Analysis of Lab_#2: Mix of (Abstract) Factory and Decorator Pattern

1 Analysis of Lab 2

In this lab, we are required to solve some real-world problems by fusing patterns including *Abstract Factory Pattern*, *Factory Pattern* and *Decorator Pattern*. The question provides situations: vendors → stores → client and we select **case 2** to complete.

First of all, we implement the vendors side with *Decorator Pattern*, all of the components are capable of offering not only the price of the computing blocks itself, but also the previous (former, inside) prices of products inside the wrapped, each of which also performs as a certain type of blocks say `CPU` and so on. Second, we build the selling store with an ability of dynamic components providing alteration, undertaking tasks of constructing the blocks from piece to piece, to a more concrete and entire object computer, which also maintains an interface of inspection, to the later supervision side. Then comes the department ensuring the quality of the products, which counts the components types to three, any of which belows three will be denitely labeled as disqualied. Note the report is illustrated based on conventions [1] in previous one.

2 Patterns

Figure 1: Patterns coupling

We strive to illustrate the patterns we harboured in this project in this section2. We dene the computing blocks to be the decorator, each of which is also able to wrap the `ComputerBase` (actually we do need a base representing the labor of the company). Then we pick the vendors `Intel` and `Samsung` out to be the *Abstract Factory* for they should behave like exposing the interface of producing three types of computing blocks (if a factory provides more than one product, then it is an abstract factory). Then comes the upper-level abstraction of the task, the store (also known as `LaptopCompany` and `PCCompany`), which is implemented in `Factory Pattern` for they should offer behaviour resembling providing the client with only ONE product.

3 Non Pattern Things

Patterns are straightforward and easy to understand as illustrated in gure 1, still, addressing consideration that the task is presented with some behaviour we can hardly extract a pattern, we carry out abstractions as order, order-decoding and decoration-unfolding. Thus well cover the thoughts of our design and tricks of implementation.

3.1 Order

For a set of requirements of the client, *ArrayList<Pair<String, Pair<String, String>>>* is utilized for growth-allowed list with nested pair like Intel, CPU, i7. The abstraction is so real that the client should make any order with ease.

But things have become a little weird for the programmer to make these requirements done correctly, aiming at solving which, the `Company` takes responsibilities of the order decoding.

[1] package, `class`, **le** , *method*, *pattern*

3.2 Order Decoding

On the stage of receiving requisites of the customers, the company first, between the lines of code, loops through each of the nested pair and send it to the *produce()* method, each of which will be dispatched to certain factory with a previous reference of decorator to be the parameter of current constructor, so as to hold a *ArrayList<String>* representing chaining of prices. This behaviour is quite simple, for in real-world, the seller would do the same as redirections of building blocks to the vendors of components.

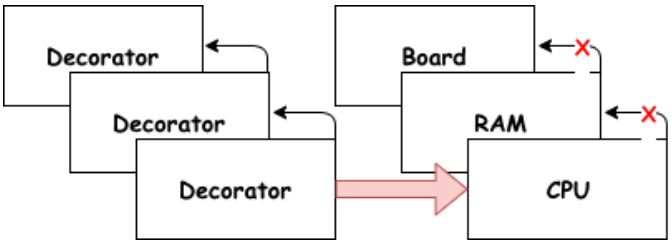

Figure 2: Decorator as wrapped object

Storing as a wrapped object, any of the abstraction can hardly know the amount and types the computer consisted of, as it leads to our tricks: *instanceof* and Java reflection.

For each of the computing blocks, as we define "price" to represent as a recursive alteration of decorator, though we can modify the decorator chain to be a more complex object, still we carry out the feature "reflection" of Java to extract the class name of the object as *obj.getClass().getSimpleName()* to obtain the name of the components as it's more convenient.

3.3 Decoration Unfolding

Obviously as displayed in figure2, one can hardly identify all types of the these computing blocks from only the outest decorator, expecting to overcome this, we propose a method using recursive methods and Java *instanceof* features. To be specific, every time the company is in need of a storage of the components as *ArrayList<Product, Integer>*, it will not only ask the outer decorator to return an addition of his price and the former prices in *ArrayList<String>* but also for the type of these components through condition branches with *instanceof* as all computing blocks implement the types (`CPU`, `RAM` and `Board` respectively) for types recognition. The algorithm is presented in this chart3.3.

