# 常见设计模式

2020/11/19

# 什么是设计模式

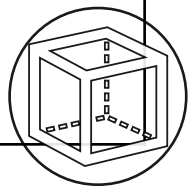
# 什么是设计模式

模式：在某一背景下对某个问题的一种解决方案

设计模式：对被用来在特定场景下解决一般设计问题的类和相互通讯的对象的描述

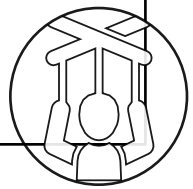
- 每一个设计模式确定了所包含的类和对象、它们的角色和协作方式以及职责分配

在构成上



- 每一个设计模式都针对一个特定的面向对象设计问题或设计要点，描述了约束条件、使用时机和使用效果等

在使用上



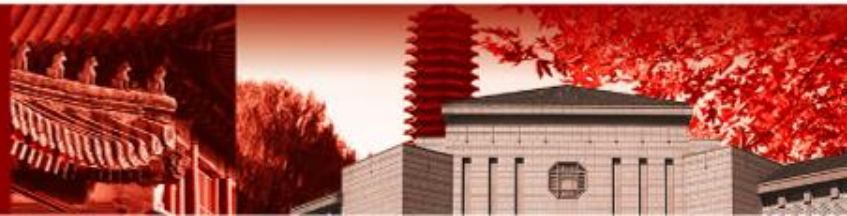
# 设计模式的作用？

更好、更快地完成系统设计

有利于软件的复用、维护和扩展

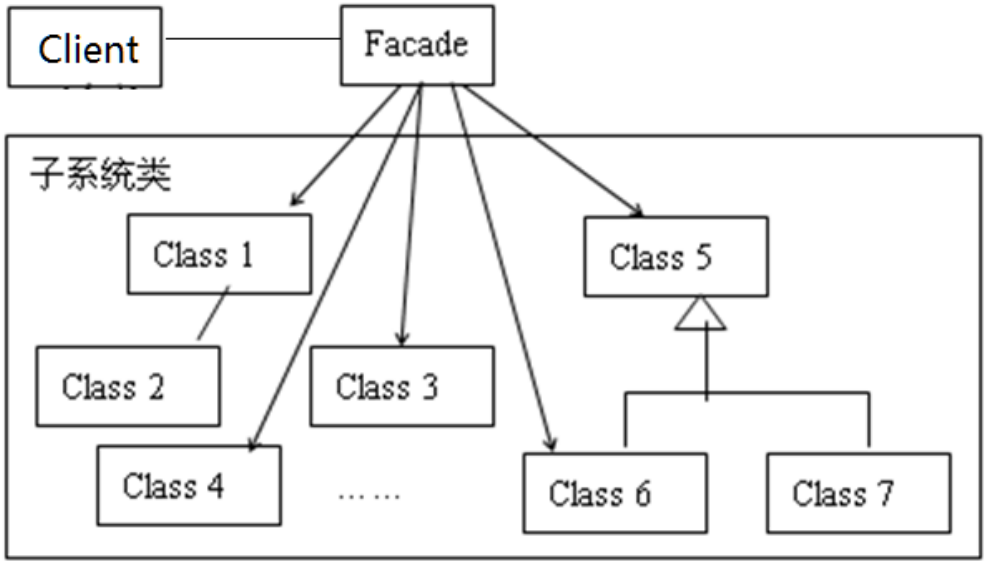
通过对设计模式的掌握，可加深对面向对象思想的理解





# 1、外观模式

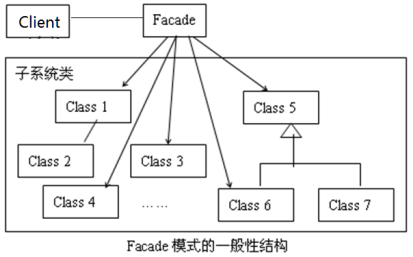
# 概述




Facade 模式的一般性结构

为子系统中的一组对象提供统一接口，以使得系统更易于使用
或者是，在原系统的基础上，再在高层定义接口，通过这样的接口只展示出特定（部分）的功能，隐藏原系统复杂性

# “外观”的思想



Facade 模式的一般性结构

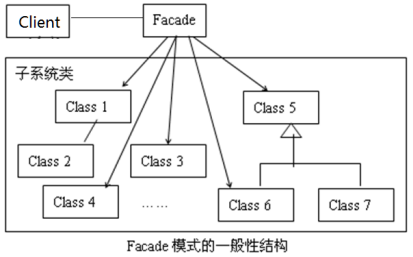
一位路过的买买买人


网购APP：系统的“外观”

网购系统

使用系统的客户端不与系统耦合，而外观类与系统耦合

# 如果不用外观……



Facade 模式的一般性结构

系统的IP地址是……
我的用户ID是……
要买的商品ID列表是……
我的身份验证token是……
……
还没发出请求东西已经抢光了（

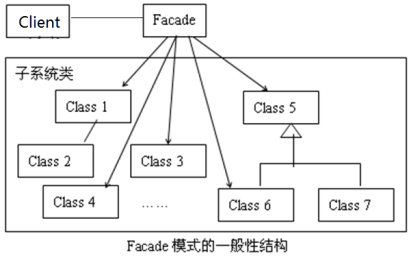
一位路过的买买买人


我的心凉凉了

一位悲伤的买买买人

网购系统

# 如果系统做出了变化……

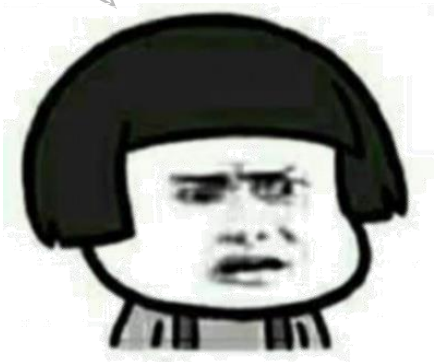


Facade 模式的一般性结构

别走啊，新的IP地址是多少啊（

我们把服务器搬迁了，所以你得往新的IP地址发购买请求才行hhh

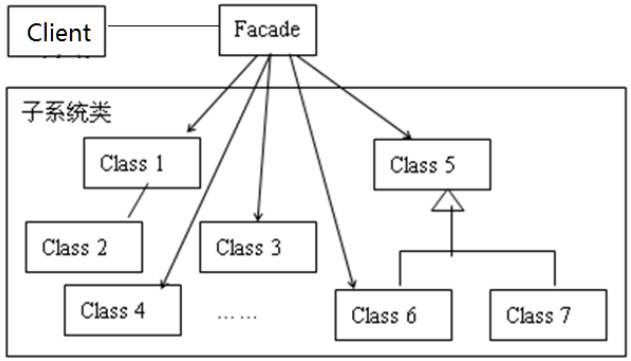

一位路过的买买买人



一位和系统耦合过度的买买买人


网购系统

# 外观模式的典型应用



Facade 模式的一般性结构

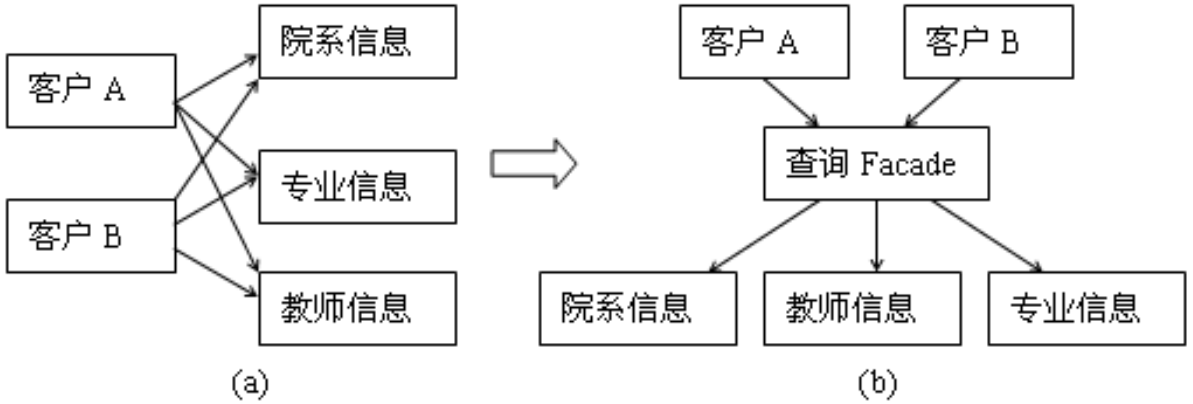


图 14－3 Facade 模式应用示例

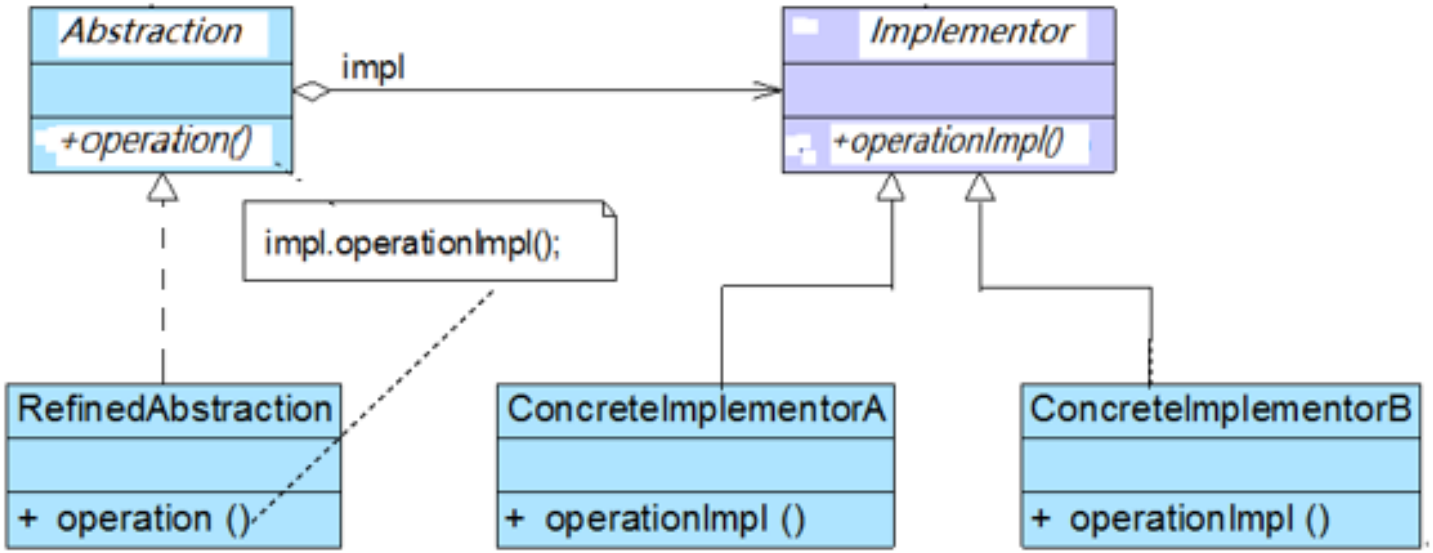
为原系统增加新的功能，通过新增接口的方式体现出来，而不改动系统已有的接口

对遗产系统进行包装，通过提供新的接口，以新的面貌出现

为了有助于监控系统，使所有的访问要通过新增的接口

# 2、桥接模式

# 概述




**将抽象部分与它的实现部分分离，使它们都可以独立地变化**

# 是否要为所有的泛化关系无脑使用继承？

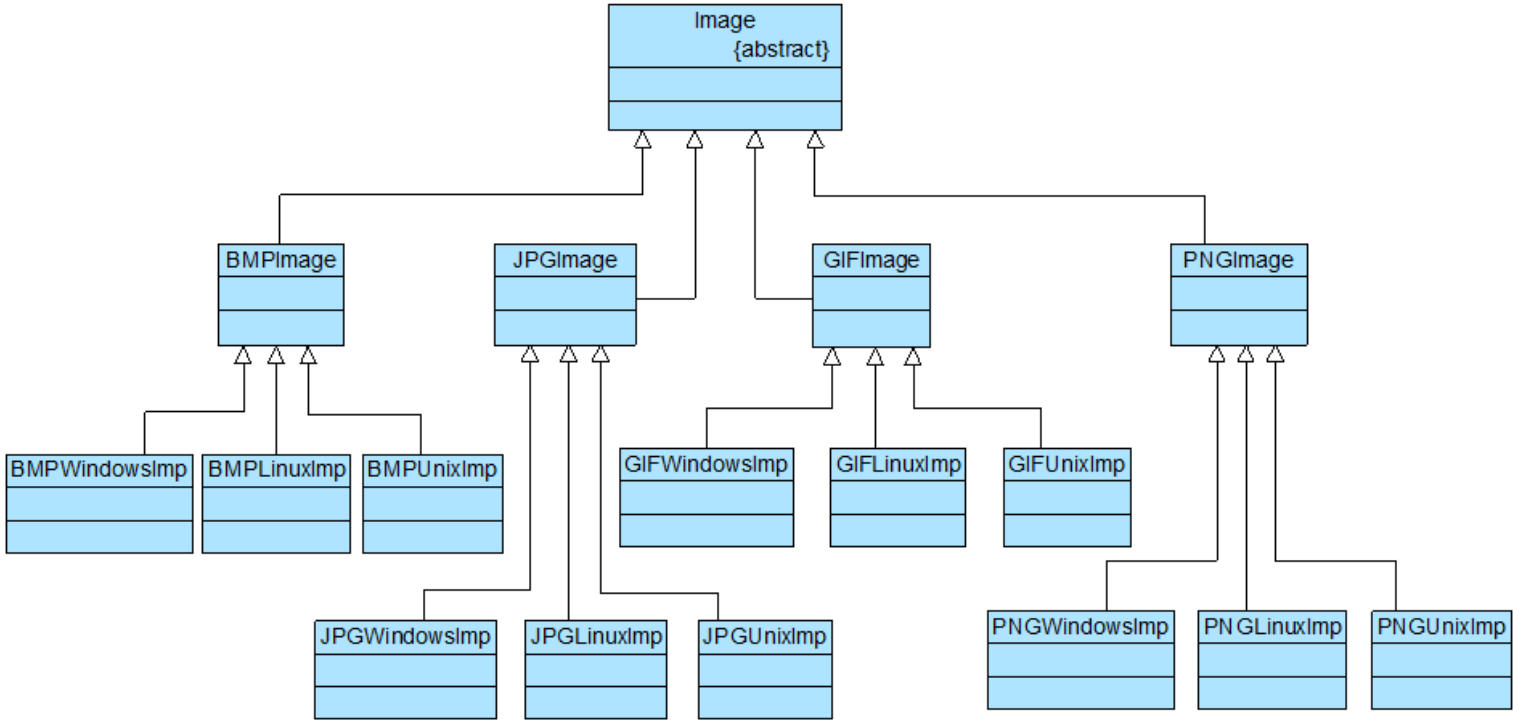



图 10-1 跨平台图像浏览器系统初始结构图

- 如果存在多个继承维度：类的数量爆炸式增加

# 桥接模式——善用组合关系，减少继承负担

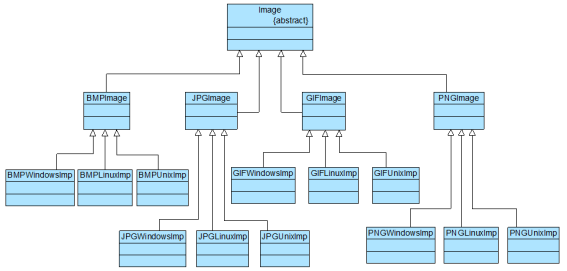
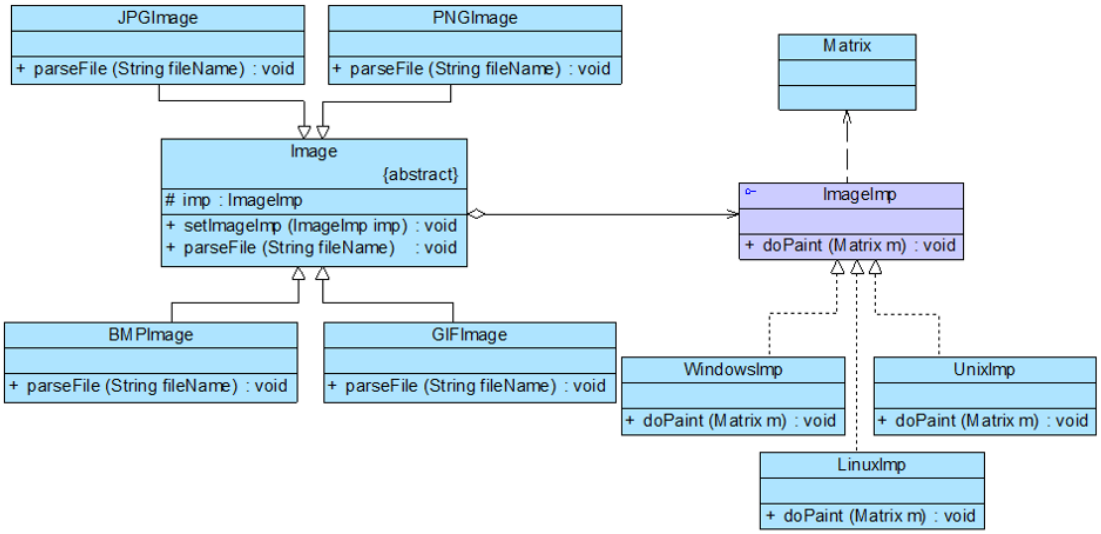



