CS320 Project 2

Michael Tekin

10/24/21

My unit testing approach for the three features of our program was specifications based or a black box approach. I analyzed the requirements and designed the tests to specifically address those requirements and verify that they were working properly. For example, with each of the features, the requirements document dictated that certain objects needed to be able to be updated. Using that information, I created tests specifically for updating objects. I attempted to update them and had the test throw an error if the update failed for any reason.

Using coverage percentage, I was able to verify that each branch in my tests was being executed. Knowing that the tests were being executed, I was able to confirm that the programs were working properly – in other words, the tests were carefully designed to throw errors if the code didn’t perform according to requirements, and since I verified that the tests were being executed, I can say that the tests were effective.

Verifying my code was technically sound involved ensuring that complex requirements were met. Examining the more complex functions (verifying that they worked properly) allowed me to verify that the simpler or more basic operations that those complex functions are based on also worked properly. For example, in module 3’s assignment, I verified that I could add and remove contacts to a contact list. I did so by using assertions to test that the contacts existed as they should (or ceased to exist) after running those functions. In my ContactServiceTest class, I used assertEquals on line 49 to verify a contact was added to my contact list and that the contact that was added had the correct name after running my addContact method. On lines 61-63 (after running my removeContact method), I used a assertThrows assertion to verify that trying to access the contact from the list that had been removed would properly throw an out of bounds exception.

Using the same technique of testing complex requirements (requirements that relied on simpler functions working properly) I was able to efficiently test all requirements. For example, again, in my ContactServiceTest class, I used assertEquals on line 49 to verify a contact was added to my contact list and that the contact that was added had the correct name after running my addContact method. Simply verifying that a contact existed in my contact list with the correct name allowed me to confirm that the Contact constructor and addContact functions were working. This lets me test more requirements with less code.

My testing relied heavily on equivalence partitioning. Equivalence partition testing involves partitioning values into chunks or groups that can be tested as a whole using a single representative value. The practical value of this kind of testing comes from greatly reducing the number of tests we need to do allowing us to save time and other resources. In our milestones, we needed to test that certain values were not greater than 10 characters in length. Using equivalence partition testing, we can create three partitions, a partition for no characters, a partition for values at or under 10 characters in length and a partition for values over 10 characters in length. Using these partitions, we can create three tests, a test with an empty string, a test with a string of 5 characters, and a test with a string of 11 characters.

Testing techniques I did not use include decision table testing and boundary value analysis. Decision table testing is primarily used when a combination of variables work together to create a unique situations. The practical value of this kind of testing partially comes from helping testers limit the possibility of forgetting to test a certain scenario or situation. For these milestones, there were no situations that relied on a set of variables working together in a complex variety of ways. For example, in milestone 5 we tested if certain variables were either too long or null. We could create a decision table for this but it would be so simple as to offer no substantial benefits (if it’s too long or null then an exception should be thrown). If there were an additional condition involved (like if the string were to be an even or odd number of characters or contain a specified substring) then a decision table may be beneficial.

The practical uses for equivalence partitioning include greatly reducing the amount of testing that needs to be done. By reducing the amount of data that needs to be tested to representative values testers are able to save time writing tests and save resources through having to run fewer tests. Decision tables are immensely beneficial in helping simplify complex scenarios and helping testers ensure that all possible scenarios are accounted for and being tested. This is crucial for reducing the number of defects that may be released.

As a software tester, it’s important to have a degree of caution in your work. In many cases, those developing tests are the last line of defense against defects before they can get released. The number of ways that things can go wrong is directly correlated to the complexity of software projects. It’s important to keep this in mind when designing and running tests. It’s not enough to test a few components or just the “main” ones. It’s essential to test every component involved with these massive projects because in this case, the chain is only as strong as its weakest link. For example, when removing objects from our lists, it’s not enough to verify that the intended object is removed, it’s also important to verify that all other objects shift afterwords. And when we start dealing with shifting lists of objects, if we’re not careful, we increase to possibility that we’re going to run into some sort of out of bounds error.

I tried to limit bias in my review of the code by clearing my mind of work done writing code and re-establishing what it is the code is meant to do through repeated reviews of the requirements document. I strongly believe that bias would be a concern if software developers were responsible for testing their own code as they had a certain frame of mind that went into building the program and they may hold onto that when they go to test the program but that just ensures that they’re going to be testing their own plan which may have been limited in perspective to begin with. Having a fresh pair of eyes try to come up with ways that the code could break should help improve the quality of the code.

It's important to be obsessed with commitment to quality for many reasons. Cutting corners can result in serious negative impacts on our reputations as individual professionals, on the reputations of the companies we work with, it can also mean huge costs to fix those mistakes later on down the line, and if we’re working on safety critical projects it can even mean serious injury and loss of life. I plan to avoid technical debt as a practitioner in this field by avoiding cutting corners but also by staying up to date on the latest techniques and tools that are meant to help make creating high quality work easier. For example, learning more about what junit is capable of will surely help make creating good comprehensive tests easier.

**References**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition).* BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb