Michael Bauer

Summary and Reflections

My unit testing approach was to test the features for failures in how they worked or what data functions accepted. I really do not know how you can test something without knowing what the program is suppose to allow or how it should work. Knowing, for example, that the ID needed to be unique I tested to make sure that the user could not add the same ID if it was already used. Without knowing the software requirements as a tester, you would have to guess what the requirements were. This could lead to not testing what actually needs to be tested and instead testing what you think the requirements should be. My JUNIT tests were effective since they covered 100% of the testing for the functions. I made sure to test the functions to allow and not allow specific lengths of characters depending on what the requirements were. I also tested for anything that could break the functions. After running these tests I covered 100%, this gives the confidence to say that the quality of my JUNIT tests were good.

I wrote code to test all the functions with 100% coverage while trying to not have code that duplicates any other tests. I broke down the tests to test the class separate from the program. This made sure that the code written hit the specifics of the class and program without duplicating code. When possible, I limited any extra code needed to initiate a variable to be used for testing and reused the variable. I also broke down the tests into functions and named these based on the type of tests within that test function. An example of this is a function named testAppointmentClassSet, this tests to make sure that when you initially set an appointment that the data inputted to set is the actual data that is set. I feel that efficient code goes in hand with technically sound code. By making sure I did not duplicate testing of previous tests was one way to keep my code efficient. I tried to reuse variables for tests that needed a variable while also maintaining organized code. Being able to not duplicate testing for a specific function and limit the amount of variables I needed to use helped to maintain efficient code.

I mainly used specification-based techniques to test the code. Within this type of technique, I used equivalence partitioning to test for any defects. Since most of the requirements were for a variable to not exceed a specific length it was easier to use this technique. Equivalence partitioning testing technique looks to chunk similar input to reduce the number of tests needed to run. If a string accepts input no larger than 20 characters and you test for 10 it is very likely that it will also accept the other 19 possible lengths. This technique allows us to chunk the possible lengths into one group and test that one of those in the group works.

I did not use any of the other specification-based testing techniques or the structure-based testing techniques. The specification-based testing techniques include: boundary value analysis, decision table, state transition, and use case. Boundary value analysis technique checks to make sure that values that are chunked into two different groups do not overlap and allows a number in one group to be allowed in another. This technique checks the boundaries since these are the most common areas of problems. A boundary example would include one group stopping at 20 and the next group starting at 21, this technique would test 19, 20, 21 for the first group and 20, 21, 22 for the second group. Decision table testing makes sure that all the possible different combinations of outcomes from certain decisions within a program works as intended. State transition testing is similar to decision table testing except it looks at different states a program is in and was in and determines if the correct output based on an input is working the intended way in that state. Use case testing is used to test for ways a user would interact with a program and checks for weaknesses or defects.

Any project that has users should use a use case test to make sure that the user’s interactions work as intended and they cannot break the program. Equivalence partitioning testing should be used where there is input that can be grouped and if there are multiple groups on inputs then boundary value analysis technique should be done along with equivalence partitioning testing. Decision table testing should be used where there are different conditions possible that can affect the output. State transition testing would be used in projects that have different states, for example different modes of a watch. Multiple techniques would be used to cover and check for all the possible defects and the combination of techniques depends on the project. Looking back I should have employed other techniques together to cover, for example, the boundary of a variable’s length as well as equivalence partitioning to cover the specific chunk of data that is allowed and not allowed.

While working on this project I made sure to change my mindset from a programmer to a tester. My mindset when testing was to find where the defect was because I knew that there was one. The idea was to try and break the program or find a piece that was overlooked. I really did not employ caution if by that meaning caution on not trying to break the program. I tested to ensure the program did what it was suppose to do without breaking and without worrying about breaking the program, since as a tester that is the job. Realizing how the code works and how the functions interact with each other is important to make sure you are testing those relationships. Understanding how the program updates the contact name by using a function to call a method to set the name is an example of understanding the interrelationships and possible complexities of the code.

The way I tried to limit bias was by changing my mindset to a tester and acting as if the code I was testing was random code that I was given to test. This made sure I tested for all of the possible issues without trying to spare my ego as a developer. I could see bias being an issue if you were both the developer and tester. Since you do not want your code to fail and look inadequate, you would not test the code fully to either limit the number of possible defects found or to not find any. The issue here is that these defects will be found eventually and then when they are you have failed as a developer and tester.

Being disciplined in both aspects, writing and testing, is important for several reasons. First, it is important to be disciplined because the work you produce is a reflection of yourself. You should be disciplined enough to not cut corners and produce the best possible quality that you can do. Another reason for being disciplined is that by allowing yourself to be undisciplined in writing code makes more work for everyone on the team. The tester will find more defects and send the code back where the code needs to be fixed. Then the code needs to be retested, this is all extra work that could have been avoided. Being undisciplined as a tester can be worse since you can let defects go public for users to find. This can cause many issues later on and way more work, time, and money to fix. I feel there are two ways to help alleviate technical debt. One is to make sure there are no corners being cut by the writers. This will, in theory, cause fewer defects in the code and thus less rework. Another way is to make sure overall the easy or simple way is not taken over the correct way. An example is making sure to do the design and planning even if time may not seem to allow this. One other way is to make sure as a developer you do some testing of the code unbiased to make sure there are less defects to be found in testing.

Resources

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software testing: An ISTQB-BCS certified tester foundation guide*. London: BCS.