**Project Two: Software Testing and Strategies**

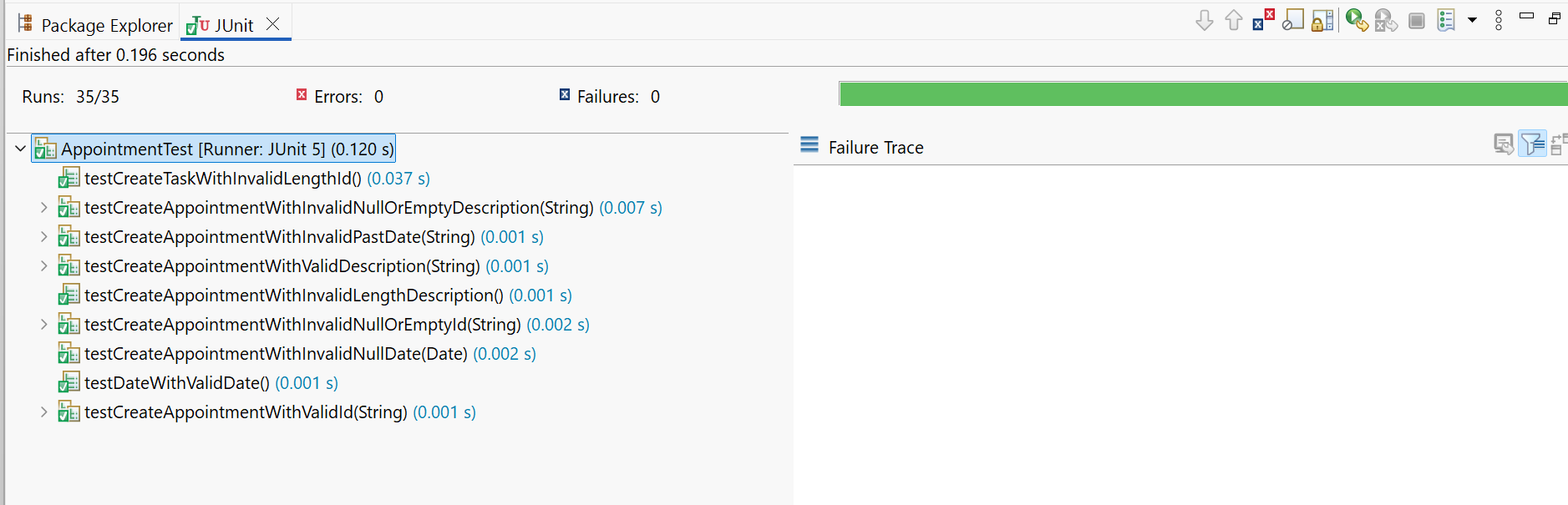
While developing the software, I prioritized the requirements above all else when coding. The contact class specified that both the first and last names must not be null, so I ensured that no more than 10 characters could be entered. In the Contact test, the JUnit executed a test to verify that the input did not exceed 10 characters.

A screen shot of a computer code

AI-generated content may be incorrect.

The task class ensured that both the first and last name were limited to a maximum of 10 characters. If a name exceeded 10 characters, the JUnit test would trigger an argument statement.

The materials in every module significantly helped increase the coverage percentage, leading to a consistent improvement in the quality of my JUnit tests each week. In contrast to the coverage percentage of the contact test, the task test exhibited a comparatively low coverage percentage, while the service test achieved the highest overall coverage percentage. Whenever a positive coverage % appeared, I was confident that the JUnit tests I executed had effectively covered the majority of the code's functions.

I made sure my code was both technically sound and efficient by conducting tests, checking for errors and coverage, retesting, and continuously repeating this process until I achieved the required results. Using JUnit testing, I confirmed that the expected outputs were accurate by employing a mix of methods and JUnit assertions. While spending more time on the Task Service could have improved the coverage further, but the requirements were still satisfied. The Appointment Service package presented the least issues in project one, though still not perfect.

