Raphael Coloma

CS-320-T4205 Software Test Automation & QA

April 9, 2023

Project 2: Summary and Reflections Report

Grand Strand Systems was tasked to develop a mobile application for a customer that would add, update, and delete contact, task, and appointment objects within the application. With developing the application, unit tests were expected to verify that the application meets the customer's requirements. The requirements defined the customer’s expectations for the application and offered a means of measuring its level of completion. When developing the application, I made sure to address each of the requirements established by the customer. Despite not needing a UI, input was expected and thus needed to be verified. Many of the requirements dealt with input validation, ensuring that the data entered fit specific criteria: the values could not be null and defined character limits needed to be established for the object variables. If an input value failed any of these conditions the application would trigger an error and end execution. Each decision statement was tested, and the resulting branches were verified to be correct.

JUnit test classes were developed for each of the object and object service classes. The object class JUnit tests verified the creation of its respective instance and that the instance variables were being properly assigned in line with the requirements.

*@Test*

*//Test Contact object Constructor*

*void testContact() {*

*//Create new Contact object*

*Contact contactClass = new Contact("Rian", "Coloma", "123456789", "Marlia St.", "5555553110");*

*//Assertions to check Contact object variable*

*assertTrue(contactClass.getFirstName().equals("Rian"));*

*assertTrue(contactClass.getLastName().equals("Coloma"));*

*assertTrue(contactClass.getContactID().equals("123456789"));*

*assertTrue(contactClass.getContactAddy().equals("Marlia St."));*

*assertTrue(contactClass.getPhoneNum().equals("5555553110"));*

*}*

The object service JUnit tests checked the application’s functional requirements. The application needed to be able to add, update, and delete contact, task, and appointment objects within the application. Since no database was required, instances were stored on in hash maps. The JUnit test classes created new hash maps for the instances and called each of the object service classes’ methods. Each test would pass input values for the new instances and validate that the methods were executed correctly.

*@Test*

*//Test AddContact Method*

*void ContactServiceAddTest() {*

*//Create a new Contact object*

*Contact contactObj = new Contact("Raph", "Coloma", "123456789", "Las Vegas", "5555551234");*

*//Method call to add contact to hash map*

*ContactService.AddContact(contactMap, contactObj);*

*//Assertion to check Contact object variables in hash map*

*assertAll("Add",()-> assertEquals("Raph",contactMap.get("123456789").firstName),*

*()-> assertEquals("Coloma",contactMap.get("123456789").lastName),*

*()-> assertEquals("Las Vegas",contactMap.get("123456789").contactAddy),*

*()-> assertEquals("5555551234",contactMap.get("123456789").phoneNum));*

*}*

The purpose of the JUnit tests was to ensure that the application was meeting the requirements of the customer. The test coverage of the application classes needed to be 80% or higher to be deemed adequate. Coverage percentage indicates how much of the code was tested and verified through execution. By executing each line of code, I ensured that the application was properly tested. After refactoring and incorporating additional tests, I was able to achieve no less than 95% cover for each of the object and object service classes.

Graphical user interface, table, Excel

Description automatically generated

Complete code coverage is desired, but not always feasible. Having a high coverage percentage ensures that the application will perform as expected with few variants.

When writing the JUnit test, it was important to fully understand the customers requirements and how they affected the application. I wanted to make sure that the application was abiding by the requirements and that the tests were properly verifying them. As mentioned, complete code coverage is not always feasible, and this is in part due to decision statements. Though my code was working correctly, when checking the coverage, I could see portions of my code not being executed. Since I was using an “or” qualifier in my “if” statements, I needed to test both conditions to confirm the appropriate code is being executed. By adding additional input value test, I was able to check both conditions for the decision statements.

