**Summary and Reflection**

Matthew A Keaton

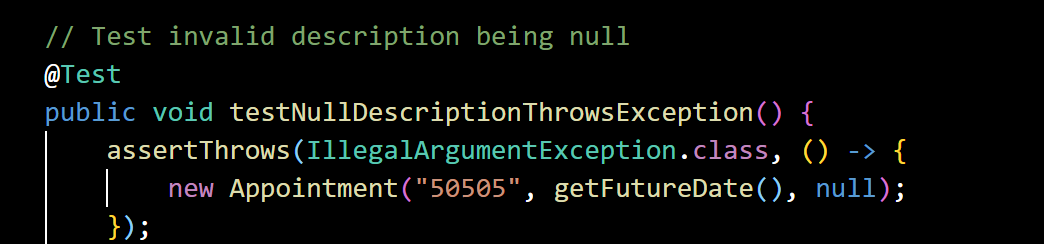
Southern New Hampshire University

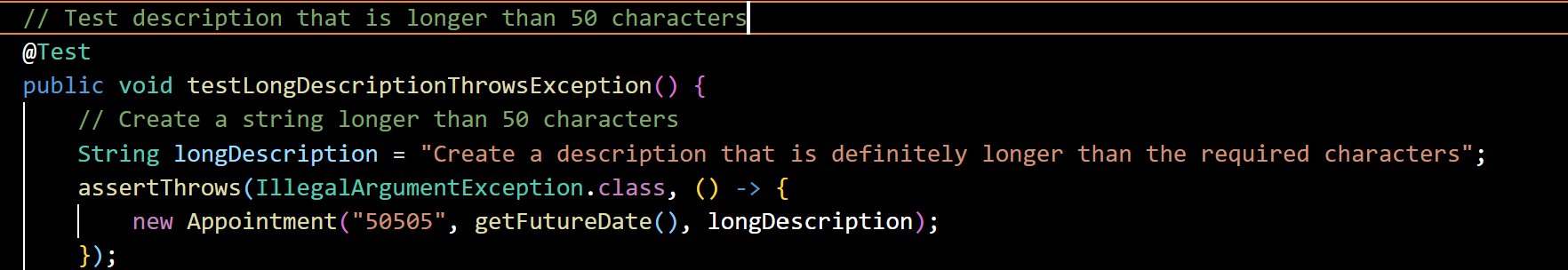
CS 320: Software Test, Automation QA

Professor Federico Bermudez

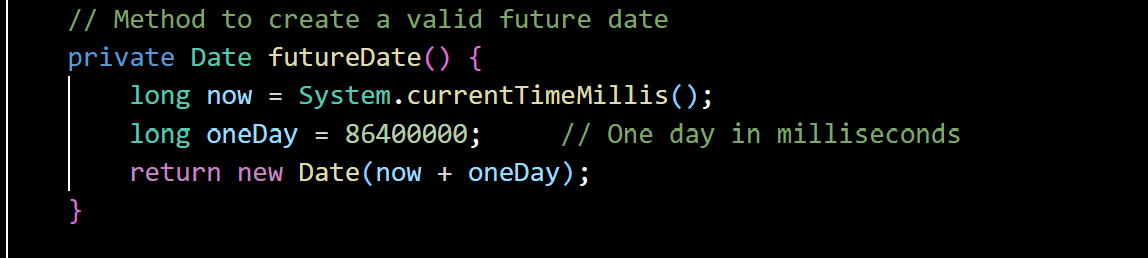
April 20, 2025

1. **Summary**
   1. To develop the mobile application, I used a JUnit testing approach to check whether each part of the contract, task, and appointment services ran correctly. My unit testing approach aligned with the software requirements because I made sure each test checked exactly what the app was supposed to do. For example, the requirements stated that each contact, task, and appointment needed a unique ID of no longer than 10 characters and could not be empty or null. I wrote tests that would fail if two items had the same ID or required information was missing. I tested that contacts could be added, updated, and deleted for the contact class. I followed the same process for tasks and appointments, ensuring the ID couldn’t be duplicated and had all the required information. My JUnit tests had a 78% code coverage. This means most of the crucial parts of the code were tested, especially the parts for adding, updating, and deleting contacts, tasks, and appointments. I tested the main things the app was supposed to do, like checking for unique IDs and making sure no important fields were missing. I also included negative test cases to ensure the services correctly handled invalid inputs. For example, I created a test case in the contactTest class that would throw an error when an incorrect phone number was entered. These tests helped ensure that I would find common mistakes and that the mobile app would work well for the customer.
   2. To ensure my code was technically sound, I tested it for good and bad inputs. For example, in AppointmentTest I tested invalid and valid description input, ensuring the input was not null, no longer than 50 characters, and worked with a valid input that meets the requirements. To maintain efficiency, I created a method called futureDate() that helped me avoid writing the same future date repeatedly in my code. By doing this, I saved time and made my code more readable and easier to maintain, promoting its overall efficiency. Specific examples from my code are shown below, highlighting areas where I implemented technically sound methods and maintained efficiency.









1. **Reflection**
   1. The software testing techniques I used for the project are equivalence partitioning and boundary value analysis techniques. According to Hambling et al. (2019), “equivalence partitioning is based on a very simple idea: it is that in many cases the inputs to a program can be ‘chunked’ into groups of similar inputs” (p. 102). I used this technique by grouping all input values to be treated the same when testing for invalid inputs. I chose one invalid option, such as a unique ID with more than 10 characters, representing all other invalid inputs over 10 characters for that partition. I used boundary value analysis to “identify just two values at each boundary” (Hambling et al., 2019, p. 105). For example, the description for an appointment must be 50 characters or less, so I tested a description exactly at the boundary of 50 and one just outside the boundary.

