Cody VanGosen

CS-320 Software Test, Automation QA

Professor Bermudez

August 18, 2024

# Project Two

The project one for this course revolved around extensive examination and refinement of testing strategies. Within this paper further examination will be applied to explain how the coding implementation in project one successfully satisfied these strict security requirements and industry best practices. Included below an extensive breakdown of the unit testing approach for each feature will be included with further information on how it aligned with software requirements, test effectiveness, and the overall experience in writing efficient and technically sound code.

**Unit Testing Approach**

*Appointment Service Testing:*

The appointment service unit testing placed a focus upon verifying proper creation, addition, and retrieval of appointment objects. This approach aligned with the software requirements as each appointment was validated to have a valid ID, a future date, and a non-null description. The various test cases covered a range of various scenarios including valid appointments, null objects, and invalid input formats. The testAddAppointment test ensured valid appointments were correctly added by using assertions like assertEquals(validAppointment, service.getAppointment("1234567890")) to validate the behavior (AppointmentServiceTest.java, line 25). According to Ammann and Offutt (2016), unit testing is critical in early identification of bugs in the development process. This aims to ensure each module functions correctly in isolation, which work to verify the effectiveness of the Appointment Service tests.

*Contact Service Testing:*

The Contact Service’s tests employed a similar strategy, wherein each contact was verified in having unique IDs, valid names, and phone numbers. The implementation of positive and negative tests, through verification of correct addition and ensuring no duplicate contacts were accepted, worked in tandem to provide more extensive test coverage. This satisfied the requirement of unique identifiers for contacts while correctly formatting details (ContactServiceTest.java, line 19). For this the strategies employed directly aligned with recommendations by Fowler (2018), where they emphasize the importance of boundary testing and handling of edge cases to avoid faults in input data. Further testing to ensure unique contacts and proper formatting work in tandem to ensure the robust handling of real-world data.

*Task Service Testing:*

The Task Service Testing focused upon ensuring tasks could be added, updated, and correctly deleted. Test coverage centered upon validation of task creation, retrieval, and updates. For instance, the test case included for adding a task verifies that when a task is added that it can then be retrieved by its ID. The test coverage report indicated all critical functions including creation, deletion, and updates were all fully covered, thereby satisfying software requirements (TaskServiceTest.java, line 30).

**Effectiveness of JUnit Tests**

The effectiveness of the included JUnit tests can be directly measured by their respective test coverage values. Within all three included services the tests achieved full coverage, thereby ensuring each critical function was validated. The AppointmentServiceTest included checks for key functionality of adding and retrieving appointments while successfully handling edge cases such as null values. The high coverage rate further validated the reliability and robust nature of the code under expected and bizarre conditions. According to Myers et al. (2011), achieving high code coverage in unit tests is critical for ensuring that most possible execution paths are exercised. High coverage results in more reliable software that better handles edge cases.

**Writing the Tests:**

*Technically Sound Code:*

Code was ensured to be technically sound through the usage of several key strategies. For instance, meaningful assertions were made, and tests were structured to verify multiple outcomes within a single test. In the testAddAppointment, the AppointmentService was ensured to correctly retrieve newly created appointments while ensuring incorrect or null appointments were being correctly rejected. The assertions such as assertEquals(validAppointment, service.getAppointment("1234567890")), ensured that the service's behavior matched the expected outcome.

*Efficient Code:*

The overall efficiency of the code was maintained through the elimination of redundancy by including helper methods where possible. For instance, the AppointmentServiceTest employed a @BeforeEach setup method to initialize objects like AppointmentService and validAppointment, thereby reducing duplication across test cases. This enhanced readability and maintainability of the test suite while minimizing chances of error.

assertEquals(validAppointment, service.getAppointment("1234567890")), ensured that the service's behavior matched the expected outcome.

**Reflection:**

* The primary technique incorporated for this project was the use of unit testing, focusing upon testing individual components in isolation. This provided a key advantage of quickly identifying bugs within isolated sections of code early in the development process (Jorgensen, 2013). Additionally, inclusion of black-box testing, focused upon the inputs and expected outputs without having knowledge of the internal workings of the system. This helped to ensure each service met the requirements while correctly handling input.
* Other techniques were not included to quite the same extent, such as integration testing and system testing. Integration testing would be excellent for ensuring interactions between services was operating as expected, while system testing would seek to validate the entirety of system functionality. For the stages examined in these projects the focus was instead placed upon individual component performance.
* Unit testing is imperative for ensuring proper function of individual components are working as expected, which is especially important in microservices architecture. Black-box testing is useful for validating external behavior. This is especially true when considering scenarios wherein internal details are abstracted away. Integration testing, though not used for this testing process, could prove useful with situations where multiple services or systems must interact effectively.

**Mindset**

* During the testing process an approach of caution was used for the entire process. This was important when understanding how even minor changes can have ripple effects up or downstream the entirety of the codebase. In testing the AppointmentService it was recognized that the date handling was complex due to the ability of future functionality being broken if implemented incorrectly. An awareness of interdependencies and attention to detail work harmoniously together to ensure prevention of future bug introduction (Jorgensen, 2013).
* Bias is a universal concept seen throughout many industries and life in general. When considering examination of one’s own code an important lens of focus must be placed on the implicit or internal bias of assuming one’s own code is bug or error free. For this reason, tests were structured around ensuring requirements were met, rather than trying to anticipate how the system would behave. The inclusion of testing for invalid appointments in the testAddAppointment demonstrates this concept by disregarding the initial developer’s assumption that a user would always input valid data.
* Maintaining discipline throughout the implementation and testing process were critical in ensuring efficiency and quality of the overall implementation. Cutting corners or costs by skipping tests or not considering edge cases, can lead to accumulation of technical debt and introduce future bugs. For example, there was a scenario where an invalid date was accepted during the testing process. If proper discipline was not employed, then this could have been ignored rather than diving deeper to investigate further. This would have led to further bugs in production down the line. Through addressing this issue immediately, the accrual of technical debt that would need to be resolved later was avoided entirely.

**Works Cited**

Ammann, P., & Offutt, J. (2016). *Introduction to Software Testing* (2nd ed.). Cambridge University Press.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley.

Jorgensen, P. C. (2013). *Software Testing: A Craftsman’s Approach* (4th ed.). CRC Press.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley.