Project Two

Robert Robbins

Southern New Hampshire University

CS-320

Professor Angel Cross

October 14, 2024

**Summery**

My testing approach was aligned to the requirements of the project in that I went through the list of requirements and created a test for each requirement. An example of this being aligned to the requirements is that each of the services required that when a object was created, that object had a unique id field that could be used to identify it. The requirements for the id field were that it was a String type, was less than ten characters, was unique, could not be “null” and could not be changed. To make sure that the requirements were met I created the id field so it was a private final type, and used a static function that created the id during construction of the object. This means that the id field can only be accessed using object methods. I then only used one object method to get and return the id field when called which means it is not possible to update the id String. To test that each object was created with a unique id I crated a test that created two objects and compared their id fields. If the id fields were different the test passed which meant the id strings were unique. The method below is from the ContactClassTest file which tests for unique ids.

@Test

void testUniqueContactId() {

ContactClass contactOne = new ContactClass("Robert", "Robbins", "1234567890", "Short address");

ContactClass contactTwo = new ContactClass("Robert", "Robbins", "1234567890", "Short address");

assertNotEquals(contactOne.getContactId(), contactTwo.getContactId());

}

The test coverage for each test class ranged from 94% to 100% coverage. Coverage is the percent of statements and conditionals that are tested. In the test classes that didn’t make it to 100% I made sure that there was not any critical statements that were missed and in the case of the test classes that didn’t reach 100% covered it was due to loops that didn’t complete or a static class that wasn’t instantiated. Test coverage is important because it will show you what areas are missed and makes it easier to ensure everything important gets tested. Another way I ensure the quality of my tests was through the use of assertions. For example when testing that a Contact object could be created correctly I used assertions to check if objects were created with the correct data. An example of this is in the ContatClassTest class I created a test to check that an object could not be created with a null name.

void testNullLastNameContactCreation() {

ContactClass nullLasttName = new ContactClass("name", null, "1234567890", "Short address");

assertNotNull(nullLasttName.getLastName());

This test creates a Contact object with null as the first name and then uses an assertion to show that the object is not created with the first name as null.

To ensure that my tests worked as desired I kept each test simple and checked for specific criteria. I went through the list of requirements and made sure each requirement had a test. An example of this is when testing the getter methods. I created a object and then used each getter method and tested that it matched what the object was created with. Keeping my tests simple also helped keep the the tests efficient in that if there is an error and the test fails it is easy to identify where the issue is. An example of this is not testing multiple requirements in the same test.

**Reflection**

The testing techniques I used for this project were white box testing and code review. White box testing is when you have access to the code your are testing and an understanding of how it works. With white box testing you can ensure that all critical areas of the project are tested. An example of how I used white box testing is creating the tests for the unique ids for each class. Because I can see and understand that there is no possible way that a id can be changed or updated I didn’t waste time trying to make a test to update the id. Code review is where a person goes through and reads through the work product to check for any mistakes or errors. This can be a formal or informal process depending on the company and what is being reviewed. During the milestones I didn’t do very much reviewing except when I encountered an error. But when submitting the final project I went through each file and checked for mistakes. This lead me to rewrite a lot of the Contact Service portion of the project.

A testing technique I didn’t use for this project is black box testing. This is testing done from the perspective of the customer or user. The person doing the testing does not have any knowledge or understanding of whats going on inside. This type of testing can uncover scenarios that might have been missed earlier in the project. An example of something that might be uncovered while doing black box testing for this project might be if an appointment object was created with the same date as another appointment object. This might cause the requirements to be reexamined and a new requirement added.

One way that it is important to exercise caution is by understanding the complexity of the project and how quickly the amount of test can grow. One example I found while writing my tests was testing that none of the object fields could violate the requirements. In the Contact class there are five fields that are assigned values when creating the object. These are the contactID, first name, last name, phone number and address. The contactId filed is automatically generated but the four other fields require input. Each of these fields is required to be within a specific length range and not be null. I structured my tests so there was one test to test if an object could be created with all the fields being valid, and then a test where one field was invalid and the rest were valid. Where I exercised caution was when I asked myself what happens if multiple fields are invalid like all fields entered null or two fields with invalid lengths and the other fields valid. Testing these other scenarios would require more tests but would it add a lot or just a few. There are three possible inputs for each field which are valid, null or invalid length, and there are four fields so the math for testing every possible scenario is 3\*3\*3\*3 or which is 81 tests. I did not write 81 tests because this project is simple and in the source code the 3 fields don’t depend on each other when running. However its easy to see from this example how great the complexity of testing can become.

It is incredibly important as a software developer and tester to strive for the greatest quality in our work and avoid being biased. An example I can think of where bias can come into play is by looking at the test coverage. All of my tests show that test coverage is greater than 94% covered which makes it look like the testing is done well and is correct. The test coverage just shows that all of the statements are tested but doesn’t show anything about the quality of the tests. The test coverage could be 100% but all of the test just have assertions validating that whatever the was tested is true or correct. For example if there is a test to show no null value is accepted, but due to a typo assertNull is used instead of assertNotNull. The value is being tested is in fact null but the test passes. The bias would be that all of the tests are solid when in fact they are not. To avoid this kind of mistake it is important to not to assume that a good test coverage translates to good tests.