This past few weeks, our focus has been on unit testing, an experience that significantly enhanced my understanding of testing within an IDE. I've found it quite intuitive to pinpoint the testable elements of a program; they often resemble decision trees, where each "node" or branch necessitates testing to confirm both correct and incorrect responses. Achieving 100% test coverage in this context seems straightforward, as it implies validating every possible branch of the decision tree.

My testing approach was directly aligned with the software requirements, which were clearly defined from the project's start. This clarity allowed for a direct mapping between the programming elements and the specified needs. The overall quality and effectiveness of my JUnit tests for both the contact and task services are evident in achieving 98% test coverage for the Task, TaskService, Contact, and ContactService classes. This high coverage percentage, confirmed by visual review during coverage testing where untestable areas were highlighted in red, indicates that virtually all code paths were exercised. For me, 100% coverage in this scenario means every "branch" of the decision tree-like program logic was traversed, ensuring that the system behaves as expected under various conditions and that all requirements have been adequately addressed in the test cases.

Ensuring the technical soundness of the code involved addressing crucial aspects identified through feedback and self-correction. A significant learning point from the first assignment was the incorrect testing of object equivalence in an AddContact method, where I was inadvertently comparing pointers rather than object content. This was rectified in the second assignment by overriding the equals method in the Task class, as demonstrated by the ability to correctly compare two separate Task instances for equality in tests like the one in TaskServiceTest (lines 34-38). Here, a new task was created with a unique ID, name, and description, then added to the HashMap which is a built-in storage structure. Subsequently, getTask was used with the same taskId to retrieve the details, which were then compared to an expectedTask using assertEquals. This process directly validates that the addTask functionality, as defined by the requirements, correctly adds and allows for retrieval of a task. This crucial step ensured that object equality comparisons were technically sound, validating that the content of the tasks matched, not just their memory addresses. Furthermore, adhering to naming conventions like testFunctionality\_withCondition\_expectedResult improved the clarity and maintainability of the tests, making it easier for me and future programmers to tell what each test was verifying (and importantly what still needed to be tested).

Efficiency was a constant consideration throughout development, primarily by reducing redundancy and optimizing data structures. The use of constants, as seen in lines 12-26 of the TaskTest class, significantly contributes to code efficiency by making the code cleaner, more maintainable, and less error prone. This practice reduces the amount of code that needs to be written and reviewed. Beyond this, the choice of a HashMap as the underlying data structure for storing tasks and contacts within the services was a deliberate decision for efficiency. HashMaps offer efficient, fast access and retrieval of data compared to other data structures like arrays or linked lists, which would typically result in slower operations for retrieval.

Software testing techniques can be divided in several ways: static and dynamic, black box and white box, and functional and non-functional. During the past few weeks we have employed multiple software testing techniques to complete each milestone.

Since we know the internal workings of the program we operate in a white box testing environment. We are able to target our testing based on the known decision coverage of the program. For example, the appointmentDescription had a requirement to be less than 50 characters, which can be tested by proving that for any description of length less than 50, the appointmentDescription is valid and created and that for any description of length greater than 50 characters would throw an error. For each of the requirements given to us at the start of our project we are able to delineate a list of conditions that should result in true or false results and then we can test for each of those categories. The Eclipse IDE has functionality with JUnit testing which allows us to test for coverage of the program. On the other hand we did not employ black box testing techniques during these few weeks because we did not have a finished user interface that would allow us to conduct testing with complete ignorance of the internal workings. Black box testing works well during User Acceptance Testing scenarios where users can give feedback about their own experiences with the product. This can also be done during system testing where all components of the product are fully integrated to ensure that the system matches the expected requirements.

