Lab Report for Software Engineering course
Lab 4: Starbubucks coffee online retailing system v3.0

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

School of Software
Fudan University

April 23, 2019

Contents

Chapter 1

Overview of this lab

1.1 The Objectives of the Project

Through this experiment, we will experience the impact of demand changes on development work, experience the software project management functions provided by Huawei's development cloud platform DevCloud, and experience the rapid deployment of application services with SpringBoot.

1.2 Specifications of the Lab

The company now hopes that the development team will develop an **Online Beverage Sales System** based on the existing system. Based on the existing system, in addition to the existing **basic functions**, the **new requirements** are as follows:

1.2.1 Add ingredients

- To meet the needs of our customers, we offer a wide range of ingredients including: milk, chocolate, cream and sugar.
- The price of adding a unit of milk and chocolate is $1.2 for the price of the beverage; the price of the unit of cream and sugar is $1 for the price of the beverage; and a portion of the beverage can be added with a plurality of ingredients.

1.2.2 Adding drinks

- In order to meet the needs of customers, we offer a variety of drinks, and now we have added drinks, so that the system can provide coffee and tea sales, including cup type matching, ingredient addition, price calculation and so on.
- Coffee types now include: Espresso, Cappuccino; Tea varieties now include: Green Tea (GreenTea), Black Tea (RedTea).

1.2.3 Price calculation

Due to the adjustment of ingredients and drinks, the price calculation of drinks is adjusted accordingly.

- The salesperson can choose the type of drink, the type of drink, or a variety of ingredients. The system will calculate the final price of the cup drink for the salesperson. The final price of the drink is calculated as follows: final price = drink price + drink cup type price + multiple ingredient price (unit: $);
- The salesperson can select the number of cups under the premise of selecting a certain type of drink type. The system will calculate the final price of the multi-cup drink for the salesperson. The price is calculated as follows: multi-cup final price = single cup final price * cup number (unit :$);
- Coffee and tea cups are currently divided into three types: large cup (3), medium cup (2), and small cup (1). Different cup types have different prices:

Large Cup	Medium Cup	Small Cup
Coffee Price + $6	Coffee Price + $4	Coffee Price + $2
Tea Price + $5	Tea Price + $4	Tea Price + $2

1.2.4 Promotional programs

In order to attract more customers at a more favorable price, the company now offers the following promotional strategies for different beverages selected by customers:

The first category: combination offer

The second category: full reduction offer

1.2.5 Basic price list

1.3 The division of work in the team

1.3.1 Division of work: Wang, Chen

(Git username: *Wang, Chen*; Student ID: *16307110064*)

He constructs the overall structure of the project, divides the entire workload into several parts so that each part can be finish the work separately. In addition, he draws the diagram of the entire project on the Huawei cloud platform that contains the parts like Epic, feature, story and tasks. Furthermore, he scratches the outline of how to implement the methods adopted in this project. At last, he summarized the general parts in the documentation and drafted some regulations for commit messages.

1.3.2 Division: Huang, Jiani

(Git username: *Currycurrycurry* Student ID: *17302010063*)

She creates the concrete classes for the diverse drinks and ingredients.

1.3.3 Division: Tang, Xinyue

(Git username: *xinyuetang* Student ID: *16307110476*)

She implements the methods related to order processing, discount processing and total price processing.

1.3.4 Division: Liu, Jiaxing

(Git username: *jiaxingliu* Student ID: *17302010049*)

He tests the implementation in this project via the interface given by the teaching assistants.

1.4 Division of work for documentation

1.4.1 Parts required in the documentation

In the requirement documentation of the lab, we are required to accomplish the following parts in this documentation:

1. Explain the design ideas of work item planning in this experiment(PLAN);
2. Explain the design ideas of code implementation in this experiment(IMPLEMENT);
3. Explain the understanding of demand changes and project management (project planning, defect management, etc.) in this experiment(UNDERSTAND);
4. Explain the problems and solutions (if any) encountered in the implementation(PROBLEM).

For the convenience of being noted, each requirement is labeled with a tag, which is used in the next part for mentioning.

1.4.2 Principal for each part of the documentation

Tag	**Writer**
PLAN	Wang, Chen
IMPLEMENT	Huang, Jiani&Tang, Xinyue
UNDERSTAND	Wang, Chen
PROBLEM	Liu, Jiaxing

Chapter 2

Improved skills on team collaboration

2.1 Consistent git commit message styles

2.1.1 Commit message requirements

The following are the requirements for the commit message in our team, this version of specifications are revised according to the commit message style recommendation from website *Commit message guidelines · GitHub*[1], *How to Write a Git Commit Message*[2], *How To Write a Good Commit Message*[3] and the git commit message recommendation from one of the most authoritive open source project **GNOME** *Guidelines for Commit Messages*[4].

