1. 创建型设计模式

创建型模式对类的实例进行了抽象，

1.1 单例模式

1.1.1 分析

动机:

对于系统的某些类来说，只有一个实例很重要，为了保证一个类只有一个实例，并且这个实例易于被访问。让类自身负责保存它唯一的方法，防止我们实例化多个对象

定义:

单例模式确保某一个类只有一个实例，并且自行实例化并向整个系统提供这个实例，这个类提供全局范围的方法。

分析:

单例模式包含的角色只有一个，就是单例本身，单例类拥有一个私有构造器，确保用户无法通过new关键字直接实例化。该模式中包含有一个静态私有成员变量与静态公有的工厂方法，该工厂检验实例的存在性并实例化自己，存储到静态成员变量中，确保只有一个实例被创建。

单例类的构造函数为私有

提供一个自身的静态私有成员变量

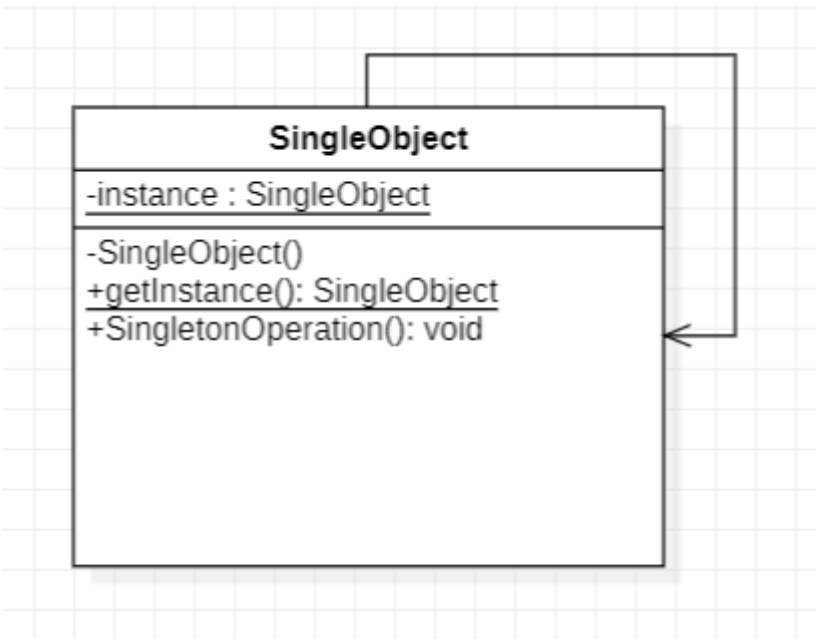
提供一个公有的静态工厂方法

使用场景：

1. 系统只需要一个实例对象
2. 调用类的单个实例，只允许使用一个公告访问点，不能通过其它途径访问
3. 在一个系统中，要求一个类只有一个实例

1.1.2 UML

UML 图：



1.1.3 代码实现：

懒汉模式

1、简单初始化方式，在`getInstance()`中判断当前实例是否被创建，如果为空，则调用构造器创建对象。

线程不安全：

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

  private Singleton(){}

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

  public void SingletonOperation(){
    ....
  }
}
```

线程安全：

使用`synchronized`加锁，但是会影响效率

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

  private Singleton(){}
```

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

  public void SingletonOperation(){
    ....
  }
}
```

双重检查：

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

  private Singleton(){}

  public static Single getInstance(){
    if(instance == null){
      synchronize(Single.class){
        if(instance == null){
          instance = new Singleton();
        }
      }
    }
    return instance;
  }

  public void SingletonOperation(){
    ....
  }
}
```

饿汉模式

直接赋值

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

  private Singleton(){}

  public static Singleton getInstance(){
    return instance;
  }

  public void SingletonOperation(){
    ...
  }
}
```

静态代码赋值

```
public class Singleton{
  private static Single instance;

  static{
    try {
      instance = new Singtон();
    }catch(Exception e){
      throw new RuntimeException("Exception occured in
creation singleton instance");
    }
  }

  private Singleton(){}
```

```
  public static Singleton getInstance(){
    return instance;
  }

  public void SingletonOperation(){
    ...
  }
}
```

用户使用的时候:

```
public class Client{
  pubic staic void main(String[] args){
    //Singleton singletonObject = new Singleton();
    //会报错

    Singleton singletonObejct = Singleton.getInstance();

    singletonObject.SingletonOperation();

  }
}
```

1.2 原型模式

1.2.1 分析

动机：

原型模式实现了一个原型接口，该接口用于创建当前对象的克隆。用于创建重复的对象，同时又能保证性能。

定义：

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。允许一个对象再创建另外一个可定制对象，无需知道如何创建的细节。工作原理是将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它。

分析：

Object类提供了一个clone（）方法，该方法可以讲Java对象复制一份，但是需要实现clone的java类必须要实现一个接口cloneable，该接口表示该类能够复制且具有复制的能力。

实现克隆操作，在Java实现Cloneable接口，重写clone()。

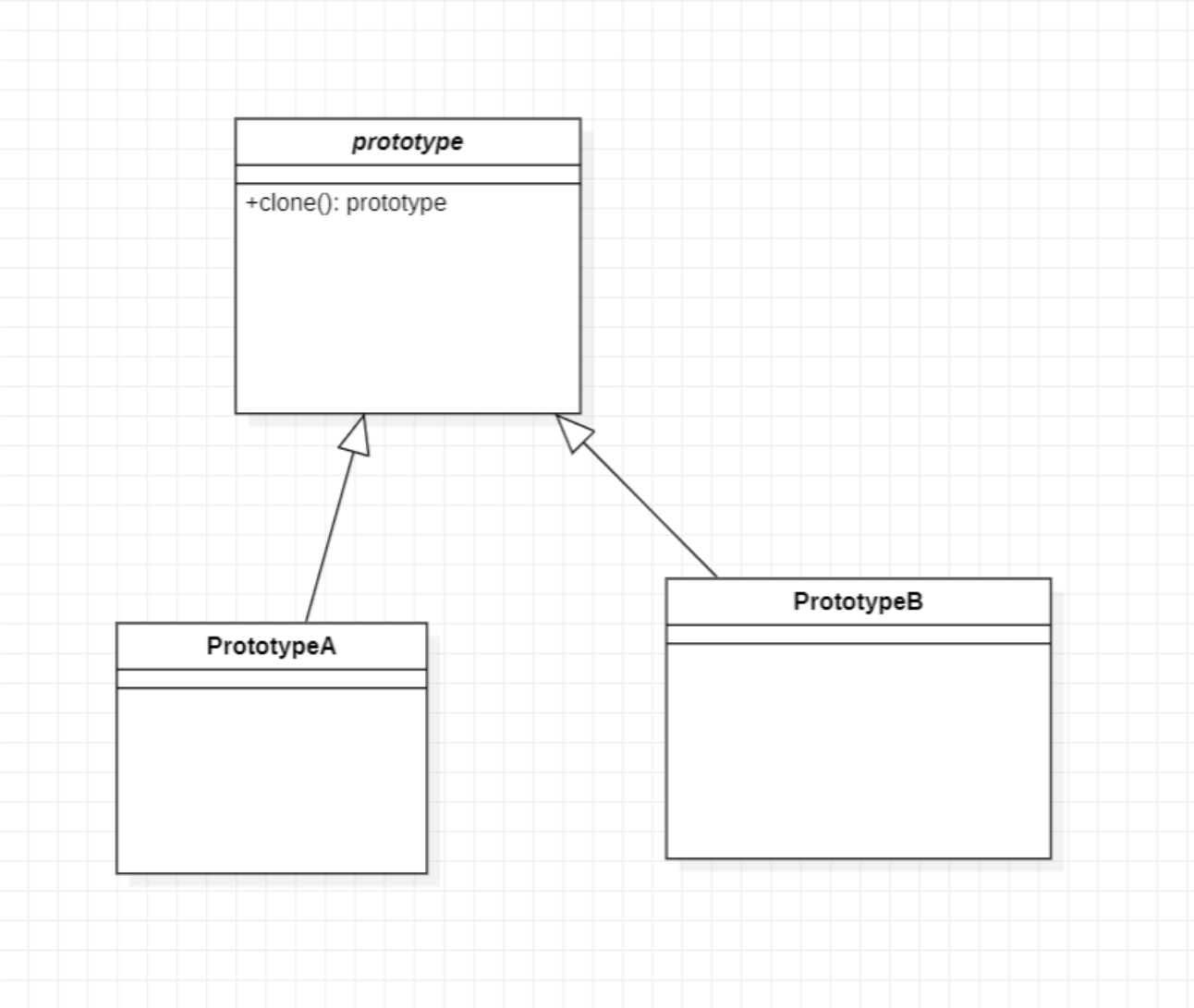
由于 `clone()` 方法的可见性为 `protected` ，需要在自定义中重写 `clone()` 方法，并将可见性修改为 `public`

`clone()` 方法声明了 `CloneNotSupportedException` 异常， 当被克隆对象对应的类没有实现 `Cloneable()` 接口时会抛出该异常

使用场景：

1. 系统只需要一个实例对象
2. 调用类的单个实例，只允许使用一个公告访问点，不能通过其它途径访问
3. 在一个系统中，要求一个类只有一个实例

1.2.2 UML图



1.2.3 代码实现

```
public abstract class Shape implements Cloneable{
  private String id;
  protected String type;


  abstract void draw();
```

```
  public String getType(){
    return type;
  }

  public String getId(){
    return id;
  }

  public void setId(String id){
    this.id = id;
  }

  public Object clone(){
    Object clone = null;
    try{
      clone = super.clone();

