Samuel Walters

09/30/2023

Project Two

Testing was aligned with requirements by assuring that the correct exceptions were thrown when passed a null, empty, or non-unique values.

*assertThrows*(IllegalArgumentException.class, () -> {  
 Task testTask = new Task("", "otherName", "OtherDescription");  
});

let me test that when I attempted to instantiate a new Task or Contact object, there should be an exception thrown. This aligned with the requirements that I had to ensure that there the task id could not be null. I also did several tests for each field to test if it was possible to create an object with null, empty, too long, and in the case of the phone number, if there was not exactly 10 digits. I also ran tests to check that a newly created contact was made, as well as making sure that by looking for the id of the contact, we could find that it equaled the contact we had just created.

public void addContactTest() {  
 Contact newContact = new Contact("John", "Lolligag", "1234567890", "Address stuff");  
 testContactService.addContact(newContact);  
 // Check for if Contact is no null, meaning it got added.  
 *assertNotNull*(testContactService.getContact(newContact.getId()));  
 *assertEquals*(newContact, testContactService.getContact(newContact.getId()));

By running the tests with coverage, I was able to see which lines of code were ran. I was then able to make changes and see that some IllegalArgumentExceptions might not have been tested, and I could then write a test for that. That ensured that I was able to test 100% of the methods, and all of the code I had written for the contact and task services and classes. I ensured that my code was efficient by including @BeforeEach  
public void setupTests() {  
 testContactService = new ContactService();  
 testContact = new Contact("Jim", "Beam", "1234567890", "123 2nd Street, Jamaica");  
 testContactService.addContact(testContact);  
}

That would create a new contact, so that for every test I would not have to constantly write the same 3 lines. I also used one Contact or Task and ran that object through multiple tests instead of creating a brand-new object every single time.

I made sure that my code was technically sound by ensuring my coverage was complete, testing each requirement in multiple ways, and fulfilling all the requirements for our application.

@Test  
public void updatePhoneNumberTest() {  
 testContactService.updateContactMap(testContact.getId(), "phone", "1111111111");  
 *assertEquals*("1111111111", testContactService.getContact(testContact.getId()).getPhone());  
 // If too long will throw and not update  
 *assertThrows*(IllegalArgumentException.class, () -> {  
 testContactService.updateContactMap(testContact.getId(), "phone", "1234567444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444890000");  
 });  
 // Checks if phone number is too short.  
 *assertThrows*(IllegalArgumentException.class, () -> {  
 testContactService.updateContactMap(testContact.getId(), "phone", "1");  
 });  
}

That is an example where I tested if I could update the phone number with it being too long or not long enough. I made sure to do this not just for the object creation, but also updating the object.

Running with Coverage from my IDE, I was able to see that I completed 100% of my four classes, 100% of my methods in Task, and TaskService, and 92% of all lines. I also completed 100% of my method coverage in both Contact and ContactService, with 92% of total line coverage.

For the module assignments I used a variety of testing, including white box, functional testing, and static testing. Static testing is what I used before I even began writing any tests or code. I made sure that the requirements were attainable and clarified any technical points. After writing the code, I used my IDE to complete more static testing regarding the syntax of the classes. Specifically, I had questions about using the Date package from java, as well as using an object, or string for representing the date of an appointment. Requirement gathering and analysis is the basis for designing any system, and I think it should be extensively used before code is even written.

I exercised white box testing due to testing the components of each class. As I wrote the code, I was also able to understand how the components fit to make the class and class services. A specific example was using coverage to ensure that all the methods were executed. By not only writing tests to ensure requirements were met, but also testing the total coverage, I was able to check every single case, and ensure that I had coverage for every if-else statement. With all of the smaller projects I am working on, having unit tests of what I write is very helpful, as well as having the components work with any existing architecture or systems. This means that I need to understand more about what I am coding, and is a great technique to catch some fringe methods that might not be integrating well.

Functional testing was used to ensure that all the requirements were met, as well as the data structure not in any way altering the data when it was retrieved from the hash maps used. I used mostly white box methods as I was already writing the code that went along with the tests. Practically, this is one of the most basic tests for delivering a product. You must have the software working to the minimum level of functionality.

There were a lot of testing techniques I did not try. Non-functional testing was not a concern, as this is such a small-scale application. I did not have to worry about performance or latency, as everything was in memory. If this was a web application, I would be more concerned about the data, and transactions of the objects, and would write more tests to cover if an object was altered or changed when inserting and retrieving from a database or some other cloud service.

My mindset to eliminate bias was to test to requirement and attempt to find flaws, not to get my tests to my pass. I of course want all of my tests to pass as that means less work, but it would also involve mean a less malleable code. I think by keeping a mindset of what will I remember a year from now, and how can I make this so I can easily change the program, helps me with all of my programs. Even when working on personal projects, I try to first test my ideas and get them to work, then I work on refactoring my code so that it can be more maintainable. After much practice I am sure that I will get to the point where I can write more readable and more maintainable code. Telling myself every time I sit down to write or design software that an extra 10 minutes will save hours of time later, is worth the commitment.