Lab Report for Software Engineering course
Lab 5: Demand Change and Prototype Development

Wang, Chen 16307110064
Liu, Jiaxing 17302010049
Huang, Jiani 17302010063
Tang, Xinyue 16307110476
School of Software
Fudan University

May 30, 2019

Contents

Chapter 1

Demands of this lab

1.1 Requirements of this lab

According to the documentation of the lab assigner, this lab should satisfy the requirments in the following perspectives:

1.1.1 Documentation

In the documentation, we need to accomplish two parts, that is, we need to: Understand the needs of this experiment, complete the requirements document; Organize overall design documentation and detailed design documentation based on requirements.

1.1.2 Code Style

Before the experiment, our group members should agree on the code specification and the unified code style.

1.1.3 Project Management

The project is generally hosted on the Huawei Devcloud platform, on which platform we are going to finish the following tasks: Project management should be arranged based on the DevCloud platform, including adding work items, assigning work items, associating work items, managing work item status, and other project management functions;

Issues in the development process need to be recorded and managed on DevCloud, including defect reports in integrated development;

Collaborative development based on Git, submit project documentation and code on a team basis.

1.2 Background on the change of demand

Starbubucks's coffee shop has received a large amount of financing and decided to open branches in different countries and regions.

In order to adapt to the habits of users in different regions, it is necessary to develop a project prototype with basic functions, and then customize the system for different regions.Customized development needs to consider the following factors:

1.2.1 Drink Customization

At least 2 beverages need to be customized for local features (beverages defined in Lab4 are system default beverages).

1.2.2 Currency change

We need to switch to the corresponding currency according to different countries and regions, for example, a cappuccino in China 20CNY, a cappuccino in the US $3 (do not consider currency exchange).

1.2.3 Language change(localization)

The system needs to switch the system language to the language of the corresponding country and region according to the language of different countries and regions. The range of language switching is visible to all system users (sales personnel) and customers (beverage buyers). For example, when the system user inputs, the input content should be the language of the corresponding country and region; after the order is successful, the customer sees The transaction content should be switched to the language of the corresponding country and region.

1.2.4 Sales strategies

A good sales strategy can bring huge benefits, and the system's default sales strategy is the sales strategy defined in Lab4. Because sales strategies can change as external markets change, systems are required to add and remove sales strategies in a pluggable manner.

1.3 Specifications of the Lab

The following requirements are to be met for the custom development factors described in the background in the previous section:

1.3.1 Drink Customization

Customize two beverages with reference to the previous lab implementation.

1.3.2 Currency change

There may be multiple currencies in different countries and regions. For example, Hong Kong can circulate Hong Kong dollars and Chinese Yuan, mainland China circulates Chinese Yuan, and the United States circulates US dollars. The prices of different currencies for the same item are different depending on the value of the product itself. The system needs to specify the corresponding

price of the corresponding commodity for different currencies (the pricing of different currencies for each commodity, and the exchange rate pricing can be reasonably justified). Currently, the system only needs to support RMB and Hong Kong Dollar (HK$).

1.3.3 Language change(localization)

Different countries and regions may have multiple official languages, and the system needs to switch between different languages according to different countries and regions. The system must be able to switch between English and Simplified Chinese.

1.3.4 Sales strategies

The sales strategy is added and deleted in a pluggable manner. The sales strategy in lab4 is the default sales strategy and cannot be deleted. The source of information for all sales strategies is order and system information (date and location). On the basis of lab4, the requirements of (1)(2)(3) and the pluggable components of the sales strategy can be regarded as the prototype of the project.Based on this prototype, add the following sales strategy:

1. When the time zone of the system is November 11th, the total price of all commodities (including all prices) will be 50% off.

2. When the order contains at least one tea item and at least one item of the Coffee category, the total price (including all prices) is 15% off.

1.4 Additional precautions of this lab

1.4.1 Switch interface of currency and language

The switching of currency and language can depend on a “switch”, which is an abstraction rather than a concrete implementation, since the opening and closing of the switch may depend on different external conditions.

1.4.2 Prototype-based development

The addition and deletion of the sales strategy is based on the fact that the prototype of the project has been developed, and the sales strategy is added and deleted through the interface of the sales strategy provided by the prototype. When using the prototype, you can’t modify the code of any prototype, and you can only perform secondary development on the basis of the prototype.