图 10-1 跨平台图像浏览器系统初始结构图

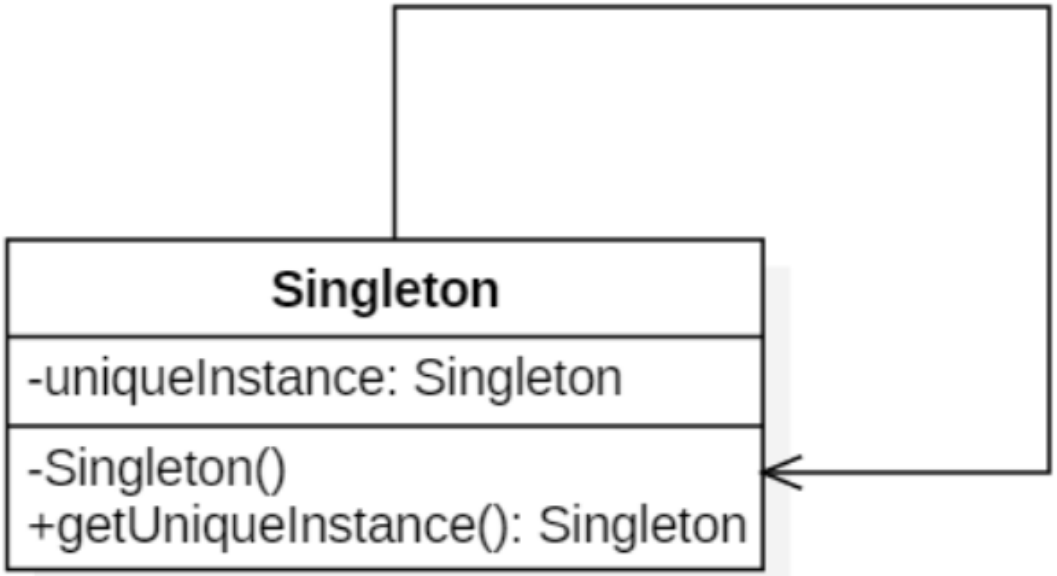



跨平台图像浏览系统结构图

- 识别出一个类所具有的两个独立变化的维度，将它们设计为两个继承等级结构，并建立抽象耦合
- 在程序运行时再动态确定两个维度的子类，动态组合对象

# 1、单例模式

# 概要




指确保一个类只有一个实例，并提供该实例的全局访问点

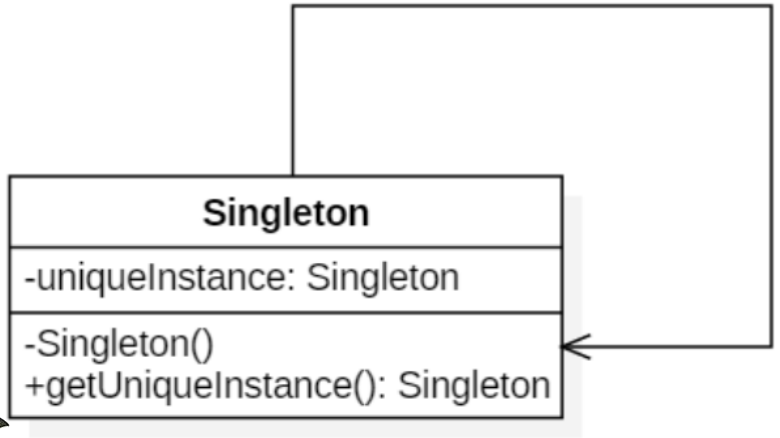
# 概要

单例有什么用?

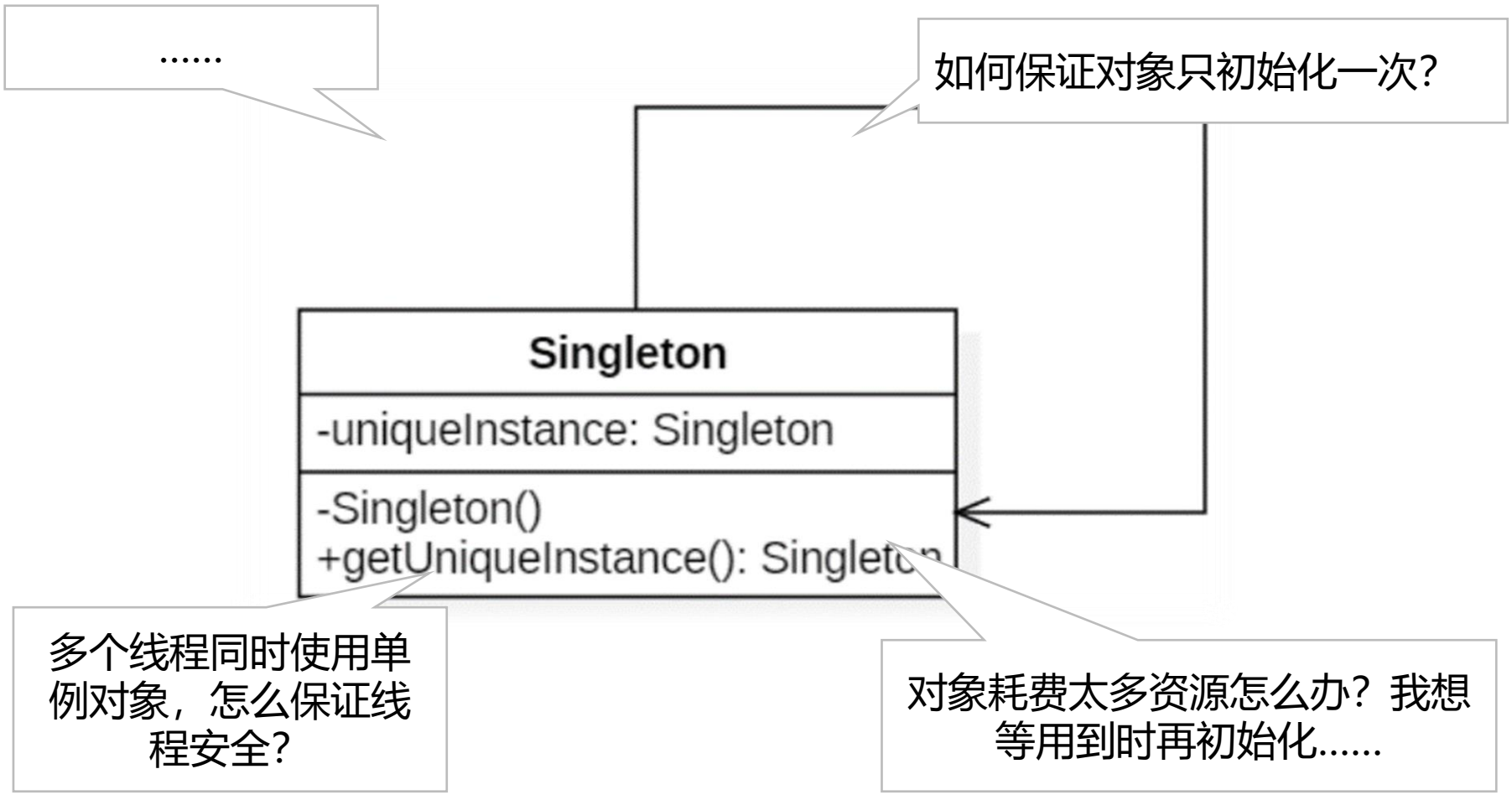
有些对象你只需要一个：线程池、数据库连接池、日志对象、全局计数器……这种对象你搞了很多反而有问题

？那我不能就整个全局变量么

小火鸡，现实比你想象的复杂……



# 概要


……
如何保证对象只初始化一次?
Singleton
-uniqueInstance: Singleton
-Singleton()
+getUniqueInstance(): Singleton
多个线程同时使用单例对象，怎么保证线程安全?
对象耗费太多资源怎么办？我想等用到时再初始化……

# 实现一：懒汉式、线程不安全

对象耗费太多资源怎么办？我想等用到时再初始化……

- 单例对象被延迟实例化
  - 如果没有用到该类，那么单例对象就不会实例化，从而节约资源。
- 这个实现在多线程环境下是不安全的，可能导致单例实例化多次

```
public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

# 实现二：饿汉式、线程安全

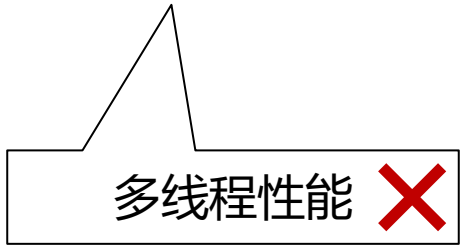
保证对象只初始化一次 ✓

对象耗费太多资源 ✗

- 直接在系统启动时就初始化单例
- 杜绝了多次实例化问题
- 但是可能带来较大的启动开销

```
private static Singleton uniqueInstance = new Singleton();
```

# 实现三：懒汉式、线程安全




- 对获取单例对象的方法加锁
- 避免了多次实例化问题，但是试图获取单例对象的线程会阻塞，造成性能问题

```
public static synchronized Singleton getUniqueInstance() {
    if (uniqueInstance == null) {
        uniqueInstance = new Singleton();
    }
    return uniqueInstance;
}
```

# 实现四：双重校验锁、线程安全

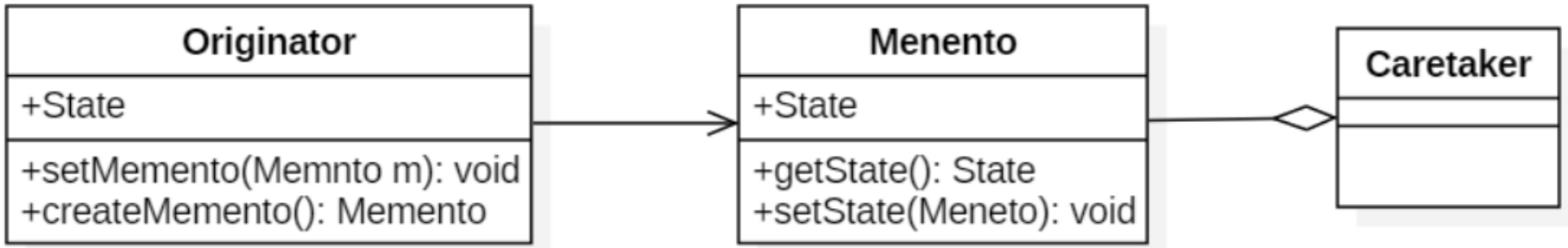
- 先判断单例是否已实例化，如果没有才对实例化语句进行加锁
- 是一种比较好的解决方案

```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

# 2、备忘录模式

又名：快照模式

# 概述





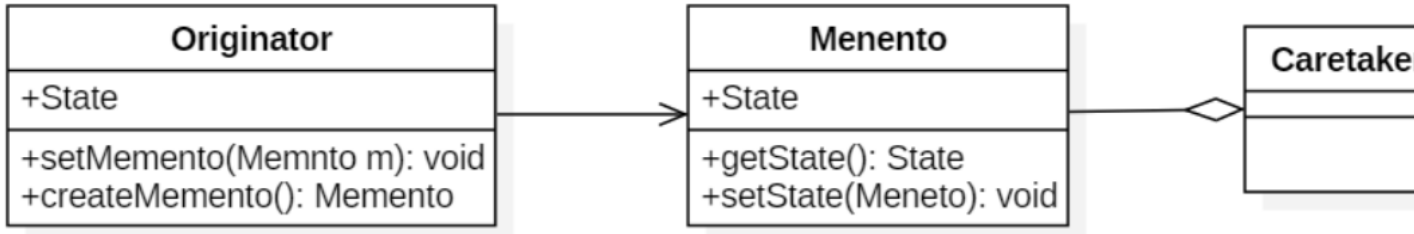
捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态

