Christopher Williams

10/15/2023

[CS-320-R1878 Software Test Automation& QA 23EW1](https://learn.snhu.edu/d2l/home/1374795)

**Summary**

My approach was perfectly aligned with the software requirements because I wasn’t trying to do anything other than fulfill the requirements and I didn’t run into any significant trouble that kept me from fulfilling any of the required functionality. My only goal was to build these basic functions, link them together, then use Junit to test them. For example, one of the specifications for the Contact class was that it needed to require a ContactId with ten characters or less to create a Contact object. To test for this, I created the Contact class with a constructor and getters and setters. In the constructor, I specified that a Contact object can only be made if it has a valid Id. Then, in the test class, I created an instance of Contact with a null Id. Then, Junit came in handy for specifying a test to see if that Contact existed or not. The idea was that if the Id is null, then the Contact should fail to be created. With a test like this, you can really spell out what exactly needs to be done, then make sure it worked. It is also necessary to “Run with Coverage” so that you can see if you missed any chunks of the source code.

I found Junit to be fairly fun and easy to use. I can imagine using it on lots of other projects. For a while, my tests kept failing, but apparently it was a build issue, followed by some really silly logic errors. I had thought that everything was spelled out so simply that there was no way I could have messed it up, but it seems that sometimes the classes I was testing had strange behavior. The UserService class kept retrieving the first stored User when I was trying to call non-existent Users. I had to alter the service class to fix this. The only thing that I did to make my code efficient was that I tried to use minimal code. Here is a bit of code (the constructor and destructor for AppointmentService.java) that demonstrates the efficiency and soundness of my program:

*// Add an appointment with a unique ID* **public void** addAppointment(Appointment appointment) {  
 **if** (appointment != **null** && !**appointments**.containsKey(appointment.getAppointmentId())) {  
 **appointments**.put(appointment.getAppointmentId(), appointment);  
 } **else** {  
 *// Handle duplicate ID or null task* Log.*e*(***TAG***, **"Duplicate ID or null appointment"**);  
 }  
 }  
  
 *// Delete appointment by ID* **public void** deleteAppointment (String appointmentId) {  
 **appointments**.remove(appointmentId);  
 }  
}

As you can see, this magnificent work of art showcases succinctness by having a brief comment before each function and by handling the if/else in the constructor as simply and explicitly as possible.

**Reflection**

One of the software testing techniques that I used for the milestones was white box testing. With this technique, I used already existing functions, like the Application class, to test for functionality. Each unit of code was built, then a bunch of test cases were built to check that they behave the way we want them to.

There is a lot of software testing techniques that I didn’t use. One example is boundary value analysis. Errors tend to cluster around boundaries in input, like when the Contact Id’s length is supposed to be less than or equal to ten, the boundaries of length are one and ten. Errors are most likely to occur if you enter some value close to or at the boundaries, in this case that would be an Id length of ten. To test this would just require that I write a few extra tests in each class, but I have not done that yet.

A practical use of the white box testing technique is to test functions after they have been created. The main difference between white box and black box techniques is that black box techniques don’t take the actual implementation into account, while the white box techniques do. Because of this, most white box techniques are meant to be built after the implementation is finished.

An implication of boundary value analysis is that there will be several tests for any function that tests a range of inputs. For some programs, this could be an awful lot of tests, perhaps more than necessary. Supposedly, testing is never truly exhaustive, so I think that testing all the boundaries of only the most suspicious functions may be necessary. Perhaps just testing one boundary per unsuspicious function would be sufficient.

Mindset

To be honest, I don’t think I showed much caution on this project. I built things, then built something to test it, saw it didn’t work, then rebuilt it all over again. It was more of a mindset of persistence. I didn’t appreciate the complexity and interrelationships that I was testing until I ran into issues. For example, by the time I saw that the tests were all failing, I was pretty much finished writing the tests. I don’t think it was very wise to write 5+ tests before trying to run them.

I don’t feel like I did anything to limit bias. I don’t imagine I have any special bias towards my code because I don’t have that much faith in my programming abilities. If something works, then I generally assume it’s good, otherwise it’s bad. I figure code that is concise and well commented is good, but at least the concise part might be a bias of mine. Shorter snippets of code with lots of fancy symbols looks like whoever wrote it knew what they were doing and was being tidy about it. Of course my code isn’t quite like that, but it is my bias to figure code like that (even if it’s undecipherable) is really good.

I suppose another bias I might have is that the code I am currently writing is wrong and old code is right, even if I wrote both the old and the new. I for some reason tend to trust past and future me more than present me. It also just seems logical that the class that seemed to work just fine yesterday is still fine, but the tests I’m writing now are somehow at fault. I definitely changed my test code several times before I ever changed source code.

When you cut corners in software engineering, you tend to accrue technical debt. You create little errors that are hard to pinpoint in the future. Oftentimes, there will be something that works in the moment, but since it wasn’t well thought out, it will cause some compatibility issue in the future that will take way more time to resolve than if you were to just have taken more time in the first place. For example, I could have probably saved some time if I had just built my project properly in the first place. Instead, I spent hours troubleshooting weird, random issues until I found out it was a build error and decided to find a tutor to help me.