第二十一章 常用设计原则和设计模式

21.1 常用的设计原则（记住）

21.1.1 软件开发的流程

- 需求分析文档、概要**设计**文档、详细设计文档、编码和测试、**安装和调试**、维护和升级

Note：设计、调试环节较为耗时

21.1.2 常用的设计原则

- **开闭原则（Open Close Principle）**

  对扩展开放，对修改关闭。为了使程序的扩展性好，易于维护和升级。

  能不修改就尽量不修改。E.g. 增加 Person 类的 attribute：建立 SubPerson 类 extends Person 类。

- **里氏代换原则（Liskov Substitution Principle）**

  任何**基类**(父类)可以出现的地方，**子类**一定可以出现，多使用**多态**的方式。(建议多使用多态)

- **依赖倒转原则（Dependence Inversion Principle）**

  尽量多依赖于抽象类或接口而不是具体实现类，对子类具有**强制性和规范性。**

  e.g. 尽量不要继承普通的实现类，而要继承 abtract class / 实现 interface。普通实现类管不住子类。

- **接口隔离原则（Interface Segregation Principle）**

  尽量**多使用小接口**而不是大接口，避免接口的污染，**降低类之间耦合度（关联）**。

  <I> Animal{run, fly}　　<C> Dog implements Animal (×，Dog 不能飞接口污染)

  <I> RunAnimal{run}, <I> FlyAnimal{fly}　　<C> Dog implements RunAnimal (√)

- **迪米特法则**（少知道原则: 知道的越少越好）（**Demeter Principle）**

  一个实体应当尽量少与其他实体之间发生相互作用，**使系统功能模块相对独立。**

  高内聚，低耦合。　　**E.g.老师应具备老师应有的技能，而不是修水龙头、打印机的技能。**

- **合成复用原则（Composite Reuse Principle）**

  尽量**多使用合成/聚合的方式**，而不是继承的方式。

  e.g. Class B 调用 Class A 中的 show()方法，不要直接 B extends A，而是在 B 类的的私有变量中，声明 private A a; 然后构造器里 this.a = a（用于网络编程）

  <在当前类中使用其他类中的成员>

```
public class B/* extends A*/ {
    private A a; // 合成复用原则

    public B(A a) {
        this.a = a;
    }

    public void test() {
        // 调用A类中的show方法，请问如何实现？
        a.show();
    }
}
```

```
public class A {

    public void show() {
        System.out.println("这是A类中的show方法！");
    }
}
```

21.2 常用的设计模式

21.2.1 基本概念

- **设计模式(Design pattern** 是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。设计模式就是一种**用于固定场合的固定套路**。
-

21.2.2 基本分类

- **创建型模式** (对象的创建) - 单例设计模式、工厂方法模式、抽象工厂模式、...
- **结构型模式** (结构关系) - 装饰器模式、代理模式、...
- **行为型模式** (行为效果) - 模板设计模式、...

21.3 设计模式详解（重点）

21.3.1 单例设计模式

- **单例设计模式**主要分为：**饿汉式** 和 **懒汉式**，**推荐饿汉式**。∵ 懒汉式需要对多线程进行同步处理。

Step 1: 私有化构造器，使用 private 关键字修饰
Step 2: 声明本类类型的引用，指向本类类型的对象，并使用 private static 修饰
Step 3: 提供公有 getter 方法，并将上述对象返回出去，并用 public static 修饰
懒汉式会有多线程问题：若两个线程同时进入 getInstance()方法，有可能会创建两个对象。
因此，**饿汉式**需用 synchronized 关键字

```
public class Singleton {  饿汉式

    // 2.声明本类类型的引用指向本类类型的对象并使用private static关键字修饰
    private static Singleton sin = null;

    // 1.私有化构造方法，使用private关键字修饰
    private Singleton() {}

    // 3.提供公有的get方法负责将上述对象返回出去，使用public static关键字修饰
    public static synchronized Singleton getInstance() {
        if (null == sin) {
            sin = new Singleton();
        }
        return sin;
    }
}
```

