Garrin Neelly

CS320

SNHU

12/23/21

When testing for these three features, the important testing was static and dynamic testing. To ensure that the features both compile and function logically. To do this I set up in each feature the Junit tests to ensure that correct data was being input into the objects and that they could be correctly updated. Each feature had its own requirements; however, a common utility was the ability to create, update and delete entries in each system. I tested this in all of them and got positive test results for each, ensuring that appointments, contacts and tasks. An example of this is in the appointments service testing file. “

@Test

public void testAdd() {

LocalDate date= LocalDate.now();

AppointmentService as = new AppointmentService();

Appointment test1 = new Appointment("1234567890", date, "First meeting to gather requirements");

assertEquals(true, as.addAppointment(test1));

“This test adds a new appointment and ensures the appointment exists at the end of the test.

Testing for these kinds of errors also ensures multiple other values are correct, not just the entire entity. With certain tests one can ensure that the values in the entity are correct for the objects' requirements as well. When adding an object, the objects themselves check that the data entered follows the requirements, so when an object is successfully created, the values follow the requirements without having to explicitly test for it.

My experience with Junit tests was positive. For most of the tests I ensured that the final value equaled some value that would ensure the correct outcome and show that there were no logical errors occurring for the code. And that the file would compile and run ensured there were no static errors as well. Junit tests in this regard ensure both static and dynamic testing happens with fewer errors. I did run into some logical errors that would break the program, and while the system would compile it would not run all the way through, this was only visible clearly when the Junit tests would fail. These tests helped me find where the error was occurring and enabled me to fix the error. In lines 32-38 of contactTest shows how the tests are efficient, I created the same test for each variable to get and set to ensure the correct value after it was run.

For these projects I used White Box Testing, as I was aware of the source code and the way the code should normally operate. I did try and test for out of parameter issues, but most of my testing was to ensure that the correct values could be input in the varying objects. White Box testing is when the tester has intimate knowledge of the source code and what the code should be accomplishing, this is good for optimizing the code, ensuring correct values can be input, and you can test for each utility of the program.

White box is not useful on too large a project, or as a whole. The testing technique is used for finding specific issues or ensuring small portions of the code are working as intended. Black box testing is used more for substantial amounts of code. While black box is difficult to determine the tests, it is like looking at the program from a user perspective, the tester does not have access to the source code and must instead infer what can be input.

When developing these projects and employing Junit tests to ensure functional code, I tended to be more direct in my approach. I could have been more cautious by implementing more tests to ensure multiple different error vectors were covered, however, I feel that with the tests and intimate knowledge of the source code, I could simply test to ensure a bad input was caught and good one was then input correctly. While I was not as diligent in my testing as I could have been, I also did not cut corners when it came to overall testing. I tested each facet of the program to ensure that the requirements were met, each had their own requirements like appointments required specific lengths of each variable to not be exceeded. I tested both that appointments were working on a basic level and then tested the service to ensure that it was able to acquire the correct information. Not testing each tier of the code could have drastic consequences later in development. This could have happened if the appointment were not checking for the length of its ID must be less than 10. If this were not implemented correctly, later when testing the service, the test would fail if a value with too long a length was implemented, but it would fail at the service level and not the original tier. “

public Appointment(String Id,LocalDate date,String description){

System.out.println(LocalDate.now());

if(date !=null ) {

this.date=date;//if the input date is valid

}

else{

System.out.println("Invalid LocalDate");

//System.exit(0);

}

if(Id.length()>0 && Id.length()<=10) {

this.Id=Id;

}

else{

this.Id="";

System.out.println("Invalid Appointment Id");

System.exit(0);

}

if(description.length()>0 && description.length()<=50) {

this.description=description;

}

else{

System.out.println("Invalid Appointment Description");

System.exit(0);

}

}”

In the original constructor the length is all tested to ensure it does not exceed the requirements. Then later in service, when updating the information, the length of the update needed to match requirements as well or else the length would be too great and would fail the test. In both instances the values must be checked, if you do not ensure a valid entry, then later you will have greater issues beyond failed Junit tests, the program will start to give the wrong output and this can lead to unhappy customers, and users of the programs.