Algorithm 1 Chain of Recursive Calling

Require: $Interface\ CPU$, RAM, $Board$ in $Types$ and a $reference\ D$
Require: Global counter as cnt[3] for three types
1: initialize a reference P for previous D
2: **if** P is **not** $NULL$ **and** is *instanceof* $Types$ **then**
3: jumps to 1 with P as D_{new}
4: **end if**
5: **if** D is in $Types$ **then**
6: cnt[D] add 1
7: **end if**
8: done

4 Conclusion

In this lab, we finished the task with patterns: *Abstract Factory*, *Factory* and *Decorator*, to construct the business logic "client → company → vendor" in figure1, for a loose coupling design. We first addressed that three of the design patterns2 we've managed could be taken advantages of, then we mentioned that there are utilities patterns can hardly cover so as being taken place of

by abstract behaviour as illustrated in section3, in which (3.2 and 3.3) we've solved utilizing the features of Java language, achieving demands of **case 2**. Hopefully we've illustrated this task and our solution clearly and accurately.

5 Appendix

We want to add more detailed illustrations of our work here [2].

5.1 Product

As there are two vendors: `Samsung` and `Intel` company, each of which is capable of providing the sellers (`LaptopCompany` and `PCCompany`) three sorts computing blocks `CPU`, `RAM` and `Mainboard`, each of which is divided into two types like "large" and "tiny", totally $2 \times 3 \times 2 = 12$ blocks, among which we define decorators inside the *price()* to return the price, inside of which then we save a temporary reference and ask the decorated to return his price.

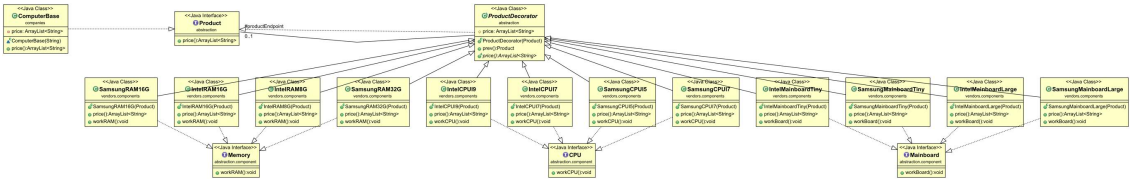

Figure 3: Product, the decorators

5.2 Company

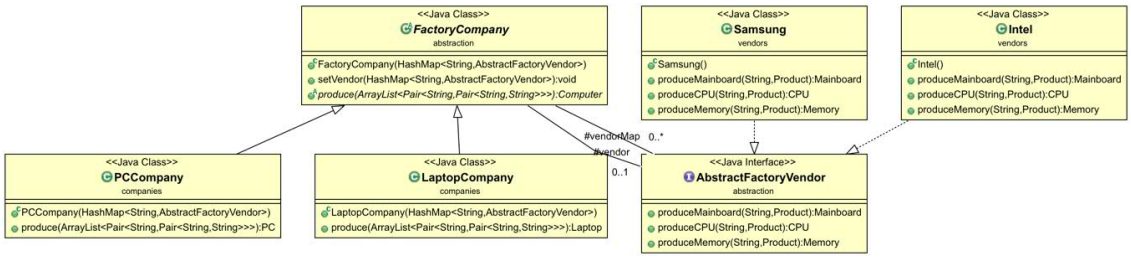

Figure 4: Company, the factory and abstract factory

In this section5.2, we implements the *Factory Pattern*, as the actually store (connect directly with the client *main()*) contain a series of actions of building a computer thing. They adapt themselves with a dynamic setter of abstract factory to deliver the customers components-blended products according to the requirement.

Abstract Factory is also included in this UML diagram, they provide brand-set computing components with multiple interface aiming at various types.

5.3 Computer

This chart5 illustrates the procedure of producing a computer (`Laptop` and `PC`) clearly and accurately. During the period of building, the company asks the the top-level decorated one for a list of components to constructing the computer, printing the price for each blocks respectively. Passing the wrapped object to SupervisionDepartment.java, the client receives a laptop or PC with an unique *UUID* and a boolean value representing a qualification of itself.