1.4.3 Methods for switch and pluggable components

The development of ”switches” and pluggable components can refer to the relevant materials.

1.4.4 Freedom of change of the existing frameworks

In order to flexibly realize the four requirements in the previous sections above, it is encouraged that every team should design independently. The fields and method signatures in the framework code provided by the assistants in Lab4 can be modified. It should be emphasized that after the prototype is completed, it cannot be modified, and it can only be developed on the basis of the prototype.

Chapter 2

Division of work for this lab

2.1 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.

2.2 Division: Huang, Jiani

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

She implements the concrete methods in the language and currency switch feature.

2.3 Division: Tang, Xinyue

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

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

2.4 Division: Liu, Jiaxing

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

He designed the interfaces for various modules of the project including the *menu* module and the *language* module. In addition, considering the actual circumstances of the product, he adopts the **singleton** design pattern, the **strategy** pattern so as to improve the encapsulation level and the ability for extension of the application.

He also finishes the testing work of the entire project, whose work is an important foundation of the *Test-Driven Development* of the project. In particular, he realizes the *JUnit* test cases within the project and the *Swagger* interface test of the project on the Huawei Cloud Platform.

Chapter 3

Analysis of the demands

3.1 General analysis of the demands

In this version of requirement, a series of new functions are proposed and they are supposed to bring the general construction of the project some prominent change. Specifically, the requirements of the project lie in two perspectives: a prototype that should work for the all future new regions and two detailed and concrete implementations for the region of China and Hong Kong specifically.

Therefore, the main work of this project has two stages:

1. Design and determine our project prototype;
2. Continue finishing the specified work and implementing the corresponding functions based on the prototype.

Therefore, in the following paragraphs, I will try to analyze the demands from the two stages shown above.

3.2 General workflow

According to the general analysis in the previous section, the total workflow of this project can be shown as the following diagram.

3.3 Project Prototype

In the requirement documentation, it is clearly specified that our project should have a clear prototype and continue with the future works based on the existing prototypes. Therefore, the first step for the team members to do is to finish the prototype.

The prototype need to well represent the newly-added features that will be future implemented. Specifically, the prototype should have all the necessary interfaces needed for the customized drink, the currency switch, the language switch and the strategy change.

Should there be future concrete implementation on the futures above, the developers (our team members here) should only need to add new classes implementing the interfaces without modifying the existing interfaces.

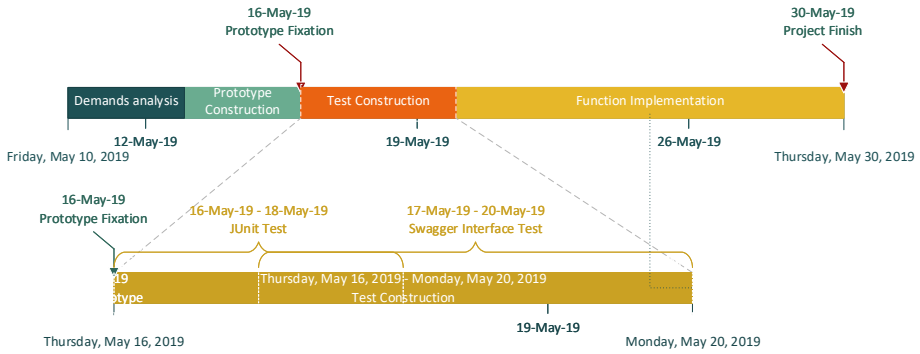

Figure 3.1: Overall workflow

3.4 Currency switch analysis

The requirements of currency switch has a lot to be further analyzed before concrete implementation. As the features of the project display, each drink and its corresponding ingredients have a price. The prices will be used in the order information and later be used to charge from the customers. However, different region has different currencies, while the drink item will also vary from one region to another. Therefore, the price of the order items should not be written directly into the code any more, which can lead more complexity to future feature extension.

Considering we are running a Spring Application, which allows a wide range of property resources lying in the *resources* folder of the application with a file extension *properties*, the price information of the order items are suitable to be stored in such places.

In such way, the price information of the project is depleted from the code section and it is much easier to maintain the data. Furthermore, should new currencies be supported in the future, we also have no need of input the price information in the code, and only need to append the configuration files.

