Abdulrahman Al-Nachar

CS320 / Project Two

In Project One I was tasked to create 3 programs each had 2 main classes and 2 testing classes, the first project was Contact, which had two Classes Contact and Contact Service, contact had these requirements: contactId String can’t be longer than 10 characters & it can’t be null and can’t be updateable, firstName , lastName, phoneNumber Strings can’t be longer than 10 characters & it can’t be null, Address String can’t be longer than 10 characters & it can’t be null, for that in contactTest I developed 7 test units to make sure that each of these requirements is being met by validating inputs, & designed them to handle errors internally. The requirements set for contactService class is to make sure that the users can add, delete, and update the contact with the contact ID, so in contactServiceTest class I developed testing units that can test that like:

**public** **void** testAddContact() {

ContactService service = new ContactService();

Contact contact = new Contact("00000", "Random", "Testing", "4697320434", "123 Testing Ave");

service.addContact(contact);

*assertEquals*(contact, service.getContact("00000"));

}

This test adds contacts using correct inputs and it will be used later to run other tests like adding duplicate contact using the same ID, EX:

**public** **void** testAddDuplicateContact() {

ContactService service = **new** ContactService();

Contact contact1 = **new** Contact("00000", "Random", "Testing", "4697320434", "123 Testing Ave");

Contact contact2 = **new** Contact("00000", "John", "Doe", "1234567890", "456 Elm St");

service.addContact(contact1);

*assertThrows*(IllegalArgumentException.**class**, () -> service.addContact(contact2));

}

For this Contact program, I believe the testing cases I developed are effective at testing the functions of the main program and my evidence for that is that my unit coverage percentage is at 80.9%, I utilized JUnit assertion methods to ensure that the results are correct and used assertion methods for one purpose at a time to isolate tests and not bulk it all in one to ensure that all functions and scenarios are tested independently.

For the second Program, I had to develop a task and taskService Class and corresponding testing classes for them, the requirement set for this project taskId String can’t be longer than 10 characters & it can’t be null and it can’t be updateable, taskName String can’t be longer than 20 characters & it can’t be null, The taskDecription can’t be longer than 50 and can’t be null, in taskService the requirements were, tasks can be added using a unique ID, tasks can be deleted using an ID and task’s names and descriptions can be updated using an ID and a new name/description. In TaskTest class I have 13 test cases meant to enforce the requirements set for the task class and make sure that errors are managed correctly, I have these many test cases to ensure that all possible scenarios are being tested like testing the name isn’t null after modification and name isn’t too long after modification for EX:

**void** validateNameNotNullAfterModification() {

Task task = **new** Task("12345", "Old Name", "Valid Description");

Exception exception = *assertThrows*(IllegalArgumentException.**class**, () -> {

task.setName(**null**);

});

*assertEquals*("Invalid name, name is null or name length is too long", exception.getMessage());

}

**void** validateNameNotLongAfterModification() {

Task task = **new** Task("12345", "Old Name", "Valid Description");

Exception exception = *assertThrows*(IllegalArgumentException.**class**, () -> {

task.setName("This name is way too long to be valid");

});

*assertEquals*("Invalid name, name is null or name length is too long", exception.getMessage());

}

and in the TaskService test class I have 9 test cases to ensure that the inputs are validated and correctly processed when using task Service class, in total my JUnit testing coverage is at 87.5% and my testing methods ensure that requirements are being met in the program.

The last piece of project one is the Appointment and AppointmentService Classes and their testing units, in short: the requirements are that the ID can’t be longer than 10 or null and cant be updateable, the Date can’t be in the past or null, and the appointment object should have a required description string that can be longer than 50 or null, for the appointment service class users should be able to add and delete appointments using the appointment ID. In appointmentTest class, I have 7 test cases and 5 test cases in the appointmentServiceTest class and it’s to test validating inputs and managing errors internally, JUnit testing coverage is at 87.5% for this program.

For this project I used White-Box (Unit) Testing to test the isolated components of my programs, my Testing also entailed exception testing or negative testing and boundary testing, & I tested the program’s behavior in the 3 service classes. For testing techniques I didn’t use, I didn’t run a performance test an example case where this would be needed is let's say the program is run by a vast number of users at once and we don’t know how the software would handle the influx of tasks given at once, I didn’t run a User Acceptance test which would mean letting potential users test it and tell us about their experience using it before fully releasing it.

The mindset to have as a tester is to think about the consequences of a vulnerable product in terms of functionality and user safety & privacy and the effects of that on the company and the reputation of individual testers and developers in that company, I make sure that I was cautious when testing this program by thinking about all the different possibilities where I could miss and then developing test units for those specific possibilities. That’s what drove me to develop plenty of test cases, each designed to test an isolated possibility and test isolated functions and behaviors to ensure that all the little pieces work together in the intended way.

To limit my bias when testing my own it’s best to automate the testing results to the computer so it can test them without bias, like utilizing assertion methods to verify results. This way I can focus on developing the code per the requirements and think about all the different cases that I would need to enforce the requirements and then code test cases for them, then let the machine without bias decide if the test cases are good enough or I need to go back and remodify something to get the intended results.

It's imperative to be disciplined when developing the test units because it shows you how all the little pieces of the program work and that requires testing all the smaller functions and behaviors of the program so that way the entire program works the way intended, not being disciplined at this stage will lead to the need to refactor the code later on when the problem is eventually found and that is called technical debt which is future more time spent on refactoring code that should’ve been right when developed the first time with less time, the way to avoid this is by taking this stage seriously and spending time at testing everything in the program no matter how time-consuming and frustrating it is of a task.