Module 7 - 2: Project Two

Dynamic and Static Testing

Stanley Niles

Southern New Hampshire University

CS-320: Software Test Automation & QA

Professor Jonathan Norman

June 18th, 2023

***Summary of Work:***

My main approach that I used for the unit testing of the Contact, Task and Appointment services involved writing independent test cases for each operation and requirements placed on the input for these services. The extent to which my implementation stayed aligned with the software requirements I believe was exceptional and thorough. When writing the actual classes themselves, I tried to encapsulate as much of the input validation as possible to reduce any redundancy within the code itself. This approach also helped improve the quality of the code as there was less opportunity for invalid input to be accepted.

Based on the 100% coverage that my JUnit testing received, I felt the independent test cases thorough caught any potential for invalid inputs. Although this was one part of my code that could be reviewed as redundant, in the frame of testing, I viewed it as being more thorough. A specific example was this test case:

*@Test*  
*void* testAppointmentInvalidId() {  
 Assertions.**assertThrows**(IllegalArgumentException.class, () -> {  
 new Appointment ("12345678987654321", LocalDate.**of**(2024, 6, 8),  
 "Lovely dinner date with my wife!");  
  });}

This test case verifies the specific constraints stated in the software requirements that the appointment ID should not exceed 10 characters. For the appointment ID specifically, I test for a valid input of less than 10 characters and for a value of “null” with separate tests to ensure coverage of invalid inputs or potential edge cases.

For the creation of the code to be technically sound and efficient, I did my best to stay within “standard best practices” to the extent that I am aware of. As stated above and with this example:

public Appointment(String *appointmentId*, LocalDate *appointmentDate*, String *description*) {  
 //Input Validation: Checking for Null and Length greater than 10.  
 //Implemented here instead of a setter method due to "final" declaration  
 if(*appointmentId* == null || *appointmentId*.length() > 10) {  
 throw new IllegalArgumentException("Invalid appointment Id: Must be less than 10 characters and not null");  
 }  
 this.appointmentId = *appointmentId*;  
 setAppointmentDate(*appointmentDate*);  
 setDescription(*description*);  
}

The appointmentID had a constraint that it could not be changed, therefore I had to implement the input validation for the ID directly into this function due to the appoindmentID being “final,” which also had to be repeated in the AppointmentService class for the addAppointment() function. After that, I was able to use the encapsulated input validation for the appointmentDate and description variables. This way, there was less repetition throughout my code. Another way to improve efficiency was to implement a HashMap to store the appointments which allowed for O(1) lookup, addition, and deletion. The overall experience with JUnit tests was enlightening, I had to keep a good focus on the project requirements closer. It also emphasized the importance of mindful coding to avoid bugs and implementation of proper code behaviors.

***Reflection:***

The main method or technique employed was White-Box testing. This method is where the test cases are derived from the design of the code itself and the tester understands the inner workings of the code. As I designed the structure of how the code was to execute, I was able to write JUnit tests to verify the necessary constraints and that the code functioned as it was designed to do. The methods I used also fall under the scope of Acceptance testing, as the testing verified compliance with the business/software requirements provided.

A few methods that were not employed would be Black-Box testing, Integration Testing and Security Testing. Black-Box testing is the opposite of what I performed, where the design structure of the code is not known to the tester. This approach can be beneficial as it can supply extraneous tests that might not have been thought of. The value of “not knowing” can be a powerful tool for creation. Integration testing involves testing software components together to check for compatibility with each other. This testing is beneficial as it allows for smaller chunks of code to be checked, which makes debugging simpler. Finally, Security testing is a process to check for inherent flaws in any security methods implements to protect the data of the system. The importance of security testing is vital so malicious actors cannot modify, obtain or destroy a system.

My mindset for the implementation of the code was more meticulous because I was keenly aware of the strict software requirements and future interrelationships that the system was supposed to have. An example, understanding that an appointment’s attributes could affect each other, which increased the importance of proper input validation.

Bias was limited by writing tests that strictly adhered to the defined requirements. Every requirement was treated as a critical test case to pass, regardless of personal assumptions or beliefs about the system's functionality, especially as I was the developer. It is important to remember that we are all fallible and make mistakes. The important part is catching these mistakes early, if at all. However, under stressful deadlines or other situations I can see where being a developer and tester of the same code could pose a rather significant conflict.

Discipline in following best practices and not cutting corners is paramount to creating quality software and being a professional developer. Trying to take the “easy way” out of just not performing quality testing can have major implications down the development road. The more complete a system is, the greater potential a “small” bug will cause more complications and be harder to find. As I did with the testing of these coding projects, although it was repetitious to write multiple test cases for potential edge cases for the same function or variable. In the end, the few extra minutes spent performing more thorough testing will help avoid a mountain of technical debt created by early laziness at the start of a project.