    }catch(CloneNotSupportedException e){
      e.printStackTrace();
    }
    return clone;
  }
}
```

1.3 建造者模式

1.3.1 分析

动机：

在开发软件系统的时候存在一些复杂的对象，它们拥有多个组成部分。建造者模式可以将部件和组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，无须知道其内部的具体构造细节。

由于组合部件的过程很复杂，因此，这些部件的组合过程往往被外部化到一个称作建造者的对象中，建造者返还给用户一个已经建造完成的产品对象，用户无序关心该对象所包含的属性以及它们的组装方式。

定义：

建造者模式将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

创建者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建他们，用户不需要知道内部的具体构造细节。

分析：

建造者模式包含以下角色

- Builder：抽象建造者
- ConcreteBuilder：具体建造者
- Director：指挥者
- Product：产品角色

抽象建造者类中定义了产品的创建方法和返回方法。建造者模式的结构中还引入了一个指挥者类Director，该类一方面隔离了客户与组装的过程，另一方面负责指挥产品组装的过程。指挥者针对抽象建造者编程，客户端只需要知道具体建造者类型，即可以通过指挥类调用建造者的相关方法，返回一个完整的产品。

在客户端代码中，无须关心产品对象的具体组装过程，只需要确定具体建造者的类型即可，建造者模式将复杂对象的构建与对象的表现分离开来，这样使得同样的构建过程可以创建出不同的表现。

用户使用不同的具体构造者既可得到不同的产品对象。每一个具体建造者都相互独立，而且与其它的具体构造者无关，因此可以很方便地替换具体构造者，或者增加新的具体构造者。

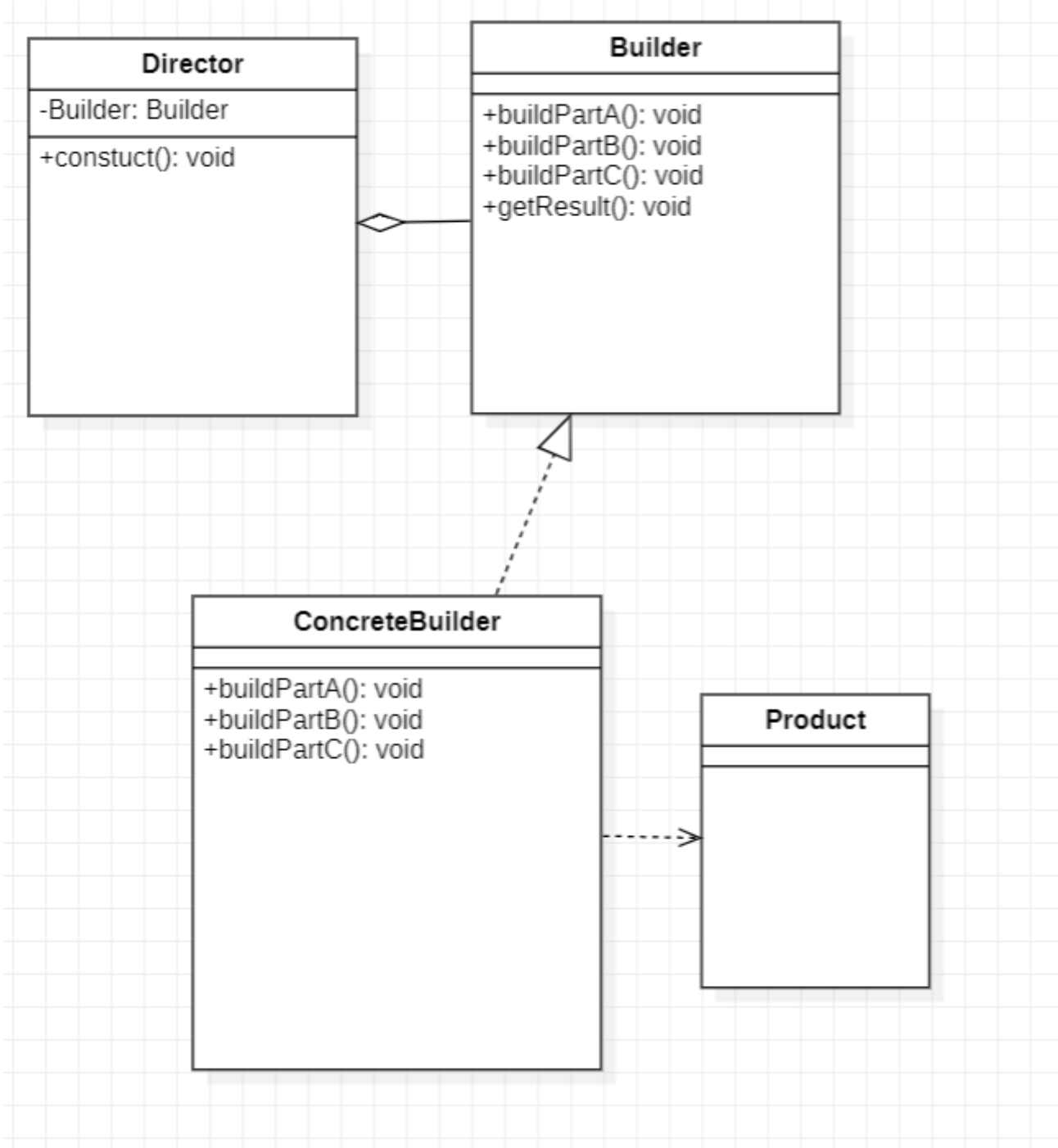
更精细控制产品创建过程。将复杂产品的创建步骤分解在不同的方法中，是得创建过程更加清晰，更方便使用程序来控制创建过程。

增加新的具体创建者无需修改原有类的代码。指挥者类针对抽象建造者类编程扩展更加方便。

使用场景：

1. 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。
2. 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。
3. 对象的创建过程独立于创建对象的类。

1.3.2 UML图



1.3.3 代码例子实现

以一个餐厅的套餐为例子

套餐A： 汉堡、薯条、大鸡腿

套餐B：汉堡、薯条，小鸡腿，可乐，披萨

```
// Product类，一个具体产品对象
public class Meal{
  //套餐都含有的
  private String hamburger;
  private String fries;

  //选择
  private String chicken;
  private String coke;
  private String pizza;

  //必点
  public Meal(String hamburger, String fries){
    this.hamburger = hamburger;
    this.fries = fries;
  }
  //选择
  public void setChicken(String chicken){
    this.chicken = chicken;
  }
  public void setCoke(String coke){
    this.coke = coke;
  }
  public void setPizza(String pizza){
    this.pizza = pizza;
  }


  //
}
```

```
//Builder类，表面需要建造的东西
public interface BuilderP{
  void setChicken();
  void setCoke();
  void setPizza();
  Meal getMeal();
}
```

```
//ConcreteBuilder类，实现抽象接口，构建和装配

//套餐A
public class MealA implements Builder{
  private Meal meal;

  public MealA(String hamburger, String fries){
    meal = new Meal(hamburger, fries);
  }

  public void setChicken(){
    meal.setChicken("大鸡腿");
  }

  @override
  public void setCoke(){
    meal.setCoke(null);
  }

  @override
  public void setPizza(){
    meal.setPizza(null);
  }

  @override
```

```
  public Meal getMeal(){
    return meal;
  }
}


//套餐B
public class MealB implements Builder {
    private Meal meal;

    public MealB(String hamburger,String fries){
        meal = new Meal(hamburger,fries);
    }
    @Override
    public void setChicken() {
        meal.setChicken("小鸡腿");
    }

    @Override
    public void setCoke() {
        meal.setCoke("小可乐");
    }

    @Override
    public void setPizza() {
        meal.setPizza("小披萨");
    }

    @Override
    public Meal getMeal(){
         return meal;
    }
}
```

```
//Director类，构建一个builder接口对象，用于创建复杂对象

public class Director{
  public Meal build(Builder builder){
    builder.setChicken();
    builder.setCola();
    return builder.getMeal();
  }
}
```

```
//test类
public class Test{
  public static void main(String[] args){
    System.out.println("套餐A");
    Builder meal1 = new MealA("大汉堡", "小薯条");
    Meal mealA = new Director().build(Meal1);
    System.out.println(mealA);

    System.out.println("套餐B");
    Builder meal2 = new MealB("大汉堡", "大薯条");
    Meal mealB = new Director().build(Meal2);
    System.out.println(mealB);
  }
}
```

1.4 简单工厂模式

1.4.1 分析

动机：

考虑一个简单的软件应用场景，一个软件系统可以提供多个外观不同的按钮（如圆形按钮、矩形按钮、菱形按钮等），这些按钮都源自同一个基类，不过在继承基类后不同的子类修改了部分属性从而使得它们可以呈现不同的外观，如果我们希望在使用这些按钮时，不需要知道这些具体按钮类的名字，只需要知道表示该按钮类的一个参数，并提供一个调用方便的方法，把该参数传入方法即可返回一个相应的按钮对象，此时，就可以使用简单工厂模式。

定义：

简单工厂模式，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其它类的实例，被创建的实例通常都具有共同的父类。

引入一个专门负责创建具体类对象的工厂类，解除了测试类与具体产品类之间的紧密耦合关系，使其不需要了解具体类的细节。

分析：

- Factory：工厂角色负责实现创建所有实例的内部逻辑
- Product：抽象产品角色是所创建的所有对象的父类，负责描述所有实例共用的公共接口
- ConcreteProduct：具体产品角色是创建目标，所有创建的对象都充当这个角色的某个具体类的实例

所有对象的创建和对象本身业务处理分离可以降低系统耦合度，使得两者修改起来容易。

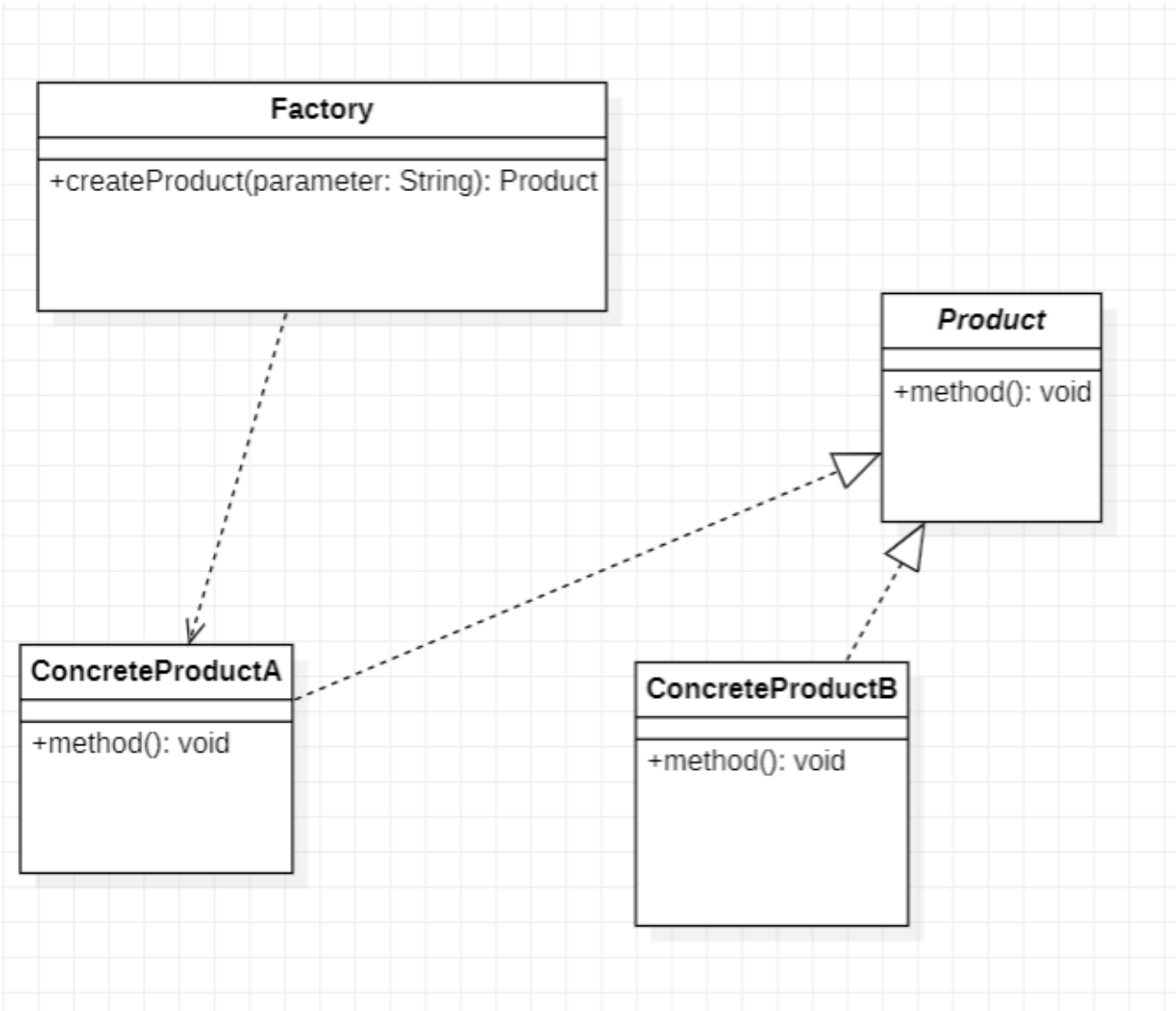
简单工厂模式主要在于只需要传入一个你需要的正确参数，就可以获取你所需要的对象，无序了解细节。

使用场景：

负责创建的对象比较少，不会造成业务逻辑太过于复杂

客户端不需要关系创建细节，只需要知道类对应的参数

1.4.2 UML 图



1.4.3 代码实现