1. Only ASCII characters are allowed in the entire commit message
2. All commit messages must start with one of the types identified in the following table, all words are lowercase
3. It is best to have an associated work item, associated with the work item, followed by type, space # number space followed by content, such as fix #123 content
4. The total number of characters recommended in the subject *(note that it is the number of chars instead of the number of words)* is less than 50, and the maximum number is not more than 74 characters (including the previous type and item number, etc.)
5. There is no need to add a period at the end of the head
6. After the type in the subject line, the first letter of the first word after the task number (if any) is capitalized and that indicates the beginning of a sentence

[1] `https://gist.github.com/robertpainsi/b632364184e70900af4ab688decf6f53`
[2] `https://chris.beams.io/posts/git-commit/`
[3] `https://api.coala.io/en/latest/Developers/Writing_Good_Commits.html`
[4] `https://wiki.gnome.org/Git/CommitMessages`

7. Use the imperative tone in the subject sentence (although it is you who have actually done the work)
8. The tense of the subject is the general present tense
9. It is recommended that for commits involving complex modifications, body should be added in addition to the subject for further explanation. The method is as follows: break a new line and then write is the body, [do not follow the head without line break]
10. For the body part of the commit message, there is no requirement other than writing Chinese, and it is relatively free. It is also recommended to write more than one line instead of one line in order to facilitate reading.

2.1.2 Types allowed in the subject of the commit message

Table 2.1: Commit message types

Type	**Description**
feat	A new feature
fix	A bug fix
wip	While working on a fix/feature
docs	Documentation only changes
style	Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
refactor	A code change that neither fixes a bug or adds a feature
test	Adding missing tests
chore	Changes to the build process or auxiliary tools and libraries such as documentation generation

Chapter 3

Designed Ideas of work planning

Chapter 4

Design ideas of implementation

4.1 Entities implementation

The implementation of entity package can be divided into one package and one class: the package of drinkEntity and the class Ingredient.

At first, I apply this.getClass.getName() to simplify the constructor by avoiding inputting string every time. However, it is evident that the tedious information in className (the long package name) may cause other related problems. So in later modification, the constants created for drink-names in the class InfoConstant are used in the constructor method, which is also very simple and clear.

4.1.1 drinkEntity

At present we have four different kinds of drinks: Cappuccino,Espresso,GreenTea and RedTea. But considering the possibilities of adding other kinds of coffee and tea and the rationality of logic, the abstract classes of Coffee and Tea are applied to construct the whole inheritance relationship.That is, Both Coffee and Tea will inherit the OrderItem class, and all specific kinds of coffee and tea will inherit the abstract classes of Coffee and Tea.

We should pay attention to the constructor method of the different concrete classes. Since the initialization only need the different basic price of drinks, we only need to call the setPrice() method in constructor method. And all the generic method of these classes will be implemented in the parent class, the OrderItem class, such as cost() and size2price() methods. Thanks to the inheritance tree, in the size2price() method, we can use the instanceof keyword to simplify the judge of cup-size.

4.1.2 ingredientEntity

At first I use four concrete ingredients inheriting the Ingredient class in dto package: chocolate, cream, milk and sugar. However, I realized that the concrete

classes of ingredients actually are unnecessary. By using the API – Hashmap and ArrayList, we can simplify the representation of different ingredients.

4.1.3 OrderItem

Two methods are added for the class: size2price and cost in this class in order to make all its subclasses use this method.

4.2 Order services implementation

4.2.1 get total price of an order

add a new function getTotalPrice() to the Order Class,which calculates the total price of an order and return it back.

the logic of promotional functions.

4.2.2 rule1:Twenty percent off for every two cups of large espresso

count the number of large espresso in the order and save it in an int named by count,then the discount for espresso is (count/2)*0.2*20*2.

4.2.3 rule2:buy 3 get 1 for free in tea sales

count the number of GreenTea and RedTea and save them in two ints named by countGreenTea and countRedTea. Then the number of tea for free is (countGreenTea + countRedTea)/4. To get largist discount, if the free cups of tea is less than the cups of RedTea(which is more expensive than GreenTea), the discount of tea is countRedTea * 18. else the discount of tea is countRedTea * 18 + (freeNumber - countRedTea) * 16.

4.2.4 rule3:The second cup of Cappuccino is half price

count the number of Cappuccino in the order and save it in an int named by countCappuccino,then the discount for Cappuccino is (countCappuccino / 2)*22*0.5.

