Project Two

CS-320 23EW3

Jared Smith

02/19/23

Summary:

While designing the Junit tests for the Contact, Task and Appointment classes, my approach was heavily focused on the requirements. I started developing each class by first writing a to do list, containing the requirements outlined for each class. I then wrote unit tests which tested for the basic functionality of each test, and ensured that they failed as expected. I then proceeded to write out the basic code for each class. As I finished each class, I opened the corresponding unit test class, and wrote out what I viewed as the groups of potential errors for each unit test. I then wrote unit tests that tested each of those potential errors, and ran them to check my code for vulnerabilities. To cite an example: the ID field on the Contact class was not to be more than ten characters long. I started by designing a test that ensured an ID of ten characters was accepted by the method. It failed, because the method was not written yet, so I went ahead and wrote the method. When my initial test passed, I determined the ranges of invalid arguments would be an ID of exactly ten characters, an ID with less than ten characters and an ID with more than ten characters. So I proceeded to design tests made sure that an ID of less than ten characters was accepted by the class, and an ID of more than 10 characters was rejected by the class.

I believe the strategy was very effective at providing a high coverage percentage, but it was not without flaws. The most glaring of which was my inexperience with writing Junit tests, followed by my inexperience with typical error margins. This can be seen in my earliest tests most clearly. If you look at the test I wrote for the Contact class, I wrote the tests in one big messy pile. All three invalid inputs share one test, so if any of them had failed, I would have had to spend a lot of extra time trying to figure out which invalid input was being flagged. I also spent more time than I really needed to rewriting my tests in a way that I thought was clever, but ended up just being very confusing. In my first few tests, I also missed checking the border of each valid input range, which, especially for input validation, is an important principle in testing known as boundary value testing. This is a testing methodology that asserts that most errors in logic happen near the boundaries, in our case the character limit on the ID field of the Contact class. My under limit test was only a few characters, and my over limit test was far above character limit for the Contact class ID. These are issues that are resolved by learning more basic principles behind testing philosophy and with experience. In future tests, I did a much better job at segregating my tests and testing for different kinds of flags. This saved me a lot of time and coding space, and in a real world scenario, would have allowed me to test for more complex and potentially niche errors.

Reflection:

When it came to my mindset behind writing my code and tests for this project, I would say it changed pretty significantly over the course of my work. Specifically, I read about test-driven development, which helped me change my mindset significantly from the first few tests I wrote. Test-driven development is a methodology where you write a simple test that checks for a single function or aspect of the program. You then see that the test fails, because you haven’t written any code yet, and then you write only enough code you need to get the test to pass. Then you kind of repeat this process until you have a whole program. This is a methodology that really helps me a lot, because I tend to be really unfocused when I’m writing code. I used to just let my mind go and write whatever made sense until I had a functioning program. This was not really a great strategy, and I’ve wasted a lot of time backtracking work because I ran down the wrong path for a whole day.

Another important consideration that I think test-driven development helps a lot with is limiting bias. While working on other projects, I’ve noticed I really just want my program to work, and I have a tendency to cut corners that come back to bite later down the projects life cycle. Adopting a careful test-driven development process can really help you stay focused on writing clean, efficient and functional code that will set you up for success later in the project. This discipline helps focus my creative mind and logical thinking skills together to get as much done as possible in the best way possible. Another aspect of test-driven development that really helps with development discipline is the emphasis on refactoring your code to make it accomplish its designed purpose as efficiently as possible. This helps you out in two big ways. First: it helps you design code that is easy to reuse later. A big element of object oriented programming is reusability, and making your pass its unit test in the best way possible can frequently lead you to having a nice piece of code to build from to pass other unit tests. Secondly, it refines your skill as a programmer. I have learned so much shorthand and very excellent simplifications in both C++ and Java while learning to write unit tests, that I am totally sold on test-driven development.

References:

*What is Test Driven Development (TDD)? | Agile Alliance*. (2015, December 6). https://www.agilealliance.org/glossary/tdd/#q=~(infinite~false~filters~(postType~(~'page~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1)

‌

*Boundary Value Testing | What is Boundary Value Testing and Examples*. (2020, April 14). EDUCBA. https://www.educba.com/boundary-value-testing/

‌