7-2 Project Two

Byron Dewayne Staton

Southern New Hampshire University

CS-320-T6628 Software Test Automation & QA

Professor Linda Gaston

August 14, 2022

The unit testing approach I decided to take for each of the three features involved the white box testing technique. I wrote code to test the functional behavior of each of the program’s components. I started by writing comments that specified each of the customer’s requirements inside my IDE. From that point, I would write test cases for each requirement that I identified. For example, in the ‘Contact’ class, the user’s phone number could not be more or less than a specific number of digits, and it must not be null; since we are in the United States, that number must be ten digits long. I wrote methods that checked these requirements inside the final overloaded constructor found in the ‘Contact’ class file.

After writing a test case based on a single user requirement identified in the assignment documentation, I would run the code as a ‘JUnit Test.’ If the test results returned a green bar, I knew that my test case was successful. If I received a red bar, I knew something was wrong with my test code. Inside the class files, I included exceptions that would be thrown and added custom error messages. This technique helped me quickly identify (almost) exactly where the problem was occurring. After adjusting the code, I repeated this technique until the program returned a green bar. More than ninety percent of my code had successful testing outcomes.

My experience writing JUnit tests was new and exciting for me. When reflecting on when I wrote methods based on the requirements, and the JUnit tests for the ‘Contact’ and ‘Contact Server’ classes, I can see how much of an improvement I made when writing the similar code for the ‘Appointment’ and ‘Appointment Server’ classes and tests. As I mentioned earlier, I wrote requirement checks inside the last constructor in the ‘Contact’ class. Here is what I wrote:

public Contact (String contactID, String firstName, String lastName,

String phoneNumber, String address) {

if (contactID == null || contactID.length() > 10) {

throw new IllegalArgumentException("Invalid ID");

}

if (firstName == null || firstName.length() > 10) {

throw new IllegalArgumentException("Invalid first name");

}

if (lastName == null || lastName.length() > 10) {

throw new IllegalArgumentException("Invalid last name");

}

if (phoneNumber == null || phoneNumber.length() > 10) {

throw new IllegalArgumentException("Invalid phone number");

}

if (address == null || address.length() > 30) {

throw new IllegalArgumentException("Invalid address");

}

this.contactID = contactID;

this.firstName = firstName;

this.lastName = lastName;

this.phoneNumber = phoneNumber;

this.address = address;

}

This worked out okay, but I thought a better approach would be to create individual functions for checking the length of the variables and whether they are null or not null outside of the constructor. If no problems are detected, then the constructor can proceed to initialize objects. It makes the code easier to read and understand so I applied this approach in the ‘Appointment’ class. Here is a code snippet to illustrate this approach:

// checks if date is in the past and not null

protected void **checkAppointmentDate**(Date date) {

if (date == null) {

throw new IllegalArgumentException("The Date Cannot Be Empty");

} else if (date.before(new Date())) {

throw new IllegalArgumentException("Cannot make an appointment in the past");

}

this.appointmentDate = date;

}

Inside the constructor, I simply added the following:

public Appointment (String appointmentID) {

Date date = new Date();

**checkAppointmentIdLength**(appointmentID);

this.appointmentDate = date;

this.appointmentDescription = "INITIALIZER";

}

The software testing technique I employed in the project unit testing; specifically black box testing, and white box testing. The black box testing technique is how I was able to check the functionality of the program from the perspective of a user. This was useful because black box testing is only concerned with the external behavior of the program. I used the white box testing technique for checking the structure of the application’s code base. What this means is that white box testing is a technique useful for analyzing the logic of the program.

The software testing techniques that I did not use for the milestones are system testing, integration testing, performance testing, and acceptance testing. I did not use these additional testing methods because I do not have a fully developed system yet. The system testing technique is used by a QA team, or even developers to analyze how all the components in the application interact together. Integration testing is useful if checking how different components interact with each other. I did not use any performance techniques because this is more useful when checking the speed of the application under a certain workload. Acceptance testing was not performed because this technique involves the user checking the application to see if it meets their expectations.

There are practical uses and implications when using black box and white box testing techniques. When using black box techniques, access to the code base is not necessarily needed. The disadvantage of this is that without access to the code base, writing test cases are cumbersome. This leads to another risk of getting back inefficient testing results. White box testing techniques allow for testers to have full coverage of the code base. This makes it much easier to detect defects and errors sooner than discovering them after extensive testing. The disadvantage with white box testing is that there are simply too many test cases one can think of, so it is almost impossible to have a perfect application. Another implication involves the skill level of the tester. With white box testing, the tester must be highly skilled, and therefore, the costs associated with the program increase.

I employed caution every step of the way through the development and testing process. I am remarkably familiar with writing the usual constructors, getters and setters, methods, and variables, but this is my first time writing my own test cases. I think for that reason alone it made me more mindful of the quality of code I wrote. The code snippets I provided earlier are a notable example of envisioning code from the mindset of a developer, and the mindset of a developer that understands the role of a software tester.

Tests should be written with the user in mind because it is highly unlikely that they will have knowledge of how all the components work together in the program. I was careful not to write the code from how I, the developer, thought the program should work. I wrote the tests based solely on the requirements provided to me. Stephen Margheim asserts that “automated tests are biased by their author, a developer.” If I were testing my own code, I imagine that bias would be a concern simply because I am the one developing and writing the code based on my own requirements. This is not to say that my own requirements are the best for what I am working toward accomplishing.

It is of the utmost importance for me to maintain discipline in my commitment to quality as a software engineering professional. Ensuring that code is well-structured, secure, and testing has high coverage is critical when developing software. Cutting corners in testing can lead to expensive, and sometimes devastating, outcomes for a company using my code in a production environment. My goal is to keep technical debt at a minimum by always maintaining a high percentage of code coverage, using good, well-known architectural coding practices, and never ignoring problems with production code. Technical debt is when “developers write software that violates good architectural or coding practices resulting in structural flaws in the code that, if left unfixed, put the business at serious risk” (DeLoach, 2020).

**References**

Margheim, S. (2020, December 17). *How to Combat Bias in Software Testing*. Test IO. Retrieved August 14, 2022, from <https://test.io/resources/blog/how-to-combat-bias-in-software-testing>

DeLoach, E. (2020, October 20). *How to Evaluate, Manage, and Avoid Technical Debt*. Metova. Retrieved August 14, 2022, from <https://metova.com/how-to-evaluate-manage-and-avoid-technical-debt/>