**Reflection on Software Testing Techniques for Milestone**

For the milestones, I employed unit testing and boundary value analysis. Unit testing focused on verifying the functionality of individual components, such as methods in contact and task services. These tests ensured that each function worked as intended in isolation, identifying errors early in development. Boundary value analysis tested edge cases, such as maximum and minimum input limits, ensuring the system handled extremes effectively. A great example would be Module 4, where I tested input validation for task descriptions limited to 50 characters to ensure compliance with specifications.

Techniques such as integration testing and regression testing were not applied during the milestones. Integration testing ensures that different modules or components of the software work together as expected, focusing on data flow and interaction. Regression testing checks whether new changes introduce errors in previously working features. These techniques are typically applied in later stages of the development process.

Unit Testing is practical for projects requiring high accuracy in individual components, such as financial or medical software. Boundary Value Analysis is useful for input-heavy applications where edge cases, such as form validation systems, could lead to failures. Integration Testing is essential for projects involving multiple interacting components, such as e-commerce platforms. Regression Testing is critical for iterative development models to ensure stability after updates or patches.

**Part B**

My mindset as a software tester in this project was predominantly shaped by the need for caution and diligence. As a software tester, I approached the task with a good mindset of discovering potential errors early and ensuring that everything functioned as needed. Attention was especially needed because even small mistakes in one part would propagate and result in serious issues downstream, particularly in large systems. As Myers (2004) stated, "tests provide a critical safety net against unforeseen consequences.". Therefore, I employed caution when designing my tests to not overlook edge cases and to ensure that I was testing components in isolation before they had touched the rest of the system.

Realizing the complexity and intertwinement of the code was at the core of how I went about testing. Software systems, as they grow larger, are intricate webs of modules interacting. Being aware of these interrelationships enabled me to anticipate any issues that would be created by combining the different modules. For instance, when testing the task service, I had to remember how an update to a task's information would affect the contact service (e.g., if task deadlines or status affected contact notifications). This intertwinement accentuates the reason caution was needed while testing—ensuring that changes did not interfere with other functions was critical.

One example of where I exercised caution was in the area of input validation. When testing the task name field using boundary value analysis (i.e., ensuring that it was limited to 50 characters), I realized that the task naming convention could impact other parts of the system, such as reports or search functionality. I thus performed extra testing to ensure that input limitation did not sacrifice functionality in other parts of the application.

In so far as minimizing bias is concerned, I attempted to look at the code review process from an objective perspective, taking special care not to overlook or favor certain code patterns or elements simply because they were familiar or authored by someone with whom I was working in close collaboration in a real-life scenario. I was cautious to note that the bias could happen both in a positive and negative way, especially in a situation where I could have been reviewing my own code. As an instance, when I was reviewing my own work, I attempted to set aside any initial attachment that I had to the solution that I developed. As Kaner (2002) states, "it's hard to objectively look at our own code since we tend to get too close to it, and it becomes more difficult to see the problems with it". To get around this, I performed code reviews after not having looked at the code for some time so that I could come back with a fresh mind.

A specific example of minimizing bias was when I was reviewing the task service. It was tempting to disregard potential errors in my logic because I had written the code myself and was convinced that it functioned. However, through unit testing, I was able to avoid the bias that would have led to failing to detect likely edge cases. Another instance that could have helped in this case is peer reviews as the peer could provide some test cases with unexpected input formats, which allowed me to detect a subtle bug that I had not previously noticed.

If I had been asked to test my own code, I could have fallen prey to the "confirmation bias," in that I could have looked for tests that would only demonstrate my code was behaving as I intended. By adopting an attitude of testing failure modes, though, I ensured that I was forcing my code to behave in ways I had not anticipated.

Discipline in maintaining high levels of quality is key to long-term software success. In my testing and coding, I was careful to follow the best practices, regardless of time constraints or pressure for on-time delivery. Shortcuts in software development can lead to technical debt that, if not curbed, grows over time and can significantly slow future development velocity, maintenance costs, and system stability. As Jones (2002) explains, "the buildup of technical debt can have a snowball effect, causing future bug fixes and updates to become increasingly harder".

One specific example of this discipline was ensuring my unit tests were complete and followed the given requirements, even when it would possibly have been quicker to write simpler, less comprehensive tests. For instance, rather than merely testing that the name field of a task would take 50 characters, I tested boundary cases where the input exceeded the limit or under the limit if applicable. This proactive step ensured that any issues with the code were caught before they escalated.

As a practitioner, I will avoid technical debt by taking the time to write clean and readable code and prioritize refactoring when necessary. For example, rather than presenting "quick fixes" that will last for the short term, I'll go for more concrete solutions that ensure the long-term scalability and sustainability of the code. Additionally, by engaging in regular code reviews and revisiting the code written previously as the project progresses, I can see where technical debt is accumulating and addressing it prior to its accumulation.

In brief, a quality mindset that is disciplined, bias-free, and knowledgeable about the potential consequences of shortcutting is required in software development. Maintaining high standards ensures that the software will remain stable, maintainable, and scalable in the long term.

**References:**

Beizer, B. (1990). Software Testing Techniques. Van Nostrand Reinhold.

Kaner, C. (2002). Testing Computer Software. Wiley  
.

Jones, C. (2002). Software Engineering Best Practices: Lessons from Successful Projects in the   
Top 10 Software Companies. McGraw-Hill.

Myers, G. (2004). The Art of Software Testing. Wiley.