3.5 Language switch

The switch of language is a common requirement in all the international software. To improve maintainability and readability, in addition to reduce future cost of maintenance, all the display language tags are stored in some files as constants. For example, in Android applications, all the *String*s that need to be displayed are encouraged to be saved into a separate XML file and then read via the constant names. Meanwhile, the the global language switch, either controlled by the Android Operating System or controlled by the application, can determine which *String* to be displayed. That is, the same display info are stored and accessed via the same variable name, while the variables direct the application to different files when different languages are set.

Therefore, in our application, we are encouraged to store the display information via constants and a global language switch can determine which language's constant will be transferred to the application.

3.6 Sales strategies

Sales strategy is an important part of the application. However, in order to better utilize the market, sales strategies need to be easily switched on or off from the pricing services.

Considering such circumstances, each sales strategy need to be a single extract-able method or class so that in the future configuration, we can add on or off such strategies with ease.

Chapter 4

General design for the implementation

The general prototype design of the project is shown as the Figure 4.1 below.

The UML diagram reveals how the components of the application interact with each other. Meanwhile, the connecting lines indicate the relationship between the classes in the project.

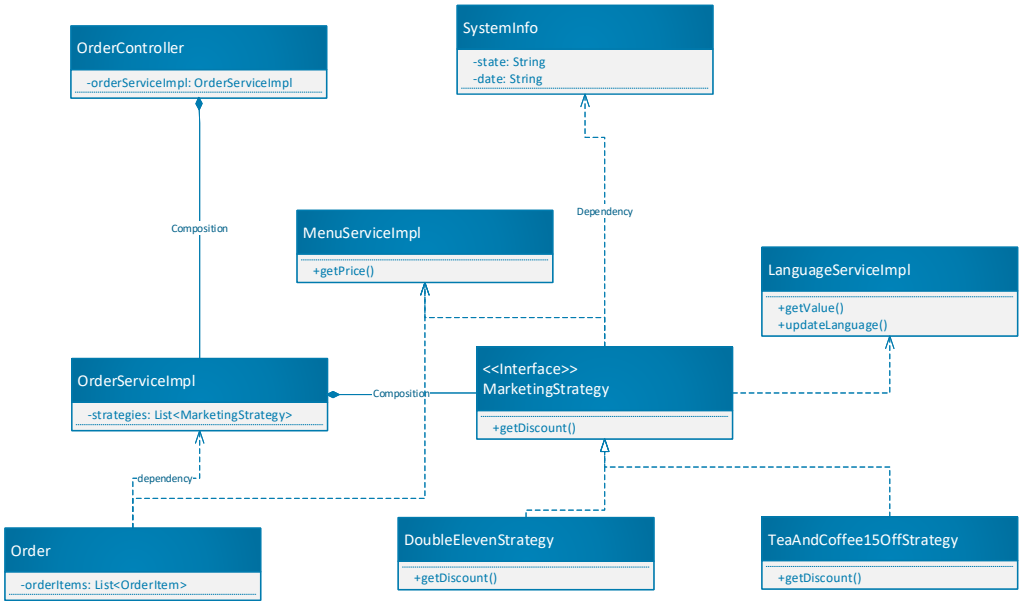

Figure 4.1: Overall workflow

Chapter 5

Detailed design for the implementation

5.1 Test design: for Test-Driven Development

5.1.1 Test components of the project

In this project, we have various types of tests so as to ensure that all components are working properly.

Specifically, we have JUnit test to be utilized during the initial stages of development. These unit tests give clear guidance of the project. In addition, we have utilized the Huawei Cloud platform with its strong Interface test function. The interface test function enables the entire project can function well with its public interfaces. What's more, the interface tests are integrated into the project as Swagger Interface test component. In this component, there is a standalone package that has a front-end user interface and will be convenient to present interface test locally.

5.1.2 Swagger Interface Test

The front-end user interface of the local Swagger Interface Test is shown as the following figures(Figure 5.1 and Figure 5.2):

5.2 Switch Language Implementation

The switch of language will be mainly displayed in the user interface, so all the information that need to be multi-translated will be separately placed into different constant files. In this iteration of implementation, we only instantiate the Chinese and English versions. And the correspondent service classes will use a typical mechanism called reflection to implement the switch of different constant language files.

In the following two sections, the detailed design of constant files and language service classes will be respectively clarified.