```
//product类，建立接口
public interface Shape{
  void draw();
}
```

```
//ConcreteProduct类,实现接口
public class Rectangle implements Shape{
  @override
  public void draw(){
    System.out.println("Rectangle draw method"))
  }
}


public class Square implements Shape{
  @override
  public void draw(){
    System.out.println("Square draw method");
  }
}


public class Circle implements Shape{
  @override
  public void draw(){
    System.out.println("Circle draw method");
  }
}
```

```java
//Factory 类
public class ShapeFactory{
  public Shape getShape(String shapeType){
    if (shapeType == null){
      return null;
    }
    if (shapeType.equalsIgnoreCase("circle")){
      return new Circle();
    }else if (shapeType.equalsIgnoreCase("rectangle")){
      return new Rectangle();
    }else if (shapeType.equalsIgnoreCase("square")){
      return new Square();
    }
    return null;
  }
}
```

```java
public class Factory{
  public static void main(String[] args){
    ShapeFactory shapeFactory = new ShapeFactory();

    Shape shape1 = shapeFactory.getShape("circle");
    shape1.draw();

    Shape shape2 = shapeFactory.getShape("rectangle");
    shape2.draw();

    Shape shape3 = shapeFactory.getShape("square");
    shape3.draw();

  }
}
```

1.5 工厂方法模式

1.5.1 分析

动机:

现在对该系统进行修改，不再设计一个按钮工厂类来统一负责所有产品的创建，而是将具体按钮的创建过程交给专门的工厂子类去完成，我们先定义一个抽象的按钮工厂类，再定义具体的工厂类来生成圆形按钮、矩形按钮、菱形按钮等，它们实现在抽象按钮工厂类中定义的方法。这种抽象化的结果使这种结构可以在不修改具体工厂类的情况下引进新的产品，如果出现新的按钮类型，只需要为这种新类型的按钮创建一个具体的工厂类就可以获得该新按钮的实例，这一特点无疑使得工厂方法模式具有超越简单工厂模式的优越性，更加符合“开闭原则”

定义:

工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样子做的目的是将产品类的实例化操作延迟到工厂子类中完成，通过工厂子类来确定应该实例化哪一个具体的产品类。

分析:

与简单工厂类相似，包含着如下角色:

- Product： 抽象产品
- ConcreteProduct：具体产品
- Factory: 抽象工厂
- ConcreteFactory：具体工厂

工厂方法是简单工厂模式的进一步抽象和推广。使用了面向对象中的多态。在工厂方法中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类来做了，这个核心类仅仅给出具体的工厂必须实现的接口，而不负责哪一个产品类被实例这个细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

简单的说，就是工厂类没有了对于传入参数的if else判断，这个过程交给子类来实现。

简单工厂：

```
if(arg.equals("A")){
  return new ConcreteA();
}else if(arg.equals("B")){
  return new ConcreteB();
}
```

工厂方法：

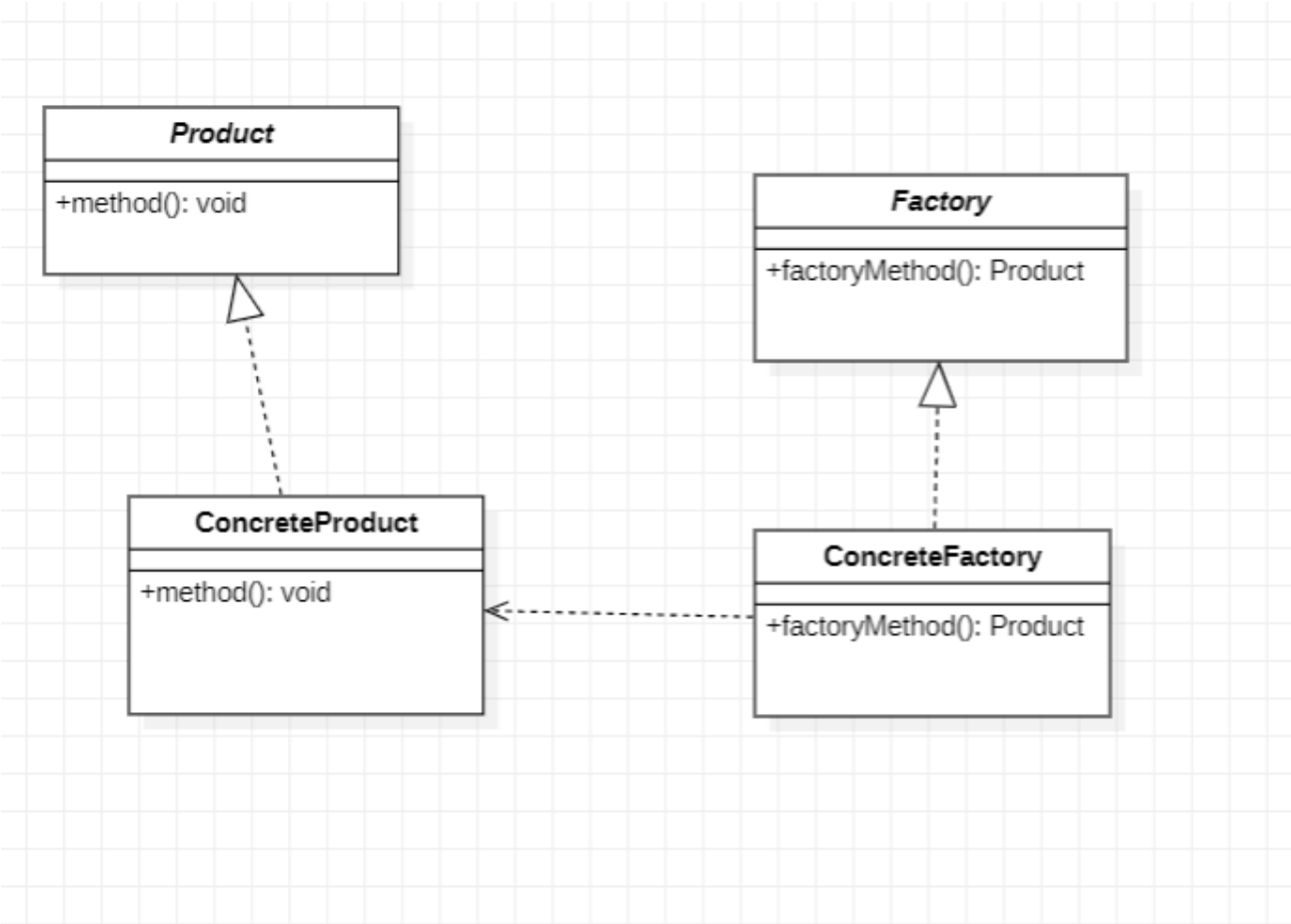
```
public ConcreteFactoryA implements IFactory{
  public Product method(){
    return new ConcreteA();
  }
}

public ConcreteFactoryB implements IFactory{
  public Product method(){
    return new ConcreteB();
  }
}
```

使用场景：

一个类不知道它所需要的对象的类。在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类

一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

1.5.2 UML图



1.5.3 代码

实现简单工厂的例子

```
//Product接口
public interface Geometry{
  public void draw();
}
```

```
//ConcreteProduct类
public class Square implements Geomerty{
  @override
  public void draw(){
    System.out.println("Square");
  }
}


public class Rectangle implements Geomerty{
  @override
  public void draw(){
    System.out.println("Rectangle");
  }
}


public class Circle implements Geomerty{
  @override
  public void draw(){
    System.out.println("Circle");
  }
}
```

```
//Factory接口
public interface Shapefactory{
  public void GeometryCreate();
}
```

```
//ConcreteFactory类
public SquareFactory implements Shapefactory{
  @override
  public void GeometryCreate(){
    return new Square();
  }
}

public CircleFactory implements Shapefactory{
  @override
  public void GeometryCreate(){
    return new Circle();
  }
}

public RectanleFactory implements Shapefactory{
  @override
  public void GeometryCreate(){
    return new Rectanle();
  }
}
```

```
//Client
public class Test{
  public static void main(String[] args){
```

```
        //Square
        ShapeFactory fatory = new SquareFactory();
        Product geometry = fatory.GeometryCreate();
        geometry.draw();

        //Circle
        factory = new CircleFactory();
        geometry = factory.GeometryCreate();
        geometry.draw()