Decision table testing is a software testing technique not used in this project. This technique involves creating a decision table that helps ensure all the combinations and conditions that may occur are covered (Hambling et al., 2019, p. 107). A more practical use for this technique is a software development project that has a lot of rules or conditions controlling what it does. For example, loan approval software would approve or deny requests based on different conditions. A decision table will help ensure every rule, condition, or combination is checked. Another software testing technique not used in this project is state transition testing. This technique is used when a system has different states and will behave based on what state they are in (Hambling et al., 2019, p. 109). A software development project where this technique would be more practical is a system that uses a login. State transition testing will help make sure the app goes from one state to another correctly, without failing if the user does it wrong.

* 1. For this project, I focused on being careful and cautious while testing my code. I checked not just the standard, expected inputs, but also the ones that could cause errors. For example, in the AppointmentTest class, I tested what would happen if someone tried to schedule an appointment in the past, which should not be possible or allowed. This helped make sure the program worked correctly in all situations, not just the good ones. It was essential to appreciate the complexity and interrelationships of the code because different parts of the program depend on each other and affect each other. For example, the unique contactId is used in multiple methods to add, update, find, or delete a contact. By understanding the relationship, I was able to write effective tests that do not update or reuse the contactId, affecting other areas of my code.

To limit bias while testing my code, I tested for valid and invalid inputs. For example, testing my code with an invalid input should throw an exception. If an exception was thrown with an invalid input, I did not just stop there and assume that the code would work perfectly with a valid input. This helped eliminate bias by assuming my code worked and allowed me to test every requirement fully. Bias is a concern when testing your own code because I have confidence in my skills and believe it will work. Another bias is that thinking about how I would use the app may not be the way others will use it. For example, as a contact address, I use addresses that are short and to the point that would meet the requirements of 30 or fewer characters, whereas other people may like to list entire addresses that go over 30 characters. Understanding this bias allows me to test all inputs that could be invalid.

Being disciplined and committed to quality as a software engineer is extremely important. Disciplined about quality ensures that you are taking time to fully write and test your code rather than rushing to complete it. Ignoring code quality leaves more room for bugs and errors that build up over time and cost time and money. For example, in the contactService class, if I cut corners and did not test for duplicate IDs, it would build up over time and become harder to fix. If there were multiple duplicate IDs, after a certain amount of time and updates to that contact, it would be hard to determine which contact is the original and which one is most recent. Focusing on quality and not cutting corners will help minimize the risk factor in a project. A test manager uses project risks, like technical debt, to manage the capability to deliver (Hambling et al., 2019, p. 159). To avoid technical debt in the field, I will identify the probability of a risk happening and what the impact would be from the risk. Then, I can form a plan that helps mitigate the chances of that risk happening or reduce the overall impact it will have.

**Reference**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.