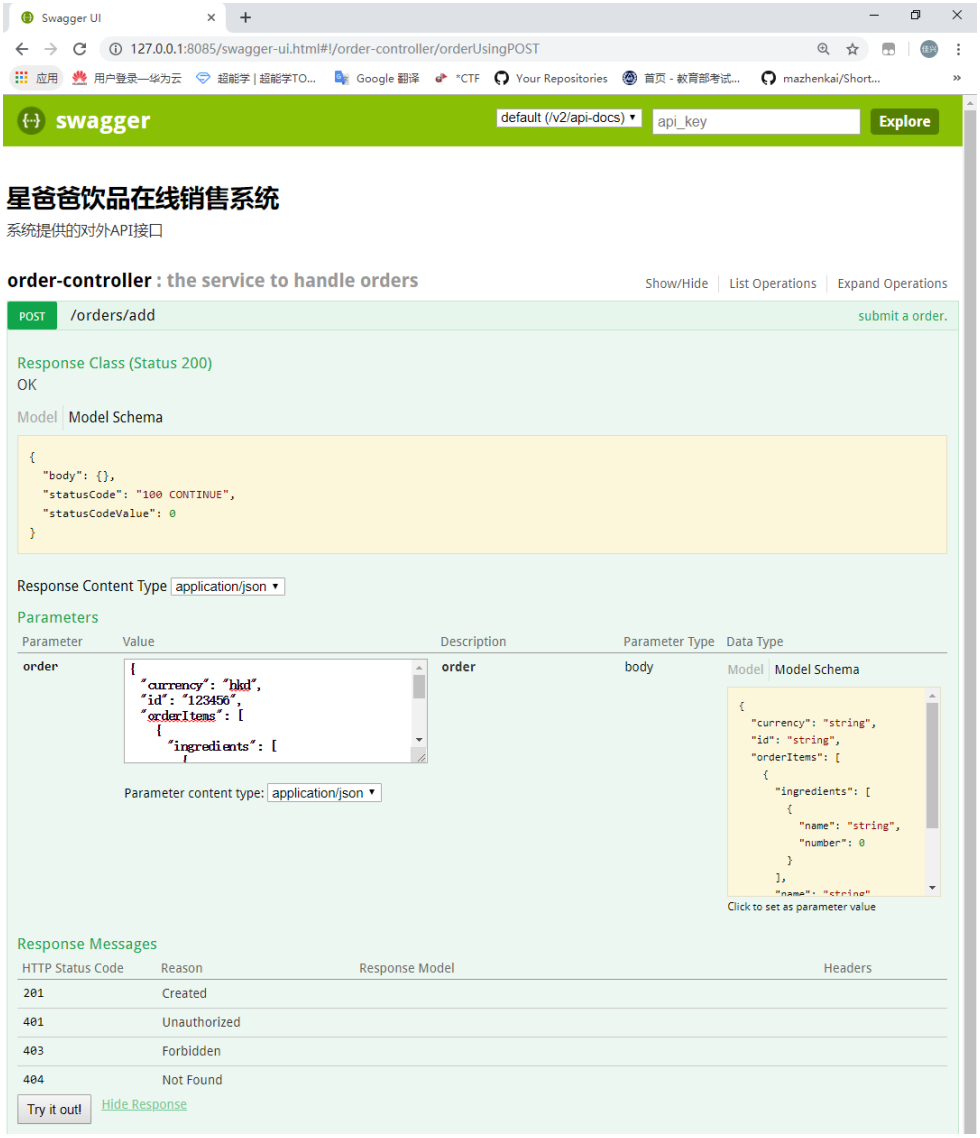

Figure 5.1: Overall workflow

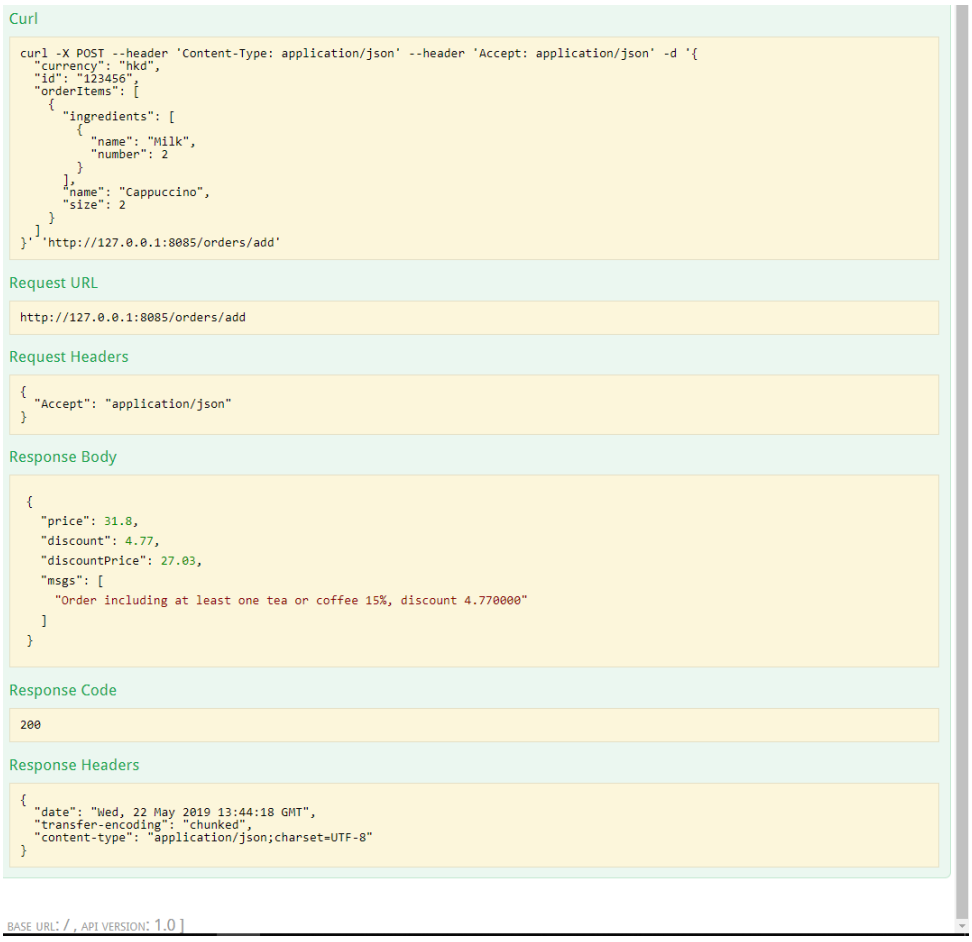

Figure 5.2: Overall workflow

5.2.1 Language Constant Files

The two constant files are positioned in the constant package. In these two files, all the variables (there are no methods) are qualified with public static final String since they are all constant strings.

To our attention, all the necessary variables should have the same names in all the language files to maintain the availability of reflection.

5.2.2 Language Service Classes

To follow the idea of prototype development, all the concrete service classes should implement their corresponding interfaces. The interface make it clear what the service will implement and its parameters. The most notable design pattern in this class is single-instance pattern.

5.3 Switch Currency Implementation

5.3.1 Currency Property Files

The different country-version menus are put in the resources directory. Temporarily, we have application-menu-hkd.properties and application-menu-rmb.properties for example.The information are stored as key-value mappings, in which keys are drink names and values are prices.

5.3.2 Currency Service Classes

All the read of resources file is done in the static procedure, and what we have is a map of maps of different prices of drinks in different country. The traversal of all the files is finished by another position2currency file.

Then the access of price will become easier by just use the get( ) method of Map API.

5.4 MarketingStrategy implementation

5.4.1 DoubleElevenStrategy class