# 概述

每日一问：备忘录模式又有啥用呢？

哦！这还真就是个存档呗！

备忘录模式在实现很多应用场景时使用：如浏览器等的访问回溯、数据库的备份与还原、文本编辑器的撤销、虚拟机快照与恢复、棋牌游戏系统悔棋……

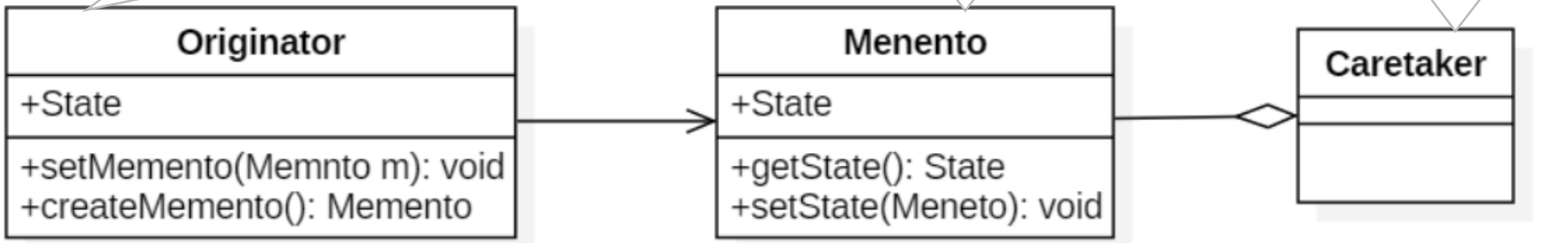


# Take a closer look…

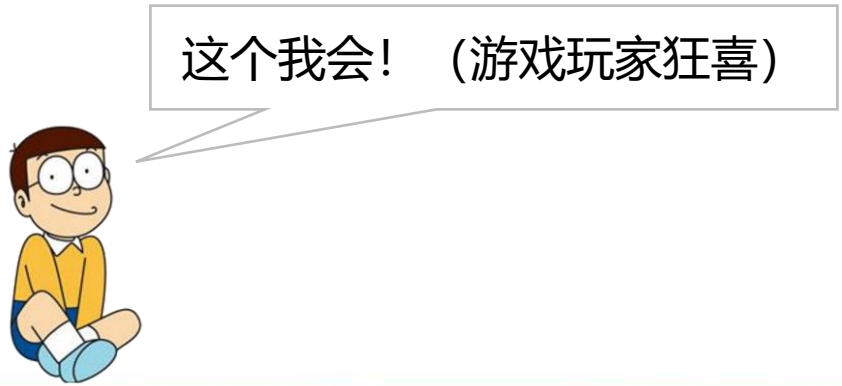
Originator：被保存的对象
想象你的游戏……

Menento：备忘录
想象你的存档……

Caretaker：接管备忘录
想象你的存档槽……







# 示例：支持访问历史回溯的简单计算器

每次计算有两个操作数，你可以把这个记下来，形成一个menento（备忘录），交给一个caretaker对象看管，这样你想知道你之前算了个什么的时候就可以回溯了

你要实现一个计算器，能进行简单四则运算，就像上面这种

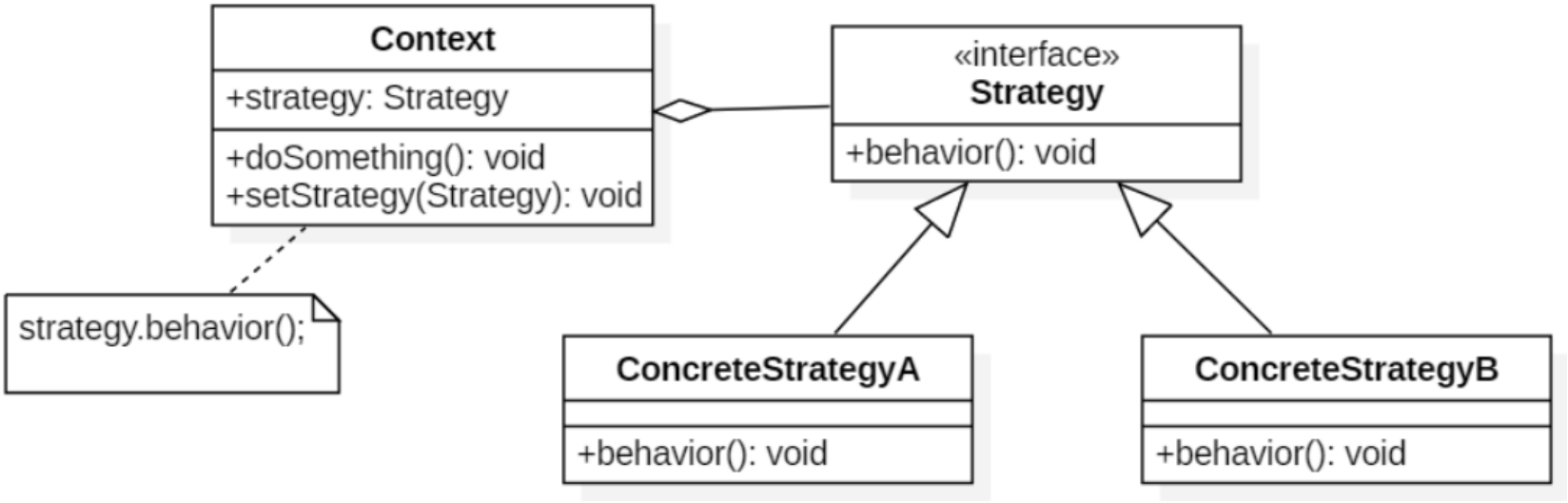
# 示例：支持访问历史回溯的简单计算器

```java
public class CalculatorImp implements Calculator {

    private int firstNumber;
    private int secondNumber;

    @Override
    public PreviousCalculationToCareTaker backupLastCalculation() {
        // create a memento object used for restoring two numbers
        return new PreviousCalculationImp(firstNumber, secondNumber);
    }

    @Override
    public void restorePreviousCalculation(PreviousCalculationToCareTaker memento) {
        this.firstNumber = ((PreviousCalculationToOriginator) memento).getFirstNumber();
        this.secondNumber = ((PreviousCalculationToOriginator)
memento).getSecondNumber();
    }

    @Override
    public int getCalculationResult() {
        // result is adding two numbers
        return firstNumber + secondNumber;
    }

    @Override
    public void setFirstNumber(int firstNumber) {
        this.firstNumber = firstNumber;
    }

    @Override
    public void setSecondNumber(int secondNumber) {
        this.secondNumber = secondNumber;
    }
}
```

```java
public class PreviousCalculationImp implements PreviousCalculationToCareTaker,
        PreviousCalculationToOriginator {

    private int firstNumber;
    private int secondNumber;

    public PreviousCalculationImp(int firstNumber, int secondNumber) {
        this.firstNumber = firstNumber;
        this.secondNumber = secondNumber;
    }

    @Override
    public int getFirstNumber() {
        return firstNumber;
    }

    @Override
    public int getSecondNumber() {
        return secondNumber;
    }
}
```

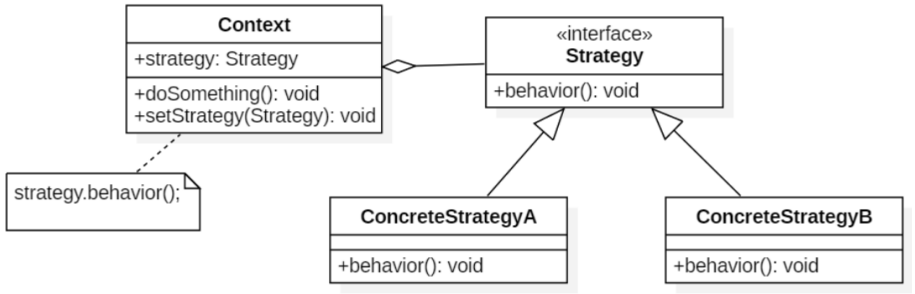
# 3、策略模式

# 概述




策略模式定义一系列算法，封装每个算法，并使它们可以互换；策略模式可以让算法独立于使用它的客户端。

# 概述




- 想象你是一家具有创新力的公司，创新到每个月都要开一场发布会
- 因为很麻烦，所以你想也许你可以写一个系统来自动帮你开发布会（怎么会想出这种主意的？？？）

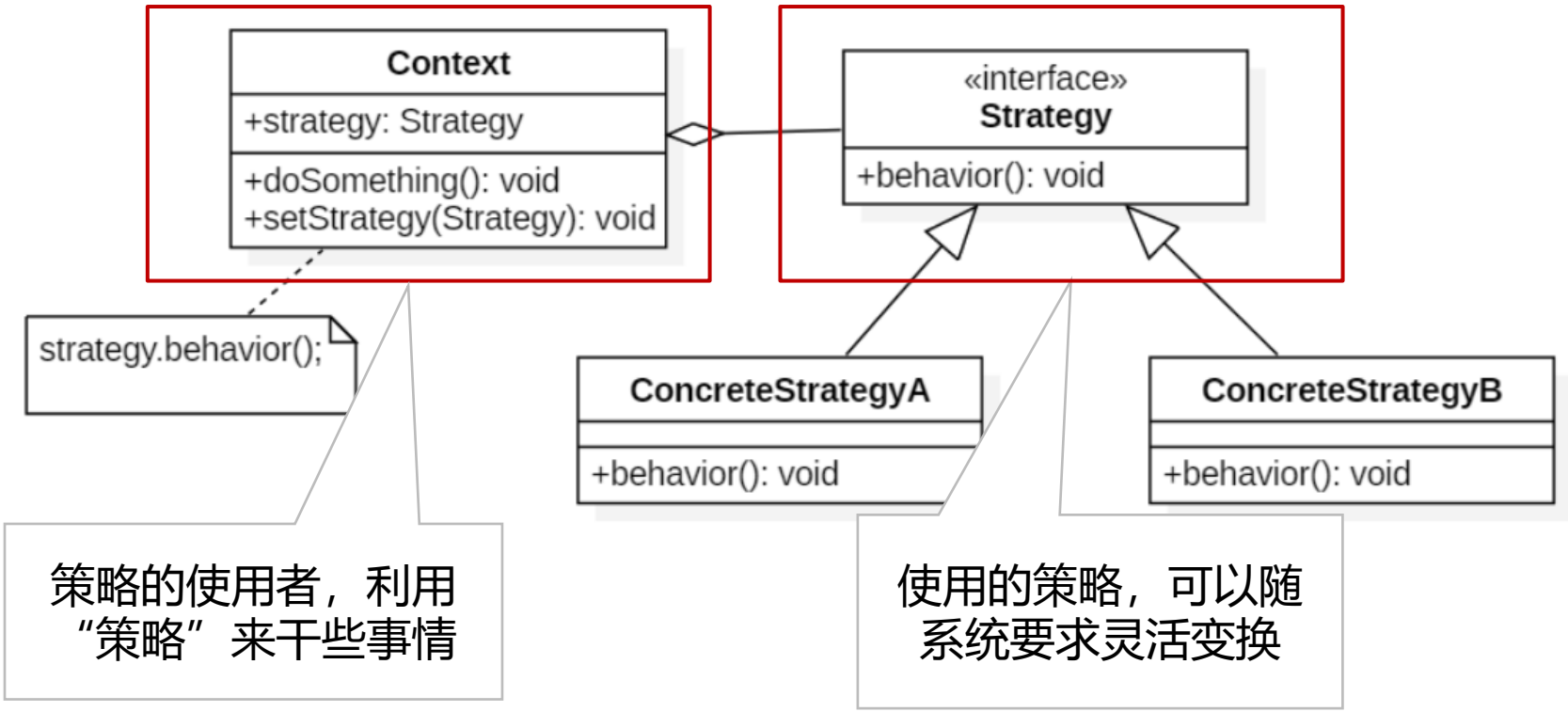
# 如果你的系统没那么灵活……


2014年


2019年

# 所以也许你需要时不时整点活……

# 言归正传




- 优势：低耦合！策略可以自由切换，系统扩展性、灵活性更高
- 如果不使用策略模式，则难免会造成对原代码的反复修改……

# 示例：发放奖励的不同策略

- 思路
  - 声明发放奖励的接口（Strategy）
  - 实现不同的发奖策略发奖
- 其它的用法？
  - 后端实现不同的对请求的过滤策略
  - 后端过滤器设置不同策略对象，实现请求拦截策略灵活配置

```java
public interface PrizeSender {

  /**
   * 用于判断当前实例是否支持当前奖励的发放
   */
  boolean support(SendPrizeRequest request);

  /**
   * 发放奖励
   */
  void sendPrize(SendPrizeRequest request);

}
```

```java
// 积分发放
@Component
public class PointSender implements PrizeSender {

  @Override
  public boolean support(SendPrizeRequest request) {
    return request.getPrizeType() == PrizeTypeEnum.POINT;
  }

  @Override
  public void sendPrize(SendPrizeRequest request) {
    System.out.println("发放积分");
  }
}
```

# 4、工厂方法模式

# 你怎么创建一个对象？

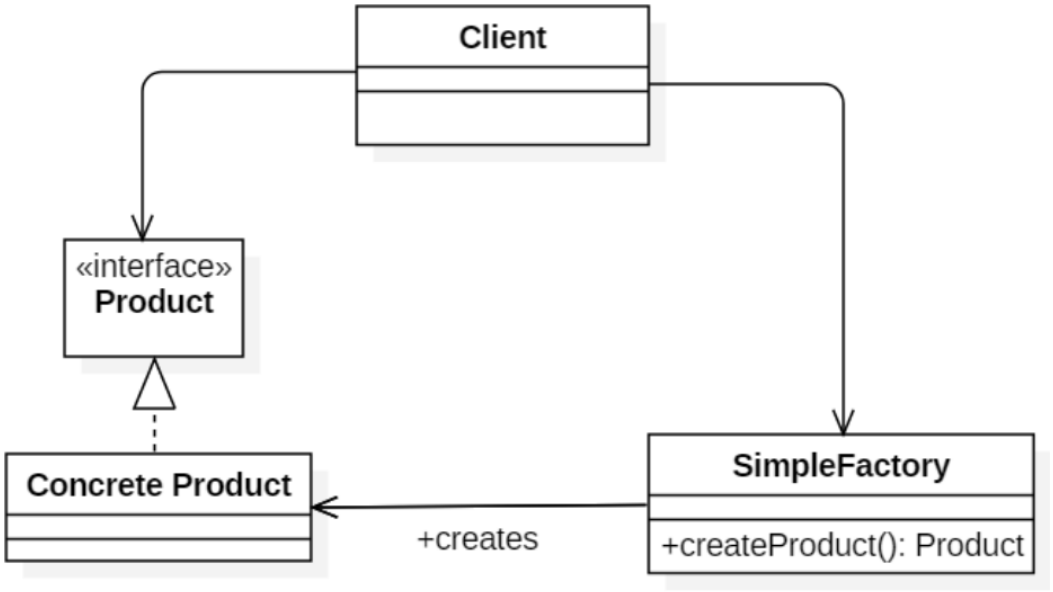

这还不简单，我直接new就完事了

那要是这个对象的构造函数改了参数呢?

嗨，那就把new那一行改了呗

改来改去无穷尽也?

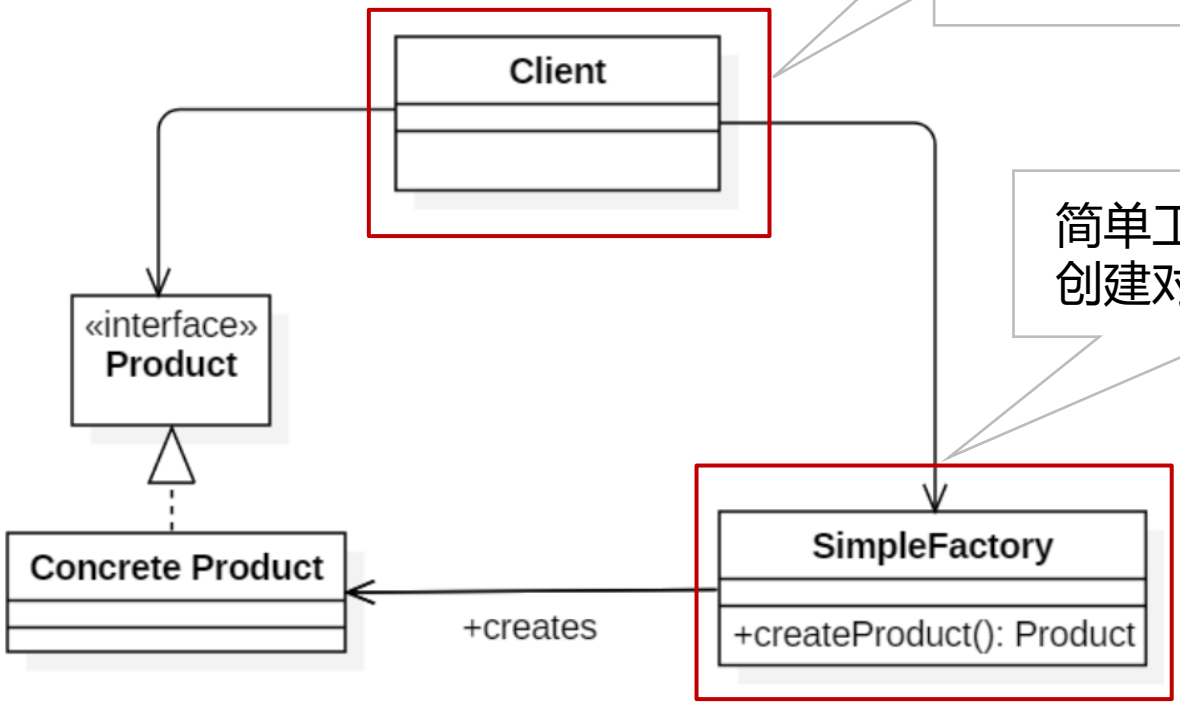
# 简单工厂模式




把实例化的操作单独放到一个类中，这个类就成为简单工厂类

# 简单工厂模式

再也不用自己new了！
以后对象构造函数改了也不怕了




简单工厂提供了一个创建对象的通用几口

Sounds good!

