Lab Report for Software Engineering course
Lab 3: Starbubucks coffee online retailing system v2.0

Wang, Chen	Liu, Jiaxing	Huang, Jiani	Tang, Xinyue
16307110064	17302010049	17302010063	16307110476

School of Software
Fudan University

April 1, 2019

Contents

Chapter 1

Overview of this Lab

1.1 The Objectives of the Project

We are going to try to learn to use the code unit testing tools, e.g. JUnit/JMock, in this lab so as to experience test-driven development and the influence of the change in code quality and requirements on the process of development.

The specifications, division of work and the detailed implementation of the work is shown in the sections below.

1.2 Specifications of the Lab

In this lab, we are required to accomplish the tasks including designing and implementing new interfaces, performing unit testing design and development and some other works. Specifically, the works are in the form of the following parts:

1. Implement the interfaces as required in the lab requirements;
2. Perform code unit testing design and develop code unit test;
3. Link the code commits with the project work items in the project planning;
4. Develop in groups based on Git and hand in the lab report.

   Our group members acted actively in their own roles together to finish the whole project and below are the detailed working results of our group.

1.3 The division of work in the team

1.3.1 Division of work: Huang, Jiani

Build the JUnit environment and write all required environmental methods except the *@test* method, such as *@before* and *@after* methods; write the *@test* method for the *login()* method.

1.3.2 Division of work: Liu, Jiaxing

Complete the *checkName()* and *checkPassword()* test method.

1.3.3 Division of work: Wang, Chen

Implement the *checkName()* and *checkPassword()* methods.

1.3.4 Division of work: Tang, Xinyue

Finish the test method of *signUp()*, *checkStatus()* and *cost()*.

Chapter 2

Tools adopted for quality analysis

2.1 Junit

2.1.1 Description of JUnit

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit.

JUnit is linked as a JAR at compile-time; the framework resides under package *junit.framework* for JUnit 3.8 and earlier, and under package *org.junit* for JUnit 4 and later.

2.1.2 Test fixture of JUnit

A JUnit test fixture is a Java object. With older versions of JUnit, fixtures had to inherit from *junit.framework.TestCase*, but the new tests using JUnit 4 should not do this. Test methods must be annotated by the *@Test* annotation. If the situation requires it, it is also possible to define a method to execute before (or after) each (or all) of the test methods with the *@Before* (or *@After*) and *@BeforeClass* (or *@AfterClass*) annotations.

We have adopted *JUnit version 4.12* and the library is from *Maven* remote repository. In our implementation of JUnit test fixture, we have well utilized the *@Before* and *@After* method to initialize and discard objects that will be used in all other *@Test* methods.

2.2 JMock

JMock is a library that supports test-driven development of Java code with mock objects.

In this Lab, JMock objects can help us design and test the interactions between the objects in your programs.

The jMock library has the following advantages, it

1. makes it quick and easy to define mock objects, so you don't break the rhythm of programming.
2. lets you precisely specify the interactions between your objects, reducing the brittleness of your tests.
3. works well with the autocompletion and refactoring features of your IDE
4. plugs into your favourite test framework is easy to extend.

Chapter 3

Features added to the project

3.1 User name checking

3.1.1 Requirements for user name checking

The username should satisfy the following requirements:

1. The username must start with **starbb_**;
2. The username can consist of **letters**, **numbers** and **underline**, excluding any other symbols;
3. The username should have a length greater than or equal to 8 and less than 50.

3.1.2 Interface for the checking method

```
/**                                                    1
 * Check whether the given name is valid              2
 *                                                    3
 * @param name the given name to check                4
 * @return whether the name is valid                  5
 */                                                   6
boolean checkName(String name);                       7
```

3.1.3 Implementation of the checking method

```
/**                                                    1
 * Check whether the given name is valid              2
 * Any user name should start with starbb_, contain only  3
     characters, numbers and underscore
 * The user name should also be of a length between 8 and  4
     49
 *                                                    5
```

```
 * @param name the given name to check                    6
 * @return whether the name is valid                     7
 */                                                       8
@Override                                                 9
public boolean checkName(String name) {                   10
    return name != null && name.matches(``^starbb_[a-zA-  11
        Z0-9_]{1,42}$'');
}                                                         12
```

3.2 Password checking

3.2.1 Requirements for user name checking

The password should satisfy the following requirements:

1. The password can consist of **letters**, **numbers** and _, excluding any other symbols;
2. The password must consist of all the three types, i.e. **letters**, **numbers** and _, excluding any other symbols;
3. The password should have a length greater than or equal to 8 and less than 100.

3.2.2 Interface for the checking method

```
/**                                                       1
 * Check whether the given password is valid             2
 *                                                       3
 * @param password the given password to check           4
 * @return whether the password is valid                 5
 */                                                       6
boolean checkPassword(String password);                   7
```

3.2.3 Implementation of the checking method

```
/**                                                       1
 * Check whether the given password is valid             2
 * Any password should only contain and must contain all 3
    the three types of characters below:
 * 1) English characters(upper case or lower case);      4
 * 2) Numbers;                                           5
 * 3) Underscore.                                        6
 * In addition, the passwords should be of a length      7
    between 8 and 99
 *                                                       8
 * @param password the given password to check           9
 * @return whether the password is valid                 10
 */                                                       11
@Override                                                 12
public boolean checkPassword(String password) {           13
```

```
    return password != null && password.matches
        (""^(?=.*[0-9].*)(?=.*[A-Za-z].*)(?=.*[_].*)[a-zA-
        Z0-9_]{8,99}$"");
}
```

Chapter 4

Features tested in the project

4.1 Login method

The whole test for login can be divided into two functions: the one for login successfully and the other for login failure.

