Design Pattern About Factory and Abstract Factory

> Designed by Zhu Yueming

Part 1 Factory

Experimental Objective

Learn how to refactor interface-oriented code to simple factory design pattern, static factory design pattern and factory method design pattern respectively. In the process, you can appreciate the role of factory model in practical project.

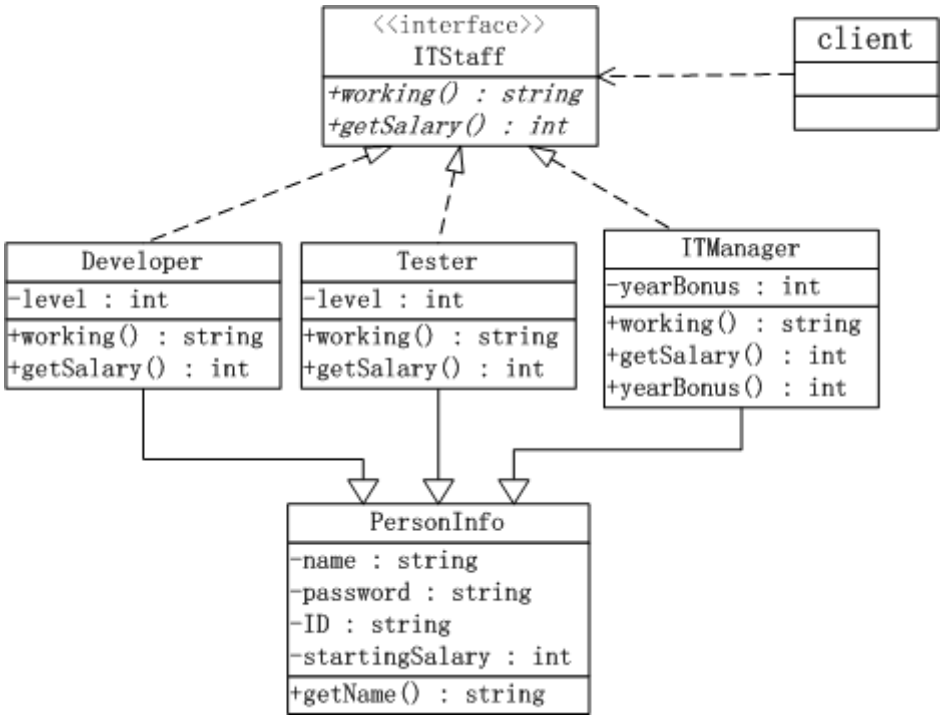
Introduce of source code

1. Requirement introduction

In development department, the types of staff and their salary are shown in following table.

ITStaff	**Description of work** `working`	**His salary** `getSalary`
Developer	Coding	10000+level * 2000
Tester	Testing	8000+level * 1500
ITManager	IT Manager	30000 (he has additional bonus)

2. Class Diagram



3. How to use it

We can use it in console window.

Input 1 means an instance of ITManager has been created.

Input 2 means an instance of Developer has been created.

Input 3 means an instance of Tester has been created.

Input 4 means print out all user information by order of salary.

Input 5 means print out all user information by order of working.

Input 0 can stop the program.

```
1 2 3 2 1 4 5 0
All information:
Testing     name: 10003 Tester   , salary:  9500
Coding      name: 10002 Developer, salary: 12000
Coding      name: 10004 Developer, salary: 20000
IT Manager  name: 10001 ITManager, salary: 30000, bonus in the end of year 21000
IT Manager  name: 10005 ITManager, salary: 30000, bonus in the end of year 3000
All name:
Coding      name: 10002 Developer, salary: 12000
Coding      name: 10004 Developer, salary: 20000
IT Manager  name: 10001 ITManager, salary: 30000, bonus in the end of year 21000
IT Manager  name: 10005 ITManager, salary: 30000, bonus in the end of year 3000
Testing     name: 10003 Tester   , salary:  9500
```

4 Disadvantage analysis

How can we do if an interface are given to us without understand which classes can implement it?