[2] All UML diagrams are automatically generated by ObjectAid at https://objectaid.com/home

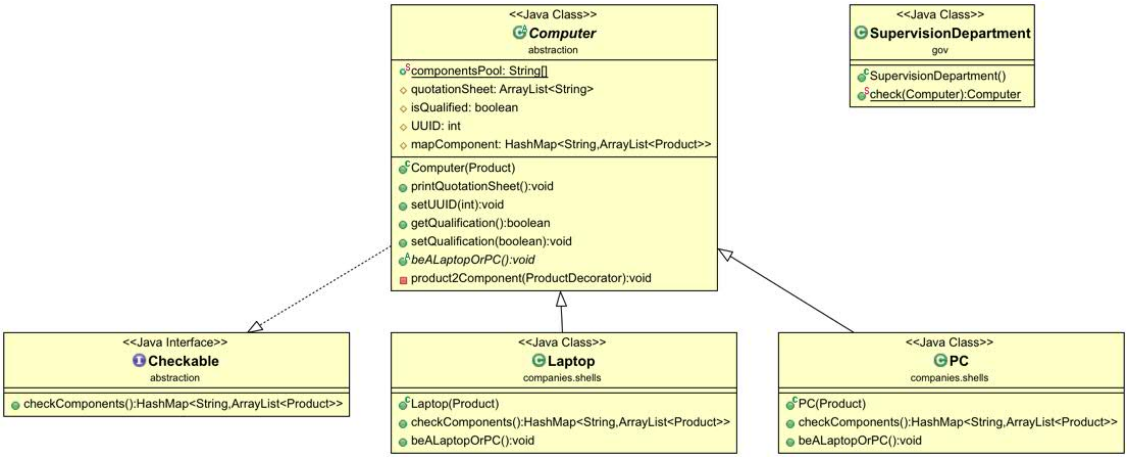

Figure 5: Computer

5.4 output

```
15$ for base
25$ for SamsungCPUI7
12$ for IntelMainboardTiny
10$ for IntelMainboardLarge
35$ for SamsungRAM32G
20$ for IntelRAM16G
I'm a PC, and is a qualified PC
15$ for base
25$ for SamsungCPUI7
35$ for SamsungRAM32G
35$ for SamsungRAM32G
I'm a PC, and is not a qualified PC
20$ for base
30$ for IntelCPUI7
35$ for IntelCPUI9
35$ for SamsungRAM32G
35$ for SamsungRAM32G
12$ for SamsungMainboardTiny
I'm a Laptop, and is a qualified Laptop
```

6 Additional Discussions

In this section, we want to discuss the differences among several patterns associating with thoughts of wrapping an object.

6.1 Decorator Pattern

Conceptually a *Decorator Pattern* is a pattern that helps the building the attachment of additional behaviour or responsibilities of an object dynamically without changing the wrapped object inside the decorators.

In the realm of coding, the decorators and the base object share a same interface that incurs a chain of calling of the desired attributes. First we create a base object, then wrap (or decorate) it with a optional decorator in the manner of composing a reference to the inside object at constructing time. And we do that again and again until the object is finished. Every time we

want an attribute, we ask the most outside layer to return an added attribute of the one just inside of which is the closest to the wrapper outer layer.

On implementing this pattern, one should give consideration to detailed syntax and features in the language he was planning to use, resembling compositions and recursive matters. For instance in **Java**, `protected` reference of the wrapped object and the endpoint of recursive calls of the chain of the dynamical attributes should be contained.

6.2 Façade Pattern

Façade Pattern is the easiest pattern among which we've learned. Just implement a wrapper of a set of methods to make it be called only once once which were collected in a single function, same effect as they were called separately.

Note that advanced data structure could be included in this pattern, for each of the client which reacts variously, like tree or graph, is a traversal of the operation set.

6.3 Adapter Pattern

Basically an *Adapter Pattern* is a means of converting an interface of an implemented object to another methods without altering the interface itself by offering methods of the adaptee inside the interface, appearing to be a fake cover cause no one knows what is going on behind the scene.

So long as the adapter provides an implicit alteration of the exposed connection, there exists two routes for building a *Adapter Pattern*. One is by carrying out the compositon of the adaptee's reference, and the other one is to `extends` (implicitly refering by `this`) the adaptee to utilize the desired methods.

The *Adapter Pattern* is quite straightforward and easy to understand. We can manipulate any objects with any functions even some not belong to the objects providing a mapping adapter from the covered to the original.