Jordan Schlinger

2/20/25

CS-320

Project 2: Summary and Reflection

**Summary**

Unit Testing for the Three Combined Projects

i. **Alignment to Software Requirements:** My approach to unit testing for the Contact Service, Task Service, and AppointmentService was focused on the software requirements. Frustratingly I made some mistakes and had some overlooks- because of this despite my maximum possible increase being ~5/150 points I resubmitted it as that is what I would do in the real world- if something is inadequate it must be corrected. In the first go around most tests validated the constraints imposed by the requirements, but there were mistakes in the contact class- specifically problems with the null value check and unique Contact IDs. In all other areas this was implemented- such as unique and immutable task IDs, field length restrictions, and it allowed updates only for certain fields. One example comes to mind- in the Task Service I implemented JUnit tests that ensured that task IDs were at most 10 characters, names were at most 20 characters, and descriptions did not exceed 50 characters, exactly as ordered. In this way I ensured that the software fit the strict requirements given to us by the instructions, and if this had been the real world would have ensured that the client was happy.

ii. **Effectiveness of JUnit Tests:** JUnit tests were effective in almost all of the required areas of the service. Test cases included valid inputs, boundary values, and negative conditions such as attempting to update an immutable ID. As mentioned I did have problems with ContactService and Contact classes- but this are now rectified. Coverage analysis indicated that the tests exercised most critical paths, with coverage being high but not quite at the required level (76.8% as opposed to the requisite 80%). This means that unexpected problems will be unlikely to pop up.

b. **Junit Test Writing**

i. **Technical Soundness:** One thing I implemented was exception handling. It needed to be relevant, it needed to tackle any problems raised, and it needed to be effective. Here is an example could do (did if I recall correctly) for exceptionally long names.

Exception exception = assertThrows(IllegalArgumentException.class, () ->

new Task("12345", "ThisNameIsWayTooLongToBeValid", "Valid Description"));

assertTrue(exception.getMessage().contains("Name must be at most 20 characters."));

This met requirements and threw up the exception whenever necessary.

ii. **Efficiency:** Efficiency is always a goal- efficiency in time, which I had next to none of this semester, and efficiency in the code itself to prevent it from becoming spaghetti code. One method I used to do this was to keep the code as brief as possible- another method was by grouping similar things together. This means when reviewing code I could rush to the areas that need to be corrected.

**2. Reflections**

a. **Testing Techniques**

i. **Techniques Used:**

* **Boundary Value Testing**: Boundary value was something that, if I remember correctly, I had had trouble with previously. I had tested to make sure that often things were not too long- now I made sure they were not to short OR long.
* **Negative Testing**: Another thing I failed with the first time I did contact service, and something corrected- it makes sure the correct exceptions are thrown.
* **Unit Testing with JUnit Assertions**: Confirmed that all three worked correctly under all expected and even unexpected conditions. JUnit testing covered the vast majority of the possible issues I could run into- and my coverage was pretty darn strong!
* **Static Testing**:I literally just looked over the code to look for obvious flaws- something that caught quite a bit.

ii. **Other Techniques Not Used:**

* **Integration Testing**: To put it simply each group was graded by itself. So there was no need to test the integration as a whole.
* **Performance Testing**: Execution time was not graded or looked at- because of this I saw no reason to focus on it. I still made my code efficient of course- but there was no bar or specific tests I did for it.

iii. **Practical Uses and Implications:**

* Boundary value testing is useful in ensuring that software adheres to strict input constraints while negative testing is crucial for preventing security weaknesses and ensuring data integrity- simply put combining them gives a far better ultimate work product.
* Unit testing is great- it ensures each individual item works! However, if this was a real world item we would ideally want to do many other types of testing afterwards as well.

b. **Mindset**

I had a few mindsets while working on this. The first was **Caution** was routinely exercised- I wanted to make sure every single thing worked (and now it does!). Caution is key and critical- you don’t want to be to overeager but rather take things step by step. The next was to watch out for **Bias:** Bias is always present when working on my own code, same for anyone. It is easy, especially in static testing, to just go quickly. To address this, I followed the rubric and paid attention to everything necessary- I also looked over my previous mistakes all to try and get my code on point. Peer reviews would have been nice to eliminate bias- however that was impossible. Nonetheless I did look at, for instance, GitHub- not to copy code but rather to compare how they approached issues to get new insights. Finally, I had a **Disciplined** mindset. I had to go piece by piece. I also had, as mentioned, many things going on. Thus, I tried to limit myself and go slowly and effectively as opposed to just rushing- this ensured I got what I needed. Writing comprehensive tests ensured robustness and maintainability, reducing the likelihood of future defects. In real life especially this is important- after all look at how many tragedies have happened due to poor code!

Works Cited

Coder Academy. (n.d.). *When coding goes wrong*. Medium. <https://medium.com/@coderacademy/when-coding-goes-wrong-e46d84c6565f>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS, The Chartered Institute for IT.