Analysis of Simple Factory Pattern and Factory Pattern

1 Analysis of Assignment

In this assignment, we are required to analyse the condition of the reduction from a *Factory Pattern* to a *Simple Factory Pattern*. Considering answering When and How to accomplish the degradation, one need three basic theory to explain this work. First is the paradigm of creating *Simple Factory Pattern*, second is the paradigm of implementing *Factory Pattern* and the third is the detailed way we harbour to make the former evolve to latter. Note that all discussions are based on condition discribed in section 2.

2 Simple Factory Pattern and Factory Pattern

To illustrate the problem discussed above, we visualize the relationship of classes, concrete objects and interfaces of the two patterns as below (Figure 1 for *Simple Factory Pattern* and gure 2 for *Factory Pattern* with description in section 6). To be specic, the former concretes its case switching in the unmoving class in the `SimpleFactory` while we extract the Factory level of the second design to substantiate an interface (as well as an abstract class in some degree), with the rst design, its more convenient for the client but is a trade-o ff with scalability.

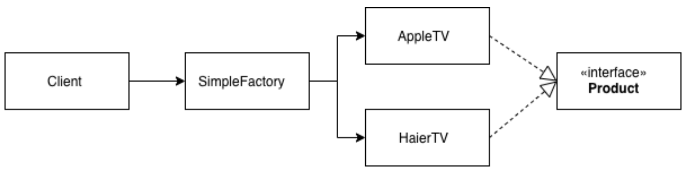

Figure 1: Simple Factory Pattern

As it shows, *Factory Pattern* is different from the aspect of Factory level, During a single runtime, one can only produce certain type of products, and to add new product (no matter the brand or process of producing is different), programmer should modify a concrete class like `SimpleFactory`.

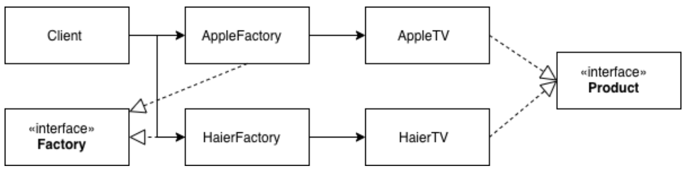

Figure 2: Factory Pattern

Paradigm is changed when it comes to *Factory Pattern*. There exists abstraction of products (`Product`), as well as the delayed implementing of case switching in all kinds of creator as an interface (`Factory`). With this pattern, one can build the same amount of sorts of products with limited types of classes compared to the simple pattern, but altering the factorys (various channels of producing things) to create multiple products at runtime [1] with ease.

[1]Its also supported for *Simple Factory Pattern* to build various objects at runtime with a wrapper with a factory setter.

As for editing, to create a new class (new factory), one only have to make the new class inherited from the abstraction (`Dependency Reverse`) and should and will not reconstruct the backbone of the original design (`Open Close`).

3 Condition of Reduction

Now it's easier to discuss the situation we are faced with, to begin with, if we are based on "producing products", and if there are:

- Certain and limited types of products that the factory should produce with no extraction of the abstraction. "Factory" downgrades to "Simple Factory", automatically.
- The number of abstraction level is only ONE, like all the products are "TV", among which differences are only various brands, then only implementing the `TVFactory` should be fine [2].

On the basis of "code modifying", from the perspective of a programmer, to answer question "When", if situations are:

- Only limited amonut of classes, with a pure design of structure, removing the abstraction causes no harm to structure and lead to limited time and labor.

4 Means of Alteration

As illustrated above, one can reduce the *Factory Pattern* to *Simple Factory Pattern* as long as the conditions is set up. More specifically, to make the former perform like the latter at code-editing time or at runtime, we can, repectively:

- Ignore the abstraction (should still do inheritance, or the code couldn't compile), then we can also create various brands and multiple types of machines with ease. Once a factory contains two or more levels of abstraction, say `AppleTVOLEDScreen` (see 6.4, totally three levels (╯°□°)╯︵┻━┻), then it works as a simple factory (ONE factory contains ALL!!!).
- Ignore the concrete interface. Adopt `AppleFactory` to create an `AppleTV` as in 6.3 instead of using `Factory` in section 6, then it performs just like a simple factory [3] [4].

5 Conclusion

To conclude, we mentioned the details of two patterns with concrete example in section 2 that *Simple Factory Pattern* contains only an abstraction of the products, and the *Factory Pattern* constructs itself with an abstraction of the factory itself, providing extendability and ability of alteration, avoiding modifications of the structurized code of the pattern or the interface.

We illustrated the possible conditions in section 3 of the required reduction and multiple channels of modifying the code from the perspective of the client 4 and the programmer 4 in section 4, described in 6.3 and 6.4 repectively. Hopefully we stated the problem and solutions correctly and accurately.

[2] `AppleTV` and `HaierFridge` have an abstraction level of brand, as well as the types of machine, totally two levels.
[3] *Simple Factory Pattern* uses `SimpleFactory` to build a simple-product.
[4] Duck test: If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

6 Appendix

6.1 Simple Factory

```
class SimpleFactory {
    public Product createProduct(String type) {
        Product newProduct;
        /* case switching */
        return newProduct;
    }
}

public static void main(String[] args) {
    SimpleFactory fty = new SimpleFactory();
    Product p = fty.createProduct("type");
}
```

6.2 Factory

```
abstract class Factory {
    protected abstract Product createProduct(String type);
}

class AppleFactory extends Factory {
    public Product createProduct(String type) {
        Product newProduct;
        /* case switching */
        return newProduct;
    }
}

class HaierFactory extends Factory {
    public Product createProduct(String type) {
        Product newProduct;
        /* case switching */
        return newProduct;
    }
}

public static void main(String[] args) {
    Factory[] fty = new Factory[2];
    fty[0] = new AppleFactory();
    fty[1] = new HaierFactory();
    Product p1 = fty.createProduct("type");
    Product p2 = fty.createProduct("type");
}
```

6.3 Factory Reduction 1

```
/* . . . */

public static void main(String[] args) {
    AppleFactory appleFty = new AppleFactory();
    HaierFactory haierFty = new HaierFactory();
    Product p1 = AppleFactory.createProduct("type");
    Product p2 = HaierFactory.createProduct("type");
}
```

6.4 Factory Reduction 2

```
/* . . . */

class AppleFactory extends Factory {
    public Product createProduct(String type) {
        Product newProduct;
        if (type.equals("AppleTVLEDScreen")) {
            /* . . . */
        } else if (type.equals("AppleTVOLEDScreen")) {
            /* . . . */
        } else if (type.equals("AppleFridge...")) {
            /* . . . */
        } else {
            /* . . . */
        }
        return newProduct;
    }
}

/* . . . */
```