4/14/2023

Dalton Walls

**Project Two: Summary and Reflections Report**

This is my summary and reflection on the work I accomplished with the recent mobile application we worked on. There were three specific services included in that app: the contact service, the task service, and the appointment service. Before releasing these products, I used unit testing with JUnit. My approach was aligned to the software requirements to a certain extent, but not all of my unit tests nor program logic stemmed directly from the requirements. Some of them came from ideas I had that I knew would help fulfill the required functionality. One specific example of me designing tests around requirements had to do with input validation. Contact, Service, and Task objects each had certain fields (a lot of them in common with each other) and each of these fields usually had a requirement of being under a certain length or not being null. Therefore, I used throw statements along with unit tests designed to trigger these exceptions. However, there were some things that I knew the program would need to work, but weren’t given directly by the requirements. I would say it was given indirectly in a sense, because although they weren’t mentioned, they were implicit in a functional program involving those requirements. An example of what I’m talking about here is when I made sure to check that an ID (the unique identifier we assigned to each instance of Contact, Task, and Appointment objects) was actually unique (wasn’t already being used by another object) before adding that object to a list. Based on my coverage percentage, I know that my JUnit tests were effective because they covered at least 80% of the tested classes in each instance.

I ensured that my code was technically sound by using Constructors for each of my objects and making sure that those constructors used the given arguments when instantiating a new object. You can see this proper assignment of data in the following lines of code:

**(Appointment.Java)**

**this.appointmentID = id;  
this.appointmentDescription = description;**

**this.appointmentDate = date;**

One example of how I wrote efficient code was when I wrote the Update functions. Each object was supposed to be updatable to a certain extent. This did not include their IDs, which were not supposed to be updatable. When I wrote my Update functions, I took in an ID so that I would know which object was being updated. However, as you can see, that when the object was found and updated, I did not include a line to update the ID, since that was not required and would have been extra code:

**(AppointmentService.Java)**

**appointment = appointmentList.get(id);**

**appointment.SetDescription(description);**

**appointment.SetDate(date);**

One software testing technique that I used in this project was Unit Testing. Unit Testing involves testing specific lines of code or functions by themselves to see if they exhibit the expected behavior (which is usually written separately at the beginning and then cross-referenced later). One of the unit tests I did was to make sure that a null field was throwing an exception for the objects.

One testing technique that I did not employ for this project was analyzing the runtime performance. Analyzing the performance of software involves seeing how fast it runs with different input sizes, figuring out its runtime complexity, and then possible needing to come up with a more efficient algorithm if it is deemed too slow.

A practical use and implication for analyzing a software’s performance is when a program is running too slow during testing, the tester’s could analyze the lines of code to see where it is taking too long. For unit testing, these are used all the time in software development, sometimes even quickly be the developers themselves to test their functions before moving on to writing a new one.

I had to take on a testing mindset while working on this project. I employed caution in the sense that I did not just start writing code that I thought would work well for the project, instead, I carefully analyzed the requirements first before I started taking my first steps. It was important to understand how all the different components of the code would interact with and work with other parts. This is because in order to integrate these components together and form a complete program I would need to understand how they are to work. One specific example of this was how the Service objects were responsible for Adding, Deleting, and Updating objects. Because these classes would be responsible for organizing, storing, and updating these objects I decided to use a data structure inside of the Service classes to store the objects (hash tables). This component would then be responsible for storing the objects as well, instead of making the hash table inside of the Class Objects themselves, which wouldn’t have made as much sense (having them store themselves). It made more sense to have the Service objects do this which came from my examination of all the components’ relationships with each other.

I tried to limit bias when testing my code by not avoiding tests that I knew would fail. For example, I was having trouble testing that an Appointment date wasn’t before the current date. This trouble came from the fact that in Java’s Date class, years weren’t represented the same way that we represent them in every day life. I had to read the documentation closely to overcome this obstacle. They do say that developer’s shouldn’t test their own code to avoid bias, so I imagine that it would be a concern if they were to do that.

Being disciplined is also important when developing software for much the same reason as I pointed out with my bias example. Had I not had the discipline to keep trying to figure out why I wasn’t getting a thrown exception for a previous date, then that functionality would not have made it into the final product (despite being a requirement). I would have been cutting corners if I had just left it the way it was, even if someone else along the line wouldn’t have caught my mistake. I plan to avoid technical debt by writing things as accurately as I can at the beginning instead of rushing through and having to refactor later. This includes testing as I go and reviewing each line of code and function.