this class has the isvalid() method to check on whether the marketing strategy can be applied to the order or not. If the date is November 11th, this method returns true.

If isvalid() returns true, getDiscount() returns 50 percent of the total price as discout, if it returns false, getDiscount() returns 0 as discount.

5.4.2 TeaAndCoffee15OffStrategy class

this class has the isvalid() method to check on whether the marketing strategy can be applied to the order or not. If the order contains at least one cup of tea or coffee, then isvalid() returns true.

If isvalid() returns true, getDiscount() returns 15 percent of the total price as discout, if it returns false, getDiscount() returns 0 as discount.

5.4.3 MarketingStrategy interface

this interface is designed to has getDiscount(Order order) method, which returns the PaymentInfo. This interface makes all those marketing strategies pluggable.

Chapter 6

Problems encountered in this project

6.1 Problems in coding

6.1.1 Wrong usage of built-in menu item price

When testing the strategy of *CombinationDiscountStrategy*, it occurs an error. The discount is illegal when the currency is hkd. Therefore, I create a work item #2282258. After checking the implementation of the class, I found the reason. In fact, there may be more than one error, but of the same type. For example, according to the strategy, when there are two cappuccino, we can get the second one for half price. But the involved method just returns a discount of 11 yuan and it is illegal. When the currency is hkd, the discount is 12.5 hkd. So in order to cope with the problem, we use the method *menuService.getPrice(Ljava/lang/String,Ljava/lang/String):LD* to replace the built-in price. And the error disappered.

