Cody Gregory

[CS-320-T6629 Software Test Automation & QA 22EW6](https://learn.snhu.edu/d2l/home/1116032)

**Summary**

When it comes to developing software that is in line with requirements, effective, efficient, and technically sound the first thing that should come to mind is testing. In our most recent project at Grand Strand Systems we developed the back end for a mobile application in Java, and thus utilized the JUnit suite to accomplish these goals.

Perhaps the largest benefit of unit testing is the role it plays in requirement adherence. This was accomplished by turning the literal requirements as provided by the client into JUnit tests. One such example in the appointment handling portion of the software is the specification for an appointment ID. The requirement directly states, “the appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable.” From here it is a matter of creating a test that handles each facet of this requirement. A test is made if the ID is a duplicate, if the ID is longer than 10 characters, to check that the ID cannot be null, to ensure that the ID cannot be updated. Once these tests have been implemented and the code passes them, we can be confident that our code aligns with the software requirements.

While the aforementioned tests sound great in theory, we need to know that they are great in practice. That is to say we need to know we have quality tests that test for all the different ways an expression can be evaluated. To do this we once again leverage the tools of JUnit and our IDE Eclipse to check our coverage. Wikipedia defines code coverage as, “a percentage measure of the degree to which the source code of a program is executed when a particular test suite is run.” which simply put is the percentage of code that is actively run by your tests. In the case of our mobile application we know that our coverage is over 80%. This serves as a good metric to know that we are on the right path and that the majority of our code is actively run through our tests.

Not only do tests provide a framework for requirement alignment and effectiveness, but they also ensure technical quality and efficacy as well. In my mind I think of these two categories as intertwined. For software to be technically sound it should be using the proper data structures, types, and methods. For software to be efficient it should avoid unnecessary complications. We can reflect on how these two concepts relate to each other through our mobile application’s task service. The task service manages task objects by reference to an ID. One such way you could go about creating the task service could be through utilizing multiple array lists and a for each loop. While this strategy may work and achieve functionality, it wouldn’t be technically sound as another data structure exists that is better suited for this job. And it wouldn’t be efficient as the quantity of objects and loops you are handling is more than what would be required through different means.

Ultimately for all of our service classes we employed a hashmap to store the given object in reference to the key which we set as the ID string. We know intrinsically that this strategy is more efficient as it is less code (and does not contain recursion). We know that our code is technically sound as we were able to clearly produce tests that functioned precisely as anticipated.

**Reflections**

Throughout the development of our mobile application the main form of testing was dynamic, specifically unit testing. I found that I naturally gravitated to an iterative approach that allowed me to get coding immediately. Once I had something actionable I would then create a unit test for it, which the very process of doing acted as a form of reviewing my logic, yet alone the changes that would need to be made to pass the test itself. One such component that highlights this experience was in the appointment class utilizing java’s date class. When writing my code I leaned on the abstraction of knowing what a date is and how it should behave to create my methods. However when it came to constructing and passing unit tests involving dates it became imperative that I knew how a date functions in Java inside and out. I needed to know how to create a date, to alter one, to refer to each component such as day / month / year, otherwise I would not be able to create proper tests that verified my code's functionality. In this way testing actively required that I become more knowledgeable on the subject at hand and made me a better programmer.

The testing techniques I did not use mainly fall into the category of static testing. You could make an argument that there was a degree of static testing in the form of technical review as each component of the final project had been submitted for grading prior to the final compilation. But there were no walkthroughs, and I had done no diagramming prior to coding. I do feel that static testing is an invaluable resource when dealing with large scale, or generally more complex projects with more parts than our mobile application. As it is far easier, and way less expensive to solve a problem before it happens then towards the end of development where one issue could have dire ramifications such as the altering of your entire program, as a wise man once said, an ounce of prevention is worth a pound of cure.

While we have spent the majority of this review stressing the importance of testing there is another factor that should not be overlooked: mindset. When working on the various classes of our mobile application there were a multitude of factors that needed to be considered at all times. How is this code going to affect that code? How am I going to integrate these components together? How am I going to test this? All of these questions can bog you down if you let them. As such it is important to apply caution, but it is also important to allow yourself to fail. While at present time the application is complete, functional, effective and efficient it did not begin that way. I would write code, then realize when I went to test it that I couldn’t and would have to go back to the drawing board. I remember this clearly in the contact service class, going through a couple of iterations such as using array lists before I landed on using a hashmap as my data structure.

While a good mindset is to be embraced, when it comes to bias our goal is to distance ourselves. Bias is a particular problem for developers since by being the one that created the code you have an intuitive knowledge of how the program ought to work, and therefore are less inclined to see ways of how others might interpret the same. I recently watched a video explaining survivorship bias that used a diagram of a plane from World War II with red dots all over it, which represented where the plane had been shot. The question at hand was with this information, where should you reinforce the plane? At first thought you would probably suggest where the red dots are, as that is where the plane was shot, but in reality you want to reinforce the plane in all the spots where there are no red dots, as planes shot there didn’t make it back to be able to get added to the diagram. In a way this logic can be adapted to programming, whereas as a developer you are so focused on the red dots you wind up missing the point of it all.

Lastly we are left with discipline. Personally I attribute my attention to detail as a personality trait. Once I start something I can’t leave well enough alone. In reality however you are bound to have good days and bad, and it is important that you hold yourself to a quality standard so that you can avoid the various pitfalls and technical debt that creeps up when you cut corners, or skip testing. Through our discussion this week we were assigned to research cases where things went wrong, and it is not difficult to find instances of human errors through taking shortcuts or lack of testing that resulted in catastrophic losses which shows that even with a lot on the line people will still get lax and let things slip. I feel on a personal level the best thing is to respect the process, and do your best to learn from others mistakes so that you can avoid having to make your own.