Maxwell Sheehan

6/19/25

CS 320

For project one, and the week leading up to it, we implemented Junit based testing for the three core services, ContactService, Task Service, and AppointmentService. The unit testing I implemented always had a few core design pieces when they were being made. Validating the boundary of the unit, by testing length constraints of the variables as set out by the customer. Negative testing, by deliberately testing invalid inputs or values, as well as duplicate values that couldn't have a shared value. Behavioral testing, by confirming when an object changed either through deletion, adding, or updating of a value. And lastly, we wanted to ensure that each test was independent, this separation of each feature and its tests allows better scalability, and keeps the code as modular as possible. We can see this within specific code examples, within different core services. Our arguments and test cases did a great job of logically covering a high percentage of errors, not only do we validate objects on their creation, but we also validate them when they are edited. The code snippets below are just some examples of how we reject creating an object that doesn't meet the software requirements, in separate services. Code is most efficient by making sure it is correct when it is created, and these are both examples of how ensuring we create a valid object increases overall coverage, and lowers potential realm for error.

```
assertThrows(IllegalArgumentException.class, () ->
contactService.addContact(duplicateContact));
ssertThrows(IllegalArgumentException.class, () ->
appointmentService.addAppointment(pastAppointment));
```

Reflecting on more code that I used, I think my practical experience, and effectiveness in

covering many instances shows when looking at null values. Throughout the errors and trying to ‘catch’ mistakes null values are a constant example of things that can cause errors in code, so creating specific unit testing that catched error is a great way to handle edge cases.

assertThrows(IllegalArgumentException.class, () -> taskService.addTask(null));

Within specific techniques used, this is an example of Black-box testing, as I focused on the input/ output of the software without knowledge of how it will be fully implemented. We also used boundary value testing, this was shown in the appointment creation, making sure to manage the length a variable should have, as well as validating the dates in use of the appointments.

I did not, and was not able to test the integration of each feature fully, System testing in the form of end-to-end testing of the entire mobile application was out of scope for this project. Practically speaking the unit testing and avoidance of collision was the most effective use of time in the early stages of our development to avoid bugs, and create clear structured code. I was very defensive in making the back bones of the code, to try to make the software as scalable as possible, and reduce the surface area of error that the design could evolve into. We see this in specific arguments, throwing illegal examples of someone trying to edit an object, we don’t want simple misinputs, or malicious attacks to damage our code base. I think this caution shows, good discipline as a developer, we need to think of all possible inputs when designing somethings sound, and although most people would never try to damage or ruin a service, sometimes it’s not maliciousness that harms us, but simple incorrect inputs.

assertThrows(IllegalArgumentException.class, () -> contactService.deleteContact("999"));

Caution shows itself in many ways as a developer I functioned as combined dev and tester for this assignment, so sometimes it’s best to show the code works and handles errors,

instead of just assuming and confidently assuring it does. We did this in meticulously testing every instance of creating an object, deleting an object, and updating an object. This cautious testing is very robust, and is a deliberate effort to make sure that we are testing potential edge cases that could occur within our software creation. I think that this creation of our early stage code is a great tool in making our code as scalable as possible, and in the future I will do my best to try and improve further from this, the modular nature of the unit testing created will allow us to further refine if we are seeing a pattern of errors slipping through. And I think this further speaks to the caution of the design and testing. Setting yourself up for success in this way helps eliminate the bias and pride you might feel in the creation of your code. I previously mentioned how it's better to show your testing results instead of assuming them, I think this is a great way of removing personal bias and ego from what you're developing that objectively increases code quality.