        //Rectangle
        factory = new RectangleFactory();
        geometry = factory.GeometryCreate();
        geometry.draw();
    }
}
```

1.6 抽象工厂方法模式

1.6.1 分析

动机:

工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但有时候我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

当系统所提供的工厂所需生产的具体产品并不是一个简单对象，而是多个位于不同产品结构中属于不同类型的具体产品时需要使用抽象工厂模式。

定义：

抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体的类。

分析：

抽象工厂是工厂方法的一种，同样包含如下角色

- AbstractFactory：抽象工厂
- ConcreteFactory：具体工厂
- AbstractProduct：抽象产品
- Product：具体产品

当系统所提供的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中同属于不同类型的具体产品时所需要使用抽象工厂模式。

抽象工厂模式是所有工厂模式中最抽象和最具一般的一种。

抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象创建。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族的所有对象，抽象工厂模式比工厂方法模式更为简单有效。

使用场景：

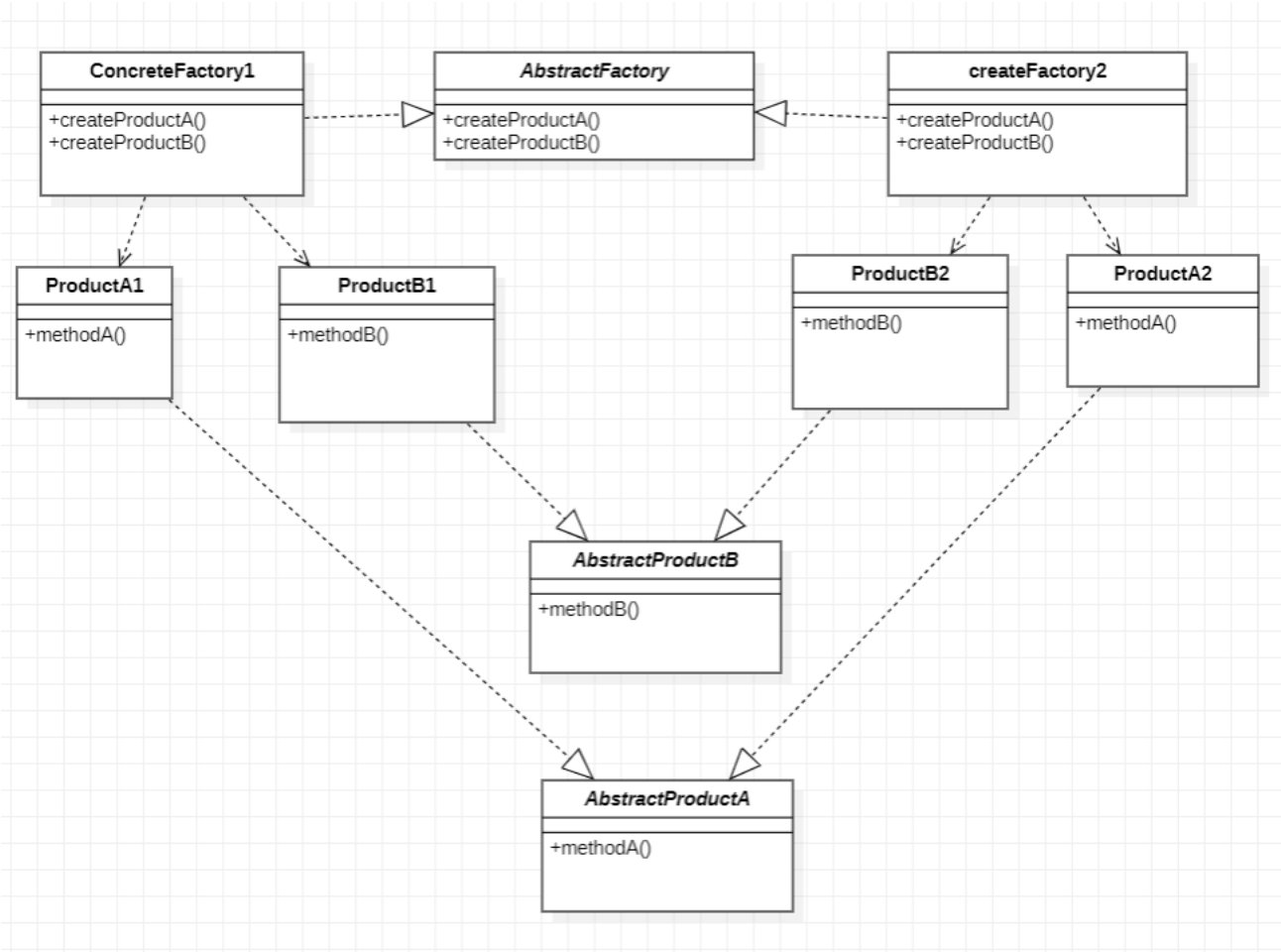
一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的。

系统中有多于一个的产品族，而每次只使用其中某一产品族。

属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。

系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

1.6.2 UML图



1.6.3 代码

```
//AbstractProduct 1
public interface Shape{
  public void draw();
}
```

```
//ConcreteProduct类
public class Rectangle implements Shape{
  @override
  public void draw(){
    System.out.println("Rectangle draw method");
  }
}


public class Square implements Shape{
  @override
  public void draw(){
    System.out.println("Square draw method");
  }
}


public class Circle implements Shape{
  @override
  public void draw(){
    System.out.println("Circle draw method");
  }

}
```

```
//AbstractProduct2
public interface Color{
  public void fill();
}
```

```
//ConcreteProduct2
public class Red implements Color{
  @override
  public void fill(){
    System.out.println("Red filled");
  }
}


public class Green implements Color{
  @override
  public void fill(){
    System.out.println("Green filled");
  }
}


public class Blue implements Color{
  @override
  public void fill(){
    System.out.println("Blue filled");
  }
}
```

```
//AbstractFactory 类
public abstract interface AbstractFactory{
  public abstract Color getColor(String color);
  public abstract Shape getShape(String shape);
}
```

```
//ConcreteFactory
public class ShapeFactory extends AbstractFactory{
  @override
  public Shape getShape(String shapeType){
    if (shapeType == null){
      return null;
    }else if (shapeType.equalsIgnoreCase("Circle")){
      return new Circle();
    }else if (shapeType.equalsIgnoreCase("Rectangle")){
      return new Rectangle();
    }else if (shapeType.equalsIgnoreCase("Square")){
      return new Square();
    }
    return null;
  }

  @override
  public Color getColor(String color){
    return null;
  }
}
```

```
//ColorFactory
public class ColorFactory extends AbstractFactory{
  @override
  public Shape getShape(String shapeType){
    return null;
  }

  @override
  public Color getColor(String color){
    if (color == null){
      return null;
    }
    if (color.equalsIgnoreCase("Red")){
      return new Red();
    }else if (color.equalsIgnoreCase("Green")){
      return new Green();
    }else if (color.equalsIgnoreCase("Blue")){
      return new Blue();
    }
  }
}
```

```
//Client
public class Test{
  public static void main(String[] args){
    AbstractFactory shapeFactory = new ShapeFactory();
    Shape shape1 = shapeFactory.getShape("Circle");
    shape1.draw();

    AbstractFactory colorFactory = new ColorFactory();
    Color color1 = colorFactory.getColor("Red");
    color1.fill();
  }
}
```

2. 结构性模式

2.1 适配器模式

2.1.1 分析

动机:

通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。

适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。当客户端调用适配器方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户端是透明的，客户端不直接访问适配类。适配器可以使由于接口不兼容而不能交互的类可以一起工作。

定义：

在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。

适配器模式将一个接口转化为客户希望的另一个接口，适配器模式使接口不同的那些类可以一起工作。在适配器模式中可以定义一个包装类，包装不兼容接口的对象。

分析：

适配器模式包含如下角色

- Target：目标抽象类，定义客户要用的特定领域的接口
- Adapter：适配器类，可以调用另一个接口，作为一个转换器，对适配者和抽象目标类进行适配
- Adaptee：适配者类，定义了一个存在的接口，这个接口需要适配

客户端这对目标抽象类进行编程，调用在目标抽象类中定义的业务方法。

适配器模式用于将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作。

在类适配器模式中，适配器类实现了目标抽象类接口并继承了适配者类，并在目标抽象类的实现方法中调用所继承的适配者类的方法。

在对象适配器模式中，适配器类继承了目标抽象类并定义了一个适配者类的对象实例，在所继承的目标抽象类方法中调用适配者类的相应业务方法

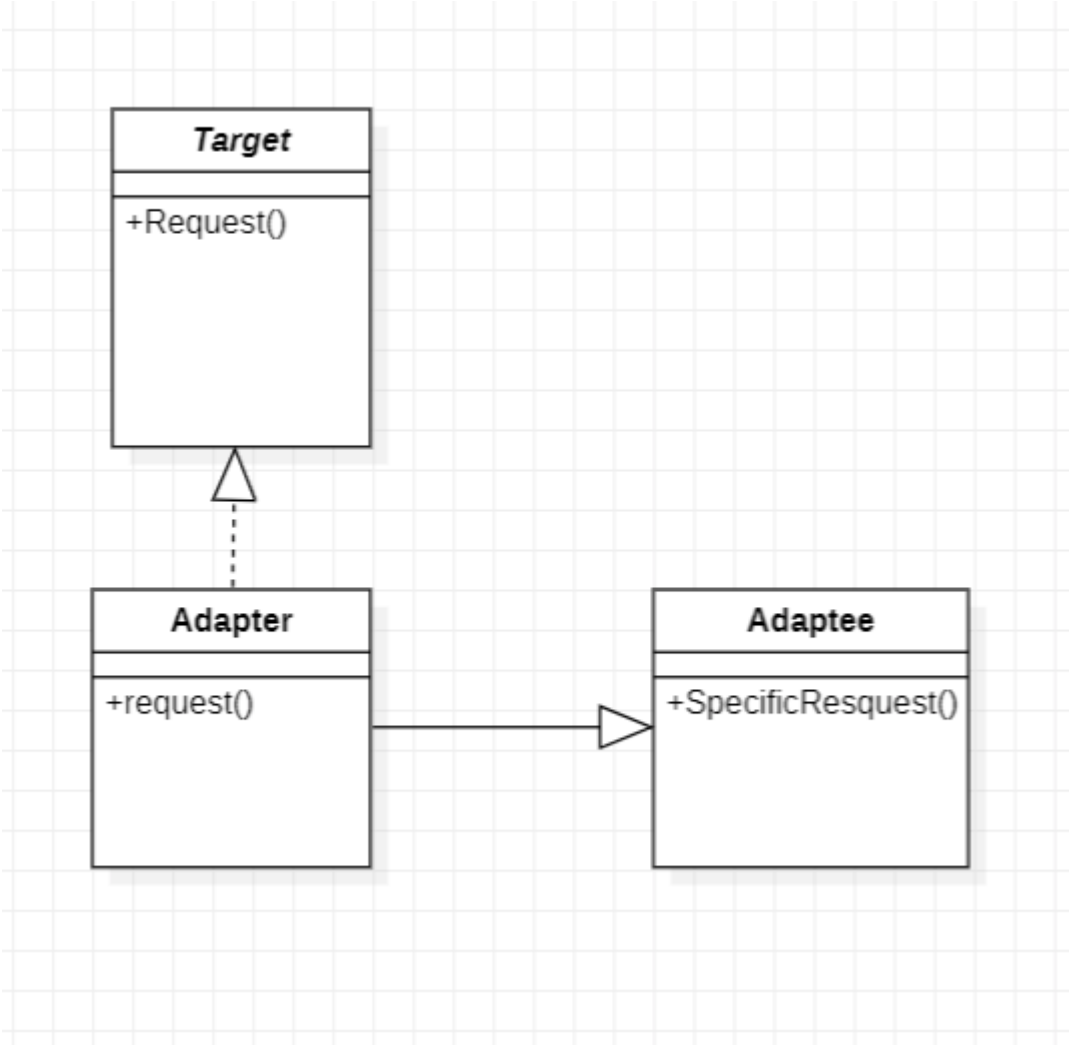
使用场景：

系统需要使用现有的类，而这些类的接口不符合系统的需要。

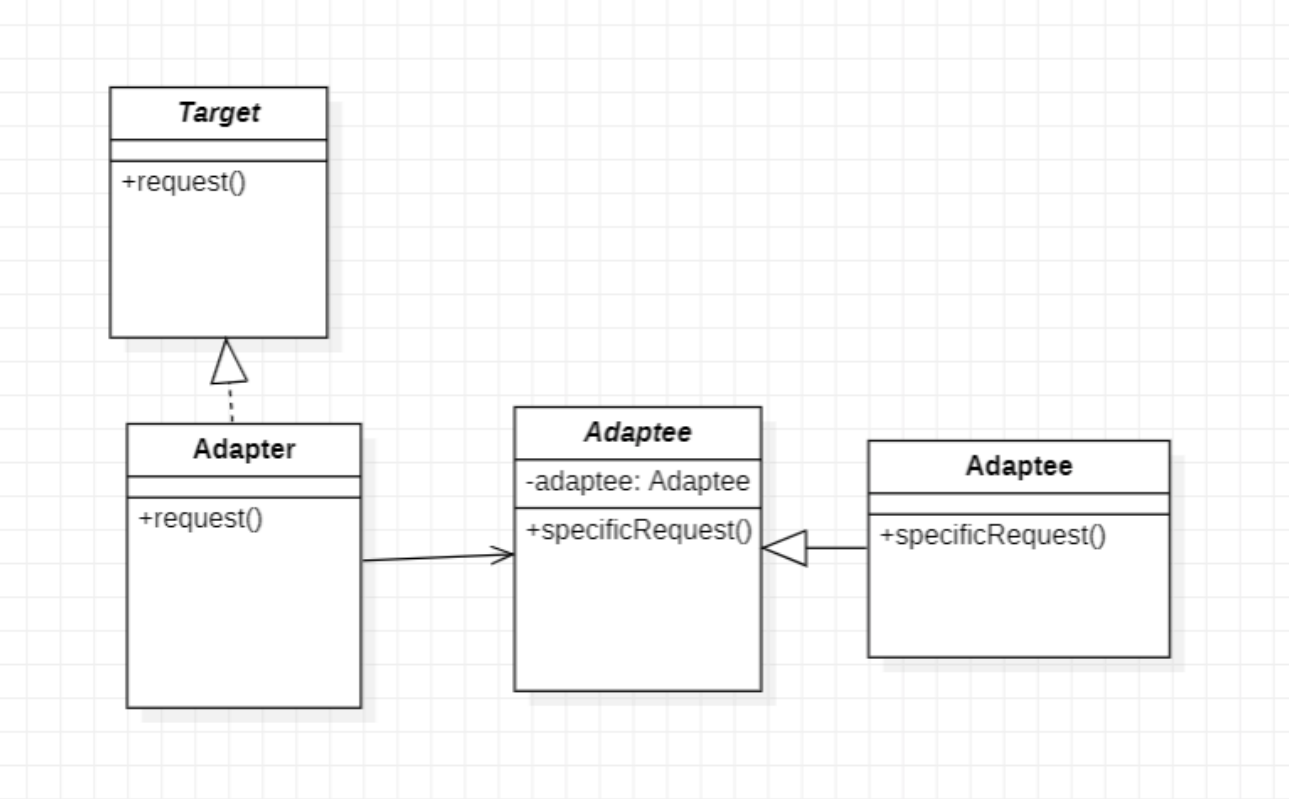
想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

2.1.2 UML图

类适配器


Target
+Request()
Adapter
+request()
Adaptee
+SpecificResquest()

对象适配器



2.1.3 代码例子

iPhone手机使用lightning接口

Android手机使用typec接口

让iPhone手机适配typec接口