# 使用简单工厂模式，还有什么问题？

汉堡店

如果有很多种相似的对象？

生产劲脆鸡腿堡的简单工厂

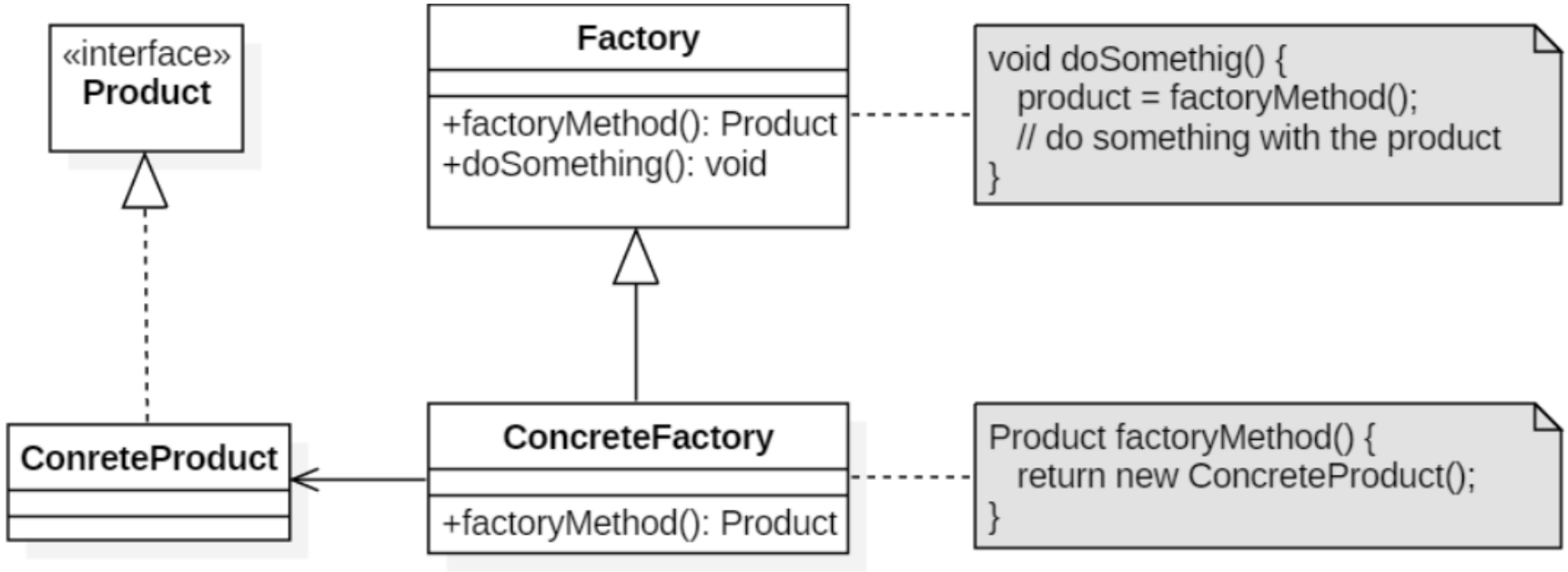
生产香辣鸡腿堡的简单工厂

生产奥尔良鸡腿堡的简单工厂

要售卖不同的汉堡，或者增加新的汉堡种类，只能：

1. 利用大量的if…else…决定使用哪个工厂，不方便
2. 每次修改原先的代码，使用不同的简单工厂，不优雅

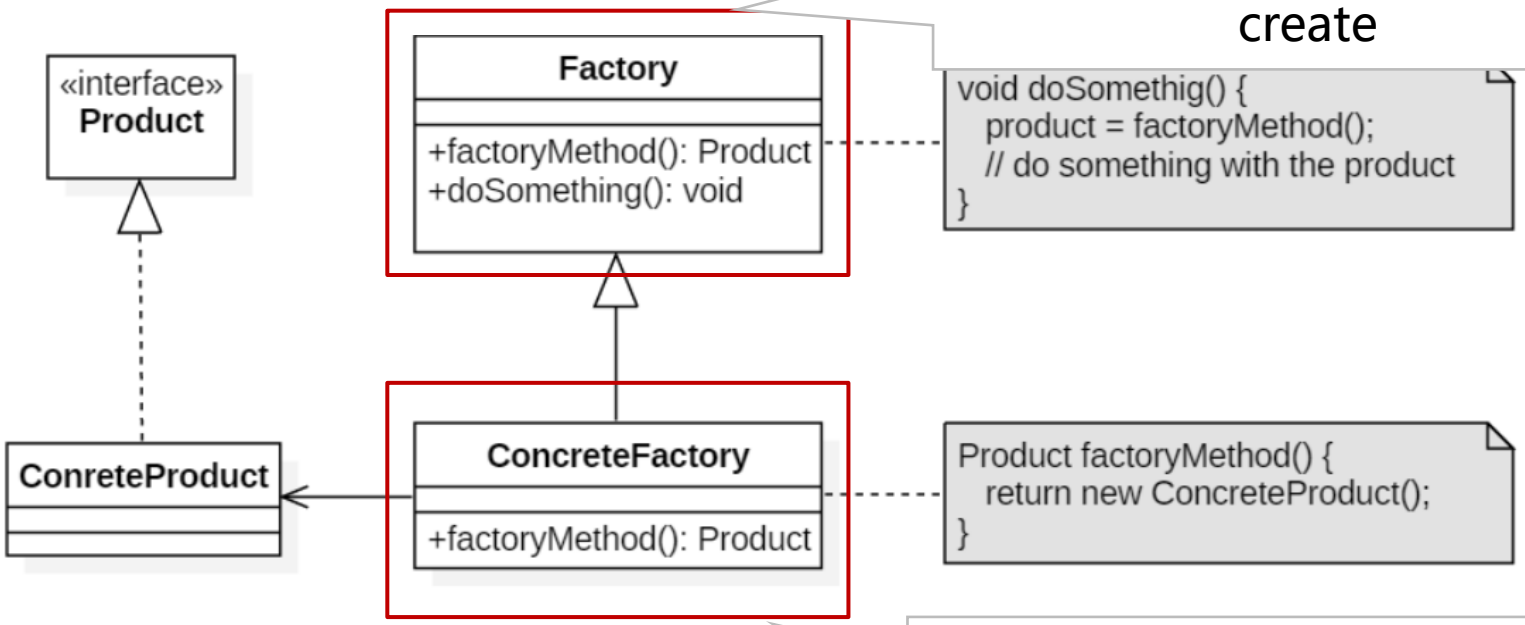
# 工厂方法模式—概述




定义了一个创建对象的接口，但由子类决定要实例化哪个类。工厂方法把实例化操作推迟到子类。

# 用了工厂方法模式？

现在有统一的factoryMethod接口，用户只需要关心do something，而不用关心how to create




增加新的产品种类，也只是增加具体工厂子类，用户的代码完全不需要更改

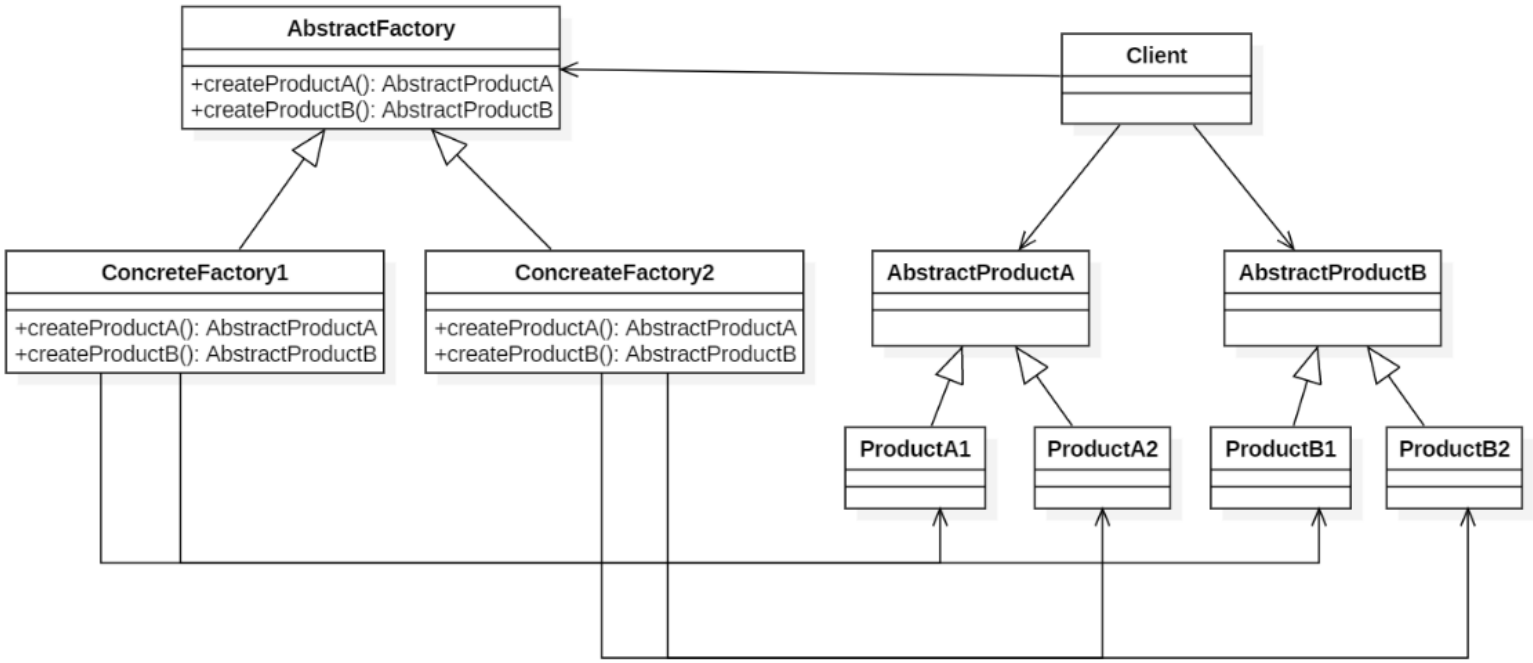
# 应用：使用不同httpClient的网络爬虫

```
public class WebCrawler {

    public WebResult getWebInfo(int clientType, String url) {
        HttpClient c = getClient(clientType);
        HtmlResult res = c.getPage(url);
        return processHtml(res);
    }

    private HttpClient getClient(int clientType) {
        HttpClient c = getFromCache();
        if (c == null) {
            c = createClient(clientType);
        }
        initClient(c);
    }

    //根据不同的类型参数来创建不同的HttpClient
    private HttpClient createClient(int clientType){
        if (clientType == 1) {
            return ATypeClient();
        } else if (clientType == 2) {
            return BTypeClient();
        } else if (clientType == 3) {
            return CTypeClient();
        } else
            ......
    }
}
```

```
public abstract class WebCrawler {

    //爬取网页数据
    public WebResult getWebInfo(String url) {
        HttpClient c = getClient();
        HtmlResult res = c.getPage(url);
        return processHtml(res);
    }

    //HttpClient是接口或者抽象类，下文统称为接口
    private HttpClient getClient() {
        //如果缓存中不存在client则创建
        HttpClient c = getFromCache();
        if (c == null) {
            c = createClient();
        }
        //创建之后对client进行初始化
        initClient(c);
    }

    //提供一个抽象让子类来实现创建client
    //这个抽象方法就是“工厂方法”
    protected abstract HttpClient createClient();
}
```

