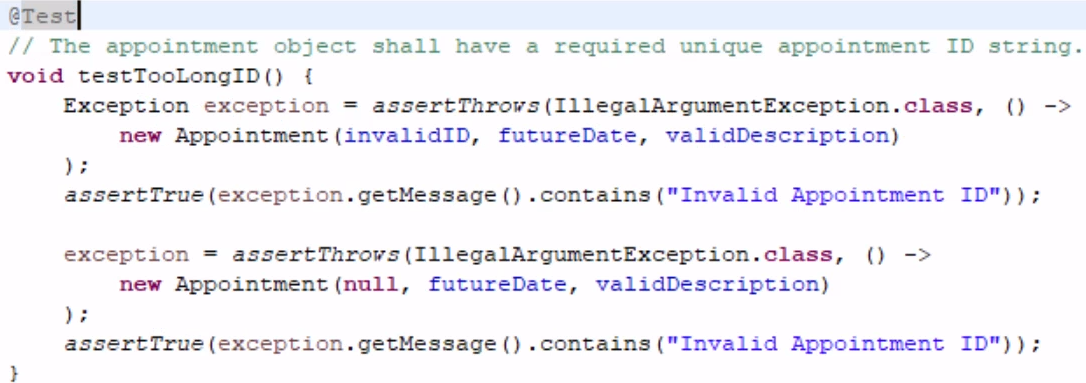
Module 7: Summary and Reflections Report

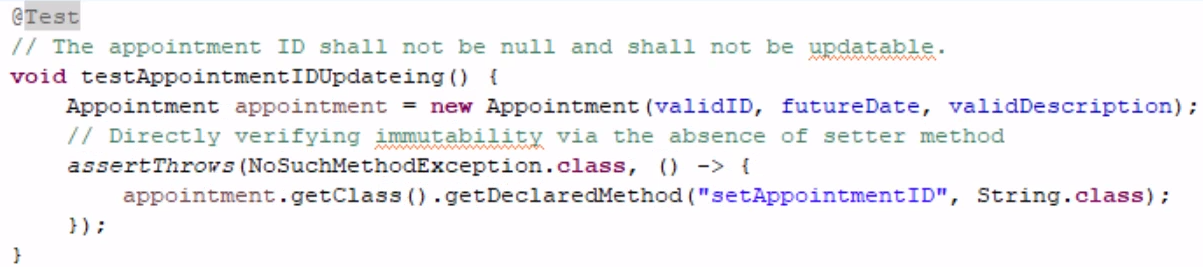
# **Software Test and Automation**

Version 1.0

Ronny Z. Valtonen

The project highlights the importance of testing in software development. I deployed various testing techniques from assertions, self-testing as well as JUnit testing. The testing employed ensures the reliability of the mobile application for Grand Strand Systems. Due to the coverage percentage being over 90%, this means that the overall effectiveness of the JUnit tests was high. The majority of the code was tested which means most of the hidden bugs were found. Not only were every requirement tested thoroughly using JUnit test, every function and statement were tested using other methods. For example, one of the requirements from the Appointment Service states: “The appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable.” This requirement for the Appointment class is complex and was deciphered entirely. This requirement can be broken down into 4 different requirements. Each of these were tested. Another example shows that the Appointment Service requirement were entirely tested:





These JUnit tests accurately test these requirements fully, ensuring the reliability of the program as a whole. To ensure efficiency, tests were concise that avoided redundancy which helps maintain readability and manageability of the tests:

* assertNotNull(taskService.getTaskById(taskId));
* taskService.deleteTask(taskId);
* assertNull(taskService.getTaskById(taskId));

For example here, in the TaskServiceTest.java, the testDeleteExistingTask combines checks for task existence before and after deletion. This way you aren’t trying to delete a non-existent task, we check that it exists first. To ensure technical soundness, the utilization of assertions such as “assertEquals” and “assertNotNull” were used to validate the expected outcomes:

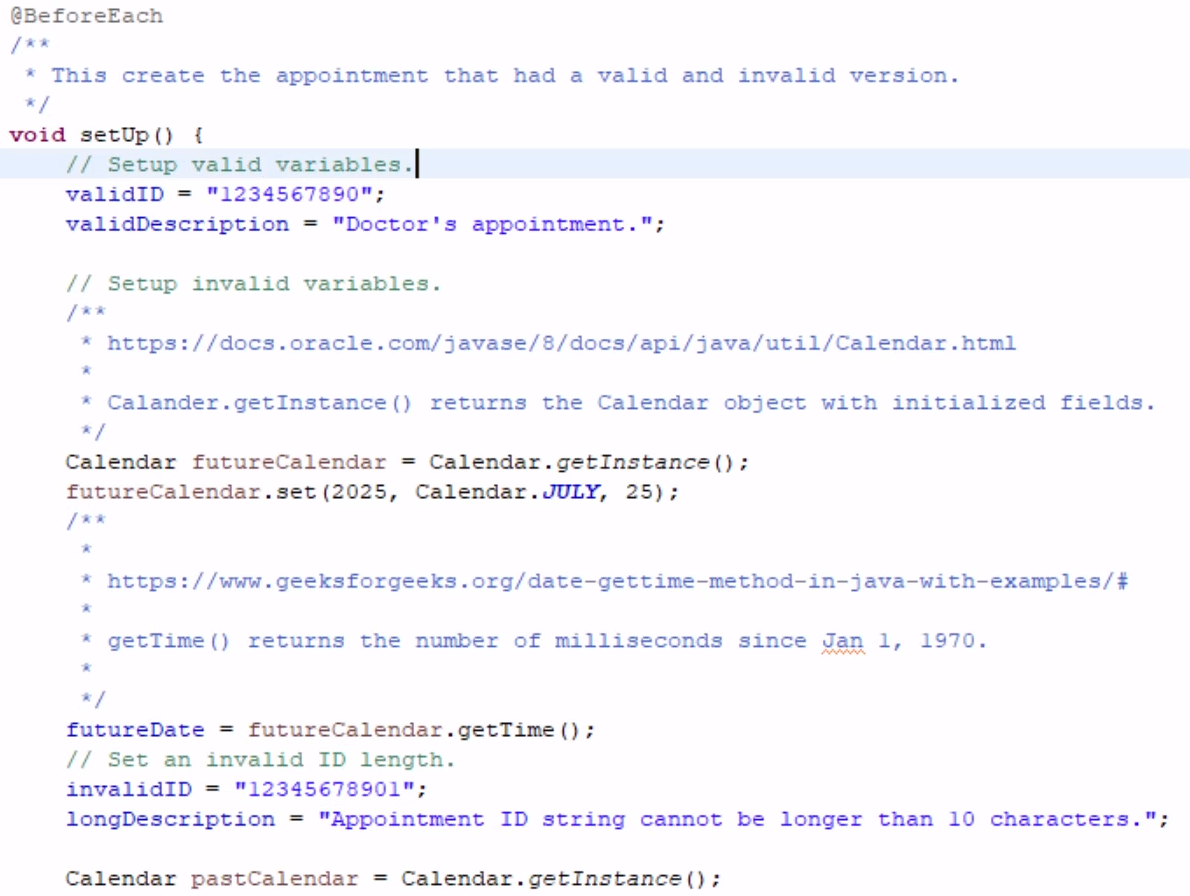
* assertEquals("Task Name", task.getName());
* assertNotNull(taskService.getTaskById(taskId));

These assertions ensure the code will behave as expected and catch discrepancies early.

The software testing techniques that were employed in this project were unit testing and black-box testing. Unit testing (using the JUnit framework in this case) focuses on individual components to ensure each feature functions correctly in isolation to match the requirement(s). Unit tests were automated and provided quick feedback on code changes. Black-box testing validated the functionality without looking at the internal code structure, ensuring that the application would meet the requirements given by the project guidelines. Other software testing techniques that weren’t employed were system testing and regression testing. During system testing, you test the entire application as a whole, and because there was no UI and these were various components of a bigger system, it was not needed at this point. This testing would be required, however when these components were put together for the full project with UI and all. Regression testing would be a valid technique to ensure any new code added would not adversely affect the current functionality of the code. All the JUnit tests do is test individual requirements which may still get met when other components fail, which regression testing would prevent. Unit testing is ideal for early development stages and quick bug catching and fast changing code/projects. System testing is useful for final code validation before a full public release. Regression testing is important for long-term projects that have updates, this maintains the quality of the code over time.

I adopted a cautious mindset during the development of the project. Understanding each requirement thoroughly was important to prevent introducing bugs. For example, while testing updating tasks, I made sure that changes to task detail did not affect features like displaying the task. It was important to appreciate the complexity because if you did not take caution, making one change may seemingly work, while it breaks something else in the background. I tried to limit bias in my review of the code by reviewing it in different perspectives. The code not only had to just meet the requirements of the project, the code itself had to be free of bugs and memory leaks. Although the end-user may not experience a bug or realize the inefficiency of code if it meets the requirements and expectations, the programmer should know about these and take appropriate measures to ensure reliability and efficiency.

This is why it is important to have a commitment to quality in software engineering. Providing code that ‘just works’ isn’t enough. You must ensure the code not only works, but is efficient in the background, is reliable to run every time, and is maintainable in the future when updates are inevitably needed. This means you need to have appropriate comments that explain the functionality enough that when you come back in a year, you and anyone else coming to it understand how it works and can update and change code effectively.



For example, here I gave a link to where I learned how to do something, so that when I come back to this, I can easily go back to my sources and understand exactly what I used, how it works and how I can use it again when/if needed.