4.2.5 compare the two kinds of promotion plan

calculate the discount in the two kinds of promotion plan separately and choose the larger one.

Chapter 5

Understanding of demand changes

Chapter 6

Testing of the features

6.1 Testing methods adopted

Since the primary interface is the method *pay* in class *OrderServiceImpl*, the testing for the class is necessary. Given that the method is very complex and contains many factors, it's very nice to add a testing class for the class *Order*.

6.1.1 Testing for class Order

To generate a legal order, I design a method *getOrder*, which accepts some order items and return a order with an id generated with the date. And it's a helper method and it can reduce the repetition. Assuring that the total price computing is right is very important, so I design a method *testGetTotalPriceOK* to test it, where I generate a order and check whether the return price is right. What's more, there may exist some unexpected exceptions. To be easy, I take three types of exception into account: the size of order items may be illegal, the order items may be null and the ingredients may be null. Well, there may be other unexpected exceptions. To make the program much safer, I leave them in the other testing class.

Many people may think the testing for the class is unnecessary, but it's not true. I think only the implementation of base functions is convincing, can we make a larger programn well.

6.1.2 Testing for class OrderServiceImpl

Compared with the former test class, it is much more important to generate a good testing class for class *OrderServiceImpl*. Since the logic of promotional functions can be devided into two parts: the one of full reduction and the one of combination promotion. To assure each function is implemented well, I design some method to test them partly.

Testing for full reduction is very easy. I just generate a total price and pass it to the method *fullReduction*. According the logic for full reduction, I can easily get the right discunt. So the next step is clear. I just need to check whether the return value is equal to the expected one. Here is the code:

```
assertEquals(orderService.fullReduction(160),
    30.0, 0.01);
```

Testing for combination promotion is much more complicated and I need take three situations into account. The three situations are twenty percent off for every two cups of large espresso, buy 3 get 1 for free in tea sales and the second cup of Cappuccino is half price. So there are three methods to test them and the key part is check whether the return discount is right.

Testing for the method pay is most important. I should design two orders which will partly choose two promotion strategies. According to the order, I can guess what the discount may be. So the next step is to check it. And the two testing methods are *testPayWithFullReduction* and *testPayWithCombination*.

Well, testings for some important fucntions are over, but there may be some unexpected exceptions. I take some exceptions into account: the order may be null, order.orderItems may be null, one of order.orderItems may be null, orderItem.ingredients may be null, one of orderItem.ingredients may be null and one of orderItem.ingredients may have illegal name or size. In fact, all the exceptions are generated by assertion statements. So to check the exception is to catch an instance of *AssertionError* and check the message.

6.2 Testing results

Testing is to assure that our program is essentially safe and creditable, so it is very important. While testing, I find some problems and also some good ideas.

6.2.1 Some problems

The first problem I find is that if I want to check some key methods partly, I must modify their access rights. And I think it is very uncomfortable. On one hand, if not, the testing may be incomplete and may be very complex. On the other hand, if so, it will make the structure of the class worse. And it is very sad that now I still don't work out a good solution. And I choose the second choice to solve it. It's very nice that we finally find a good solution. I modify the testing methods and just test the oridinary interface without changing their access rights.

How about others? Well, I also find some bugs, such as the wrong implementation of the method *fullReduction* which primarily return (totalPrice / 100) * 30 and be modify to return (int) ((totalPrice) / 100) * 30.0 later, the wrong using of some variables and stuff. But they do not matter, because it is very easy to correct. The most important thing is that it lacks some necessary exception checking in the implementation of these methods. And later we add them.

6.2.2 Some good ideas

In total, the methods to be tested are well implemented, especially the using of stream and labdma statements. And they can help make the program more efficent and more pithy. What's more, the design of them is very nice. I think tang has very good skills of designing methods and it is very helpful for my testing. The relationship of each methods is very fine and clear.

Chapter 7

Problems encountered and solutions

7.1 the Improper Use of getClass().getName() Method

The first try to simplify the constructor method of entity classes proves to be infeasible.Since the whole project is composed of many organized packages, and this function will return a whole path string of the package name it belongs to, the class name we finally get will be long and confusing.

Thanks to the help of Jiaxing Liu, in the final version we decided to still use the string constants of drink names in constructor method. Every time we construct a new kind of drink class, we need to add the drink name in the infoConstant class.

7.2 the Defect of Ingredient Class Design

For each kind of ingredient, a unique class is applied at first, which bring about the problem of the concrete classes with only a few attributes (a bad design).

Therefore, we use the key-value data structure – map to represent each kind of ingredient, and arraylist for the total collection of ingredients added.

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`