques that I did not use. Some of them, such as the decision table, state transition, and use case techniques, were suited more for diagrams instead of actual testing of the code, so I did not implement them. The decision table lists all the input conditions and what possible outputs it would produce. The state transition test is a sort of diagram that shows the logic behind the inputs and outputs. The use case is a diagram that shows how an outside force interacts with the system. Another technique I didn’t use was boundary testing, since my inputs were mostly strings, I did not feel I could test the boundaries well.

The black-box testing techniques that I did not use, such as the state transition and use case tests, would be good tests for a project that needs the process tested. These two tests look at the way that outside forces interact with the system, and even what other systems the current testing system will interact with. These two tests really test the logic behind the project. The decision table, which also tests the inputs and outputs of the project, tests for logic as well. However, I feel this testing technique is better for a system with fewer input options. Such as a system that takes true or false Boolean types as the inputs and outputs, or even a mathematical system that uses few variables. Boundary testing, which tests the boundaries of the possible inputs, is a testing technique that would be best used with numbers. Then it can test the very limits of the system.

While working on this project, I employed caution by using many tests for many different variables and ensuring that my test coverage was at least ninety percent. This allowed me to cover as much as possible and allowed me to know what I was and was not testing. It can be overwhelming staring at a long code block, but with taking it one step at a time and using the Junit tests for each method, I was able to cover most, if not all, of my project. Using the coverage testing function, I was able to see how each of my tests interacted with the classes they were testing. The test for checking for duplicate task IDs went through the Task Service class to the Task class where the isDuplicate function is declared, for one example. This allowed me to better understand how each section was connected, which could be very useful for a tester in understanding code that is not written by them.

To eliminate bias while testing my code, I wrote tests for everything that I could think of even if the tests were failing. In that instance, I tried to rewrite the code and tests until I got the code fixed and tests passed. The delete object function in each of my service classes was the biggest example of this. I kept trying to run the tests, and the tests kept giving me errors. Continually I rewrote the method itself and the tests, until I was sure both were correct, and the test was passing. If I were a software developer, I could see how bias could come up while testing my own code. There were a few times that I felt that there was no way my code was wrong, but then after careful examination I could see a simple error, such as a missing semicolon.

It is important to be disciplined in quality testing because usually that testing is the last line of defense from the product going out into the public world. It is important to not allow defects to make it past the testing phase because those defects cost more money to fix once released and can even sometimes cause harm to people. Cutting corners or skipping important tests could be what causes a major financial loss, and there are many examples of software errors that were released and cost the companies hundreds of millions of dollars, in some extreme cases. One such example came from NASA, as they released their Mars Climate Orbiter. The spacecraft had a simple error in the coding from standard to metric measurements, which caused it to burn up in Mars atmosphere. Such a simple error could have been detected in testing and would have saved millions of dollars if it had been caught before launch.

Resources:

Stableford, B. (2023). Mars Climate Orbiter and Mars Polar Lander Are Launched and Lost. Salem

Press Encyclopedia.