*//Phone Number Setter*

*protected void setPhoneNum(String phoneNum) {*

*if(phoneNum == null || phoneNum.length() != MAXNUM) {*

*throw new IllegalArgumentException("Invalid Phone Number");*

*}*

*this.phoneNum = phoneNum;*

*}*

*@Test*

*void testContactClassPhone() {*

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

*new Contact("Rian", "Coloma", "98765431", "Marlia St.", "5553110");*

*});*

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

*new Contact("Rian", "Coloma", "98765431", "Marlia St.", null);*

*});*

*}*

Efficiency is an important factor to consider when developing code. After checking the test coverage, I could see blocks of code not being executed due to redundancies in the object class methods. The major issue lay with the constructor methods conflicting with the mutator methods. I corrected this issue by refactoring the code to have the constructor methods call the mutator methods to assign the passed input value to the instance variable when created.

//Constructor passing all Contact Class variable arguments

public Contact(String firstName, String lastName, String contactID, String contactAddy, String phoneNum) {

//Contact ID requirement

if(contactID == null || contactID.length() > MAXID) {

throw new IllegalArgumentException("Invalid Contact ID");

}

//Call setters to assign class arguments

this.contactID = contactID;

setFirstName(firstName);

setLastName(lastName);

setContactAddy(contactAddy);

setPhoneNum(phoneNum);

return;

}

After refactoring, the test coverage percentage went up and I could see all the methods were being executed. The JUnit tests helped to evaluate the efficiency and effectiveness of the application code.

As described, the primary white-box testing techniques I used were coverage and decision testing, but I also incorporated different black-box testing techniques in this project. For input validation, I used equivalence partitioning. To test the character limit for the object variables, I partitioned the inputs based on character counts and determined how the system would react given strings of varying length. I used decision tables to track how instances satisfied the application requirements. By defining the conditions for success or failure, I was able to create tables that would plot the outcome of iterations and help develop how I would test input values. I also used state transition tests to evaluate how the hash maps handled instances. I wanted to make sure that adding, modifying, and/or deleting an instance in the hash map did not affect the other instances being held. While having its own merits, I did not use boundary value analysis because I was primarily dealing with strings and not integers. Also, I did not use test cases, though testing a system through various perspectives is a valuable approach when evaluating possible errors. But, in this instance I was testing from the perspective of the end-user since other roles were not accounted for. Lastly, I did not create diagrams to illustrate the possible choices and outcome for the system, though I see the value in visualizing the steps a user would have to take when operating the software. Granted, a flow diagram would have assisted in ensuring my data was being properly stored and processed.

Going into development, I focused on the requirements first. I used them as a foundation for the rest of the application. Caution came when I started testing. Though the application appeared to be working, the coverage tests showed that the code execution was not flowing as initially planned. In addition to the assertion statements in the JUnit test classes, I needed to add print statements to see how the input values were being stored and what was being passed. Understanding how your code works will help design tests that will best determine its effectiveness. As mentioned, I was able to find defects in my code only after testing and checking the coverage percentage. In that same vein, developers should be aware of their own bias. I am of the mind to not have a developer test their own code. Since they created it, they understand it better than anyone. With this inside knowledge, it is possible to design tests that do not exercise the effectiveness of the code, but instead succeed due to this bias. I do not believe malicious intent is the motive but with pride and time constraints, I think tests could be developed to preserve those.

While I am confident in my code, I believe there could be more tests I could conduct to bolster the security of the application. I tested the code I developed, but there still exists factors that were not safeguarded against that would prolong development if incorporated. Once an application is release, it is free to be scrutinized and dissected by anyone. The testing period is miniscule compared to the lifespan of an application once it is released. It would be impossible to test and review all the possible outcomes during the testing phase. Depending on the level of risk involved, this period will vary. Still, remaining disciplined in your commitment to quality will ensure that you will produce a sound product. Software development starts before writing actual code. It is important to plan and review all available resources and requirements to ensure complete understanding of the application. Failure to adequately test code could result in devastating loss of revenue, time, reputation, and sometimes life.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An*

*istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.