```
//被适配类对象接口
public interface Lightning{
  public void recharge();
  public void useLightning();
}

public interface TypeC{
  public void recharge();
  public void useTypeC();
}
```

```
//被适配类
class iPhone implements Lightning{
  private boolean connector;

  @override
  public void useLightning(){
    connector = true;
    System.out.println("Lightning connected");
  }

  @override
  public void recharge(){
    if (connector){
      System.out.println("Recharge started");
      System.out.println("Recharge finished");
    }else{
      System.out.println("error");
    }
  }
}

class Android implements TypeC{
```

```
    private boolean connector;
    @override
  public void useTypeC(){
    connector = true;
    System.out.println("TypeC connected");
  }

    @override
  public void recharge(){
    if (connector){
      System.out.println("Recharge started");
      System.out.println("Recharge finished");
    }else{
      System.out.println("error");
    }
  }
}
```

```
//适配器类接口
public interface Target{
  public void Charge();
}
```

```
//适配器类
public ChargeApater implements Target{
  private Lightning lightningPhone;
  private TypeC typecPhone;

  public ChargeApater(Lightning lightningPhone){
    this.lightningPhone = lightningPhone;
```

```
  }
  public ChargeApater(TypeC typecPhone){
    this.typecPhone = typecPhone;
  }

  @override
  public void Charge(){
    if (lightningPhone == null ){
      typecPhone.useTypeC();
      typecPhone.userecharge();
    }else if(typecPhone == null){
      lightningPhone.useLightning();
      lightningPhone.recharge();
    }else{
      system.out.println("Error");
    }
  }
}
```

```
//client
public class client {
    public static void main(String[] args){
        iPhone iphone = new iPhone();
        Android vivo = new Android();

        ChargeApater charger1 = new ChargeApater(vivo);
        charger1.Charge();

        ChargeApater charger2 = new ChargeApater(iphone);
        charger2.Charge();

    }
}
```

2.2 装饰模式

2.2.1 分析

动机：

一般有两种方式可以实现给一个类或对象增加行为。一种是继承，使用继承给现有类添加功能，可以使子类拥有自身方法的同时还拥有父类方法。关联机制，将一个类嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为。一般把嵌入的对象称为装饰器。

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任，可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

定义：

装饰模式，动态地给一个对象增加一些额外的功能，同时又不改变其结构。

分析：

装饰模式有以下几个角色

- Component：抽象构件，定义一个抽象接口说明要添加的功能
- ConcreteComponent：具体构件
- Decorator：抽象装饰类，继承抽象构件，包含具体构件实例，可以通过子类扩展具体构件功能。
- ConcreteDecorator：具体装饰类，实现相关方法。

与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是一种耦合度较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是到了软件维护阶段，由于关联关系使系统具有较好的松耦合性，因此使得系统更加容易维护。

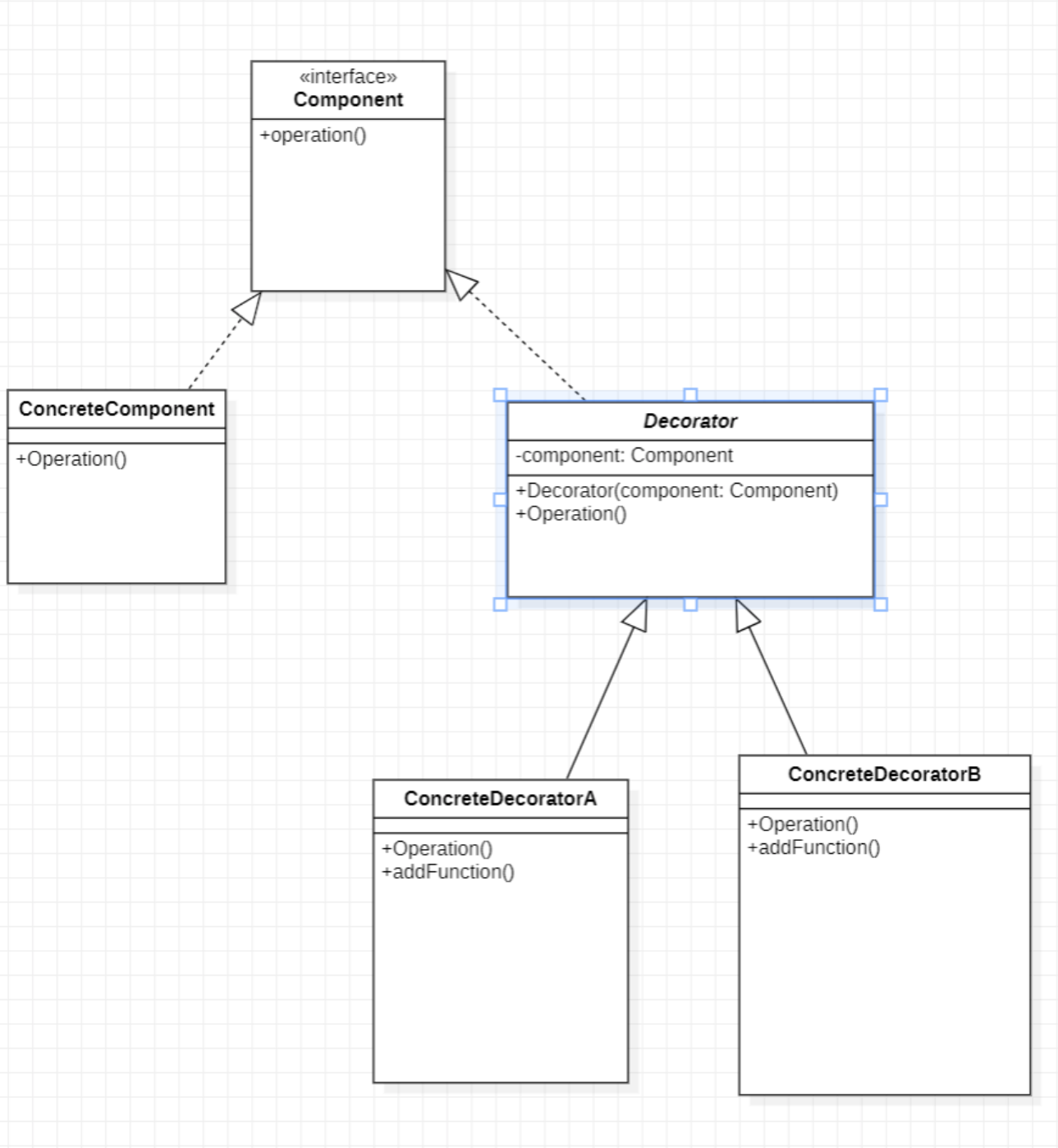
使用场景：

在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。

当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。

2.2.2 UML图



关键代码：

```
//Decorator
public void operation(){
  component.operation();
}
```

```
//ConcreteDecoration
public void operation(){
  super.operation();
  addFunction();
}
```

2.2.3 代码例子