Additionally we utilized functional testing in our software development cycle. Because we are working with the internal program and our highest aim is to match this program to its technical requirements, we focused our work on functional testing–testing that verifies that. Since we used requirement-driven development to design the program, each element of the program reflects one of the clearly identified requirements: for instance having an AppointmentId less than 10 characters. Our tests directly checked that requirement. Another example of functional testing we utilized is Unit Testing–in which individual methods of the program are testing piece by piece. On the other hand we did not utilize non-functional testing which would test for the non-functional requirements of the system–how it functions and handles itself in the real world. Security testing is an example of a time when this would be used. Compatibility testing, which tests the programs functioning across devices and operating systems was also outside of the scope of this project. If we were working with a finalized program with a user interface, it would be important that the next steps were to test its deployability across platforms.

Another software testing technique we utilized is both static and dynamic testing. These two types of testing differ because static testing does not run the code, which dynamic testing does. Static testing looks like a manual review of the code and other documents to ensure that there are no errors. Although we did not do the formal versions of static testing like peer reviews and walk-throughs, we did visually inspect the code for errors against the requirements. At one point I noticed code that had not been updated to reflect the most recent service; the code still checked that the description was 100 characters or less, but it needed to say 50 characters or less based on the requirements. In this scenario a test would not have caught the error, because the program was testing the wrong limit. Much of the testing was also dynamic because we were running the code via JUnit to see how it executes. During all five milestones, we developed clear test classes that reflected industry standard testing techniques against various types of input into the program.

Overall the choice of software testing techniques depends on the stage of development and the type of program. In our situation the program needed a lot of dynamic, functional, white box testing to ensure that we had met the requirements for this stage of the project.

Crafting robust and reliable software necessitates a testing mindset woven throughout the entire development lifecycle. During the creation of contact, task, and appointment services, we embraced this approach by continuously scrutinizing the code for potential errors and striving for comprehensive test coverage. Our methodology prioritized caution, employing thorough and deliberate programming techniques to confirm the individual functionality of each program component, as well as its collective performance when integrated. A key aspect of this meticulousness involved a deep dive into edge cases and boundary conditions, which are often overlooked but critical for software stability. For instance, we verified that cases with character lengths at 31 characters long would fail if the acceptance criteria was 30 characters long.

To bring structure and mitigate complexity, tests were meticulously designed using the Arrange-Act-Assert (AAA) pattern. Given the implicit interrelationships within the code, it was imperative to test at every interaction point. This necessitated the application of white box testing techniques, allowing us to fully comprehend backend processes and validate their appropriate operation. A crucial lesson learned was that a passing test doesn't always equate to a truly working program. For instance, a segment of code transferred from a previous service passed its tests, yet failed to meet a revised requirement, specifically a character limit change from 50 to 20. This experience underscored the importance of remaining vigilant for false positives and ensuring tests genuinely reflect evolving program requirements.

To minimize bias during code review, I adopted the persona of a software tester rather than a software developer. Through this lens, success was measured by the creation of each new test targeting an additional feature and every new section of code covered and validated. However, I remained acutely aware that a passing test alone was not sufficient; constant vigilance against false positives was paramount. Discovering errors was celebrated and not a sign of failure. The naive thing would have been to assume that the program was error-free.

Ultimately, software engineering professionals must maintain unwavering discipline to ensure the thoughtful construction of a product from its inception to completion. Neglecting thorough consideration of requirements, comprehensive testing, or the intricate inner workings of a program can introduce chaos, leading to long-term production delays due to bugs and errors. Such oversights can have catastrophic repercussions for the product and the company, resulting in wasted time and the need for extensive rework. The role of testing is critical throughout the Software Development Life Cycle (SDLC) to ensure the successful delivery of high-quality products. Testing, like security, should be a significant consideration at each stage of the development of the program. Testing itself should follow its own systematic process that mirrors the SDLC, encompassing test planning, monitoring and control, analysis, design, implementation, execution, and completion. Thorough test planning is crucial, as its comprehensiveness directly impacts the effectiveness of defect identification. By following these steps with a disciplined mindset and vigilance toward bias, we’ve developed a product we can stand behind and present to our customers with trust.