6.1.2 Wrong usage of % in a format string

According to the strategy of *TeaAndCoffee15OffStrategy*, when the order consists of one for coffee or tea, we can get drinks %15 off. So in the payment information, there may be a character of %. In most time, it is ok. But a % in the format string may be illegal, especially in the format string formatting by %d, %f or something else. In this format string, we should use %% to replace %. In fact, it is just an easy question, but not least.

6.1.3 Problems in applying swagger testing

In the project, we use two tests, one for *Junit* and one for *Swagger*. *Junit* testing is much easier than the second one, we just need to add some test classes. Before using *Swagger* testing, we use an interface testing for a special url in the cloud of huawei. But the interface testing is inefficient and there is a bad management for it, so we deside to use *Swagger* testing. But it is not essay, First, I need to know what is *Swagger* testing and how to use it. After learning it, I found it is a set of interface testing case and it can be imported to a script, so it is

essay to manage. But when I attempted to use it, some problems occur. First, I should add the dependency of the *Swagger* to the file *pom.xml*, which would load some classes when the program works. And then, to use it, I should make some annotations. And the following are some examples.

```
@RestController
@RequestMapping("/orders")
@Api(value="order",description = "the service to handle
    orders")
public class OrderController {

    @Autowired
    private OrderServiceImpl orderServiceImpl;

    @RequestMapping(value="/add",method= RequestMethod.
        POST,produces = "application/json")
    @ApiOperation(value = "submit a order.",response =
        ResponseEntity.class)
    public ResponseEntity<PaymentInfo> order(@RequestBody
        @Valid Order order) {
        ArrayList<MarketingStrategy> strategies = new
            ArrayList<>();
        strategies.add(new DoubleElevenStrategy());
        strategies.add(new CombinationDiscountStrategy())
            ;
        strategies.add(new FullDiscountStrategy());
        strategies.add(new TeaAndCoffee15OffStrategy());
        orderServiceImpl.setStrategies(strategies);
        return ok().body(orderServiceImpl.pay(order));
    }
}
```

```
@ApiModelProperty(notes="order ID",required = true,
    dataType = "String")
private String id;

@ApiModelProperty(notes="order items",required = true,
    dataType = "OrderItem")
private List<OrderItem> orderItems;

@ApiModelProperty(notes = "the currency",required = true,
    dataType = "String")
private String currency;
```

In the second example, I worked out an important problem. In the second annotation, I am confused about what the dataType should be because the dataType of *ApiModelProperty* only consists of some basic types, such as int, string, double and stuff. After searching for the solution for minutes, I still didn't find the right answer. And I deside to assign *OrderItem* to it and make annotations for the class *OrderItem*. Luckily, it works well. So I finished using *Swagger* testing.

6.2 Problems in designing the project's structure

6.2.1 Problem in applying xml files

Being influenced by the arrangement of constants in *android* projects, I often want to use *xml* files to cache some constants. But after I tried some methods to do so, I found it is much more complex than using *properties* files, which is well supported by *java virtual machine*. So to be easy, we use *properties* files to cache menu and some important constants.

6.2.2 Wrong design of interface in *MarketingStrategy*

In the interface *MarketingStrategy*, there is an important method called *getDiscount(Lfudan/se/lab4/dto/Order):Lfudan/se/lab4/dto/PaymentInfo*. Now the method is well designed. But at first, I design the return value to be a double value. And if so, to get the payment information, the program may do ssame thing in another position and it is inefficient.

Chapter 7

Measures against demand change

7.1 Measures for beverage customization

In the old version, we design some drinks, such as red tea, green tea, cappuccino and so on. But now, we need to design another two drinks, which is special for different areas. According to the design of our project, it is an easy question. In last version, we create a file named *application-menu.properties*, which contains some menu items and their price. To satisfy the new need, what we should do is just to add another two items to file and it's very navie. How about the design? Well, for China, we design a drink named *CoconutMilk*. And we design a drink named *Latte* for HongKong.