Regularly reviewing the code for mistakes and opportunities for improvement allowed me to enhance the code to its fullest potential. The knowledge gained from previous classes contributed to the code's development, but recalling many concepts can be challenging, so watching videos and learning from them helps refresh my memory. I aimed to keep everything organized and straightforward in the codes I created each time. Often, experimenting through trial and error and observing the outcomes helped me grasp the process, and frequent testing proved to be the most effective approach.

The software testing methods I used in this project included JUnit testing and static testing. Throughout the project, some modules required either Junit testing, static testing, or both. Static testing involves reviewing the code and comparing it to the defined requirements to identify any bugs or errors. If a JUnit test fails, I would then use static testing to help pinpoint and fix the problematic code. My goal was to ensure the logic of the class methods aligned with the specified requirements. While developing the JUnit tests, I would revisit the defined requirements and examine the methods to look for any errors or bugs I might have overlooked during coding. Occasionally, a JUnit test would fail, prompting me to review the code to determine the issue. Junit testing is great for regression prevention and facilitates refactoring. Static testing is great for early bug detection and fosters team collaboration (Paganini, L., 2022).

For this project, many software testing techniques were not implemented in this

milestone. Some of the techniques I didn’t use were reliability testing and performance testing. Reliability testing evaluates a system's ability to perform consistently and without failure over a specific length of time with specific conditions. Performance testing is the practice of evaluating how a system performs in terms of responsiveness and stability under a particular workload. Performance tests are typically executed to examine speed, robustness, reliability, and application size (Powell, P., 2025). Both are similar, but reliability testing is for consistency over a long time, and performance testing is for immediate actions such as responsiveness.

Working with Junit and testing was initially a foreign concept when this term started. One significant change I needed to make to ensure the implementation of quality tests was to think in terms of the bigger picture. When dealing with code, it’s easy to become engrossed in the details. By focusing on the overall picture, I could verify that the code was functioning correctly. For instance, while handling user input in the contact section, I could visualize how the code should operate. As a software tester, there are numerous factors to consider. A fundamental grasp of how each piece of code interacts is essential. Recognizing that each test code works in conjunction with others is vital to ensure the code performs as intended.

I attempted to minimize bias in my code by using a checklist that addressed all of the client’s requirements, followed by thorough testing to confirm that those requirements were met. I can see how bias could be a concern when testing your own code. It’s easy to get caught up in the belief that your work is error-free, which can result in neglecting testing and ultimately wasting time. This is why testers and code writers are advised against criticizing their own code. This allows fresh perspectives to evaluate the code during both testing and writing phases. Adopting distinct styles can help meet industry best practices and ensure the code is technically robust (Gilpin, K. 2021).

Above all else, it is crucial to maintain discipline while coding. Taking shortcuts in coding can result in a program that is hard to test, leading to overlooked errors and potential threats to the system. Additionally, it may create code that is challenging to update and modify as the system evolves. Having error-free code is vital to ensure that sensitive information is not disclosed to unauthorized individuals, that financial transactions are conducted over secure connections, and that systems providing essential services (such as power, water, and telecommunications) remain operational. I intend to prevent technical debt by adhering to industry best practices and being open to learning the specific best practices of my employer, even if they differ from the broader industry standards.

Sources

Powell, P., & Smalley, I. (2025, July 3). *Performance Testing*. Ibm.com. <https://www.ibm.com/think/topics/performance-testing>

Paganini, L. (2022, July 11). *Static, Unit, Integration, and End-to-End Tests Explained*. Medium. <https://medium.com/@lucas.paganini/static-unit-integration-and-end-to-end-tests-explained-f87a0ac40ca5>

Gilpin, K. (2021, April). *Beginners mind - How to effectively review your own code*. DEV Community. https://dev.to/appmap/beginners-mind-how-to-effectively-review-your-own-code-1h8o