Cameron Clarke

Southern New Hamspire University

CS-320

August 18, 2024

Project Two- Summary and Reflections Report

I. SUMMARY

I would probably say the testing approach we used would probably be best described by Hambling as methodical based testing which “rely on the systematic usse of some predefinted tests or test conditions such as failure based (including error guessing and fault attacks), checklist based and quality-characteristic based. This was used majority because we were looking for more functionality based requirements more so than delivering a project to be implemented into say a website and we would need to test its functionality that way.

So for the quality of the tests, I had at least 34/37 unit tests passing, and we allowed for over 90% coverage. For one of the requirements that we had four one of the three services that we provded, let’s go with one of our first ones. The Contact Service should be able to update contact fields per contact ID, which are firstName, lastName, Number, Address. To implement this we of course had these private members being able to accessed by mutator methods. Then with each member, we have a method to update inside of the contact list with the contactID as the parameter and the new member. For example.

public void updateAddress(String contactID, String newAddress) {

Contact contact = getValidatedContact(contactID);

contact.setAddress(newAddress);

}

So when I created the test, I updated all the fields by already having a contact inside with the ID of 1. And went to update it through the methods. Through the JUnit testing capabilities, I tried to verify all contact field updates by AssertAll. The reason I used AssertAll is because we wanted to make sure that every assert is done, because if we just implemented multiple asserts, it could possibly cause the entire test to pass or fail. Below is the code assigned to this JUnit test implementation.

@Test

public void testUpdateContactFields() {

contactService.updateFirstName("1", "Jane");

contactService.updateLastName("1", "Smith");

contactService.updatePhone("1", "9876543210");

contactService.updateAddress("1", "456 Elm St");

assertAll("Verify contact updates",

() -> assertEquals("Jane", contactService.getContactById("1").getFirstName()),

() -> assertEquals("Smith", contactService.getContactById("1").getLastName()),

() -> assertEquals("9876543210", contactService.getContactById("1").getPhoneNumber()),

() -> assertEquals("456 Elm St", contactService.getContactById("1").getAddress())

);

II. REFLECTION.

We implemented a few testing techniques however the most common of course was the Equivalence Partition Technique featured by Hambling (2019). “The equiavlence partition technique takes advantage of the properities of equivalence partitions to reduce the number of test caseds we need to write. Since all values in an equivalence partition are handled in exactly the same way by a given program, we need only test one of them a sa reprensatative of the partition.” (Hambling, 2019). For example for our Task, there is a requirement that the name cannot be null or under 20..

public void setName(String name) {

if (name == null || name.length() > 20) {

throw new IllegalArgumentException("Invalid task name");

}

this.name = name;

}

Instead of having to make sure that our test is valid for all lenghts of 1 through 19. We choose a value that is somewhere in the middile to reduce our test use cases.

@Test

public void testSetNameInvalid() {

Task task = new Task("12345", "TaskName", "TaskDescription");

assertAll("invalidSetName",

() -> assertThrows(IllegalArgumentException.class, () -> {

task.setName(null);

}),

() -> assertThrows(IllegalArgumentException.class, () -> {

task.setName("ThisNameIsWayTooLongToBeValid");

})

);

}

For techniques that we have yet to implement there is a vast wide of techniques we didn’t implement. Mostly because it wouldn’t suit our puposes for this project. For example, maintenance testing is used for making sure that the proccess doesn’t break when continually running. For example, the projects’ statement that it was to be implemented in a web based application. We could do maintenance testing to make sure that when it is running, the proccess does not falter in its user reequirements or as intended.

Bias and cutting corners are poisonous to any project working for anything. It goes against ethical practices that ensure the benefit to your organization, to the customer and to public safety. We try to limit bias by sticking to objective markers and requirements that have little to no room for interpretation. This also includes communication with the customer and fellow software developers in order to make sure that your perspective isn’t the only thing that matters when immplementing these tests. When we doi manage to cut these corners the consequences can be devastating from project price overrun, to delays in implementation, as well as safety concerns that plague the industry and many others.

WORKS CITED

Hambling, Brian. *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*. BCS, The Chartered Institute for IT, 2019.