需处理多线程问题

或者

```
    // 3.提供公有的get方法负责将上述对象返回出去，使用public static关键字修饰
    public static /*synchronized*/ Singleton getInstance() {
        synchronized (Singleton.class) {
            if (null == sin) {
                sin = new Singleton();
            }
            return sin;
        }
    }
```

或者（**优化，最佳↓**）

```
    // 3.提供公有的get方法负责将上述对象返回出去，使用public static关键字修饰
    public static /*synchronized*/ Singleton getInstance() {
```

```
        if (null == sin) {
            synchronized (Singleton.class) {
                if (null == sin) {
                    sin = new Singleton();
                }
            }
        }
        return sin;
    }
```

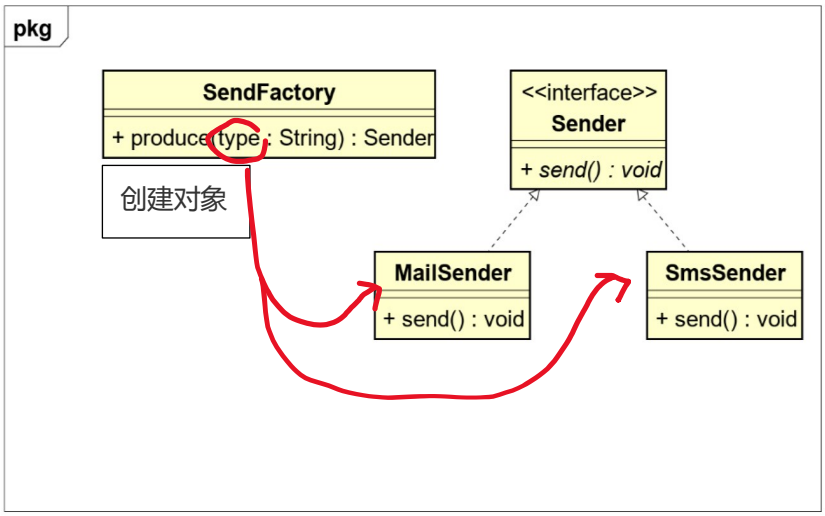
21.3.2 普通工厂模式

（1）基本概念

- 普通工厂方法模式就是**建立一个工厂类**，对实现了同一接口的不同实现类进行实例的创建。

优点：**扩展性**和**可维护性强**（可以直接在 Factory 类里改），尤其是在创建**大量对象**的前提下。把所有创建对象的工作交给工厂类。（e.g. 手机公司把手机组装委托给第三方工厂，批量生产）

（2）类图结构




```
public interface Sender {
    // 自定义抽象方法来描述发送的行为
    void send();
}
```

```
public class MailSender implements Sender {
    @Override
    public void send() {
        System.out.println("正在发送邮件...");
    }
}
```

```
public class SmsSender implements Sender {
    @Override
    public void send() {
        System.out.println("正在发送短信...");
    }
}
```

```
public class SendFactory {
    // 自定义成员方法实现对象的创建
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        }
        if ("sms".equals(type)) {
            return new SmsSender();
        }
        return null;
    }
}
```

```
public class SendFactoryTest {

    public static void main(String[] args) {

        // 1.声明工厂类类型的引用指向工厂类类型的对象
        SendFactory sf = new SendFactory();
        // 2.调用生产方法来实现对象的创建
        Sender sender = sf.produce( type: "mail");
        // 3.使用对象调用方法模拟发生的行为
        sender.send();
    }
}
```

Terminal: 正在发送邮件...

