Lecture6 设计模式3

1. 单例模式 Singleton Pattern

定义

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

问题引入

有很多对象，我们仅仅需要全局有一个

- 线程池、缓存、对话框、日志对象、设备驱动程序
- 在许多情况下，实例化多个这样的对象会产生各种各样的问题（例如，不正确的程序行为，资源的过度使用，不一致的结果）

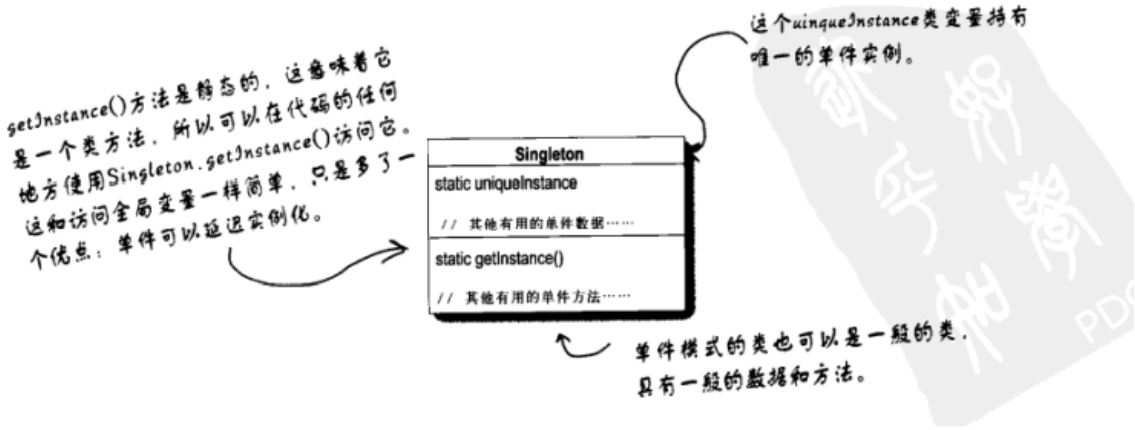
我们只想要 global static 的一个变量

- 单例模式提供了所有的优点而没有缺点

基本上，当你希望应用程序中的每个对象都使用相同的全局资源时，单例模式就会被使用

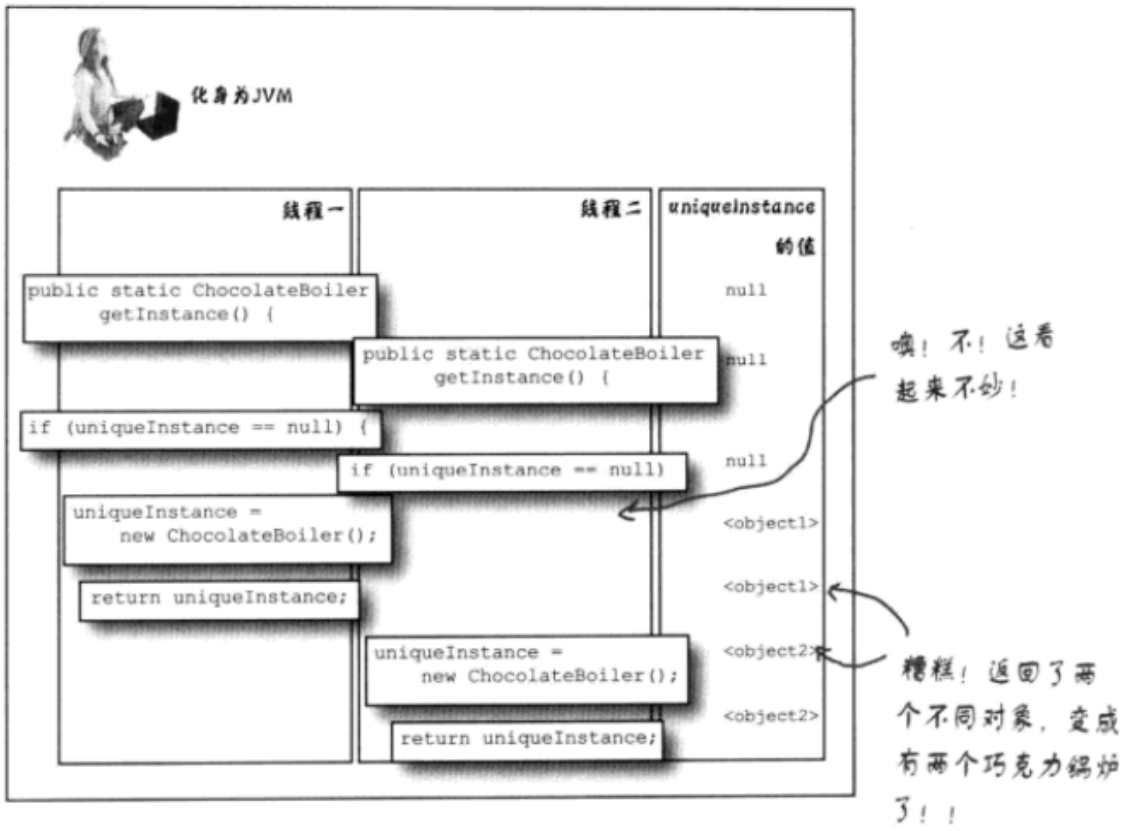
实现

```
public class Singleton {
    private static Singleton uniqueInstance;
    // …
    private Singleton() {}
    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // 其它有用的单例模式方法
}
```



