Module 4; Journal Entry

Jason Kremhelmer

Southern New Hampshire University

CS320: Software Test Automation

Karl Lewis

31 May 2014

**To what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.**  
  
In order to test the boundaries of the specification, I used the Unit Test. In the specification of the Task, it states that the full name can’t be greater than 20 characters, as well as it’s mandatory. In order to test this, no less than 4 branches are needed.   
  
if (fullName == null || fullName.length() > 20 || fullName.equals("")) {return false; }  
  
For example, I tested a valid version of the name, a null string, a very long string, and an empty string.  
  
 Assertions.assertThrows(IllegalArgumentException.**class**, () -> {

testContact.setFirstName("MyFirstNameCanNotBeThisLong");

});

}

**Defend the overall quality of your JUnit tests for the contact service and task service. In other words, how do you know that your JUnit tests were effective on the basis of coverage percentage?**  
  
With this test, 100% coverage was not my goal, instead I targeted 100% of the branch’s getters and setters, and only a good constructor, and an invalid one. There are 3+ parameters to set, I did not believe it worthwhile to test both good and bad Constructors for each parameter.   
A screenshot of a graph

Description automatically generated

**How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**  
  
In ContractService there is a method to delete an ID. This function could be tested multiple ways.

* Add three objects to the collection, delete one, and confirm the size dropped by one, to two.
* Parse the entire collection to validate that it is no longer present.

One issue with this method is you are only checking the collection size. If the wrong item is deleted, the number would still drop by one, just not the correct one, and the test would pass. Also, if the wrong item was accidentally deleted, and all you did was search for said items, you would also still pass. Therefore, doing a combination of the two tests is what I opted for:

@DisplayName("Test deleteContact")

@Test **void** testDeleteContact() {

String firstName = "Steve";

String lastName = "Thomas";

String phoneNumber = "1238675309";

String address = "123 Parker Street";

**boolean** testBool = **false**;

ContactService test = **new** ContactService();

assertTrue(ContactService.contactList.isEmpty());

test.addContact(firstName, lastName, phoneNumber, address);//object at ID 0

test.addContact(firstName, lastName, phoneNumber, address);//object at ID 1

test.addContact(firstName, lastName, phoneNumber, address);//object at ID 2

assertEquals(3,ContactService.contactList.size());

test.deleteContact("1");

assertEquals(2,ContactService.contactList.size());

//loop through to look for ID

**for**(**int** i = 0; i < ContactService.contactList.size(); i++) {

**if**(ContactService.contactList.get(i).getContactID() == 1) {

testBool = **true**;

}

}

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

As noted in the above section, just checking the size of the collection would not suffice. You must also verify that the correct item is deleted. An additional step could be to ensure the desired items are still intact.

A test that I did complete was ensuring that objects were instantiated only when requested.

ContactService test = new ContactService();

assertTrue(ContactService.contactList.isEmpty());

Also, I checked that after a failed string update the string was still as before the attempt:

tempTask.updateTasks("1", fullName, "New description"); //bad ID

assertNotEquals("New description", TaskService.tasks.get(id).getDescription());

assertEquals(fullName, TaskService.tasks.get(id).getName()); //original string