4.1.1 Login successfully

The *assertTrue* method is used in this test function. If the *login()* method returns *true*, then the test method will be passed.

4.1.2 Login failed

If we fail to login, a runtime exception will be thrown. *AssertEquals* method is used in this test function to handle the exception message. By comparing the two strings, we successfully test th login failure situation.

User Is Null

If the username input doesn't exist in the database, the exception message will be "Entity not found."

Password Is Incorrect

If the username matches but password does not in the database, the exception message will be "password or username error.".

4.2 Sign up method

The test for *signUp()* method can be divided into two functions: the one for sign up successfully and the other for sign up failure.

4.2.1 signUp successfully

The *assertTrue* method is used in this test function.

To make sure we use a name that doesn't exist in user.csv, I used the *Date* object to construct a new name.

```
Date date = new Date();
String name = (new SimpleDateFormat("yyyy_MM_dd_hh_mm_ss")).format(date);
```

4.2.2 signUp failed

assertFalse method is used in this test function and the *signUp()* function will return false if the user name already exists.

4.3 Username checking method

The test for username checking method consists of two parts: the one for legal one and the other for illegal one.

4.3.1 Legal name

I construct a string–*starbb_12AC*–and it begins with the string starbb_, and contains letters and digits at the right length. Therefore, it's legal.

4.3.2 Illegal name

It's very easy to construct an illegal name. A null string, a string with other special characters, a string not beginning with the specified string and a much shorter or longer one all are wrong names. So I design 5 methods to cover all situations. In the beginning, I forgot to consider the situation that the name may be null. Thanks to **Wang, Chen**'s advice, I take this problem into account.

Username checking methods are more complex than other test methods, and they contain many situations. But in fact, they are very normal and easy to implement. According to the classification method mentioned in class, I should design 5 methods to check whether the length of tested name is legal. But to be easy, I just implement 3 methods for three types of name: the shorter one, the right one and the longer one, which don't include the two methods for just right ones.

```
@Test
public void testNameShorterThan8() {
    //length here:7
}

@Test
public void testNameOK() {
    //length here:11
}
```

```
@Test                                                    1
public void testNameLongerThan49() {                    2
    //length here:50                                     3
}                                                        4
```

4.4 Password checking method

The test for password checking method consists of two parts: the one for legal one and the other for illegal one.

4.4.1 Legal password

I construct a string–*123_abc12*–and it contains letters, digits and underline at the right length. Therefore, it's legal.

4.4.2 Illegal password

A legal password must contain all the three types: **letters**, **numbers** and _. So in order to get a illegal password, there are many ways. Null password, a string without digits, a string without letters, a string without underline, a string with other special characters and a much shorter or longer one all are OK. So I design 7 methods to test.

There maybe a small trap in checking password. Well, if we couldn't find the differences between legal name and legal password, we'll write some wrong codes. It's because a legal password must contain all the three types but a legal name is unnecessary to satisfy the need.

4.5 Status checking method

The test for status checking method can be divided into two functions: the one for user login successfully and the other for user has not logged in.

4.5.1 status is logged in

We let a true user login and then call the *checkStatus()* method. In this test, we *assertTrue* on the return value of this method.

4.5.2 status is not logged in

assertFalse method is used in this test function. this test works well when running separately, but failed when running the whole *Lab2ApplicationTests* program.

This problem arises because we haven't designed the log out method in our application yet. Therefore, if the test method has a successful login and then calls the test-not-logged-in method, it will never pass. We tried to change the order of the methods so as to let this method appear first, but failed to change the order being tested.

After thorough analysis and further research into the *JUnit* testing implementation, we found out that the test method will be executed in the order of

the *hash code* of the method, rather than the order they appear in the source code or the order of their name. Thankfully, *JUnit* offers us a method of fixing the order of the tests and we adopted one of the methods in our implementation here in this lab.

Liu, Jiaxing fixed this problem by adding:

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

4.6 Cost checking method

The test for cost method can be divided into five functions.

4.6.1 testCostOK for cappuccino and espresso

We call the *assertEquals* method on the cost's result and the manually calculated value;

4.6.2 testCost for NumberError

We will catch a *COFFEE NUMBER ERROR* exception when the number of coffee is less than zero.

4.6.3 testCost for CupErrors

We will catch a *CUP SIZE ERROR* exception when the cup SIZE of coffee is less than 1 or more than 3.

4.6.4 testCost for Null

We will catch a *ORDER Null* exception when passing in a *null* value to the cost function.

Chapter 5

Javadoc

The full **javadoc** of the project is available in the same path under the *javadoc* folder. The source *javadoc* is also available in the source code comments for reference. Please use a web browser to navigate through the HTML files in the generated *javadoc*.

Bibliography

[1] Wikipedia contributors. (2019, March 22). JUnit. In *Wikipedia, The Free Encyclopedia*. Retrieved 14:53, April 1, 2019, from `https://en.wikipedia.org/w/index.php?title=JUnit&oldid=888928403`