# 5、抽象工厂模式

# 概述




提供一个接口，用于创建一个对象家族

# 概述

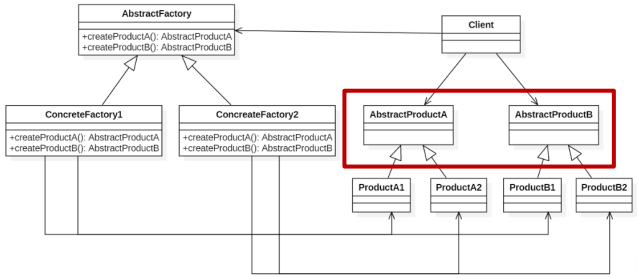
你是否永远只需要一种对象？

一名训练有素的士兵

子弹呢？？？

可以生产各种枪的抽象工厂方法

一个对象族：
你需要同时创建
这些对象



# 概述

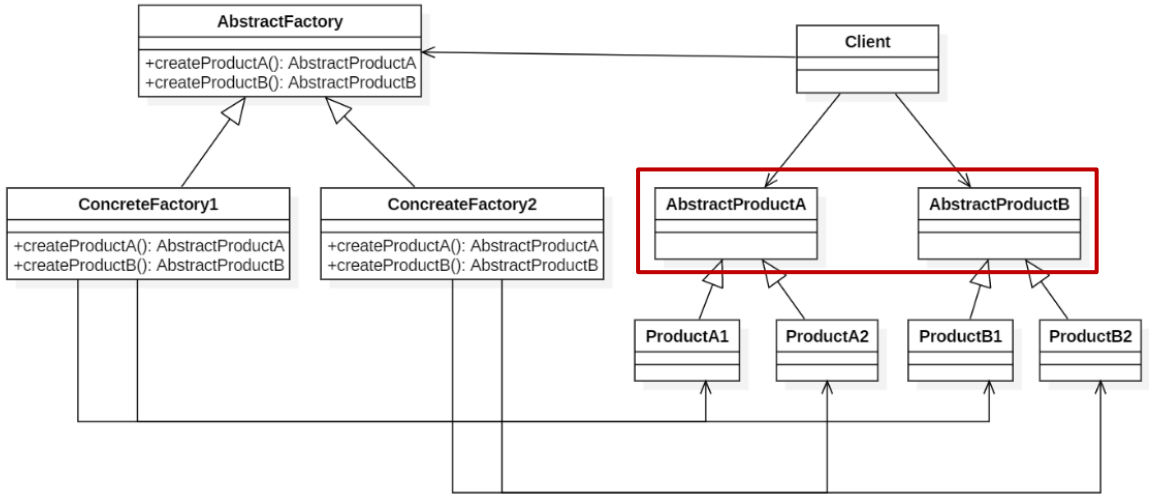
与工厂方法区别与联系

1. 抽象工厂创建的是对象家族，家族中的对象是相关的、必须一起创建出来

2. 抽象工厂模式用到了工厂方法模式来创建单一对象

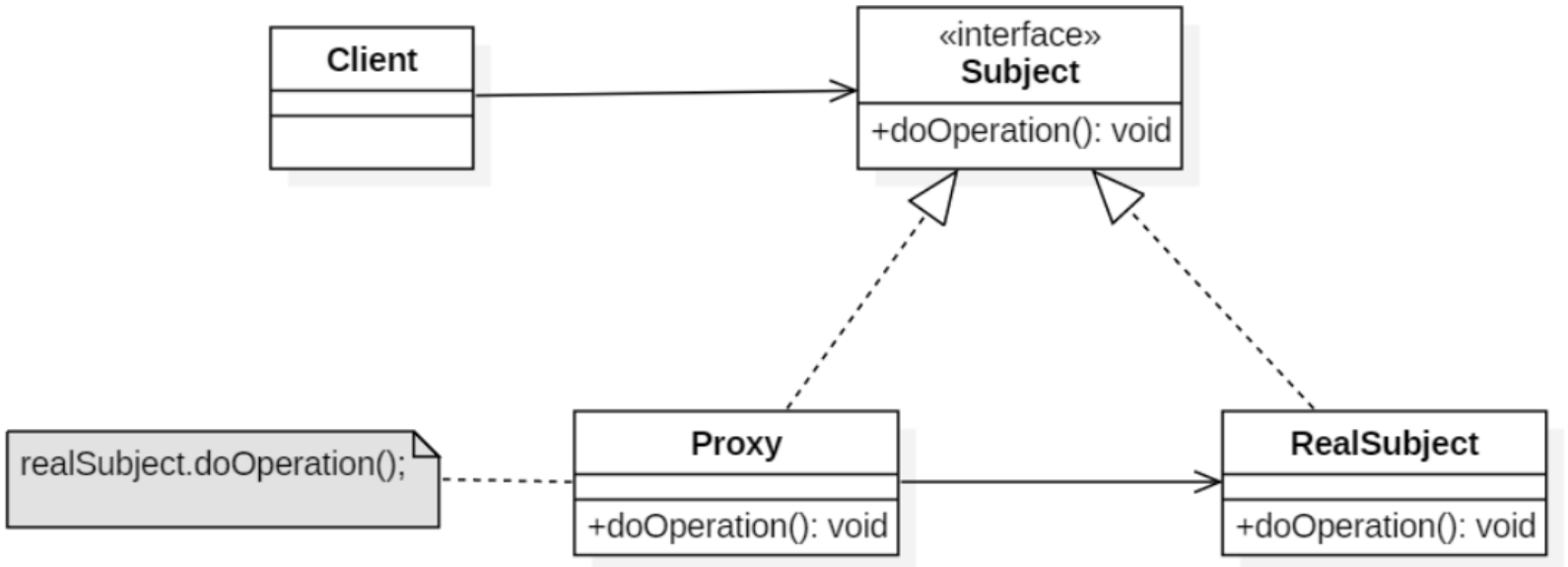
应用场景

需要创建对象族，比如枪和子弹



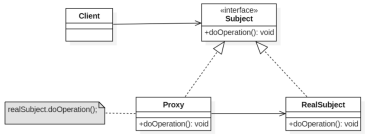
# 6、代理模式

# 概述




给某对象提供一个代理以控制对该对象的访问。
常用于扩展原实际对象的行为

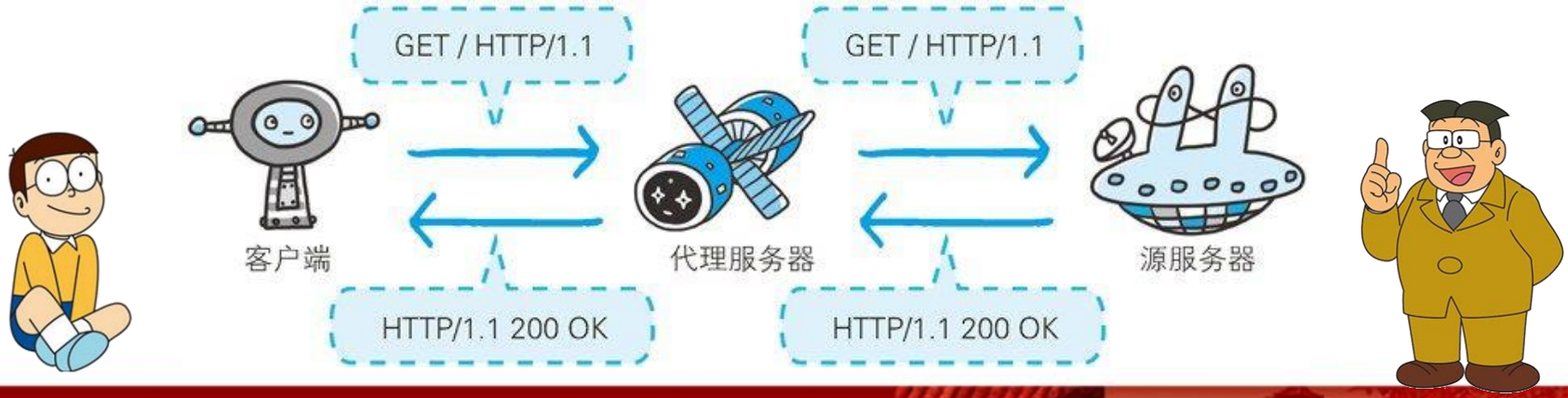
# 概述



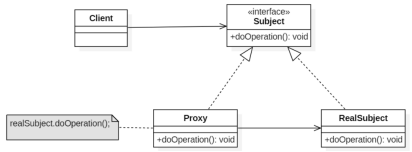

同学们的学习生活中有接触过代理这一概念吗?

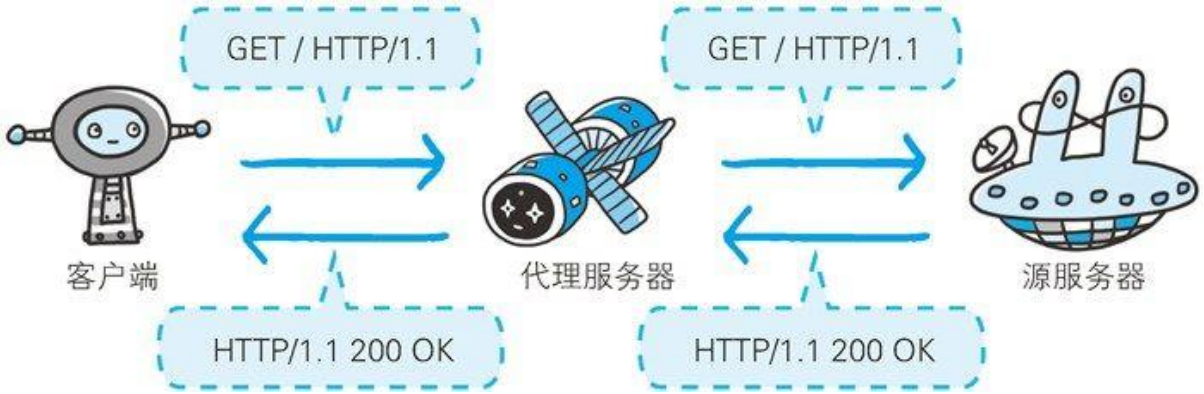
我知道！梯子!

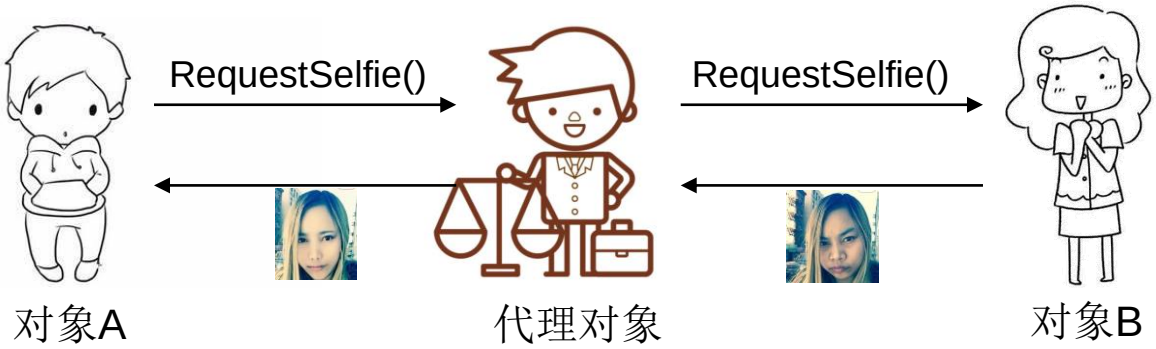
额确实……代理服务器很好地展现了代理的概念……



# 做个类比——我们看看面向对象的代理



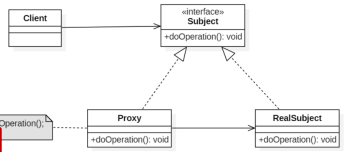



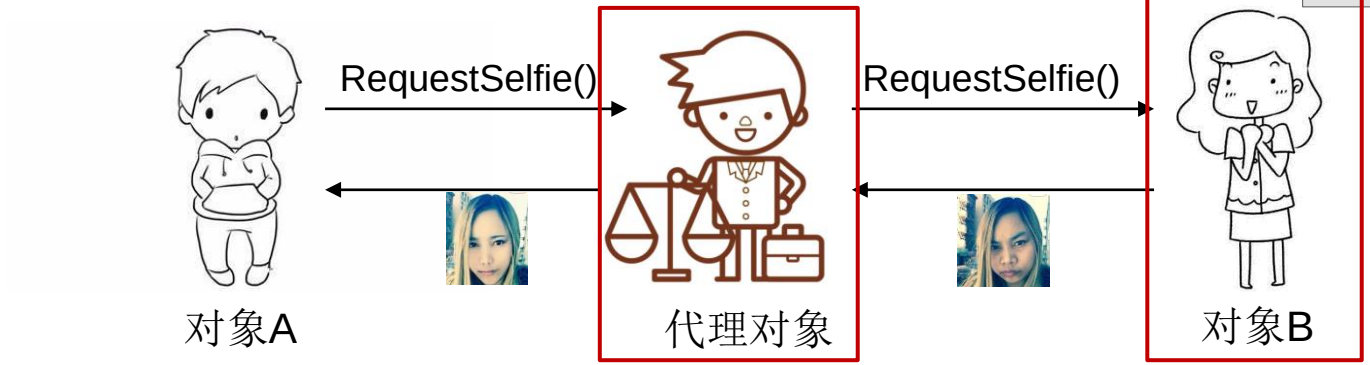




因为代理对象和对象B都实现了小姐姐接口，对象A什么都没发现

返回自拍时，代理对象又干了点多余的事情.....

# 做个类比——我们看看面向对象的代理

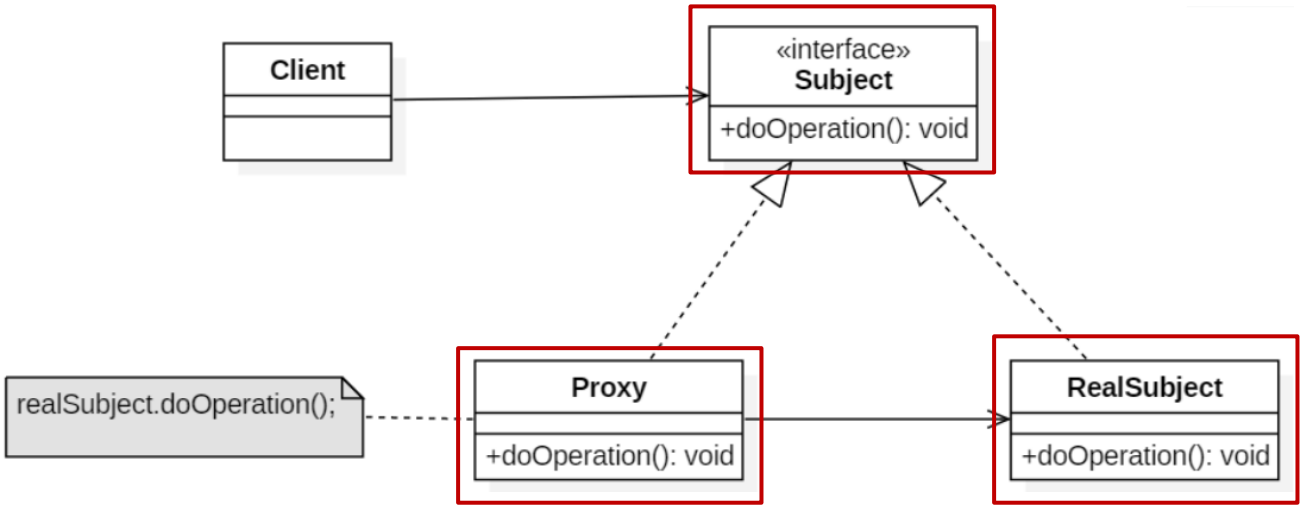



所以我为啥不直接修改对象B的代码呢?

增加代码是比修改原代码更有扩展性的方案，而且有时，对象B在你依赖的一个第三方库里，你改不了他的代码……

# 概述




- 要点
  - 代理与实际对象实现同一接口
  - 代理使用实际对象的操作，并在实际对象的操作之上添加其它操作（相当于一个中介）

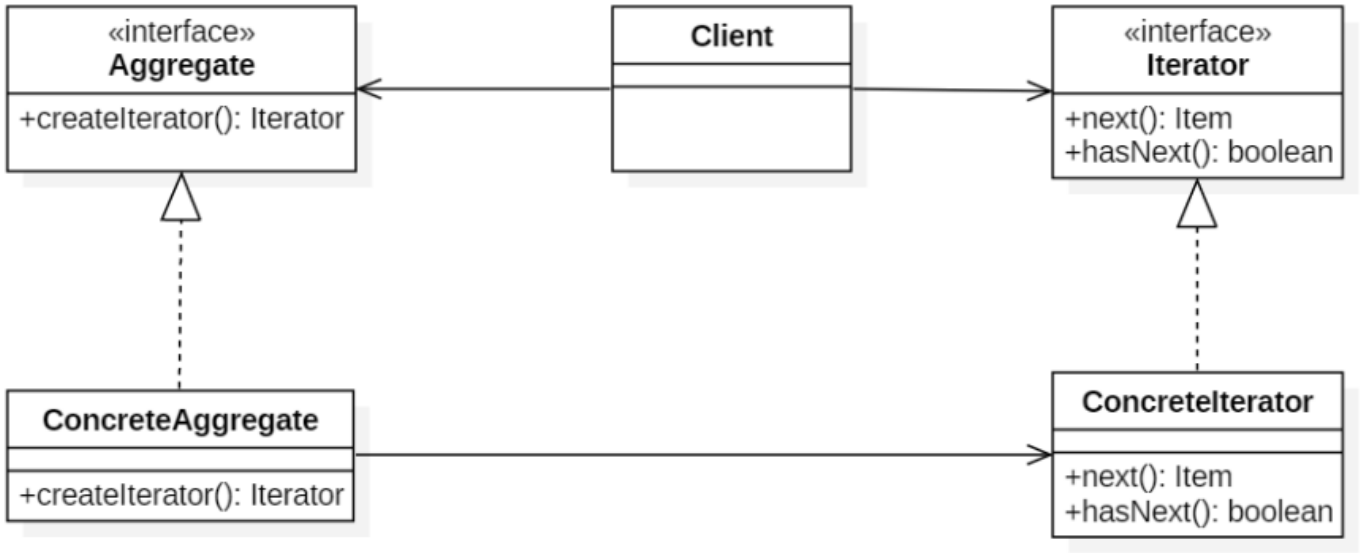
# 应用：图片延迟加载

```
public class HighResolutionImage implements Image {

    private URL imageURL;
    private long startTime;
    private int height;
    private int width;

    public int getHeight() {
        return height;
    }

    public int getWidth() {
        return width;
    }

    public HighResolutionImage(URL imageURL) {
        this.imageURL = imageURL;
        this.startTime = System.currentTimeMillis(); }
        this.width = 600;
        this.height = 600;
    }
```

```
public class ImageProxy implements Image {

    private HighResolutionImage highResolutionImage;

    public ImageProxy(HighResolutionImage highResolutionImage) {
        this.highResolutionImage = highResolutionImage;
    }

    @Override
    public void showImage() {
        while (!highResolutionImage.isLoad()) {
            try {
                System.out.println("Temp Image: " + highResolutionImage.getWidth() + " "
+ highResolutionImage.getHeight());
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        highResolutionImage.showImage();
    }
```

# 7、迭代器模式

# 概述




提供一种顺序访问聚合对象元素的方法，并且不暴露聚合对象的内部表示

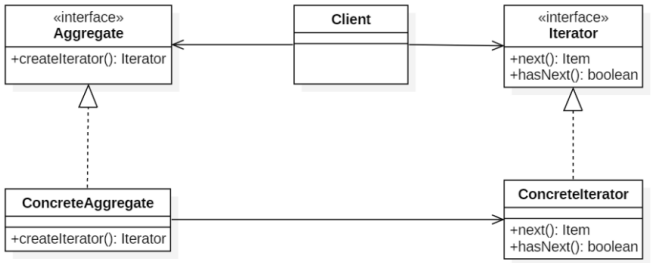
# 概述

注：很多面向对象语言内部的集合类就支持迭代器，如C++、java……

我感觉我从来都是直接
for(int i=0; i < length; i++)...

