John Mullen

CS-320

February 19th, 2022

Project Two

When it came to creating tests for this application, I needed to understand the program, what its expectations are, and what limitations are being placed on it. We often find this information within a use case, but also in information provided about the base expectations and structure of the program. For this application, we were looking to create a database that could retain various groups of information, each of which must experience its own limitations on the data. One of the most crucial pieces of information that was dealt with throughout each section within the program is the ID associated with each type of data (ContactID, TaskID, AppointmentID). The reason that this variable was most predominant in ensuring its characteristics was because the ID is never altered, never replicated, and is always linked with a certain contact, task, or appointment. This is the only variable within each class that cannot be altered. In the testing that I created, I used test cases that would intentionally cause the application to fail in order to ensure it is properly monitoring each aspect of the classes. Some of those test cases included ensuring the IDs did not exceed 10 characters, nor was it a value that could be altered once it had been set.

While I had not used Junit testing to this extent prior to this project, I felt that I was able to get a good hold on proper utilization during the Contact section of the program in module 3, and that I was able to build upon that throughout the remainder of the project. To properly ensure that my testing was both functioning appropriately and adequately testing my code, I used two different approached to the test cases utilized. First, I created test cases that would create a class with given input knowing that one of the inputs was intentionally outside of the acceptable parameters of the class.

Example:

*@DisplayName*("Task Name cannot have more than 20 characters")

void testTaskNameWithMoreThanTwentyCharacters() {

Task task = new Task("Supercalifragilisticexpialidocious","TaskDescription");

if(task.getTaskName().length() > 20) {

*fail*("Task Name has more than 20 characters.");

}

}

This example is testing that the Task Name parameters were properly functional. I did this by providing the application with a serious of input values that associate with the requested values from the program. This input uses a value for the Task Name that exceeds the 20 character limit, thus throwing an exception so we know that the parameters are set appropriately.

Along with the tests created to test the parameters put in place for the application, we also wanted to ensure that if multiple changes were made, that one alteration wouldn’t override another. For this testing, we created test cases that would run sequentially to ensure this wouldn’t be an issue.

Example:

*@Test*

*@DisplayName*("Test to update phone number.")

*@Order*(3)

void testUpdatePhoneNumber() {

ContactService service = new ContactService();

service.addContact("Dr.", "Cross", "5555551111", "123 Lollypop Lane");

service.updateNumber("5555550000", "4");

//service.displayContactList();

*assertEquals*("5555550000",service.getContact("4").getNumber(), "Phone number was not updated.");

}

*@Test*

*@DisplayName*("Test to update address.")

*@Order*(4)

void testUpdateAddress() {

ContactService service = new ContactService();

service.addContact("Dr.", "Cross", "5555551111", "123 Lollypop Lane");

service.updateAddress("555 Nowhere Ave", "6");

//service.displayContactList();

*assertEquals*("555 Nowhere Ave",service.getContact("6").getAddress(), "Address was not updated.");

}

When reviewing this section of code, we can see the @order declaration at the top of each section. This is part of the JUnit test case that follows that series of actions in numerical order (1 > 2 > 3 > 4 etc.). Each section of this test would make a different alteration to the same class in order and would verify that the changes were capable of being made without interference.

When it came to the mindset that I utilized to properly accomplish the needed outcomes of this project, I tried to ensure that at the forefront of each situation, I was reminding myself of the use cases and not overcomplicating each process being done. While the complexity of each script helped retain the understanding and expectation of each section, breaking it down to a more molecular level and reviewing each section separately helped the project from getting too overwhelming and confusing. When I initially constructed the first set of scripts, I reviewed the use case which led to a largely overprocessed and overcompensating script which I ended up rewriting with a more basic and broken-down approach to the end goal.

I think that bias was one of the largest barriers that I faced and needed to attempt to overcome. I realized after reviewing my initial code and, more importantly, my testing that I was accommodating my script and not necessarily approaching it from a neutral perspective. Having written the code, I was aware of what parameters were set and what expectations I had for the code. I am sure that even on my second attempt I still had those limitations and characteristics in mind while creating my tests. In order to completely remove bias from the tests, we would need to find someone that knows the expected use cases and capabilities a program should provide, without seeing the code itself. They would build tests based on these use cases and expectations which would truly allow us to ensure that our tests are checking for proper limitations and functionality without obscuring it for the code written.

Discipline in terms of software testing and QA is critical in ensuring that we are not jeopardizing any data or user base with anything we write. Any absence or limitations of the testing of an application can lead to unknown and unnecessary exploits and leaks within the program. While it often feels as though vulnerabilities are found and exploited in almost any application, we must do our best to remain ahead of the threats. I think that the mindset that I utilized when reviewing my code is one of the most beneficial approaches to QA and vulnerability prevention. When we hear about applications falling victim to attacks, it is often more elementary and basic than we tend to think would cause these issues. I think that we need to continue to ensure that we are remembering the basics throughout developing each application. Remembering the foundations of coding and our basic structures is the first step in building a secure and stable application.