**Summary and Reflections**

**Summary**

***To what extent was your testing approach aligned with the software requirements? Support your claims with specific evidence.***

My testing approach aligned with the software requirements by testing each function with valid and invalid arguments. If a field, i.e., Name, could accept up to 10 characters, but not more, I wrote a test passing an argument that was null, a test with less than 10 characters, a test for exactly 10 characters, and a test for over the limit of 10 characters. Doing this provides a means of testing if a user happens to enter a small string, an exact-sized string, or a string that should be flagged by an exception within the logic. This also means we can ensure that IF the spec specifies a name of up to 10 characters is acceptable, then 10 characters will be accepted, and won't cause an issue with the logic.

***Defend the overall quality of your JUnit tests for the contact service and task service. In other words, how do you know that your JUnit tests were effective based on coverage percentage?***

As stated above, my unit tests would test less than, exact sizing, oversize, and null. This would ideally catch most use cases that the customer may experience with Strings. When it comes to percentage of coverage, Task.Java has 100% coverage, TaskService.Java has 100% coverage, Contact.Java has 63% coverage, and ContactService.Java has 100% coverage. Contact.Java has a lower percentage coverage, as the Update methods of the script are tested as part of ContactServiceTest (covers 56.4%, meaning there is some overlap in the tests).

Overall, I would say my tests are very accurate on coverage, and allow us to surmise that the code is testing all aspects that the customer is hoping the code to follow.

***How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.***

I tested my code to be technically sound by testing more use cases than necessary. Testing for null, a smaller value than the limit, a value that is exactly at the upper limit, and a value that is over the bounds of the limit. If the data types had been numeric, it would have also helped to test with negatives, or under the lower bounds. I should also employ tests for special characters to ensure that no special characters can break the underlying logic.

***How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.***

For efficiency, I mainly just relied on the execution times of each test, shooting for as little of a footprint as possible. There are a few tests that do take longer to run, namely newShortTaskNameTest() in TaskServiceTest, that could definitely be made more optimal (It is slow due to the need to run SearchForTask, which could probably be sped up with a HashMap or Data Structure that is more efficient than looping all records stored in the TaskList).

**Reflection**

**Testing Techniques**

***What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.***

The method of testing used for each milestone was that of White-box testing. This is because I had access to the source code, which allowed me to see how the code functioned logically and write tests that acted on the structure and fail-safes written into the source. This ensures that there is fully functional code by giving me a means of comparing my test coverage against each specific file and determining how many of the cases, conditions, and methods I was testing (always aiming for 100% coverage). This also gives a means to check for repetitive code within the source and know if we are running redundant logic in the functions.

I also performed manual testing by reading over the logic and automated testing via the implementation of Unit Tests.

***What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.***

I did not use decision tables, as there were potentially too many variables to account for in this instance. Decision tables are tables that look at each possible scenario and give the expected behavior based on this combination of scenarios. I found this would not be suitable, as we are working with data that is not as cut and dry as True or False. We have Dates that could change output by a single millisecond being off of what is expected.

For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.

Decision Tables are akin to Cause and cause-and-effect matrices, which based on a given combination of scenarios, alters the expected output of the table. Given that it is a Black Box Testing Method, the testers would need to pre-determine all given possibilities and note them and the expected behavior with that given combination. This form of testing is generally better for simpler logic checks, i.e. User Authentication, where you are just checking if the Username matches one in a DB, and that the password matches that user. Or User creation on a password, where you need a password to contain ‘x’ amount of special characters, upper/lower, and numerics, and also a verify password on second input matches first input. This is not an ideal method for something with infinite possibilities, such as when dates, names, and IDs could drastically vary.

**Mindset**

***Assess the mindset that you adopted working on this project. To what extent did you employ caution in acting as a software tester? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.***

The mindset I adopted while working on this project was one closer to that of an end-user than what I was used to (a Developer mindset). I usually am pretty good at testing code (manually) and hit my code from different angles, however writing Unit Tests forced me to think of other possibilities, rather than brute-forcing tests and hoping I caught everything. There was more of a statistician approach by trying to ensure that my tests combined would hit 100% code coverage, whilst still maintaining accurate results.

