Chris Gollnick

Final Project 2

CS320-T3228

21 February 2021

When reviewing the mobile application that contained elements for appointments, tasks, and contacts, the initial approach was to build and understand what was necessary for the application. Upon reviewing the client requirements and the elements for each, we would attempt to implement as many of those features into the build. Afterwards, when testing was to commence, we would test each of these features and ensure they work according to how they should be used, test cases for when they are used not according to what is expected and attempt to theorize different potential situations and uses that the client and end user might attempt to utilize them. For example, in ContactTest.java we ran a test to test creation of a new object under expected protocols at line 22 -26. If the Junit test passes, then we know the creation protocol works as expected with normal values within the specified ranges. The remainder of the tests in that file were to test what would happen when a value was sent that violated the parameters of character limit. Each time we were to expect an exception and a successful Junit pass meant that was met. In the end, testing is supposed to help the program run and ensure as many elements of the program and its functions are tested to work as intended. While theorizing every possible scenario is difficult, using the run coverage option can help us isolate how much of a percentage of the application the test will cover. “Test coverage is a very important idea because it provides a quantitative assessment of the extent and quality of testing. In other words, it answers the question ‘how much testing have you done?’ … provides a quantitative measure of the quality of the testing that has been done by measuring what has been achieved. It provides a way of estimating how much more testing needs to be done. Using quantitative measures we can set targets for test coverage and measure progress against them” (Hambling, Morgan, Samaroo, Thompson, & Williams, 2019). In the case of each of our test files, they met at 80% or higher. Since 80% coverage is the acceptable baseline, and it identifies that 80% of the application is tested, then we can assume from there we are testing most of the application to run without incidents.

When writing Junit tests, we needed to be careful and mindful how each element was referenced and utilized in the main applications. When we look at TaskTest.java at lines 20-25, we can see the base scheme of the Junit tests. Essentially the thought needed to be what are we testing? And what operations occur once that function executes? In this case we wanted to test the TaskID function and creating a new ID. In order to write the test, we need to understand what happens. The TaskID function is given a String value of a number or the uniqueID. It then checks the variable to see if it is null, if not then it checks if it exists in the array where ID’s are stored, if it is not a duplicate then it checks to see if the character length is less than 10. If it is, then it stores the ID in the array at the next position. Now that we understand the function, we can then look at the Junit test writing. We know we need to call the function and pass it a string value for a unique ID. In the test, we are then going to store the ID number to a local variable of uniqueID and then call the function and give it that variable to use. We then use the assertTrue to check if the expected result equals the actual result. IF they are then the test passes. This long-routed thinking helps us identify what everything does so when we write the test, we know it should come back and pass and helps ensure that what we wrote is correct to the process of events that needs to occur. The efficiency of the writing can be understanding it to practicality and logicality of what occurs. In the above example, we may have been able to run the test without declaring a local variable and just passing the value itself. There were times when what looks complicated was the most efficient way to get a result, however. If we look at TaskServiceTest.Java at lines 31-53, we can see the test to add multiple tasks. Since the original requirements never mentioned about adding multiple items at once, it was not a feature that was coded into the main application. When writing the test, the first though was to just write the AddTaskTest multiple times in a row. This would seem highly inefficient. Instead, we landed on the idea of having an array of variables for each passed value and the loop will iterate through the list. This system may look confusing compared to other tests in the file but has proven to be the best method we found for performing this action without needless repetition of typing code.

In this project, our testing technique was mainly Junit testing utilizing the assertTrue, assertEquals, and assertThrows methods. The assertTue and assertEquals ran on similar characteristics. We passed values to the function in test, and we expect a certain result. The methods then tell the program to get the actual result and compare the actual versus the expected. If the values are the same, then the test passes. We can see this in TaskService.Java lines 76-81 where we call the test True if the value “Deleted task successfully!” comes back as a result. Likewise, in AppointmentTest.java lines 25-29 we can see the assertEquals looks for the given idNum to be returned as the expected vale and compares what exists in the getApptId function against it. If the two values are the same, then the test passes. The assertThrows was used to test when we passed values that violated the criteria or parameters of certain values like character length or null. With this method we would expect and exception to be thrown when one of the parameters was violated. If the exception is reached, then the test passes. In AppointmentTest.java lines 35-39 we can see this at work. This test was to test if the unique ID was a duplicate. We passed the value of an ID that already exists and as such we expect an exception to be thrown. If it is then the test passes.