```
//Component
public interface Shape{
  public void draw();
}
```

```
//ConcreteComponent
public class Rectangle implements Shape{
  @override
  public void draw(){
    System.out.println("Shape: Rectangle");
  }
}

public class Circle implements Shape{
  @override
  public void draw(){
    System.out.println("Shape: Circle");
  }
}
```

```
//Decorator
public abstract class ShapeDecorator implements Shape{
  protected Shape decoratedShape;

  public ShapeDecorator(Shape decoratedShape){
    this.decoratedShape = decoratedShape;
  }

  public void draw(){
    decoratedShape.draw();
  }
}
```

```
//ConcreteDecorator
public class RedShapeDecorator extends ShapeDecorator{
  public RedShapeDecorator(Shape decoratedShape){
    super(decoratedShape);
  }

  @override
  public void draw(){
    decoratedShape.draw();
    setRedBorder(decoratedShape);
  }

  public void setRedBorder(Shape decoratedShape){
    System.out.println("Border Color: Red");
  }
}
```

```
//Clinet

public class Client{
  public static void main(String[] args){

    Shape circle = new Circle();
    ShapeDecorator redCircle = new RedShapeDecorator(new
Circle());
    ShapeDecorator redRectangle = new RedShapeDecorator(new
Rectangle());
    circle.draw();
    //Shape: Circle
    redCircle.draw();
    //Shape: Circle
    //Border Color: Red
    redRectangle.draw();
    //Shape: Rectangle
    //Border Color: Red
  }
}
```

2.3 代理模式

2.3.1 分析

动机:

在某些情况下，一个客户不想或者不能直接引用一个对象，此时可以通过“代理”来实现间接引用。代理对象可以在客户端和目标对象之间起到中介的作用，并且可以通过代理对象去掉客户不能看到的内容和服务，或者添加客户需要的额外服务。通过引入一个新的对象来实现对真实对象的操作或者将新的对象作为真实对象的一个替身。

定义：

给某一个对象提供一个代理，并由代理对象控制对原对象的引用。

分析：

代理模式包含如下角色：

- Subject： 抽象主题角色，通过接口或者抽象类声明真实主题和代理对象实现的业务方法
- Proxy：代理主题角色，提供了与真实主题相同的方法，其内部含有对真实主题的引用，可以访问，控制或扩展真实主题
- RealSubject：真实主题角色，实现了抽象主题中的具体业务，是代理对象说代表的真实对象，是最终要引用的对象。

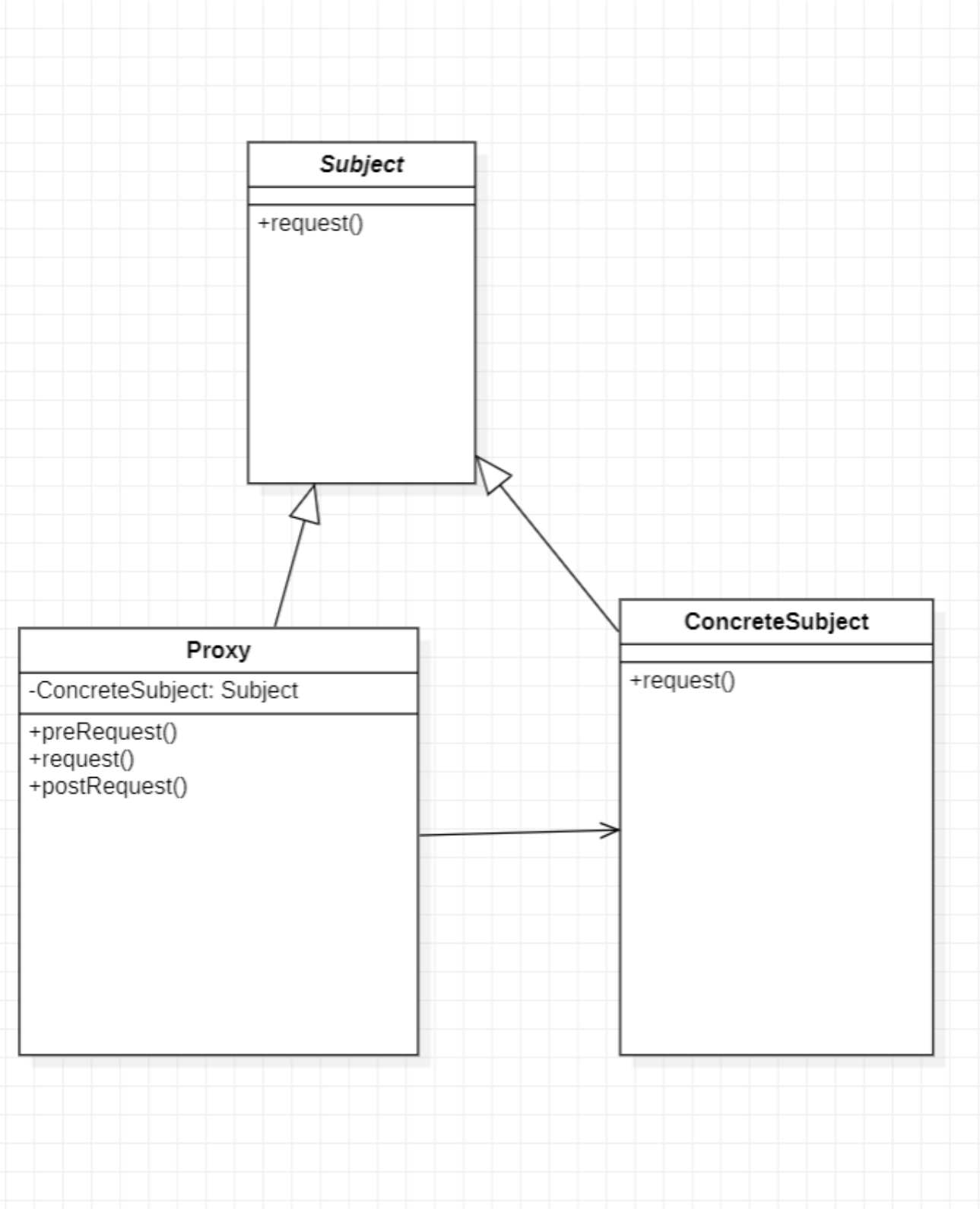
代理模式不同的形式，静态代理，动态代理

动态代理技术能事先定 义某方面的处理，在系统运行过程中针对某个特定业务类动态创建代理类，这也是面向切面编程（AOP）实现方式。JDK中的反射包中提供了Proxy类和InvocationHandler 接口来实现动态代理。

使用场景：

如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。

2.3.2 UML图



Subject类可以为接口或者抽象类

Proxy中的关键代码

```
preRequest();
subject.request();
postRequest();
```

2.3.3 代码例子

```
//Subject接口
public interface Image{
  void display();
}
```

```
//ConcreteSubject 类
public class RealImage implements Image{
  private String fileName;

  public RealImage(String fileName){
    this.fileName = fileName;
    loadFromDisk(fileName);
  }

  @override
  public void display(){
    System.out.println("Displaying " + fileName);


  }
```

```
  private void loadFromDisk(String fileName){
    System.out.println("Loading "+fileName);
  }
}
```

```
//Proxy 类
public class ProxyImage implements Image{
  private RealImage realImage;
  private String fileName;

  public ProxyImage(String fileName){
    this.fileName = fileName;
  }

  @override
  public void display(){
    if(realImage == null){
      realImage = new RealImage(fileName);
    }
    realImage.display();
  }
}
```

```
//client
public class Client{
  public static void main(String[] args){
    Image image = new ProxyImage("test.png");

    image.display();
    Sytem.display();
  }
}
```

3. 行为型模式

3.1 职责链模式

3.1.1 分析

动机:

未来避免请求发送者与多个请求处理者耦合在一起，于是将所有请求的处理者通过前一对象，记住其下一个对象的引用而连成一条链，当有请求发生时，可将请求沿着这条链传递到有对象处理它为止，而不必知道哪个接收者处理该请求。

定义:

职责链模式可以把响应请求的对象组成一条链，并在这条链上传递请求，从而保证多个对象都有机会处理请求并可以避免请求方和响应方的紧密耦合。

分析：

职责链上的角色：

Handle：抽象处理者，定义请求处理 方法，维护继任者引用

ConcreteHandler：对具体请求进行处理

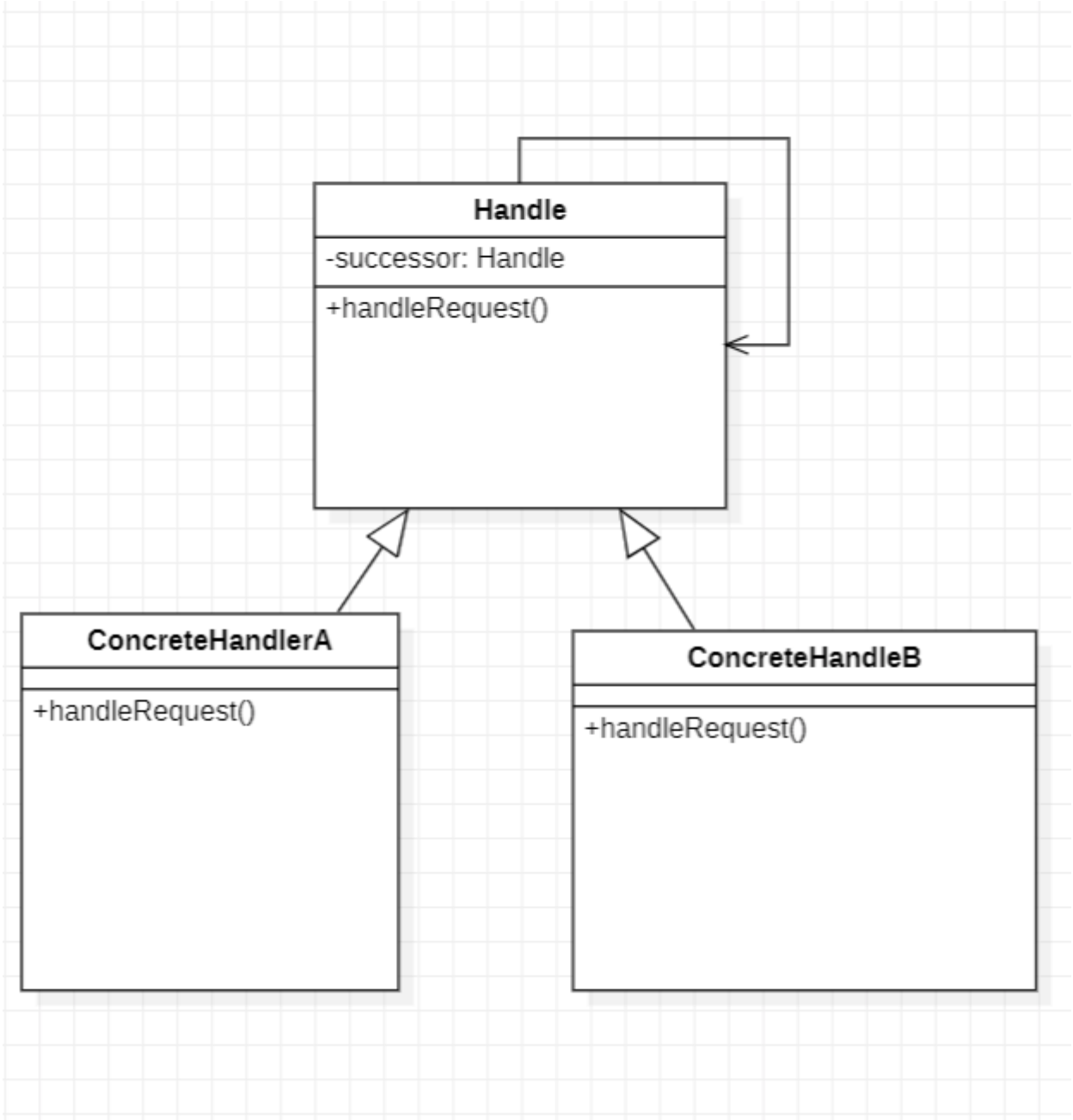
结构与装饰设计模式相似

使用场景：

应避免将请求的发送者与其接收者耦合。

应该有可能不止一个接收者可以处理一个请求。

3.1.2 UML图



3.1.3 代码实例

实验5例子:

```
//Handle
public abstract class CalcUnit {
    private CalcUnit next = null;
    public abstract boolean fit(String operator);
```

```
    public abstract double calc(double x, double y);
    public CalcUnit(CalcUnit next){
        this.next = next;
    }
    public double handle(String operator) throws Exception{
        if(fit(operator)){
            double x, y;
            Scanner input = new Scanner(System.in);
            System.out.print("arg1: ");
            x = input.nextDouble();
            System.out.print("arg2: ");
            y = input.nextDouble();
            return calc(x, y);
        }else if(next!=null){
            return next.handle(operator);
        }
        else{
            throw new Exception("没有找到合适的处理");
        }
    };

}
```

```
//ConcreteHandler
public class CalcPuls extends CalcUnit{
    public CalcPuls(CalcUnit next){
        super(next);
    }
    @Override
    public boolean fit(String operator) {
        return operator.equals("+");
    }
```

```
    @Override
    public double calc(double x, double y) {
        return x + y;
    }
}

public class CalcDIvide extends CalcUnit{

    public CalcDIvide(CalcUnit next) {
        super(next);
    }

    @Override
    public boolean fit(String operator) {
        return operator.equals("/");
    }

    @Override
    public double calc(double x, double y) {
        return x / y;
    }
}

public class CalcMinus extends CalcUnit{
    public CalcMinus(CalcUnit next) {
        super(next);
    }

    @Override
    public boolean fit(String operator) {
        return operator.equals("-");
    }


    @Override
```

```
    public double calc(double x, double y) {
        return x - y;
    }
}

public class CalcTimes extends CalcUnit{

    public CalcTimes(CalcUnit next) {
        super(next);
    }

    @Override
    public boolean fit(String operator) {
        return operator.equals("*");
    }

    @Override
    public double calc(double x, double y) {
        return x * y;
    }
}
```

```
//Client
public class Client {
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        CalcUnit calculator = new CalcPuls(new CalcMinus(new
CalcTimes(new CalcDIvide(null))));
        while(true) {
            System.out.print("输入: ");
            String commond = input.nextLine().trim();
            if(commond.equals("exit")) {
```

```
                    break;
                }
                try {
                    System.out.println("计算结果: " +
calculator.handle(commond));
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
            input.close();
        }
}
```

3.2 迭代器模式

3.2.1 分析

动机:

提供一种访问一个聚合对象的元素，而不需暴露该对象内部的表示，遍历整个对象。它支持以不同的方式遍历一个聚合对象。

定义:

迭代器模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

分析:

迭代器包含以下元素：

- Aggregate：抽象集合，定义创建迭代器的接口
- Iterator：抽象迭代器，定义访问和遍历元素的接口
- ConcreteIterator：实现访问和遍历元素的接口
- ConcreteAggregate：具体集合，实现迭代器接口

在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码

在同一个聚合上可以有多个遍历，简化了聚合类。

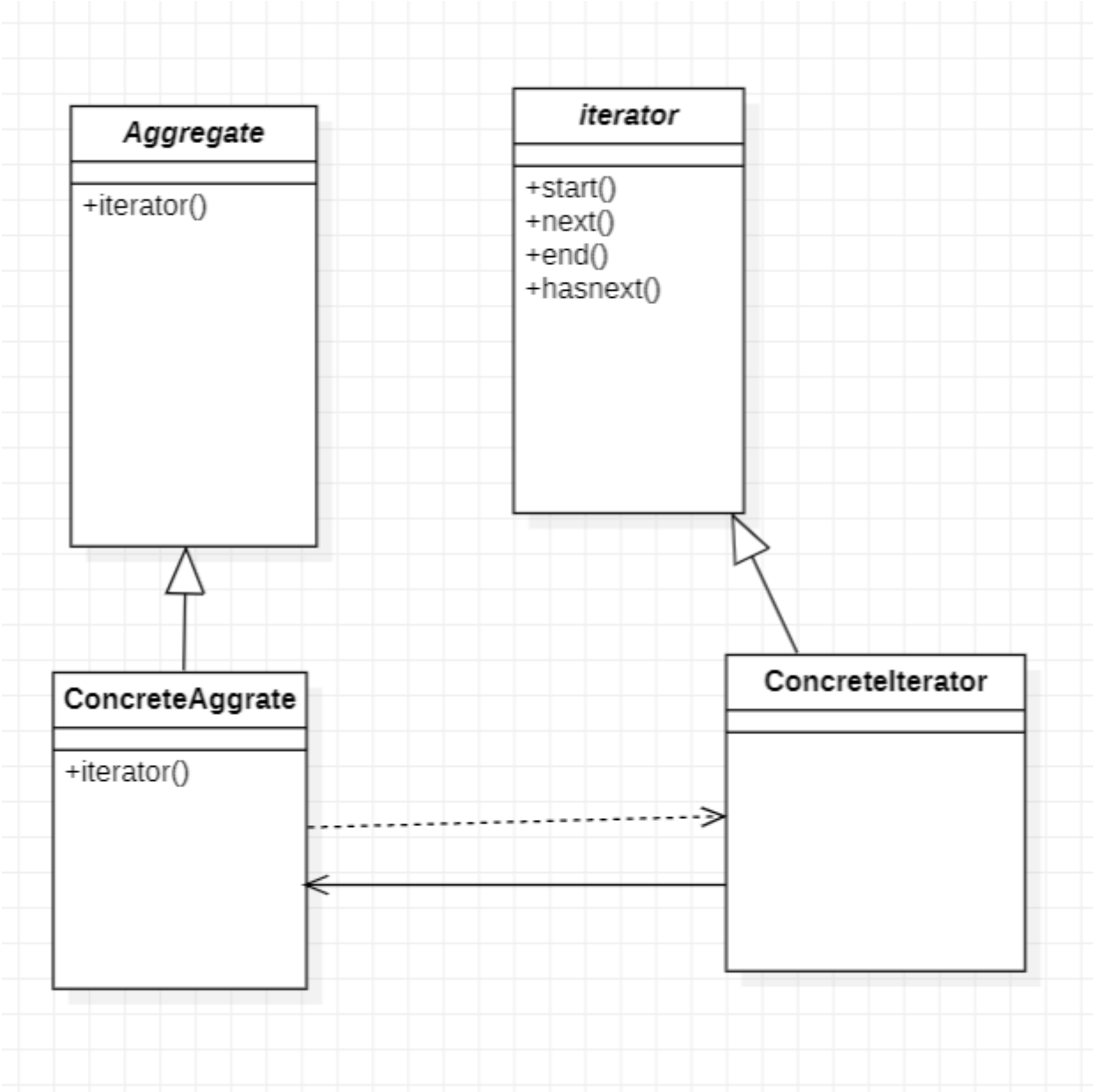
使用场景：

访问一个聚合对象的内容而无须暴露它的内部表示。

需要为聚合对象提供多种遍历方式。

为遍历不同的聚合结构提供一个统一的接口。

3.2.2 UML图



3.2.3 代码实例

例子为实验5

```java
//Aggregate
public interface Iterable<T> {
    public abstract Iterator<T> iterator();
}
```

```java
//iterator
public interface Iterator<T> {
    public abstract Object next();
    public abstract boolean hasNext();
}
```

```java
//ConcreteAggregate
public class IntegerList implements Iterable{
    private List<Integer> list = new ArrayList<>();

    public void add(Integer e){
        list.add(e);
    }
    protected int size(){
        return list.size();
    }
    protected Integer get(int index){
        return list.get(index);
    }

    @Override
    public Iterator iterator() {
```

```
        return new IteratorA(this);
    }

}
```

```
//ConcreteIterator
public class IteratorA implements Iterator{
    private IntegerList e;
    private int index = 0;

    public IteratorA(IntegerList integerList) {
        e = integerList;
    }

    @Override
    public Object next() {
        if(hasNext()){
            return e.get(index++);
        }else{
            throw new RuntimeException("");
        }
    }

    @Override
    public boolean hasNext() {
        return index < e.size()? true:false;
    }
}
```

```
//Client
public class Test {
    public static void main(String[] args){
        IntegerList list = new IntegerList();
        list.add(20);
        list.add(30);
        list.add(40);
        list.add(50);
        Iterator<Integer> iterator = list.iterator();
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }

    }
}
```

3.3 观察者模式

3.3.1 分析

动机:

建立一种对象与对象之间的依赖关系，一个对象发生改变时自动通知其它对象，其他对象将相应做出反应。发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统易于扩展。

定义:

观察者模式定义对象间的一种一对多依赖关系，使得每个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。

分析：

- Subject：被观察者，知道它的通知对象，事件发生后会通知所有它知道的对象，提供添加删除观察者的接口。
- Observer：提供通知后的更新事件
- ConcreteSubject：被观察者具体的实例，储存观察者感兴趣的状态
- ConcreteObserver：观察者的具体实现

观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。这一模式中的关键是观察目标和观察者，一个目标可以有任意数目的与之相互依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通

使用场景：

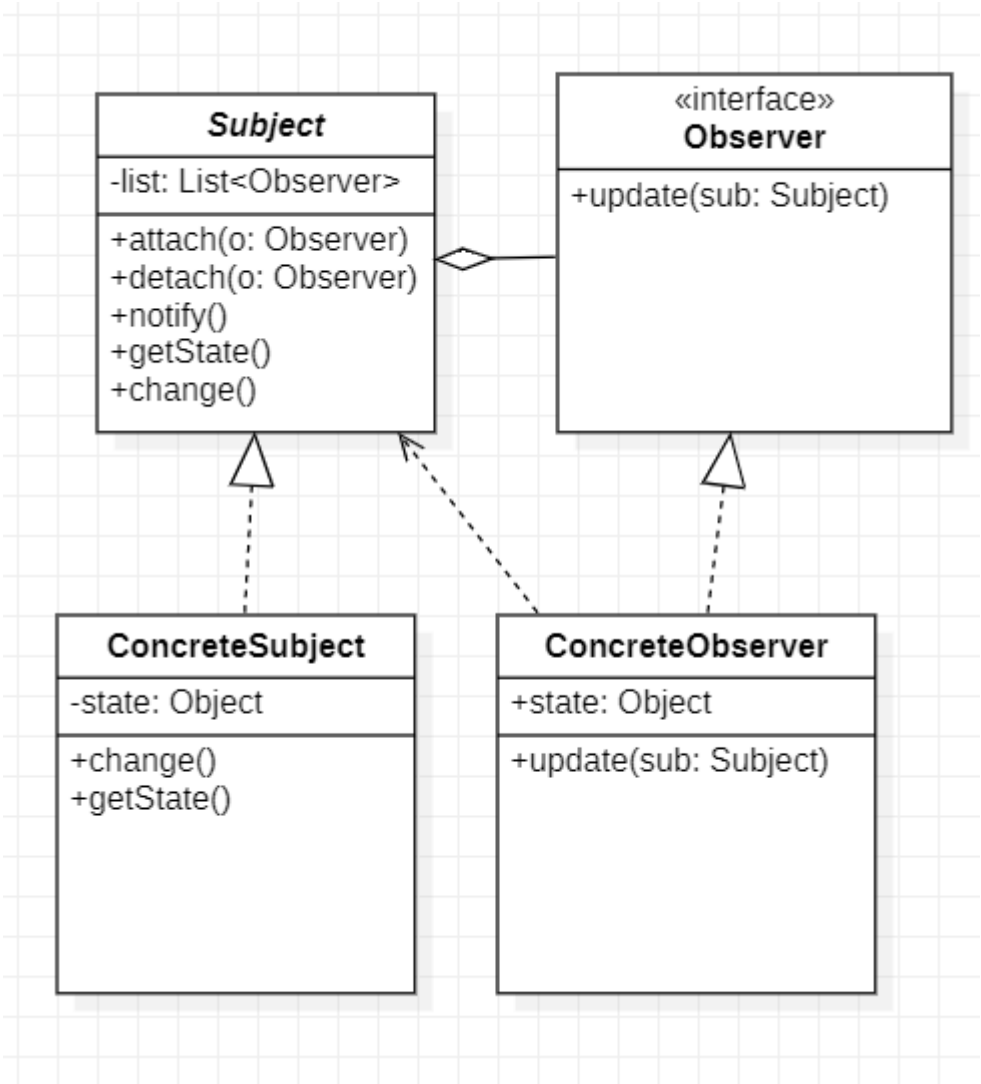
一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。

一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。

一个对象必须通知其他对象，而并不知道这些对象是谁。

需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象...可以使用观察者模式创建一种链式触发机制。

3.3.2 UML图



3.3.3 代码例子

例子为实验7

```
//Subject 类
public abstract class SensorSubject {

    private List<SensorObservers> list = new
```

```
ArrayList<SensorObservers>();
    public void addObserver(SensorObservers observers){
        list.add(observers);
    }
    public void deleteObserver(SensorObservers observers){
        if(list.contains(observers))
            list.remove(observers);
    }
    public List<SensorObservers> getObservers(){
        return list;
    }
    public abstract void notifyObservers();
    public abstract double getState();
    public abstract void change(double randomNumber);
}
```

```
//Observer接口
public interface SensorObservers {
    public void update(SensorSubject sensorSubject);
    public void StatePrint();
}
```

```
//ConcreteSubject
public class Sensor extends SensorSubject{

    private double state;
    @Override
    public void notifyObservers() {
        for(SensorObservers o: getObservers()){
```

```
            o.update(this);
        }
    }

    @Override
    public double getState() {
        return state;
    }
    public void setState(double state){
        this.state = state;
    }

    @Override
    public void change(double randomNumber) {
        this.state = randomNumber;
        notifyObservers();
        if(state >= 50){
            for(SensorObservers o : getObservers()){
                o.StatePrint();
            }
        }
    }
}
```

```
//ConcreteObserver
public class Alarm implements SensorObservers{
    private double state;
    @Override
    public void update(SensorSubject sensorSubject) {
        setState(sensorSubject.getState());
    }
    public double getState(){
        return state;
```

```
    }
    public void setState(double state){
        this.state = state;
    }
    public void StatePrint(){
        System.out.println("报警器响了");
    }
}


public class Lamp implements SensorObservers{
    private double state;
    @Override
    public void update(SensorSubject sensorSubject) {
        setState(sensorSubject.getState());
    }
    public double getState(){
        return state;
    }
    public void setState(double state){
        this.state = state;
    }
    public void StatePrint(){
        System.out.println("报警灯响了");
    }
}
```

```
//Client
public class Test {
    public static void main(String[] args){
        Sensor sensor = new Sensor();
        sensor.addObserver(new Lamp());
        sensor.addObserver(new Alarm());
```

```
        //模拟N次环境变化
        for(int i = 0;i < 10; i++){
            System.out.printf("第%d次温度变化\n", i + 1);
            double temp = Math.random() * 100;
            sensor.change(temp);
        }
    }
}
```

3.4 状态模式

3.4.1 分析

动机:

在很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的对象，这样的对象状态是事先定义号的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会发生改变，从而使得系统的行为也随之发生改变。

定义:

状态模式允许一个对象在其内部状态发生改变时改变它的行为，对象看起来似乎修改了。

分析:

状态模式包含以下角色：

- Context：环境类，保持并切换各个状态，它将依赖状态的各种操作委托给不同的状态对象执行。
- State：抽象状态类，封装了状态以及行为。
- ConcreteState：具体状态类

状态模式描述了对象状态的变化以及对象如何在每一种状态下表现出不同的行为。状态模式的关键是引入了一个抽象类来专门表示对象的状态，这个类我们叫做抽象状态类，而对象的每一种具体状态类都继承了该类，并在不同具体状态类中实现了不同状态的行为，包括各种状态之间的转换。

环境类实际上就是拥有状态的对象，环境类有时候可以充当状态管理器的角色，可以在环境类中对状态进行切换操作。

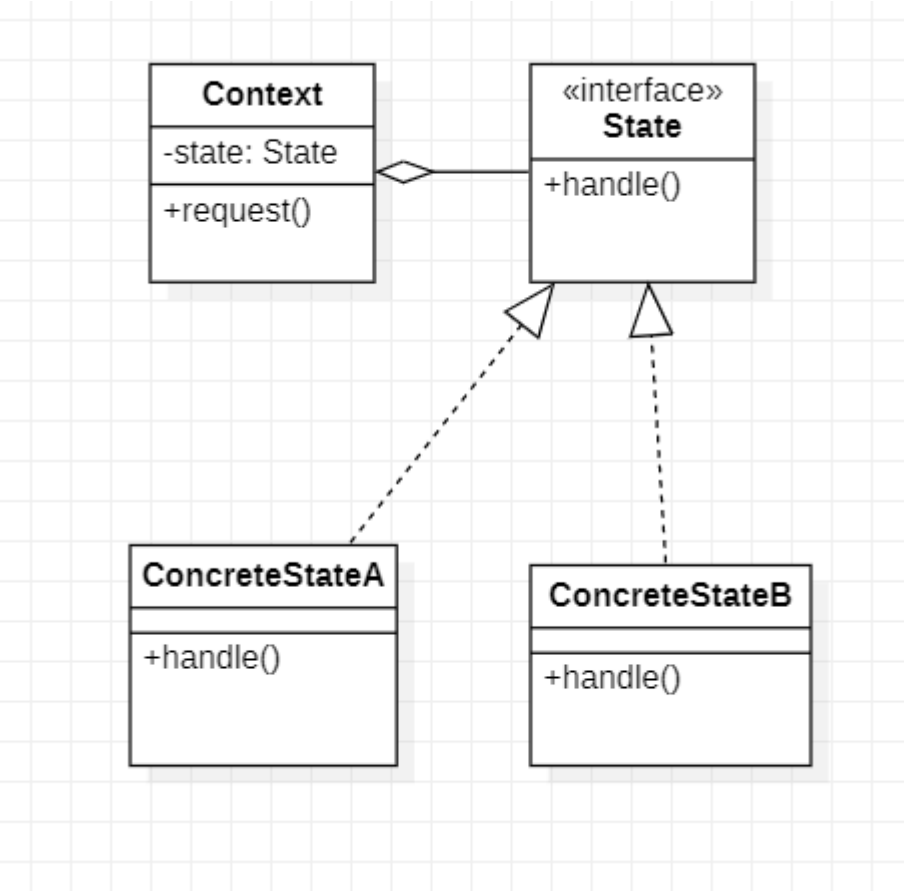
抽象状态类可以是抽象类，也可以是接口，不同状态类就是继承这个父类的不同子类，状态类的产生是由于环境类存在多个状态，同时还满足两个条件： 这些状态经常需要切换，在不同的状态下对象的行为不同。因此可以将不同对象下的行为单独提取出来封装在具体的状态类中，使得环境类对象在其内部状态改变时可以改变它的行为，对象看起来似乎修改了它的类，而实际上是由于切换到不同的具体状态类实现的。由于环境类可以设置为任一具体状态类，因此它针对抽象状态类进行编程，在程序运行时可以将任一具体状态类的对象设置到环境类中，从而使得环境类可以改变内部状态，并且改变行为。

使用场景：

对象的行为依赖于它的状态并且可以根据它的状态改变而改变它的相关行为。

代码中包含大量与对象状态有关的条件语句，这些条件语句的出现，会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，使客户类与类库之间的耦合增强。在这些条件语句中包含了对象的行为，而且这些条件对应于对象的各种状态

3.4.2 UML图



关键代码：

```
//Context中
state.handle();
```

3.4.3 代码例子

```
//State 接口
public interface State{
  public void handle(Context context)
}
```

```
//ConcreteState
public class StartState implements State{
  public void handle(Context context){
    System.out.println("Player is in start state");
    context.setState(this);
  }

  public String toString(){
    return "Start State";
  }
}

public class StopState implements State{
  public void handle(Context context){
    System.out.println("Player is in stop state");
    context.setState(this);
  }
}
```

```
public class Context{

  private State state;
```

```
  public Context(){
    state = null;
  }

  public void setState(State state){
    this.state = state;
  }

  public State getState(){
    return state;
  }
}
```

```
//Client
public class Client{
  public static void main(String[] args){
    Context context = new Context();

    StartState startState = new StartState();
    startState.handle(context);

    System.out.println(context.getState().toString());

    StopState stopState = new StopState();
    stopState.handle(context);

    System.out.println(context.getState().toString());
  }
}
```