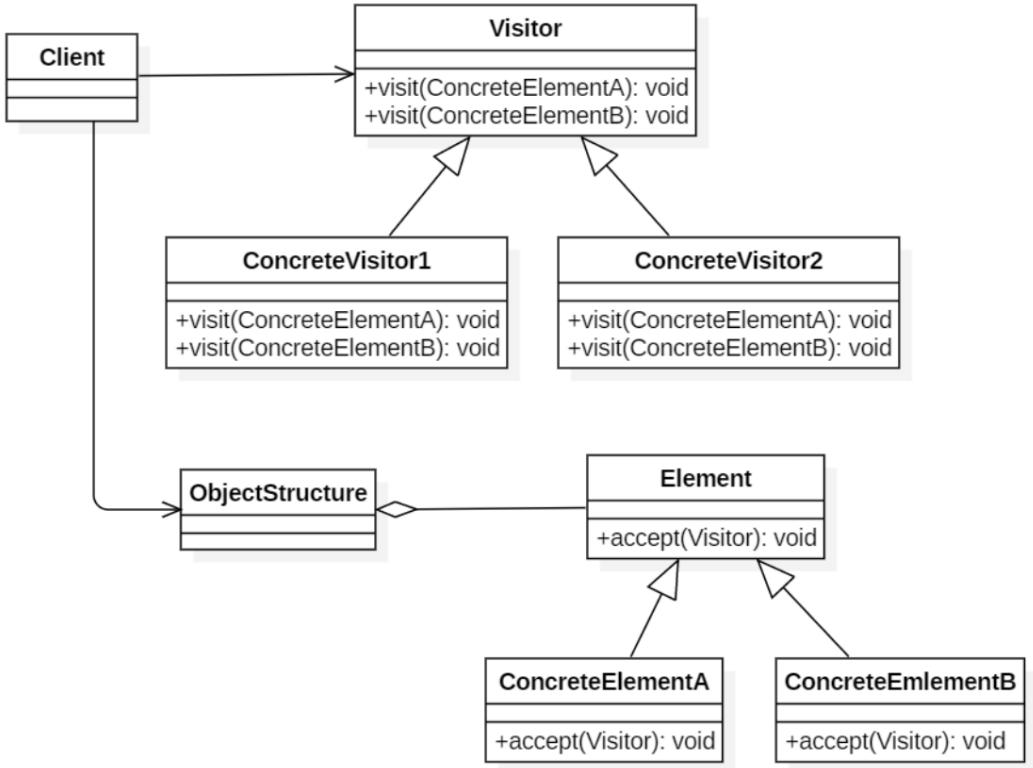
迭代器有很多优点，他是面向对象封装特性的很好体现，可以以不同方式遍历一个聚合而不需要暴露聚合对象的内部表示，增加新的迭代方式也很方便，不需要修改原代码.....

额，也许我该试试



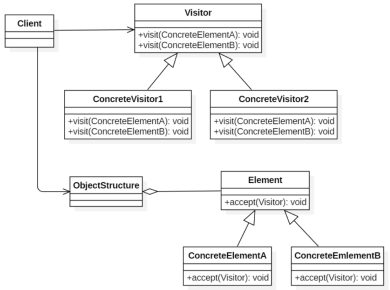

# 8、访问者模式

# 概述




将作用于某种数据结构中的各元素的操作分离出来封装成独立的类
使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作
为数据结构中的每个元素提供多种访问方式

# 概述




我怎么从来没听说过？这玩意有用么？

访问者模式通常用于需要为某种复杂的数据结构增加新逻辑的场景，这的确是一种初学者很难接触到的开发需求

“某种复杂的数据结构”到底是哪种复杂的数据结构啊

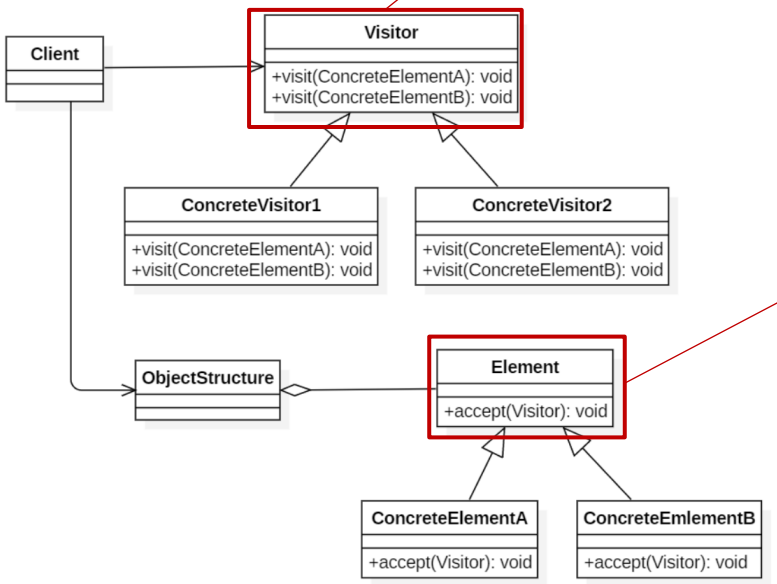
基本上包括所有可以遍历的复杂结构，包括树、图等...典型的比如语言的语法树、html页面的dom树等

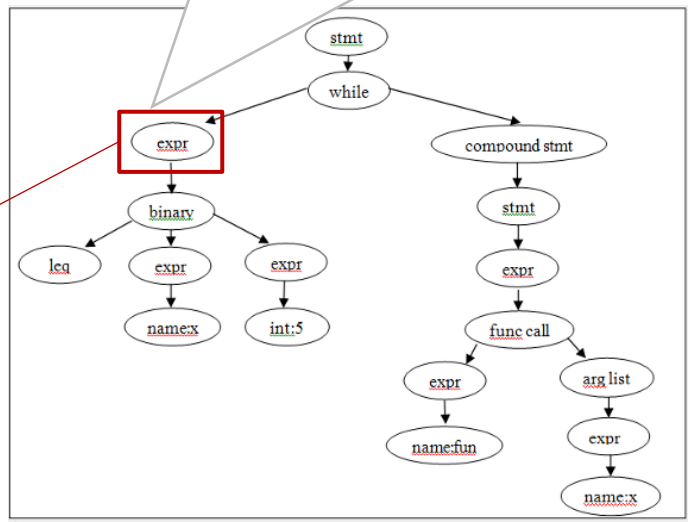
# 举个例子：编译器

在当前结点，我应该根据子节点的翻译结果，把这个结点对应的语句翻译为……

Visitor内部有访问每个结点的visit方法，在方法内部实现语言的翻译

每个语法树结点内部实现了accept方法，内部调用visitor的visit方法让他来访问自己




经过解析，程序代码被解析为一棵“抽象语法树”

# 举个例子：编译器

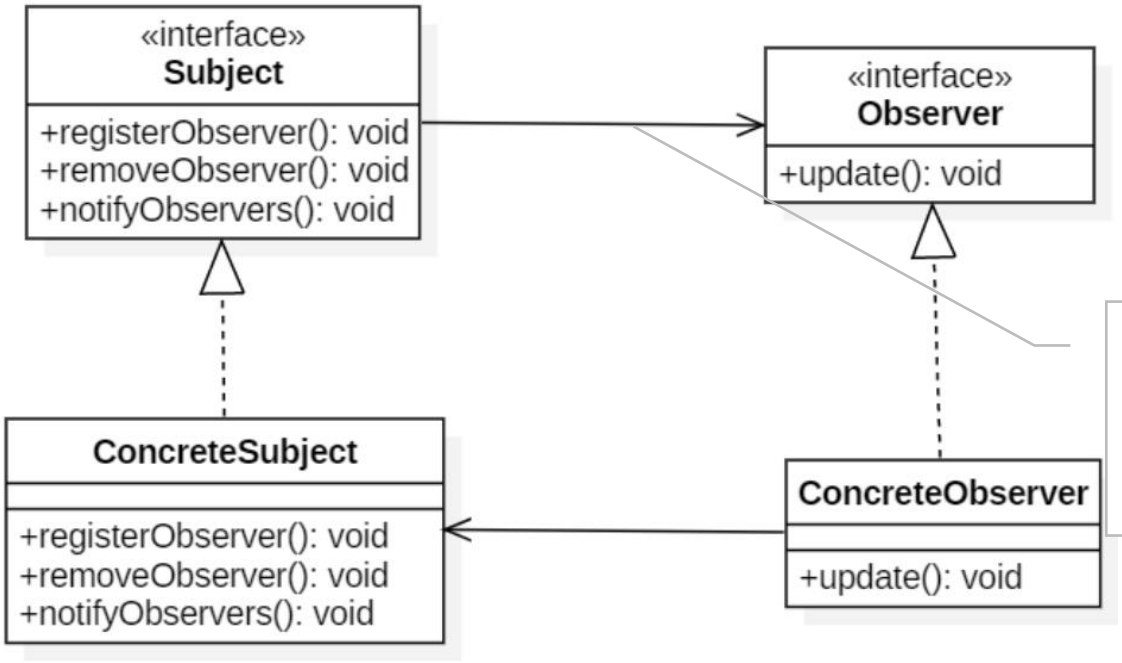
- 概念对应
  - 复杂的数据结构-抽象语法树
  - 数据结构中的元素-语法树中的节点
  - 访问者-在visit方法中实现具体编译逻辑
- 其它可能应用
  - 自定义处理HTML等

```java
public class BooleanType implements Node {
    public NodeToken f0;

    public BooleanType(NodeToken n0) {
        f0 = n0;
    }

    public BooleanType() {
        f0 = new NodeToken("boolean");
    }

    public void accept(Visitor v) {
        v.visit(this);
    }
    public <R,A> R accept(GJVisitor<R,A> v, A argu) {
        return v.visit(this,argu);
    }
    public <R> R accept(GJNoArguVisitor<R> v) {
        return v.visit(this);
    }
    public <A> void accept(GJVoidVisitor<A> v, A argu) {
        v.visit(this,argu);
    }
}
```

```java
public interface GJVisitor<R,A> {

   //
   // GJ Auto class visitors
   //

   public R visit(NodeList n, A argu);
   public R visit(NodeListOptional n, A argu);
   public R visit(NodeOptional n, A argu);
   public R visit(NodeSequence n, A argu);
   public R visit(NodeToken n, A argu);
```

# 9、观察者模式

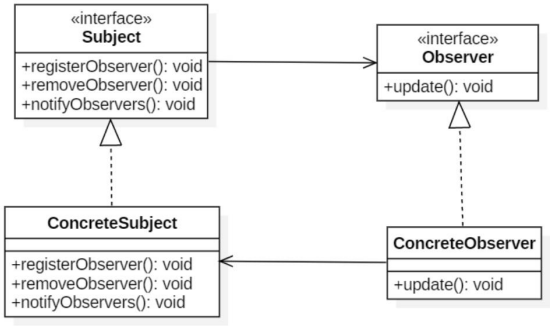
# 概述




指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新

又称作发布-订阅模式、模型-视图模式。

# 概述




嗯，感觉又是一个啥用没有的模式呢（x

并不是，你们的系统总要写前端或客户端的UI吧，其实这个模式已经潜移默化地包含在你们使用的UI框架里了

For real?

# UI中的观察者模式

以下代码是伪代码

手动刷新，不行！

```
// 这个是类似于命令的方式，在点击 count 之后手动调用刷新 UI
<button onClick="count += 1; refreshUI(); ">{ count }</button>
```

观察者模式可以观察 count 的变量，在变量改变之后刷新 UI

```
// 在 count 改变之后刷新 UI
observe(count, refreshUI)
// 这个观察通常由某些框架来完成，是自动化的

<button onClick="count += 1;">{ count }</button>
```

自动刷新，行！

count被观察了！

# 应用：天气数据分析系统

- 虽然是不同的分析对象，但是分析的是一个天气数据

```java
public interface Subject {
    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObserver();
}
```

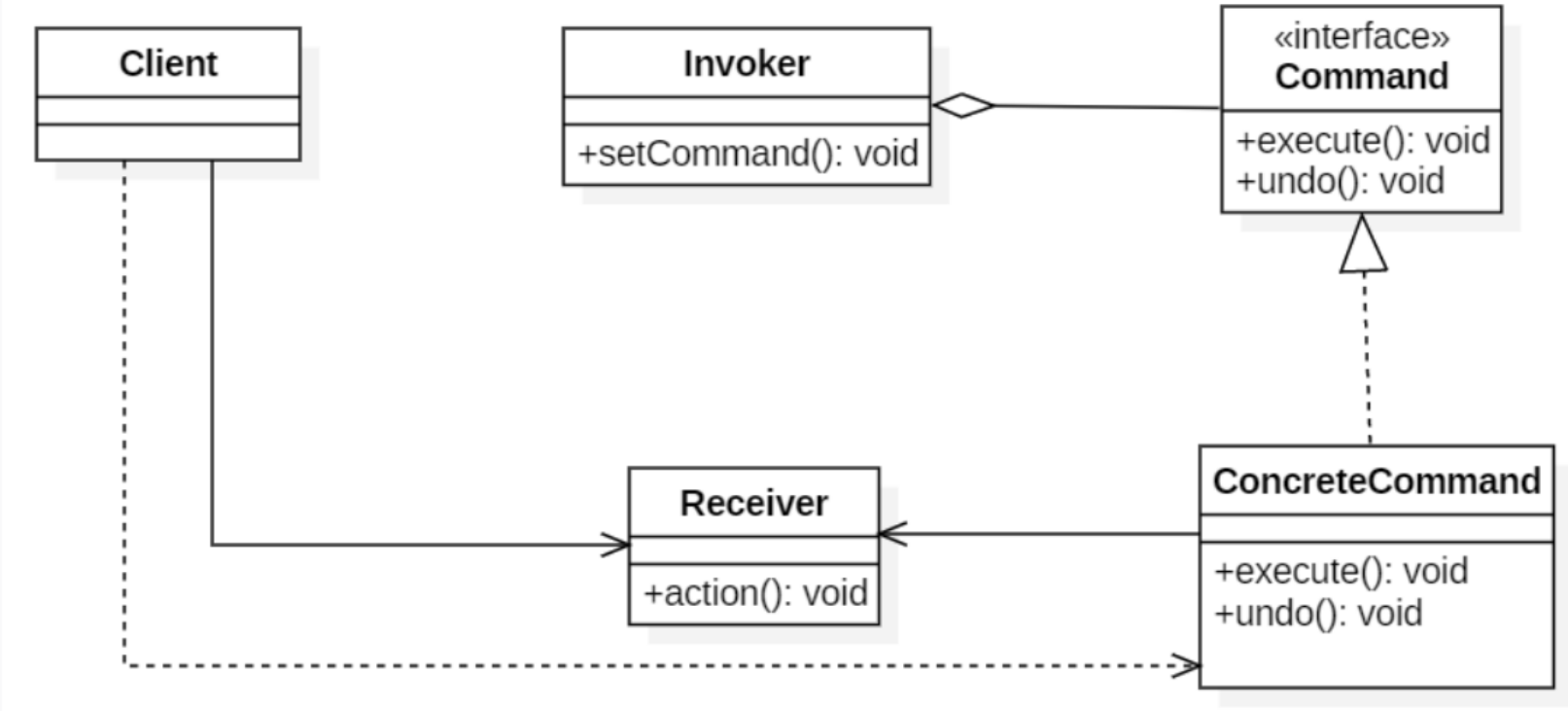
```java
public class WeatherData implements Subject {
@Override
public void registerObserver(Observer o) {
    observers.add(o);
}

@Override
public void removeObserver(Observer o) {
    int i = observers.indexOf(o);
    if (i >= 0) {
        observers.remove(i);
    }
}

@Override
public void notifyObserver() {
    for (Observer o : observers) {
        o.update(temperature, humidity, pressure);
    }
}
```

```java
public interface Observer {
    void update(float temp, float humidity, float pressure);
}
public class StatisticsDisplay implements Observer {
public class CurrentConditionsDisplay implements Observer {
```



