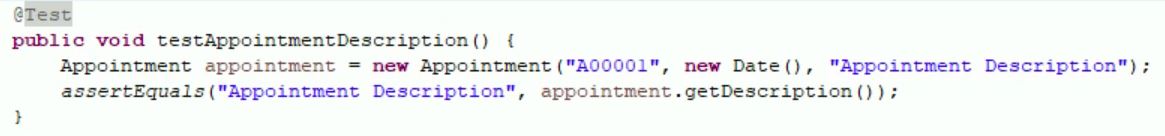
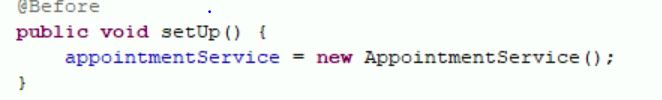
I worked to create unit tests that help to ensure security, reliability, and quality. I worked to write readable, simple tests that help ensure code works as intended. I wrote tests that aligned with the software requirements before writing code. I worked to determine the amount of code the tests cover and further understanding how effective the tests are, and how to improve them. I looked at all the classes in the project and all of my tests to figure the percentage coverage.

I worked to keep the code technically sound by keeping the tests as small and focused as possible. For example, the below section of code shows a test case that tests whether the getDescription() method of the Appointment class returns the correct appointment description that was set by the constructor.

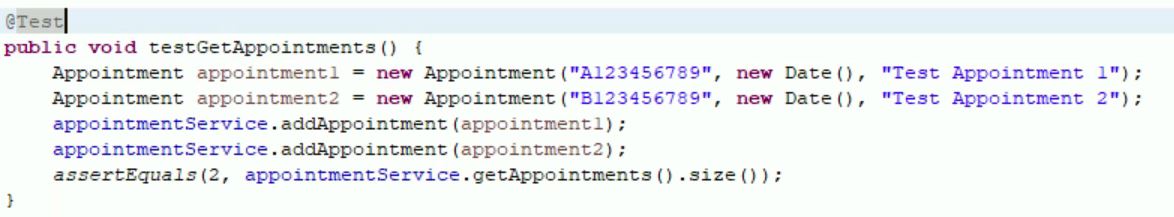


The test case creates a new Appointment object with an ID, date, and description. it then uses the assertEquals() method to compare the expected appointment description ("Appointment Description") with the actual appointment description returned by the getDescription() method.

To ensure efficient code, I made sure to use an @Before annotation to set up the AppointmentService object before each test method is executed, which is an efficient way to ensure that the object is initialized properly before each test, shown below:



Additionally, the getAppointments() method correctly returns a list of appointments with the correct size, creating two new Appointment objects with unique IDs, dates, and appointment descriptions. It then adds these appointments to an AppointmentService object using the AddAppointment() method. Finally, it uses the assertEquals() method to compare the expected size of the appointment list(2) with the actual size of the list returned by the getAppointments() method:



I used testing techniques to help identify gaps, errors, or any kind of missing requirements that may have been different from the actual requirements. More specifically, unit testing to define and run tests for individual units of code in isolation.

I used Gray box testing that is used to execute test methods and test cases as well as White box testing to analyze the internal structures of the used data structures, internal design, code structure, and the workings of the software.

Other methods of testing that I did not use are testing that requires the integration of different components of the software to ensure they work together as a system, such as in Integration testing. System testing, which tests the complete software system to ensure it meets the specified requirements as well as Usability testing, which tests the software to evaluate its user-friendliness and ease of use.

Gray box testing can be used for various practical purposes, including testing the integration of various modules of the software system to ensure that they work together as intended, and testing the security of the software system by examining the system architecture and identifying potential vulnerabilities.

Integration testing can also be used for a wide range of platforms and applications where different components or modules need to be in integrated and tested together. It is particularly important for applications that are composed of multiple components, as it helps to ensure that these components work correctly together and that the application functions as intended.

System testing is used for validation, integration, compatibility, performance, security, usability, and recovery. It can be used for various practical uses, by testing the entire system.

Usability testing is used to evaluate how easy it is to use a software product by testing it with real users. It is used to improve user experience, identifying usability issues, increasing productivity, reducing training costs, improving customer satisfaction, and reducing support costs.

I employed caution by way of following a testing plan and following it carefully. I employed different testing techniques, and documented all defects.

Appreciating the complexities and interrelationships of the code we test is important because it helps us to better understand the software system, identify dependencies and interactions, and avoid unintended consequences. For example, understanding the complexities of the code that powers the application, you can better understand how the system works, what features it provides, and what user interactions are possible. This can help design more effective test cases that cover all aspects of the system and ensure that the software product works as intended. Furthermore, understanding the complexities of the code can avoid unintended consequences that may arise when changes are made to the system. For example, a change made to a financial software system that allows users to transfer funds between accounts, may inadvertently allow users to transfer more funds than they have available. Understanding the code can ensure that testing does not introduce new issues into the system.

I limited bias in reviewing my code by taking breaks, approaching testing with a critical and objective mindset, and following coding standards and best practices. In developing code, it can be easy to be emotionally invested in it, and may have developed blind spots to potential issues or bugs, along with becoming unconsciously influenced by your own knowledge and expertise. Bias can also arise from time constraints, pressure to deliver results, and personal preferences, this can lead to focus on certain areas or test cases while neglecting others.

Being disciplined in your commitment to quality as a software engineering professional is crucial to building trust, minimizing risk, meeting user expectations, and continually improving your work. It is an essential part of being a successful and respected software engineer.

Cutting corners when it comes to writing or testing code can have serious consequences for both the software and end-users. Skipping security measures or not thoroughly testing them can have vulnerabilities that can be exploited by attackers. This can result in data breaches or other security incidents that can harm both the software and its users. It can also lead to stability issues. Unstable code can lead to software crashes, data loss, and other issues that can impact users and organizations. Cutting corners may save time and money in the short term, but it can result in greater costs and longer development times in the long run. Issues that are not caught early on can become more complex and expensive to fix later on. And lastly, delivering software that is of poor quality or that doesn't meet user expectations can damage your reputation as a developer or as an organization. This can lead to lost business and a negative impact on future projects.

I plan on avoid technical debt by making code quality a top priority from the beginning of the development process by writing clean, maintainable, and well-documented code that adheres to best practices and industry standards, improving the quality of the existing code without changing the behavior, testing code thoroughly to help catch bugs and errors before they can cause problems down the line by utilizing automated testing, unit testing, integration testing, and end-to-end testing, staying up to date with the latest technologies and industry trends, dedicating time and resources to refactoring, testing, and improving code, rather than adding new features, and lastly, by collaborating effectively with other members of my development team, including designers, product managers, and other engineers. This can help ensure that decisions are made with a long-term perspective in mind, rather than just addressing short-term needs.