（3）主要缺点

- 在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，并且可能出现**空指针异常。**

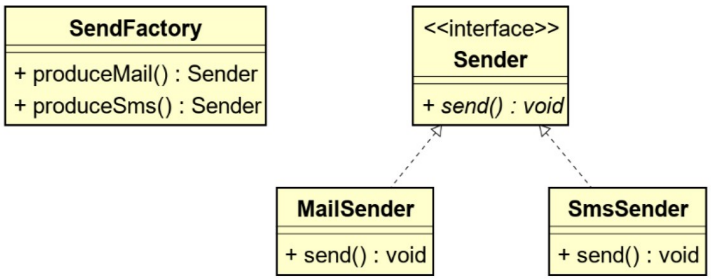
```
Sender sender = sf.produce( type: "maill");
// 3.使用对象调用方法模拟发生的行为
sender.send();
```

字符串传错了

空指针异常

21.3.3 多个工厂方法模式（改进版）

（1）类图结构



SendFactory:

```
public Sender produceMail() {
    return new MailSender();
}
public Sender produceSms() {
    return new SmsSender();
}
```

→ 避免了字符串的传入，避免了空指针异常

Main:

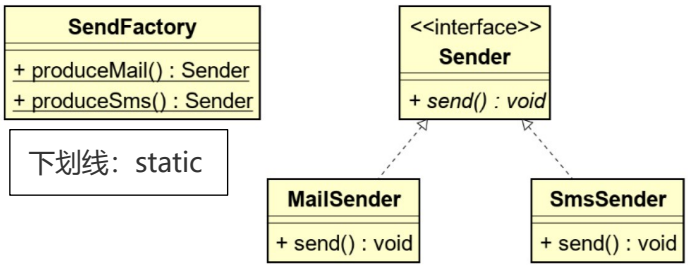
```
Sender sender = sf.produceMail();
```

（2）主要缺点

- 在多个工厂方法模式中，为了能够正确创建对象，先需要创建工厂类的对象才能调用工厂类中的生产方法。

21.3.4 静态工厂方法模式

（1）类图结构



SendFactory:

```
public static Sender produceMail() {
    return new MailSender();
}
public static Sender produceSms() {
    return new SmsSender();
```

Main:

```
Sender sender = SendFactory.produceMail();
```

（2）实际意义

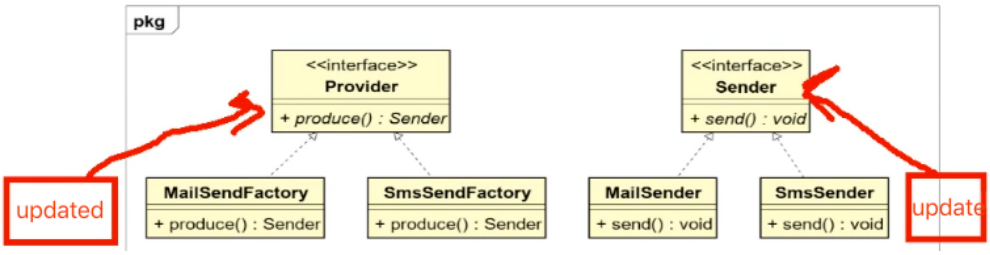
- 工厂方法模式适合：凡是**出现了大量的产品需要创建**且**具有共同的接口**时，可以通过工厂方法模式进行创建。

（3）主要缺点

- 工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序生产新的产品，就必须对工厂类的代码进行修改，这就违背了开闭原则 (对拓展开放，对修改关闭)。

21.3.5 抽象工厂模式

（1） 类图结构 （多个工厂类，各自的生产方法）



维护(加新的类)：

```java
public class PacketSender implements Sender {
    @Override
    public void send() {
        System.out.println("正在发送包裹...");
    }
}
```

```java
public class PacketSendFactory implements Provider {
    @Override
    public Sender produce() {
        return new PacketSender();
    }
}
```

Main：

```java
Provider provider1 = new PacketSendFactory();
Sender sender3 = provider1.produce();
sender3.send();
```