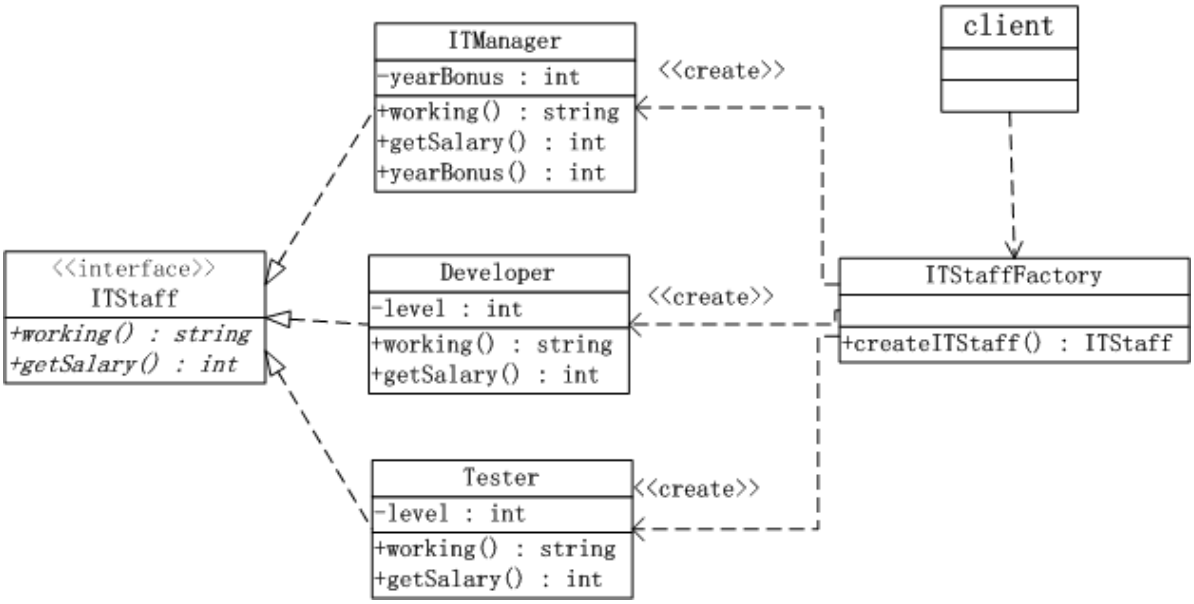
Consequently, in interface-oriented programming, clients not only need to understand interface but also need to understand for which classes can implement such interface. Actually, the implementation classes should be encapsulated by interface and isolated form client, in other words, the client does not need to know what the specific implementation classes (`ITManager`, `Developer`, `Tester`) of the interface (`ITStaff`) are.

Simple Factory

To solve the previous problem, we can provide a class, the function of which is to provide functions which can create instances. The class can be regarded as factory, and a factory can be an interface, abstract class or a class.

The main internal implementation of the simple factory is to “select the appropriate implementation class” to create the instance. Since we want to realize the select operation, we need to pass parameters, which represent different choices. This condition and parameters can be derived from client.

Class Diagram



Task 1

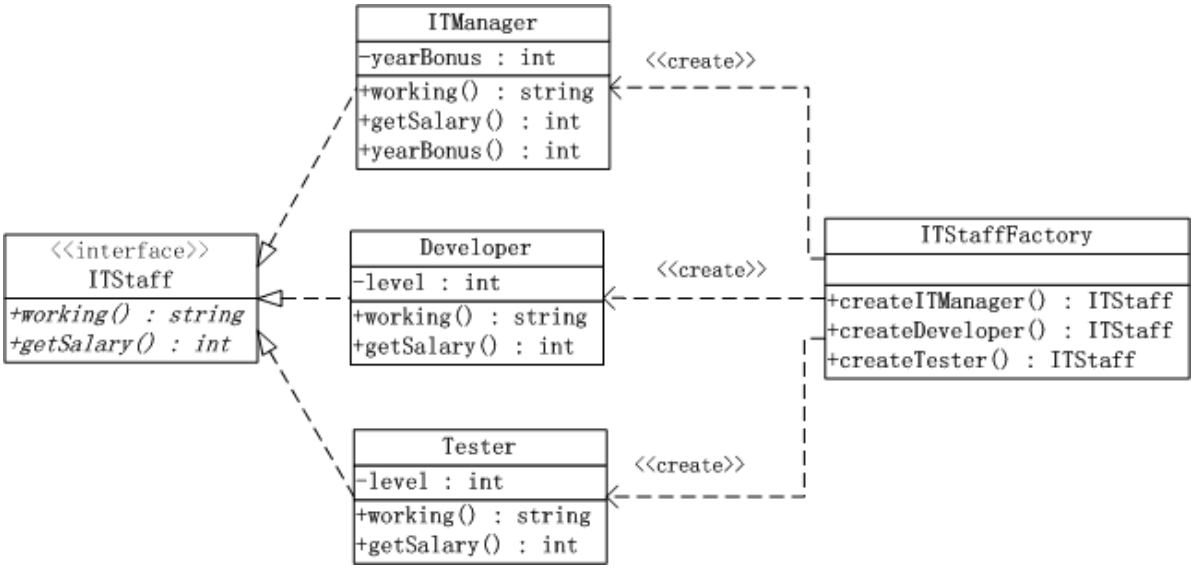
Create a package named **SimpleFactory**, then copy your source code here, and modify them to simple factory design pattern.