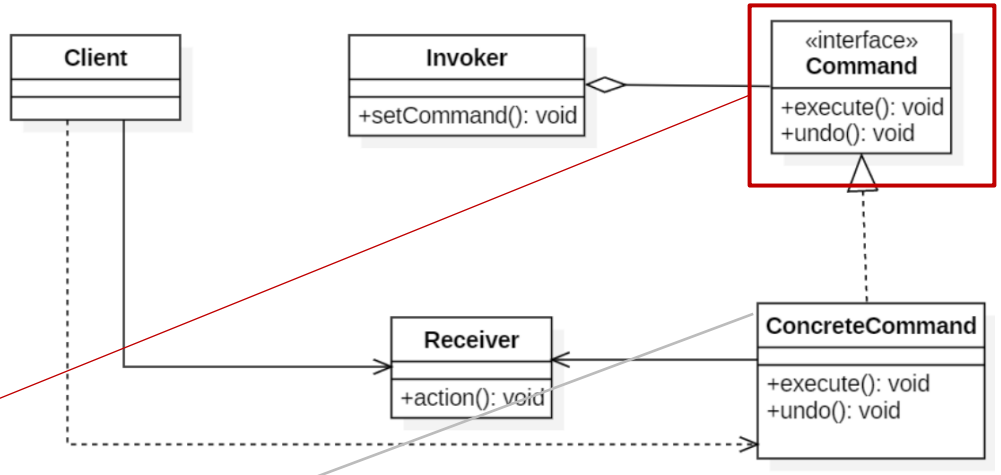
# 10、命令模式

# 概述




将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开

# 示例：遥控器

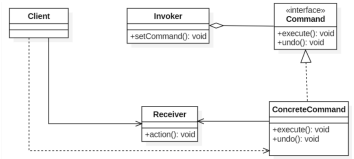



我们与系统的每次交互，都可以抽象为“我们向系统发送一条命令、系统接收命令后执行具体逻辑”。每条命令都可抽象为一个对象，就像“传令兵”

遥控器的每一个按钮可以对应每个具体的command，背后执行系统具体的操作，遥控器可以自由地使用这些command对系统发出命令

# 示例：遥控器

```java
public interface Command {
    void execute();
}
```




```java
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.on();
    }
}
```

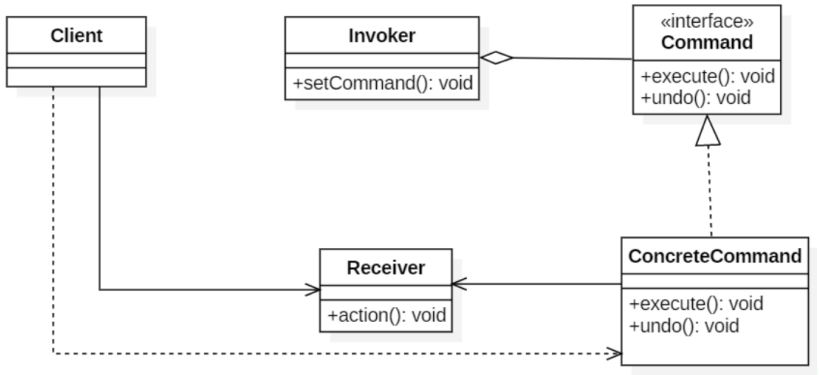
```java
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.off();
    }
}
```

```java
/**
 * 遥控器
 */
public class Invoker {
    private Command[] onCommands;
    private Command[] offCommands;
    private final int slotNum = 7;

    public Invoker() {
        this.onCommands = new Command[slotNum];
        this.offCommands = new Command[slotNum];
    }

    public void setOnCommand(Command command, int slot) {
        onCommands[slot] = command;
    }

    public void setOffCommand(Command command, int slot) {
        offCommands[slot] = command;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }
}
```

啊所以为什么我不能直接执行系统逻辑呢？非要中间再夹一个命令对象？

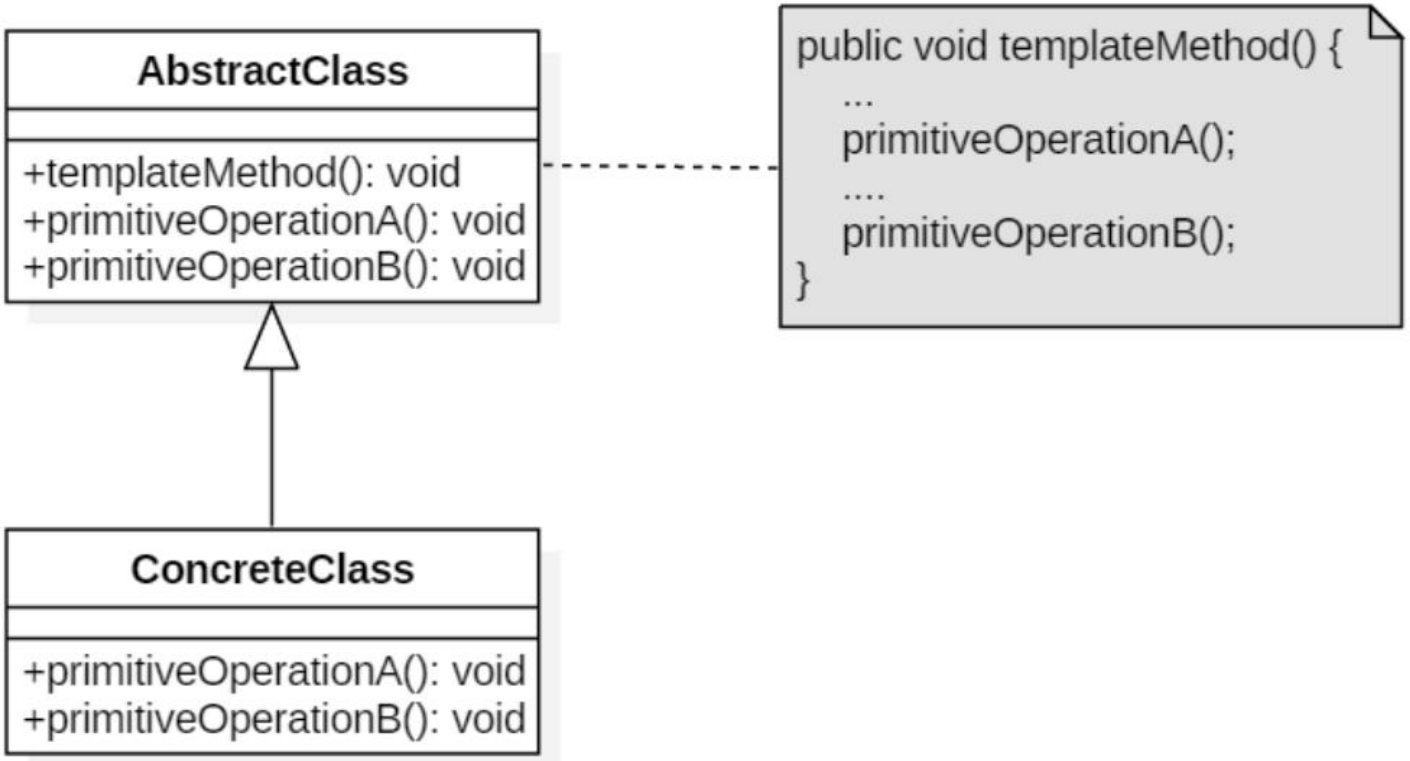
降低耦合是命令模式的一大优点，命令的使用者不再需要了解系统底层的逻辑，只需要使用包装好的command

而且对于文字编辑器等系统，命令模式有很多方便的作用：如命令的参数化、将命令放入队列、纪录命令到日志中、撤销命令等。



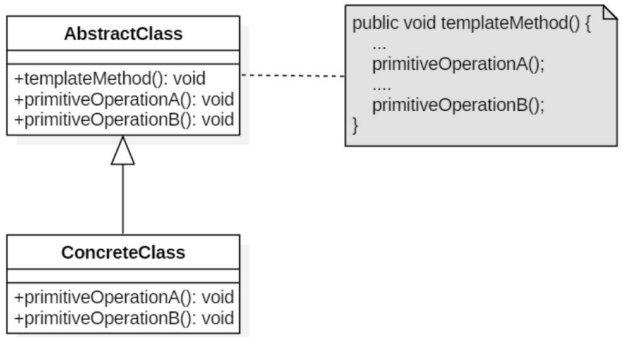

# 11、模板方法模式

# 概述




定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤

# 概述





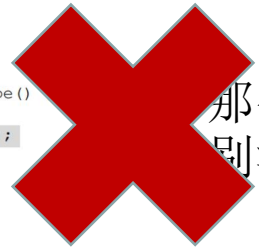
- 想象你的室友是个DDL战神，每天都熬夜肝DDL
- 好心的你决定在他肝的时候给他泡点咖啡
- 当然每天泡咖啡有点腻歪，也许你偶尔也会给他泡点茶

**Coffee**

```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

**Tea**

```
void prepareRecipe()
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

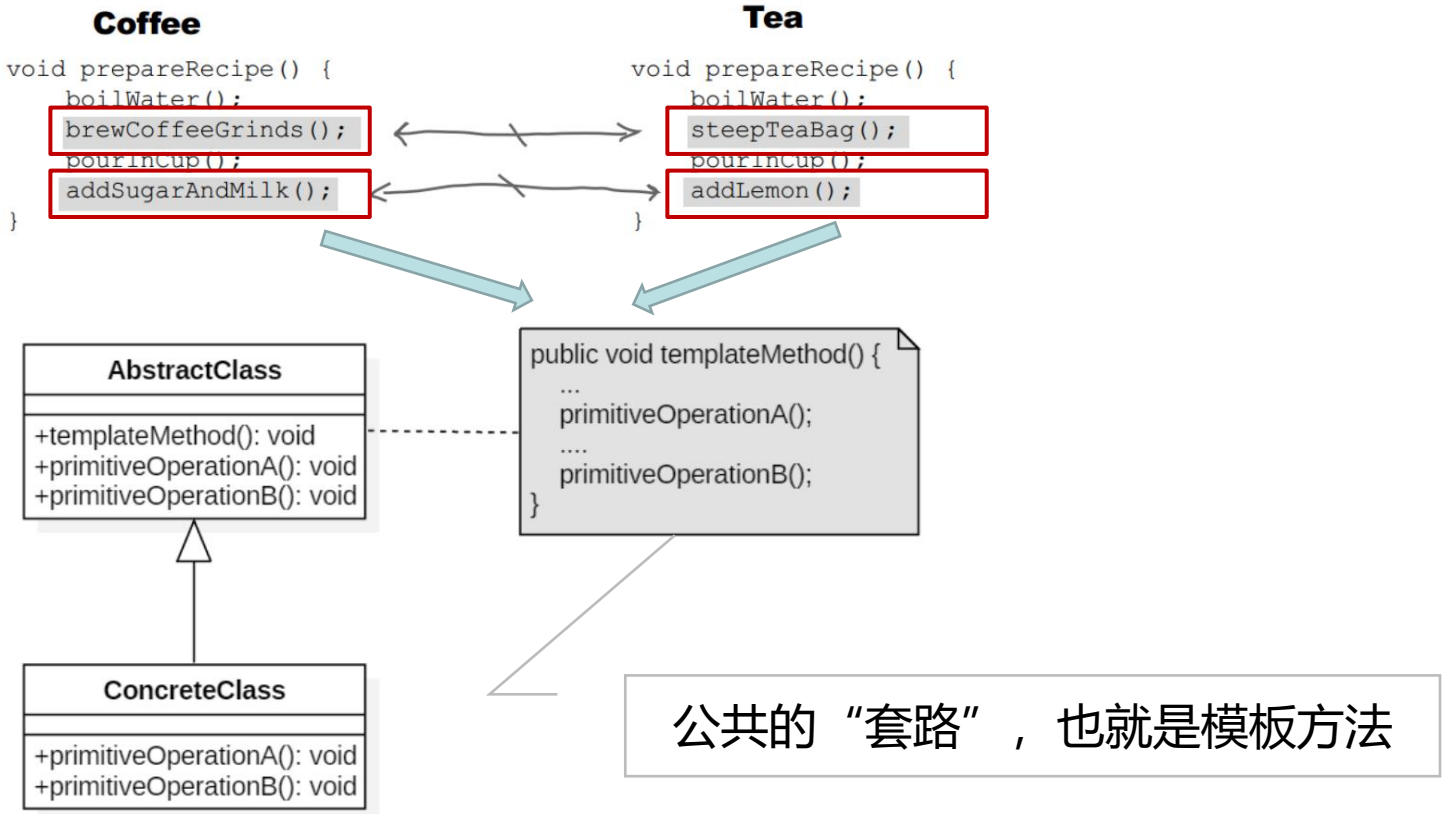


那么你要写两个函数分别泡咖啡和泡茶吗？

# 概述

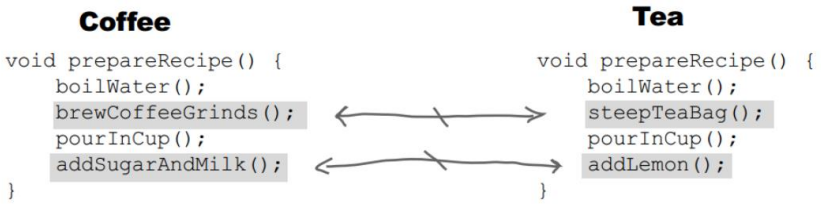
泡茶跟泡咖啡，其实都是一个套路！

可以用一个抽象类总结这个“套路”，把每一步的具体实现放到子类里




提取公共部分代码，方便系统的扩展与复用

# 示例：冲咖啡与冲茶



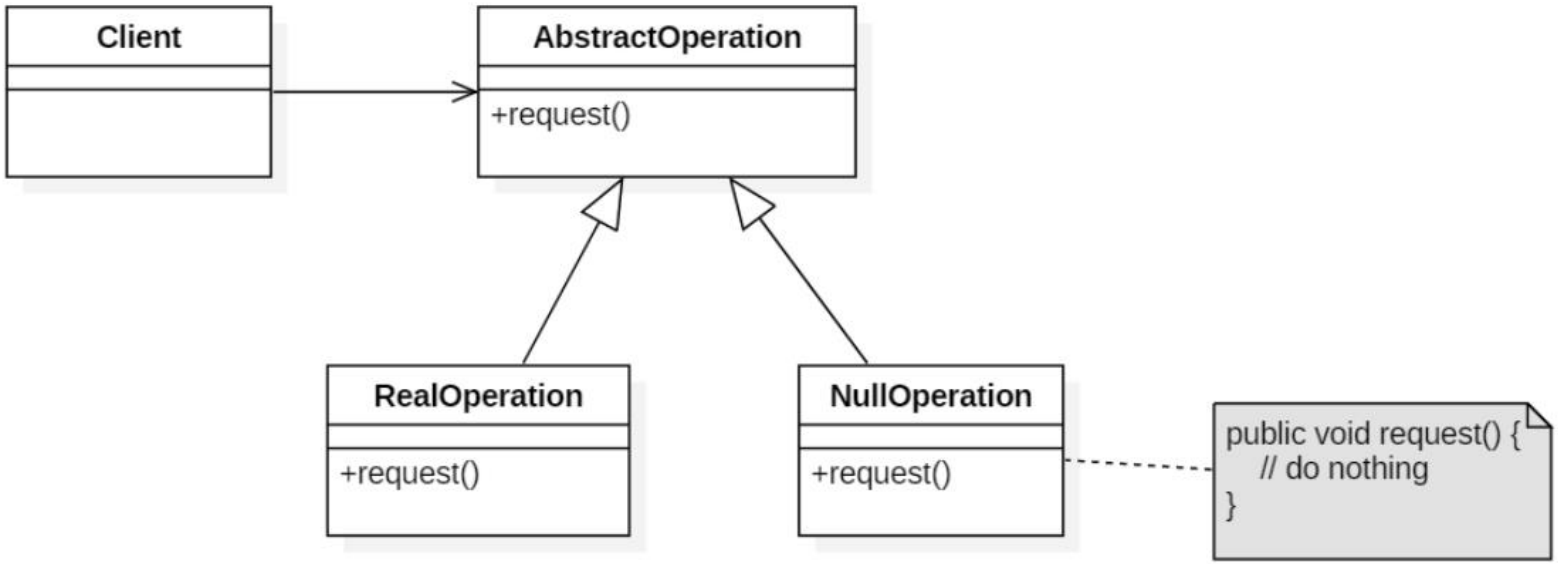

```java
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("boilWater");
    }

    void pourInCup() {
        System.out.println("pourInCup");
    }
}
```

```java
public class Coffee extends CaffeineBeverage {
    @Override
    void brew() {
        System.out.println("Coffee.brew");
    }

    @Override
    void addCondiments() {
        System.out.println("Coffee.addCondiments");
    }
}
```

```java
public class Tea extends CaffeineBeverage {
    @Override
    void brew() {
        System.out.println("Tea.brew");
    }

    @Override
    void addCondiments() {
        System.out.println("Tea.addCondiments");
    }
}
```