(2) 优势

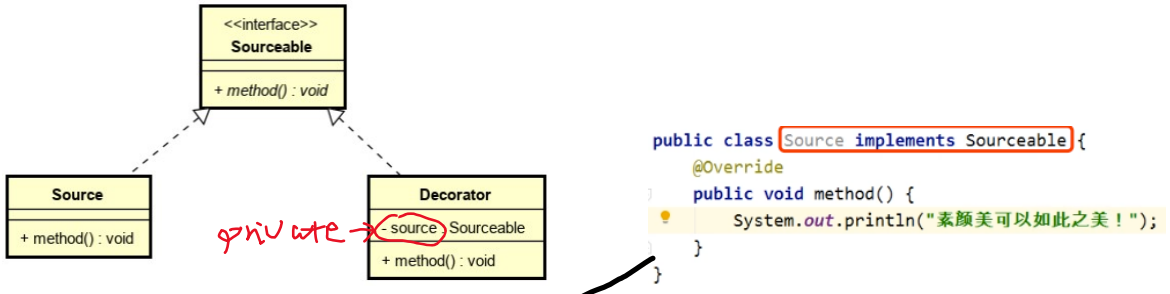
- 符合开闭原则，可维护。

21.3.6 装饰器模式

（1）基本概念

- 装饰器模式就是给一个对象动态的增加一些新功能，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

（2）类图结构




```java
public interface Sourceable {
    // 自定义抽象方法
    void method();
}
```

```java
public class Source implements Sourceable{
    @Override
    public void method() {
        System.out.println("素颜美可以如此之美！");
    }
}
```

```java
public class Decorator implements Sourceable {
    private Sourceable source;

    public Decorator(Sourceable source) {
        this.source = source;
    }

    @Override
    public void method() {
        source.method(); // 保证原有功能不变
        System.out.println("化妆之后你会更美！");
    }
}
```

原有功能不变的情况下，增加了新功能

```
// 接下来使用装饰类实现功能
Sourceable sourceable1 = new Decorator(sourceable);
sourceable1.method();
```

```
素颜美可以如此之美！
化妆之后你会更美！
```

（3）实际意义

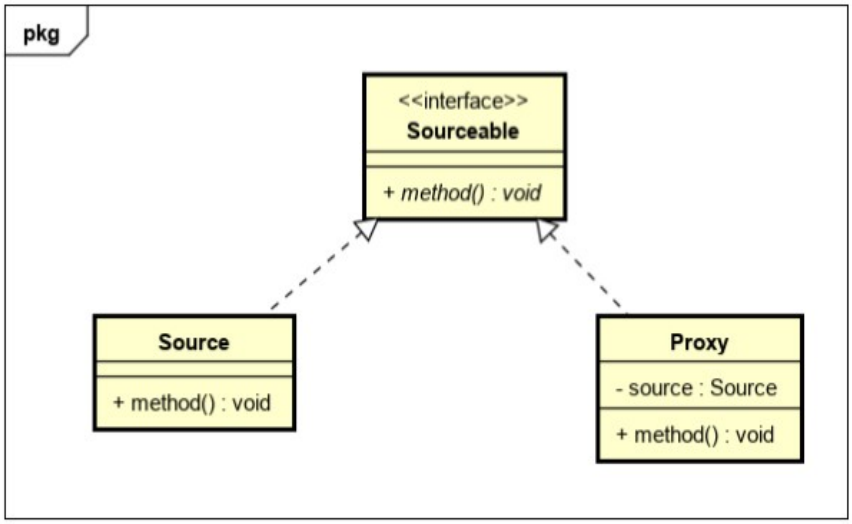
- **可以实现一个类功能的扩展。**
- 可以**动态的增加功能**，而且**还能动态撤销**（继承不行）。
- 缺点：产生过多相似的对象，不易排错。

21.3.7 代理模式

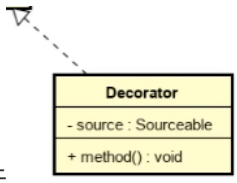
（1）基本概念

- **代理模式**就是找一个代理类替原对象进行一些操作。
- 比如我们在租房子的时候找中介，再如我们打官司需要请律师，中介和律师在这里就是我们的**代理。**

（2）类图结构



区别于




```
public class Proxy implements Sourceable {
    private Source source;

    public Proxy() {
        source = new Source();
    }

    @Override
    public void method() {
        source.method();
        System.out.println("我和装饰器模式其实是不一样的！");
    }
}
```