Hints: Create a class named **ITStaffFactory**, in which there is a method named **createITStaff()**, and we need to passing parameters in this method to distinguish which implementation class we need to instantiate.

Static Factory

The difference between static factory and simple factory is that several static methods should be defined in factory to instantiate different objects. The client needs to understand the definition of those static methods.

Class Diagram



Disadvantage analysis

1. Increase the complexity of client usage.
2. Inconvenient to expand sub-factories.

Task 2

Create a package named **staticFactory**, then copy your source code here, and modify them to static factory design pattern.

Hints: Create a class named **ITStaffFactory**, in which you need to design three static methods **createITManager(), createDeveloper()** and **createTester()** and the return value of those three methods is a **ITStaff** type.

Factory Method

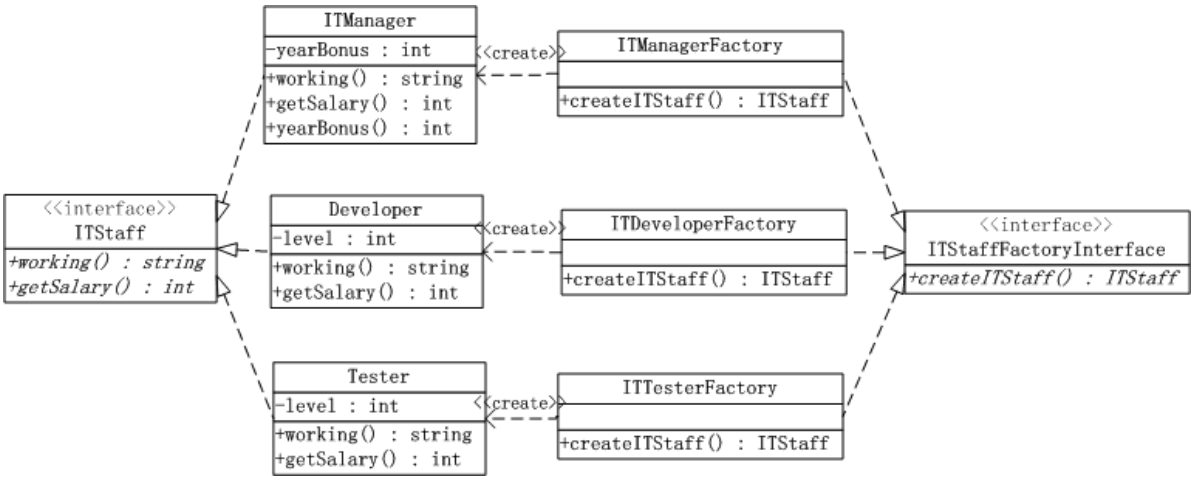
For simple factory, the process of creating instance is in the factory class by the way like "selective implementation", while for factory method, it will transfer the process to subclasses. The factory is an abstract class, and the methods inside are also abstract rather than concrete implementations.

In factory method:

1. Abstract factory: It is the core of the factory method pattern. It can be an interface that a concrete factory must implement, or it can be a parent class that concrete factories need to inherit.
2. Concrete factory: It can create objects for the corresponding specific product.
3. Abstract product (`ITStaff`): It is an interface or a parent class to be implemented or inherited by concrete product classes.

Concrete product (`ITManager` etc.): The instance of which is created by corresponding concrete factory.

Class Diagram



Task 3

Refactoring the code to factory method design pattern by following steps.

Step 1

Create a package named **factoryMethod**, then copy your source code here.

Step 2

Modify it to factory method design pattern.

