Aaron Fehir

CS 320

August 22, 2021

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.
      2. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

My unit testing approach for each of the three features utilized a checklist-based model in order to test that the software aligned to the requirements. This is known as a “methodical approach” to software testing (Hambling, 2015, p. 151).

For example, requirements governing the input of data require that entries cannot be null and must meet certain length requirements. A task ID cannot exceed 10 characters. If an ID that exceeds the character limit is entered, an exception should be thrown. Junit testing confirms this behavior.

Consider the tests for the contact service. My tests for the contact service worked as expected. When I run the tests, I get the desired green bar within Eclipse indicating no problems. However, some tests did run into problems, such as the test for the task service where the bar shows red rather green.

Coverage for the contact service test is 66.3%, which is low, but acceptable. The coverage for the task service test is 26.1%, which is very low and indicates some potential problems with the code.

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.
     2. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

Consider the following test:

@Test

void testContact() {

Contact contact = new Contact("abc123", "Bob", "Smith", "0123456789", "123 Pine Rd.");

assertTrue(contact.getContactID().equals("abc123"));

assertTrue(contact.getFirstName().equals("Bob"));

assertTrue(contact.getLastName().equals("Smith"));

assertTrue(contact.getPhone().equals( "0123456789"));

assertTrue(contact.getAddress().equals("123 Pine Rd."));

}

The above test makes sure that the parameterized function enters the values into the specified fields as desired. So calling getLastName gives Smith, for example. If these asserts were coming back negative, we would know we have a problem.

How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.

Consider the following lines of code:

//Test delete method

@Test

public void testDelete() {

ContactService cs = new ContactService();

Contact contact1 = new Contact("abc1230", "Bob", "Smith", "0123456789", "123 Pine Rd.");

Contact contact2 = new Contact("abc1231", "Joe", "Smith", "0123456789", "123 Pine Rd.");

Contact contact3 = new Contact("abc1232", "Bill", "Smith", "0123456789", "123 Pine Rd.");

cs.addContact(contact1);

cs.addContact(contact2);

cs.addContact(contact3);

assertEquals(true, cs.deleteContact("abc1231"));

assertEquals(false, cs.deleteContact("abc123x"));

assertEquals(false, cs.deleteContact("abc1231"));

}

This code creates a new contact service object and then three new contacts. The asserts show that a field with a matching ID is deleted and then checks to make sure it is no longer present in the list. The program functions as expected.

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.
      2. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details.
      3. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

I used a methoficaal approach (Hambling, 2015, p. 151) to testing this piece of software. In particular, I used the checklist approach. Checklist based testing is involves making a checklist based user stories and/or software requirements. The requirement that IDs must be a specific length, for example, is added to the checklist. And then Junit testing is used to check in each case that input failing outside the required parameters generates an rrror.

Other software testing techniques not used in the course of this project would include standard- compliant approaches. Such approached rely on the use of industry-specific standards (Hambling, 2015, p. 151), which wasn’t applicable within the confines of this project.

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.
     2. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.
     3. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

Caution must be exercised when developing tests for software. You must be careful, for example, that tests are not ambiguous. Each test should pass or fail for specific known reasons. You wouldn’t want tests to be so generic that they fail on multiple fronts. For example, you want to test specifically for length requirements and then you should test separately for character-type requirements. If a password must contain a special character and must have a length of at least 7, testing with “rock32” would not be helpful as it fails on more than one front. It would be better to test with “rock321” to check that it detects the lack of a special character and then separately “rock@” could be used to test the length.

Bias is a potentially problematic issue. It is tempted to only look for what we want to see. When testing, it is important that the tests conform to the software requirements rather than the code. The temptation is to write tests that we know will pass based on the code rather than really testing the limits by means of edge cases, for example.

Finally, an overriding value that applies to software development generally and testing specifically is discipline. It is important that we do not get sloppy and let mistakes pass in the code. Discipline requires that we diligently check each requirement and try our hardest to write test that result in high coverage.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., and Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide*, 3rd Ed.