Patrick Kempf

CS – 320

06/21/2024

Southern New Hampshire University

Project Two Summary and Reflection

My main goal when approaching project, one was to create classes that met the software requirements in an efficient way. We were given clear requirements for each class so there was a clear path to meeting them. Along the way I did misinterpret the requirements for the contact service class and made the contact as a whole updatable instead of the individual components. This was fixed in the final iteration of the class and each aspect is no updatable individually. Overall, my approach to this project was to meet the software requirements and confirm their functionality through testing.

While I do not have an exact percentage of coverage for my JUnit tests, every function is covered by a test. If we look at the contact test class it is covering mostly failed cases. Each variable of the contact class is tested by inputting an invalid entry and testing if the system catches it. An example is below.

private static void testCreateContactIdFails() {  
 try {  
 Contact contact = new Contact("12345678910", "Patrick", "Kempf", "1234567899", "1 West Essex St");  
 System.*out*.println("Id too long test failed");  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Id too long test passed");  
 }  
  
}

Since our contact and contact service classes are interconnected the test coverage for the contact class carries over to contact service. In the contact service test class, we are testing if our functions are working correctly. In those tests we are creating or updating a contact and checking if it was successful. An example is below.

public static void testUpdateContactFirstNameSuccess() {  
 ContactService contactService = new ContactService();  
 Contact contact = new Contact("1", "Patrick", "Kempf", "1234567890", "1 West Essex St");  
 contactService.addContact(contact);  
 String contactId = contactService.getContactList().get(0).getContactId();  
 contactService.updateContactFirstName(contactId, "John");  
 Contact updatedContact = contactService.getContactList().get(0);  
  
 if (updatedContact.getFirstName().equals("John")) {  
 System.*out*.println("Update Contact First Name Test Passed");  
 } else {  
 System.*out*.println("Update Contact First Name Test Failed");  
 }  
}

Since our contact test class covers tests regarding valid entries, we do not need to test for them in our contact service tests. Through the combine testing of the contact and contact service classes we ensure that every aspect of our code is technically sound and functional. Overall, I tried to use concise functions and tests for this program with little to no redundancies.

For my testing I used three main types of testing; unit testing, boundary testing, and exception testing. The unit testing consisted of testing individual functions such as the add contact test below.

public static void testAddContactSuccess() {  
 ContactService contactService = new ContactService();  
 Contact contact = new Contact("1", "Patrick", "Kempf", "1234567890", "1 West Essex St");  
 contactService.addContact(contact);  
  
 if (contactService.getContactList().size() == 1) {  
 System.*out*.println("Add Contact Test Passed");  
 } else {  
 System.*out*.println("Add Contact Test Failed");  
 }  
}

An example of boundary testing is one of the tests in contact test that check if an input that is too long is flagged by the system. An example of this is below.

private static void testCreateContactIdFails() {  
 try {  
 Contact contact = new Contact("12345678910", "Patrick", "Kempf", "1234567899", "1 West Essex St");  
 System.*out*.println("Id too long test failed");  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Id too long test passed");  
 }  
  
}

This tests checks what happens when an ID that is too long and outside the boundaries is input. If the test is successful an illegal argument is thrown and Id to long test passed is print in the console.

Lastly, we have exception tests. A good example of this is the duplicate ID test below.

public static void testDuplicateContactIdFails() {  
 ContactService contactService = new ContactService();  
 Contact contact1 = new Contact("1", "Patrick", "Kempf", "1234567890", "1 West Essex St");  
 contactService.addContact(contact1);  
  
 try {  
 Contact contact2 = new Contact("1", "John", "Doe", "9876543210", "1 Elm St");  
 contactService.addContact(contact2);  
 System.*out*.println("Duplicate ID test failed");  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Duplicate ID test passed");  
 }  
}

In this test we are checking to see if the system flags a duplicate ID that is input. By creating a second contact with an ID that matches the first the system should throw an illegal argument exception and print a test passed line into the console.

Because this project consisted of relatively insular parts many testing strategies were not practical. Load and stress testing determines how the system as a whole functions under an expected load or scale. Load and stress testing are excellent for determining how efficiently a system will run once it has a full user base interacting with it. Regression testing is also more applicable to large systems. With regression testing previous builds would be tracked and cataloged so if a bug is added developers can look back on previous builds and determine what caused the error. Lastly system testing determines how the system as a whole functions once individual classes are put together. In the case of our project the goal was to create tests for individual classes and the assembly of the system never came to fruition.

It was an interesting experience writing tests for this project. I have done testing in the past, but nothing like this. Writing additional classes to test each aspect of our code proved an interesting task. When writing the actual classes I tried to ensure that when tested the proper exceptions would be thrown and the proper entries would be returned. Writing code knowing that it is going to be heavily scrutinized, even if it is by myself, made me consider what I was writing even more. The more difficult part of the project was writing the tests. I have made classes similar to the projects before, but I have never had to test them so rigorously.

I am sure there is a lot of bias when developers look at their own code, but honestly, I do not put myself on a pedestal at all when it comes to coding. I know I make a lot of mistakes so I am certainly not above scrutiny. I imagine an ingrained developer may have more bias when testing their own code. There is incentive for developers to pass tests and it is a bit of a conflict of interests to test your own code in most cases.

For me the most important take away from this class is that in the long run it is much more efficient to test code as you go and before release than to fix things after the fact. Cutting corners just ends with more work in the future. It is vital to write efficient and functional code, check it as you go, and rigorously test it. The projects we are doing are not vital by any means, the stakes are not very high. In the future many of us may be in situation where a bug that sneaks through costs a fortune to fix after the fact. There have been many cases where lives have been lost because of errors in programs. It is important to build habits and skills that lead to our success in the future, to not cut corners, and to put our best work out into the world.