Hints: Create an interface named **ITStaffFactoryInterface**, and then created three concrete factory classes to implements the interface.

Those three factory can be named as follows:

`ITManagerFactory`

`DeveloperFactory`

`TesterFactory`

Step 3

Adding an concrete class named **ArtDesigner** and we can describe it as follows:

Attributes:

level: the value is from 1 to 5

Starting Salary: 7000

name: userID+" "+"ArtDesigner"

For other attributes are similar to Developer or Tester classes.

Method:

Constructor and tostring method are similar to which in Developer or Tester classes.

working: It needs to return "Art Design"

getSalary: startSalary+ level*1500

Step 4

Input type for client

Input 1 means an instance of ITManager has been created.

Input 2 means an instance of Developer has been created.

Input 3 means an instance of Tester has been created.

Input 4 means an instance of ArtDesigner has been created.

Input 5 means print out all user information by order of salary.

Input 6 means print out all user information by order of working.

Input 0 can stop the program.

```
1 2 3 4 5 6 0
All information:
Testing     name: 10003 Tester   , salary: 12500
Coding      name: 10002 Developer, salary: 14000
Art Design  name: 10004 ArtDesigner, salary: 14500
IT Manager  name: 10001 ITManager, salary: 30000, bonus in the end of year 27000
All name:
IT Manager  name: 10001 ITManager, salary: 30000, bonus in the end of year 27000
Coding      name: 10002 Developer, salary: 14000
Testing     name: 10003 Tester   , salary: 12500
Art Design  name: 10004 ArtDesigner, salary: 14500
```

Part 2 Abstract Factory

Experimental Objective

- According to source code, learn what is the benefit of abstract factory, in which circumstance using abstract factory is better, and how to use abstract factory design pattern in different design layer.
- Then exercise the basic usage of singleton design pattern.
- Finally, exercise how to separate the dao layer from service layer in different ways.

Introduce of source code

1. Requirement introduction

There are two different kinds of databases in a project, the one is Mysql, and the other is SqlServer. In the project, there are two entity classes (**Staff** and **Computer**), and we need to do insert, update and delete operations of those two entities in both two databases respectively.

The original code is using simple factory design pattern. For example the `ComputerFactory` can return an instance of `MysqlComputerDao` or `SqlServerComputerDao` by passing different parameter. Accordingly, the client needs to generate two instances for two different factories, and then whether we can get correct instance of `StaffDao` and `ComputerDao` is determined by passing correct parameter.

In this tutorial, the task is that how can we get the instance of `StaffDao` and `ComputerDao` in a better way.

2. How to use it

After we get two instances, we can do a simple test:

```
1 2 3 4 5 6 0
insert staff into Mysql database successfully
update Staff in Mysql database successfully
delete Staff in Mysql database successfully
insert computer into Mysql database successfully
update computer in Mysql database successfully
delete computer in Mysql database successfully
```

