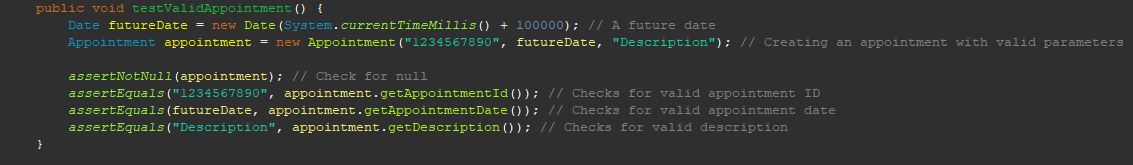
Grand Strand Systems: Summary and Reflections

My unit testing approach was fully aligned with the software requirements due to the limited and clear specifications provided by the client. I ensured that my tests met all the requirements, such as the immutability of IDs once they have been set. Additionally, to fulfill the requirement for unique IDs, I implemented a check before assigning a new ID to verify that the randomly generated ID does not match any previously existing ID.

The overall quality of my JUnit tests is demonstrated by the fact that they achieved 100% coverage of the software. I meticulously tested every function to ensure all requirements were met, including constraints on variable length, mutability, and non-null values. For the ID generation feature, I utilized repeated tests to confirm that the IDs generated were indeed unique. This thorough testing approach provides confidence in the effectiveness and reliability of my JUnit tests.

To ensure my code was technically sound, I wrote JUnit tests with variables set according to all the requirements. This approach ensures technical soundness because I only need to call these variables in my tests, rather than writing the values directly into each test. This method also facilitates quick updates to the tests if the requirements change in the future, allowing for easy adjustments to variable lengths or other constraints.

Here is a sample of my code illustrating this approach:



A screen shot of a computer code

Description automatically generated

By defining variables that meet the requirements from the outset, I ensure that my tests are both accurate and maintainable. This approach highlights the technical robustness of my code, as it promotes consistency and ease of updates across all tests.

To ensure my code was efficient, I utilized repeated tests. Instead of writing multiple tests to check for unique IDs, I used a repeated test, which makes the code more efficient and less cluttered. This approach reduces redundancy and keeps the codebase clean and maintainable.

Here is a sample of my code illustrating this approach:

A computer screen shot of text

Description automatically generated

For the project, I employed unit testing to ensure that all given requirements were able to be met. It helped break down everything and verifying that each function worked as intended. This was very crucial to finding and fixing any issues. Unit testing is practical for ensuring that individual units of software function as expected. It is an essential first line of testing that allows developers to verify code functionality incrementally as they build the project.

Integration testing examines how different components work together to ensure they function as intended within the system. It is particularly useful for identifying defects that occur when modules are integrated, facilitating early detection and resolution of issues. Integration testing is crucial for identifying defects early when two or more modules are integrated. By detecting issues early, integration testing helps save time and money, making it easier to address problems before they escalate.

Regression testing involves testing the system after changes have been made to ensure that no new bugs have been introduced. It is essential for maintaining system stability and functionality over time, especially after updates or enhancements. Regression testing is practical for maintaining system integrity after modifications. It ensures that new changes do not introduce new bugs, thereby preserving the system’s reliability and performance over time.

System testing is a comprehensive technique that tests the entire system to ensure it meets all requirements. It verifies the system's overall behavior and performance, ensuring that it functions correctly in a real-world environment. System testing is vital for validating the complete system against the specified requirements. It ensures that the system operates correctly in a production-like environment, providing confidence that it will perform well for end-users.

In working on this project, I adopted a cautious mindset to an extent appropriate for the simplicity of the requirements. Given the straightforward nature of the project, I exercised caution primarily to avoid over-testing. Appreciating the complexity and interrelationships of the code was crucial because more complex code can easily lead to over-testing or the unintentional introduction of bugs through tests. For example, a test designed to check one function might inadvertently affect another due to interrelated components.

To limit bias in my code review, I focused on maintaining an objective perspective. The simplicity of the code made it easier to ensure that the review was accurate and unbiased. However, with more complex code, ego can interfere, leading to biased reviews. This could result in unintentionally overlooking flaws due to a misplaced belief that the tests cover all sections adequately. For instance, a developer might assume their code is flawless and not thoroughly test a section, thus missing potential bugs.

Being disciplined in maintaining high-quality standards is paramount in software engineering. Quality ensures a better user experience, enhances the reputation with clients, and ultimately saves time and money by avoiding the need to redo sub-par work. Cutting corners can lead to code defects that cause security or user experience issues, which can sometimes be severe. For example, in medical software, cutting corners could result in a security breach, leading to costly legal action and the need to rework the code. This not only incurs financial costs but also damages the reputation for quality work.

To avoid such issues, I plan to keep my ego out of the equation and remain objective when reviewing my work. Additionally, working with a communicative and cooperative team is crucial. A good team can make a significant difference in ensuring quality code, especially in systems with multiple authors.

Works Cited

*System Integration Testing: A Comprehensive Guide With Examples And Best Practices*. (n.d.). Www.lambdatest.com. <https://www.lambdatest.com/learning-hub/system-integration-testing>

*System Testing vs Integration Testing: Differences and Examples*. (n.d.). Ronwelldigital.com. https://ronwelldigital.com/insight/blog/system-testing-vs-integration-testing

‌ *Software Testing Techniques: Explained with Examples*. (n.d.). BrowserStack. https://www.browserstack.com/guide/software-testing-techniques

baeldung. (2016, November 19). *A Guide to JUnit 5 | Baeldung*. Www.baeldung.com. https://www.baeldung.com/junit-5

‌ Lars Vogel (c) 2007, 2019 vogella GmbH. (2019). *Unit Testing with JUnit - Tutorial*. Vogella.com. https://www.vogella.com/tutorials/JUnit/article.html

‌ Bechtold, S., Brannen, S., Link, J., Merdes, M., Philipp, M., de Rancourt, J., & Stein, C. (n.d.). *JUnit 5 User Guide*. Junit.org. https://junit.org/junit5/docs/current/user-guide/

‌