I had to employ more caution, as it was very easy to write a test that would pass, but to actually write a test that ensured accuracy in the result, without being able to actually see the data being entered (i.e. in a field in the GUI) and testing from an automated code-based approach was far different for me. Writing the tests specifically for the Service (where it would make method calls to the base classes) makes you appreciate the interoperability between the objects, as you can catch a lot of issues in the Service calls before they are taken in by the base class. This, however, can make testing lazier, as you rely on the service to catch the issue. This made me appreciate the safety net that is provided with handler classes but also made me more cautious in my tests because I need to still remember to write tests for the cases that the service would check, to ensure that the logic in the service is sound (and would help catch issues in future patches).

***Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.***

This is common in the Game Development profession. So, I will use this as an example. Generally, when a video game is written, it is tested (hopefully) from the start of a playable build until release, however, as time passes, the testing team gets very good at the game, from all the practice. This inevitably leads to difficulty being increased due to the ease the dev/test teams feel (a bias created by them testing for so long and getting practice). Then when the game ships, the users may feel the difficulty is unfair, as they have not been practicing like the dev team for the past 2-4 years of development.

With software (non-games), the developer is probably less likely to make the mistakes that they wrote the logic for, which can cause a bias that can cause bugs/defects to go uncaught. This is why it is imperative that software is tested from other viewpoints and in other scenarios (within reason, as time is not necessarily cheap). One way I ensure that I limit bias in my review of code that I have personally written, is I either have someone else demo the software (if possible), while I watch silently, and see how they go through the product, and observe for issues.

If more time is available, I will test as I write, and then when I call it a day, I try not to think about the code. The next morning, I will test first thing in the morning (preferably when I am still out of it), this provides me a means to test without being at peak performance and hit the software differently than I normally would, which allows me to observe other functionality in different scenarios.

Lastly, I just try to act like an end-user with 0 knowledge of the underlying code. Like a child, I play pretend, and act oblivious to the underlying framework, and just use the software as I would if I were a normal user. It's surprising how many issues you can find by just disconnecting yourself from the systems you wrote.

***Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.***

It is extremely important to be disciplined in your commitment to quality as a software engineer. Your face and name may not always be prominently displayed on the product you are designing, but someone’s name is. Their business reputation is on the line. They rely on you to solve their business needs, and if you fail to do so, within reason, this will hurt your reputation as a reputable engineer, and in this world, you need recommendations from reputable sources (people and businesses). Harming a business's reputation due to poor code/testing is very likely to indirectly hurt your reputation and ability to find work. So it is always important to commit to quality in both code and testing.

Cutting corners is almost never a good idea, and depending on what the software application will be, could lead to financial loss or worse, negligence, and could lead to a loss of life. Think about the software that is written to control any important system in a car (airbags, ABS, or fuel injection/EV). You would not want to drive a car that has software issues that cause one of these systems to be problematic (exploding airbags, premature inflation, ABS lockups mid-drive, too much fuel being injected into the engine and causing combustion that results in explosions in the engine bay, or EV batteries that can bypass the voltage regulator and overheat or overcap and explode). You also should not want your name plastered on one of those above issues. So DO NOT cut corners, and always write quality code and perform quality tests.

Avoiding technical debt is a harder thing to do, especially when you commit to quality code and testing. The best way to reduce technical debt is to ensure you have proper documentation on the user case of a requested feature and understand what its needs are. You should know how the customer or user intends to use the feature before writing it to reduce the number of rewrites that will need to be done. You should also try and demo to the customer or user throughout development to get their feedback. This way, if the code deviates from what they expected, it's easier to fix the code now, rather than re-writing days or weeks' worth of progress. A third way to cut down on technical debt is just to test often as a software developer. If you are writing the feature, it is easier for you to fix the code before it deploys or moves to QA, rather than waiting for QA to test, you fix it, then they test again, or worse, it goes to preview or production and you need a Change Request or Patch to be entered, approved, written, tested, then redeployed. Testing and writing quality code is one of the best ways to cut down on technical debt, and cutting corners should not be a solution.