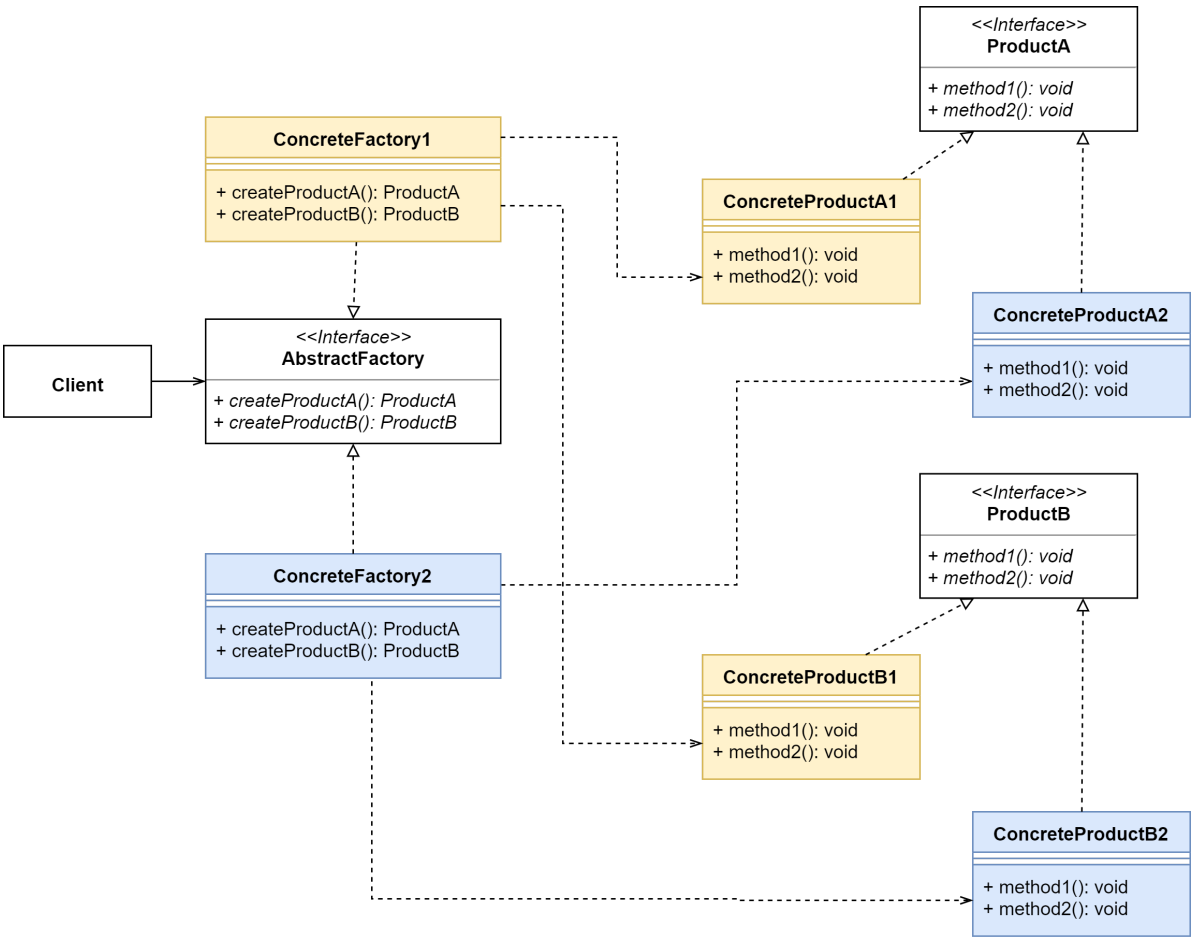
3. Disadvantage analysis

What we want is to interact with only one database for one time, however if we pass a wrong parameter, we might interact the staff information with one database as the same time interact the computer information with another database

Abstract Factory

The abstract factory design pattern can provide a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. Which means, the products created by only one concrete factory can not from different themes.

1. Class Diagram



Task 1

In `abstractFactory` package, please modify your code by using abstract factory design pattern in `dao` layer, and make sure that it can match the client.

Hints: You need to created two concrete classes including `MysqlDaoFactory` and `SqlServerDaoFactory` to implements the interface `DaoFactory`

Refactoring Abstract Factory by Singleton

The Singleton Pattern ensures a class has only one instance and provides a global point of access to that instance.

1. Class Diagram

2. Sample code

```
public class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

or

```
public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }

}
```

Using reflection to separate two layers

If we want to switch database from one to another, we need to instantiate another concrete factory accordingly, in the process the source code needs to be changed. A better way is that, we can define the name of concrete factory into configure file, and then using reflection to instantiate an instance, so that we can switch the database only by changing the configure file instead of modifying the source code. In this case, defining two concrete factories are duplicated, only one is enough.

Here is a sample code of instantiate a class from properties file.

```
public class demo {
    public String toString(){
        return "hello world";
    }
```

```
    public static void main(String[] args){
        Properties prop=new Properties();
        try{
            InputStream in=new BufferedInputStream(new
FileInputStream("src/resource.properties"));
            prop.load(in);

            try {
        Class clz=Class.forName(prop.getProperty("classname"));
       demo object= (demo)clz.getDeclaredConstructor().newInstance();
                System.out.println(object);
            }catch (ClassNotFoundException e) {
                e.printStackTrace();
            }catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            } catch (NoSuchMethodException e) {
                e.printStackTrace();
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Task 2

In **singleton** package, please merge two concrete factories into one concrete factory (`DaoFactoryImpl`), and you need make sure that the instantiate process for `StaffDao` and `ComputerDao` is by reflection, and the class name of concrete factories are from `resource.properties`.

Modify your concrete factory (`DaoFactoryImpl`) to be a **singleton**.

Your code need to match the client.