Main:

```
Sourceable sourceable1 = new Decorator(sourceable);
sourceable1.method();

System.out.println("------------------------------------------------------");
Sourceable sourceable2 = new Proxy();
sourceable2.method();
```

```
素颜美可以如此之美！
化妆之后你会更美！
------------------------------------------------------
素颜美可以如此之美！
我和装饰器模式其实是不一样的！
```

（3）实际意义

- 如果在使用的时候需要对原有的方法进行改进，可以采用一个代理类调用原有方法，并且对产生的结果进行控制，这种方式就是代理模式。
- 使用代理模式，可以将功能划分的更加清晰，有助于后期维护。

（4）代理模式和装饰器模式的比较

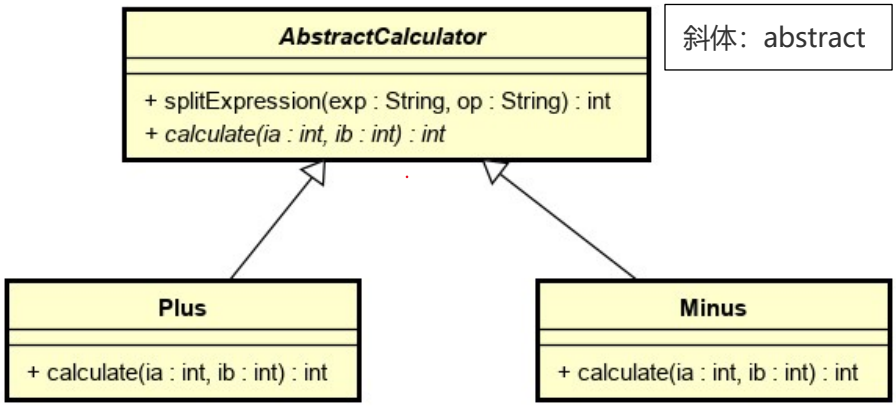
- **装饰器模式**通常的做法是将原始对象作为一个参数传给装饰者的构造器，而**代理模式**通常**在一个代理类中创建一个被代理类的对象**。
- 装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。

21.3.8 模板方法模式

（1）基本概念

- 模板方法模式主要指一个抽象类中封装了一个固定流程，流程中的具体步骤可以由不同子类进行不同的实现，通过抽象类让固定的流程产生不同的结果。

（2）类图结构



（3）实际意义

- 将多个子类共有且逻辑基本相同的内容提取出来实现代码复用。
- 不同的子类实现不同的效果形成多态，有助于后期维护。

```
public abstract class AbstractCalculator {

    // 自定义成员方法实现将参数指定的表达式按照参数指定的规则进行切割并返回计算结果  1+1  +
    public int splitExpression(String exp, String op) {
        String[] sArr = exp.split(op);
        return calculate(Integer.parseInt(sArr[0]), Integer.parseInt(sArr[1]));
    }
    // 自定义抽象方法实现运算
    public abstract int calculate(int ia, int ib);
}
```

```
public class Plus extends AbstractCalculator {
    @Override
    public int calculate(int ia, int ib) {
        return ia + ib;
    }
}
```

```
public class Minus extends AbstractCalculator {
    @Override
    public int calculate(int ia, int ib) {
        return ia - ib;
    }
}
```

```java
public class AbstractCalculatorTest {

    public static void main(String[] args) {

        AbstractCalculator abstractCalculator = new Plus();
        int res = abstractCalculator.splitExpression( exp: "1+1",  op: "\\+");
        System.out.println("最终的运算结果是：" + res); // 2
    }
}
```

正则

Module Summary:

1.常用的设计原则和设计模式

软件开发的流程、常用的设计原则、设计模式的概念和分类、常用设计模式的详解等