# 12、空对象

# 概述




使用什么都不干的空对象来代替NULL

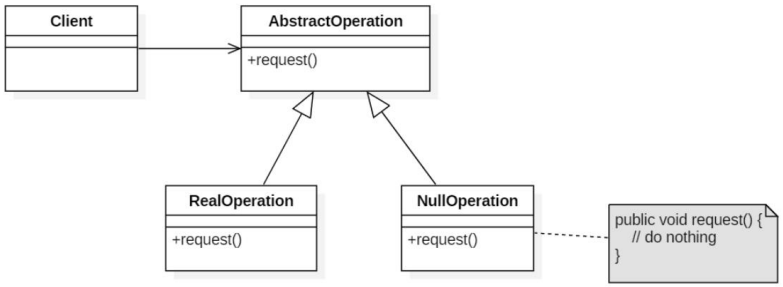
# 概述

就这？？

就这！事实上，使用空对象来替代NULL是十分有必要的事情

?愿闻其详

一个方法返回 NULL，意味着方法的调用端需要去检查返回值是否是 NULL，这么做会导致非常多的冗余的检查代码。并且如果某一个调用端忘记了做这个检查返回值，而直接使用返回的对象，那么就有可能抛出空指针异常。




# 13、mixin模式

# 概述

```
function withPhone (o) {
  if (!o.phones) o.phones = []
}

function withIPhone8 (o) {
  withPhone(o)
  o.phones.push('iPhone 8')
}

function withIPhoneX (o) {
  withPhone(o)
  o.phones.push('iPhone 9')
}

const o = {}

withPhone(o)
withIPhone8(o)

// 这个时候对象 o 就具有某种能力了
```

利用动态语言（如js）的灵活特性，通过一系列mixin函数为对象赋予灵活的能力

# 概述

```
function withPhone (o) {
  if (!o.phones) o.phones = []
}

function withIPhone8 (o) {
  withPhone(o)
  o.phones.push('iPhone 8')
}

function withIPhoneX (o) {
  withPhone(o)
  o.phones.push('iPhone 9')
}

const o = {}

withPhone(o)
withIPhone8(o)

// 这个时候对象 o 就具有某种能力了
```

- 用途
  - Mixin常用于加强一个对象，为对象加入某种能力。
- 应用场景
  - 在动态性比较强的语言中很常见，坏处是对类型推导不友好，因为啥样的都有可能混出来。
  - UI开发常见、后端不常见

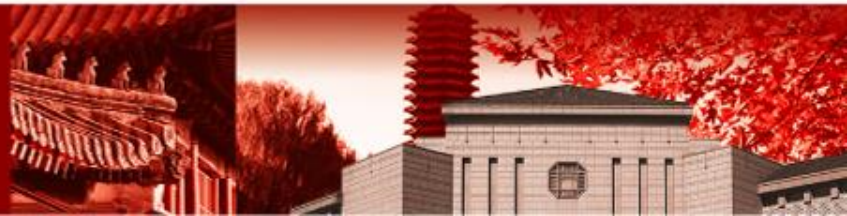
# 应用：为UI组件增加能力

- 利用mixin函数为组件组合新的能力（如对被点击等行为的处理）
- 在UI的业务开发中、组合在实践中优于继承，继承对于类的结构有很大的形态上的限制
- 但灵活的负面就是难维护，比如引入了一个别人写的mixin，可能很难弄清楚到底组合了什么能力

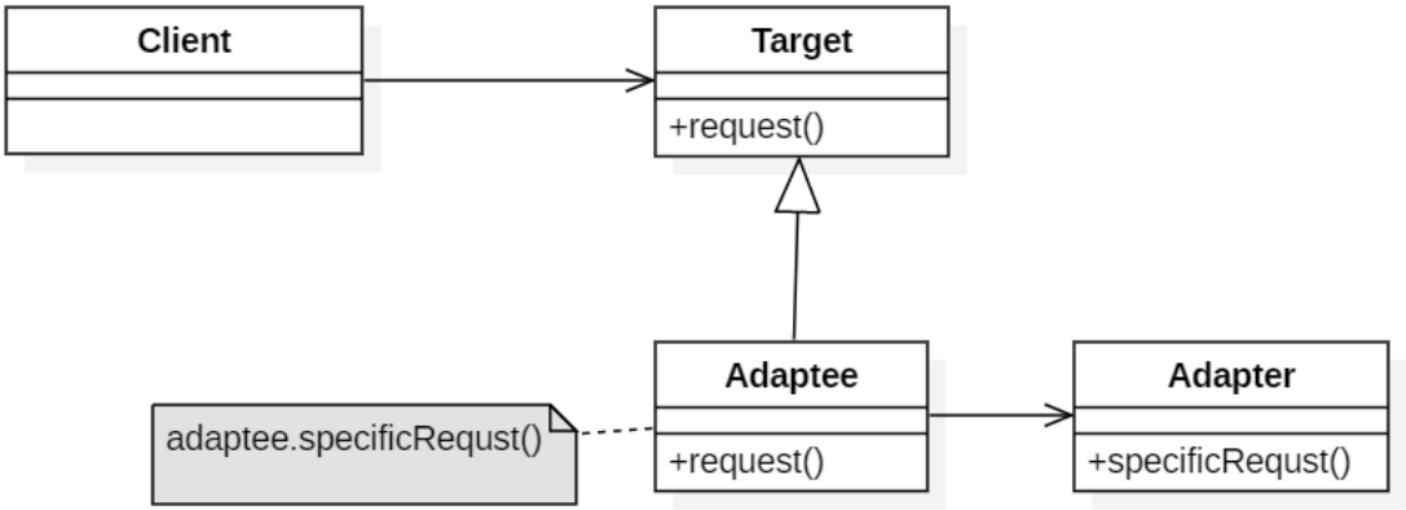
```
// 在 UI 开发中很常见，后端开发不常见
// clickable, hoverable 是一个 mixin 函数
const clickable = xxx
const hoverable = yyy

// 复用起来就很简单，很灵活
const clickableA = clickable(componentA)
const clickableHoverableA = hoverable(clickableA)
```

# 14、适配器模式



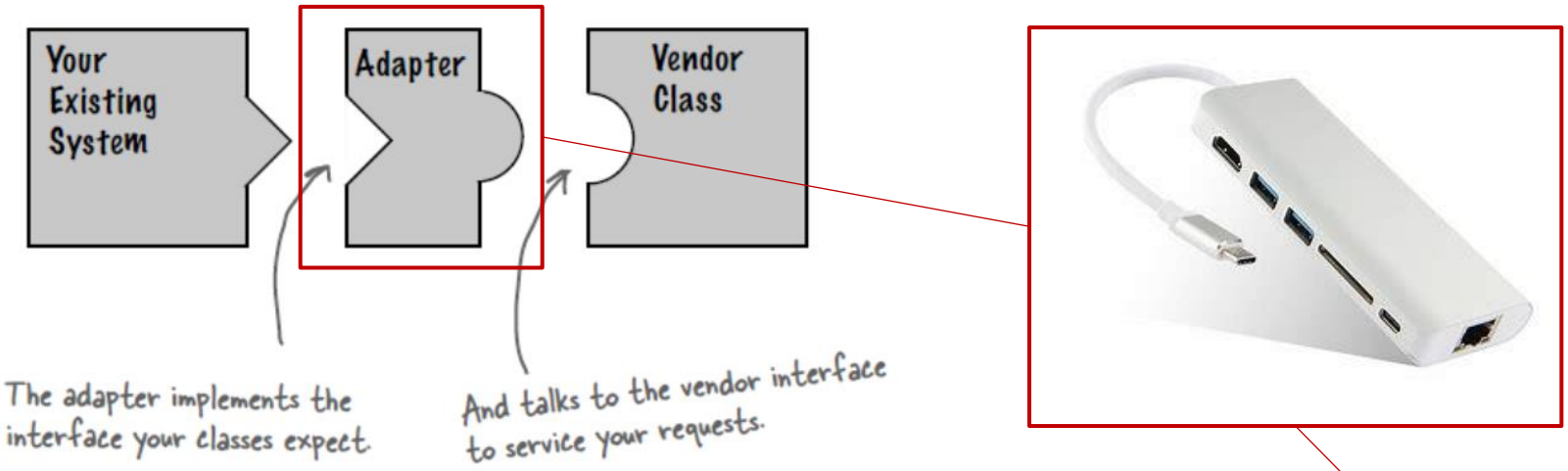


# 概述



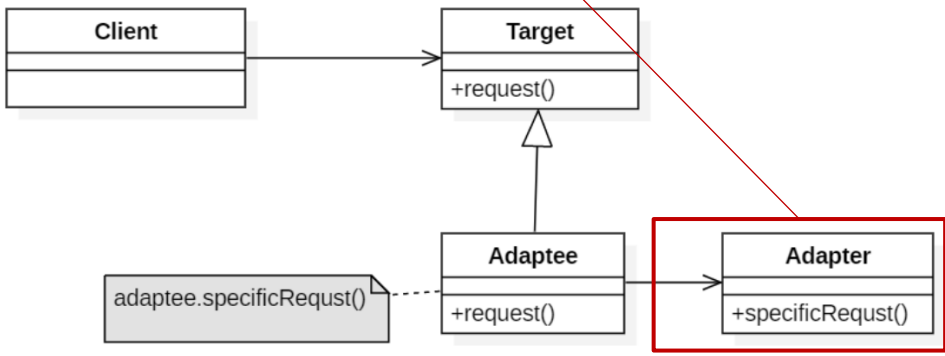

定义一个转换器，把一个类接口转换成另一个用户需要的接口

# 其实这个东西你们应该很熟悉……



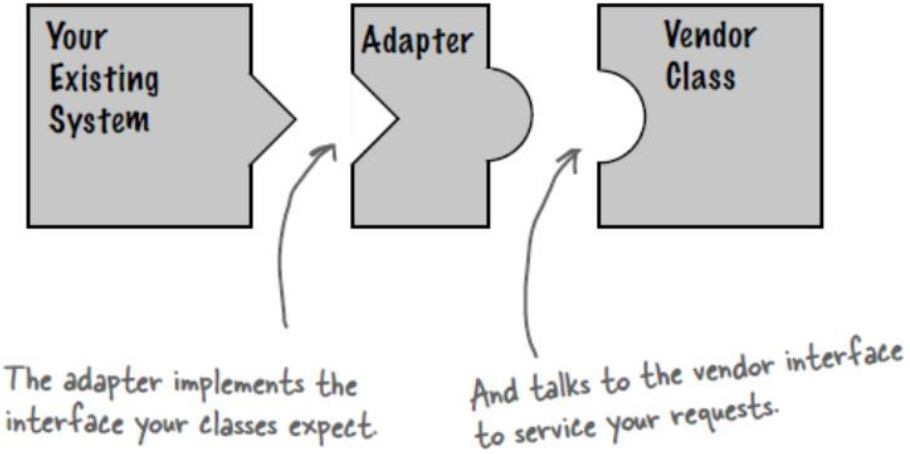


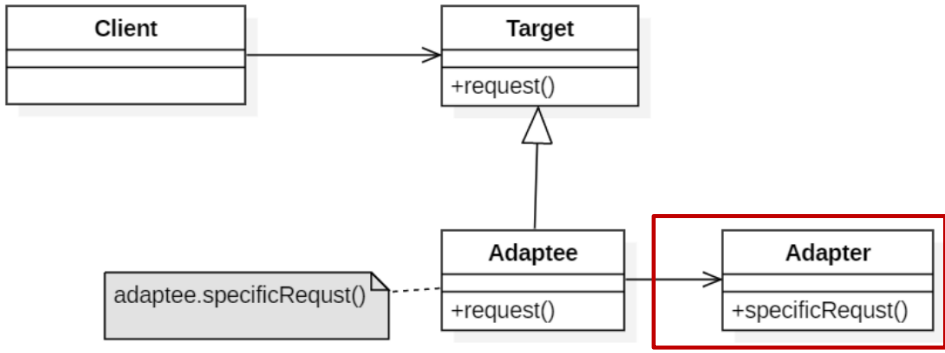




# 概述

- 实现
  - 实现适配器类作为原对象与适配目标对象的桥梁



# 实例：不修改对外接口而修改实现

比如你原先搞了个系统，提供一个接口，接受的数据格式是A，现在想改了。

```
// 比如原先暴露了个接口，让用户设定他的信息
MyModule.setUserInfo('罗智翔', '男', '我不骗人，骗人的是小猪')
// 出于某种原因，你想把接口改了，比如下面这样
MyModule.setYourFavoriteApp({
  name: '罗智翔',
  gender: '男',
  motto: '我不骗人，骗人的是小猪'
})
```

之前已经有很多人用了你写的这个东西，怎么改？

```
const MyModule = {
  // 把原先的实现挪到另一个内部函数
    setUserInfo (...args) {
        this.setUserInfoAdapter(...args)
    },
  // 适配器
  setUserInfoAdapter (...args) {
    if (typeof args[0] === 'string') this.prevSetUserInfo(...args)
    if (typeof args[0] === 'object') {
      this.prevSetUserInfo(args[0].name, args[0].gender, args[0].motto)
    }
  },
    prevSetUserInfo () {
        // 旧的实现
    }
}
```

# 15、依赖注入

# 概述

汽车类依赖轮胎类！

# 概述

你会怎么实现这台汽车?

直接来!

```
class Car {
    private Tire tire;
    public Car() {
        tire = new Tire();
    }
}
```

Nonono, 这样的耦合程度太高啦

啊? 这?

# 概述

- 对象声明自己的依赖，而该依赖由外部注入的形式为其提供
- 用途
  - 对依赖于被依赖的对象进行解耦
  - 右图例
    - 原先必须测试完整的自行车和汽车，现在轮子和车可分开测试，也可以分开实现

```
const tireA = xxx
const tireB = yyy

class Car {
  constructor (tire) {
    this.tire = tire
  }
  go () {
    this.addGas()
    this.tire.roll()
  }
}

class Bicycle {
  constructor (tire) {
    this.tire = tire
  }
  go () {
    this.addLeg()
    this.tire.roll()
  }
}

new Car(tireA)
new Car(tireB)
new Bicycle(tireA)
new Bicycle(tireB)
```

# 设计模式中的原则

面向接口编程，而不是面向具体细节

- 将实现隐藏在接口后
- 双方在遵循共同接口的前提上，均可独立变化而不影响对方

优先使用聚合而不是继承

- 使用继承，会导致一般类的变化影响特殊类
- 使用聚合，设计好整体类和部分类之间的接口，运行时可以根据需要替换部分类对象、整体类和部分类均可独立变化

# 设计模式中的原则

## 正确使用委托

- 一个对象接收一个请求，它要进行一定的逻辑分析，然后决定让哪个（些）对象来进行进一步的计算
- 便于在运行时组合对象的操作以及按需要改变组合

## 找出变化并进行封装

- 使用抽象类和接口，尽量隐蔽：数据、操作、派生类、实例化规则；——共性概念用抽象类表示，变化用具体类实现
- 类的职责要单一，将不同的变化原因封装在不同类；若多个职责总是同时发生改变，则可将它们封装在一个类中

# 设计模式中的原则

## 开闭原则

- 通过扩展增加行为（开放），而不修改代码（封闭）

## 依赖倒转

- 针对高层模块（抽象）编程
- 高层模块起接口的作用、具体类应当只实现接口或抽象类声明的操作，而不要有多余的操作供外部编程使用

# 完

Any questions?