There are certainly other methods of testing that were not utilized such as assertFalse, assertNotNull, and others. The assertFalse should pass if the expected and actual values are not the same. We did not see a real use for this in this project. The assertNull and assertNotNull methods should pass or fail based on the null condition returned. This probably could be implemented instead of exception on null tests. Outside of Junit then theres testing methods such as Integration testing that should test what happens when your module is integrated with others. “These are then tested as group through integration testing to ensure whole segments of an application behave as expected (i.e, the interactions between units are seamless)” (SmartBear, 2021). Seeing as we were not integrating this application anywhere else there arose no need to utilize this type of testing. If this project were to move to become part of a full application for a medical or business office, then we would want to consider integration testing as what we have so far will likely need to work with other systems and applications. At that point, failure to run the test during that scenario would mean serious downtime as it could lead to records and files being orphaned or inaccessible that are business critical. Acceptance Testing is another method not used that aims to test whether the final product meets the criteria of the client and the requirements and is ready for final delivery. Since we are operating within a small frame of the whole application so far, we are not near the end of where the project would be deliverable to contemplate this type of testing yet. This would become critical later in the process. While we work through an application, we can get close to what we are working on and forget about the bigger picture. Running the final acceptance testing can help allow us to take step back and re-look at that bigger picture to ensure everything works. Without it we can then send an application out that has a handful of really great features but what the client paid for was not met.

During this project employing caution became critical to writing successful tests. This meant that we really need to think and analyze step-by-step what each function was doing, what values were being passed where, and what outputs would exist. Failure to do so meant poorly written tests or tests that may pass but do not actually test what we were trying to. For example, the first iterations of the ContactTest and ContactServiceTest was written with giving the program values and letting it run. Nothing was every compared so no real test occurred. Running the Junit test at that point technically passes because no errors occur. For later submissions, this was re-worked to have actual testing parameters and look for specific results there by testing the function.

When it comes to bias, elimination of it can be easier than many others may find. When testing, we went in on the belief of that we are not experts in Java nor are we professional or junior level programmers. As such, there was no real reason to be overly proud or defensive about the code written. Secondly, separating the two roles and switching mindsets is an invaluable skill acquired over years of training that is at this point easy to accomplish. As such, switching to a tester role was seamless and easily able to look at the code, analyze what it does, and then test it. We could then easily switch roles back to developer to fix what did not work and then go test again. Testing items like Multiple Addition of objects became a clear use of this as the developer role had no indication of this existing but the tester role needed to find this as a possible use case a client might try to do and thus need to test if it can be done.

Being disciplined and doing things thoroughly and correctly is critical to being a software engineer for multiple reasons. While taking shortcuts can accomplish a goal, you need to fully understand what it is that the shortcut does. In doing so you also need to think of who else is working on the project with you and how does what you do affect them. Fore example, if you use a shortcut as the developer, does the tester know what that shortcut does? How can they write a test if you can not explain it or they cannot easily understand what it does? Does the shortcut have an impact when the project gets integrated into other modules or the final piece? If it causes problems can the debuggers or QA team find the problem and identify what it is? Also, if you die tomorrow, will another developer understand what you implemented to pick up where you left off? These types of questions are what need to be thought of since Software Engineering is not a single person job but a team-based career path. At any of these points if you cut corners and someone can not figure out what was going on, you effectively are now wasting the companies’ time and money to attempt to analyze and resolve the problem you caused.

References:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software testing: An Istqb-bcs CERTIFIED tester FOUNDATION GUIDE. In *Software testing: An ISTQB-BCS certified tester foundation guide* (3rd ed., p. 83). Swindon, UK: BCS Learning and Development.

SmartBear. (2021). Software testing methodologies. Retrieved February 21, 2021, from https://smartbear.com/learn/automated-testing/software-testing-methodologies/