7.2 Measures for currency switching

In the old version, we don't take currency into account. So we need to design some methods to do it. Frist, where can we get the currency of one order? Considering that the currency may be constantly changing with different orders, we add the currency to the class *Order*. Second, how to switch currency? We must have different menus for different currency at first. So I add two files named *application-menu-cny.properties* and *application-menu-hkd.properties*, which are partial for China and HongKong. And next, we design an interface named *MenuSevice*, in which there are a method named *getPrice(Ljava/lang/String, Ljava/lang/String):D*. The method accepts two parameters, one for the currency, the other for the menu item, and return its'price under the currency. We can easily get what the class may be implemented. We should read files and cache them in the run time environment. if can, we should read once. And we can get price of any legal menu item for any legal currency. And if we want to add some new currency, what we need to do is just to add another menu file and it is very navie.

7.3 Measures for language switching

Similar to currency switching, language switching is also new to us. Many people may think the two new need are the same and can be satisfy with one method. But it's not true. Language swithcing may be seldom changing, but we also need to add an interace for chaning language. This time, we don't choose to use properties files to help us. In stead, we use constant classes to cache some constant strings and we design an interface named *LanguageService* to do language switching, in which there are two methods, one for updating language, the other for getting the constant. The first method accepts a parameter for the language name and return nothing. And the second one accpets a parameter for the inforname symbol and return it's value. We can guess how the class may be implemented. Since we should load some different classes for different language, we can use *classLoader*, which is used for classes reflection. The implementation may be too complex, but it provide us with a good extension. If we should support other languages, what we need to do is just to add some constant classes.

7.4 Meausers for marketing strategies

This new need is much more normal than above two needs and it can be easily satified. First, we design an interface *MarketingStratgy*, in which there is a method named *getDiscount(Lfundan/se/la4/dto/Order)*. The method accepts an order and return it's payment information. If we want to add some new strategies, what we need to do is just to add some classes which should implement the interface.

Chapter 8

Tools and literature involved in this project

8.1 Swagger Interface Test

The OpenAPI Initiative (OAI) has become the de facto standard to specify what an API can do. The API community as a whole has embraced the standard, and tooling around OpenAPI, especially Swagger, has never been better, with ways to document, implement, and test APIs that adhere to the standard. This is an awesome thing for the API community. Instead of taking precious time to create homegrown standards and ways of creating, testing, and monitoring APIs, API providers can follow the specification and avoid much of the standard work to build an API, and focus on innovating by what their APIs can do.

With the facts above, Swagger is a powerful tool enabling us to test our APIs and clearify the API definitions.

Swagger is an open-source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful Web services. While most users identify Swagger by the Swagger UI tool, the Swagger toolset includes support for automated documentation, code generation, and test-case generation.

Sponsored by SmartBear Software, Swagger has been a strong supporter of open-source software, and has widespread adoption.

In this project, we are adopting Swagger Inspector to utilize the functions of Swagger Interface Test. The detailed implementation and test results can be seen from the previous chapters.

8.2 Singleton Pattern

In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. The term comes from the mathematical concept of a singleton.

Critics consider the singleton to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in

situations where a sole instance of a class is not actually required, and introduces global state into an application.

8.2.1 The general characteristics of Singleton Pattern

The singleton design pattern is one of the twenty-three well-known "Gang of Four" design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse. The singleton design pattern solves problems like:

1. How can it be ensured that a class has only one instance?
2. How can the sole instance of a class be accessed easily?
3. How can a class control its instantiation?
4. How can the number of instances of a class be restricted?

The singleton design pattern describes how to solve such problems:

1. Hide the constructor of the class.
2. Define a public static operation (*getInstance()*) that returns the sole instance of the class.

The key idea in this pattern is to make the class itself responsible for controlling its instantiation (that it is instantiated only once). The hidden constructor (declared private) ensures that the class can never be instantiated from outside the class. The public static operation can be accessed easily by using the class name and operation name (*Singleton.getInstance()*).

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`

[2] Wikipedia contributors. (2019, March 7). Singleton pattern. In *Wikipedia, The Free Encyclopedia*. Retrieved 04:09, May 30, 2019, from `https://en.wikipedia.org/w/index.php?title=Singleton_pattern&oldid=886672542`

[3] SmartBear Software. (2019, May 30). API Testing Tools — Swagger & ReadyAPI — Swagger. In *The Best APIs are Built with Swagger Tools — Swagger*. Retrieved 12:13, May 30, 2019, from `https://swagger.io/solutions/api-testing/`