单例模式的问题

- 单例模式，正如我们实现的那样，**不是线程安全**的



处理多线程 —— synchronized

只要把 `getInstance()` 变成同步的方法，多线程灾难几乎可以轻易的解决

```
public class Singleton {
    private static Singleton uniqueInstance;
    // …
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // 其它有用的单例模式方法
}
```

- 但是，其实只有当 uniqueInstance 为 null 的时候，才需要真正的同步，换句话说，一旦设置好了 uniqueInstance 变量的时候，就不需要同步的方法了
- 当然，如果 `getInstance()` 的性能对应用程序不是很关键，就不用考虑了

处理多线程 —— 使用饿汉式，而不是懒汉式的实例化做法

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();
    // …
    private Singleton() {}
    public static Singleton getInstance() {
        return uniqueInstance;
    }
    // 其它有用的单例模式方法
}
```

- 我们依赖 JVM 在加载这个类的时候马上创建唯一的单例实例
- JVM 保证在任何线程访问 uniqueInstance 变量之前，一定先创建此实例

处理多线程 —— 双重检查加锁

利用双重检查枷锁 double-checked locking

- 首先检查实例是否已经创建
- 如果尚未创建，才进行同步

这样只有第一次会同步

```
public class Singleton {
    private static Singleton uniqueInstance;
    // …
    private Singleton() {}
    public static Singleton getInstance() {
        // 检查实例，如果不存在，进入同步区块
```

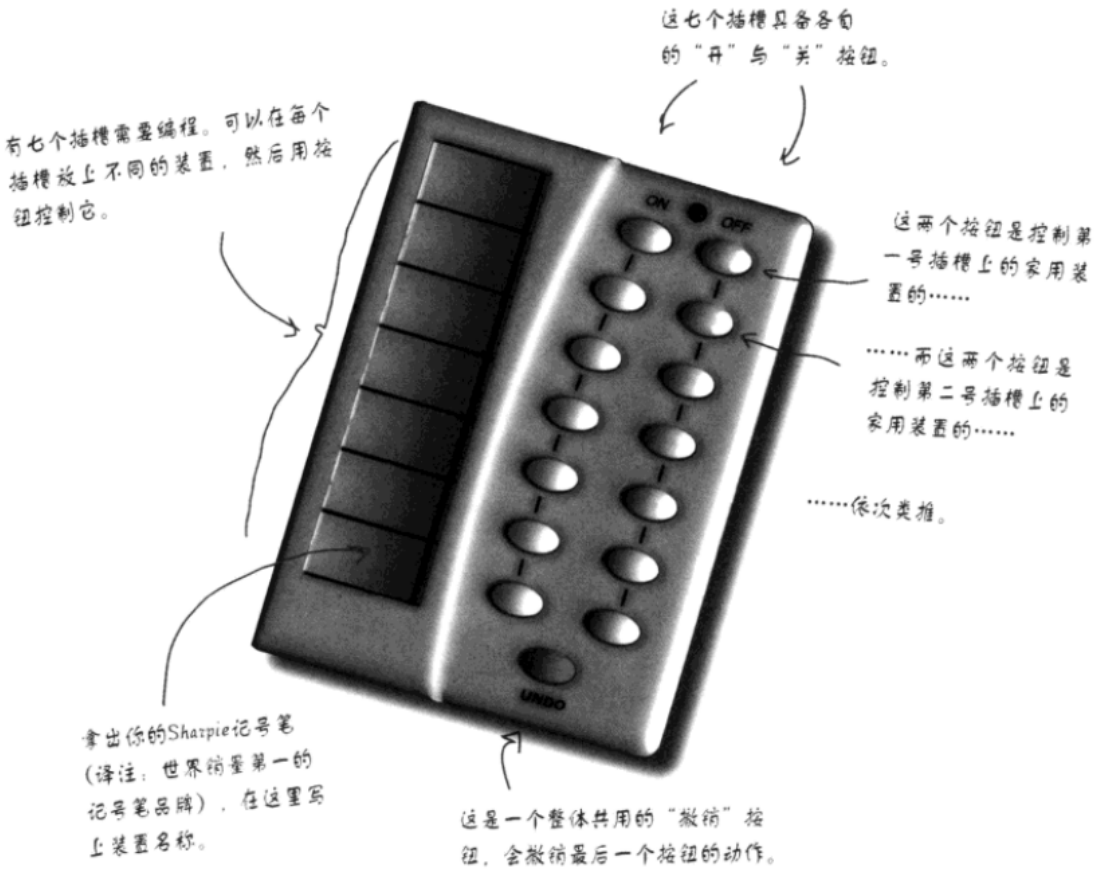
```
        if(uniqueInstance == null){
            synchronized(Singleton.class){
                // 进入区块后，再检查一次，如果仍是null，才创建实例
                if(uniqueInstance == null){
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
    // 其它有用的单例模式方法
}
```

2. 命令模式 Command Pattern

定义

> The Command Pattern encapsulates a **request** as an **object**, thereby letting you parameterize other objects with different requests, queue or log requests, and support **undoable** operations

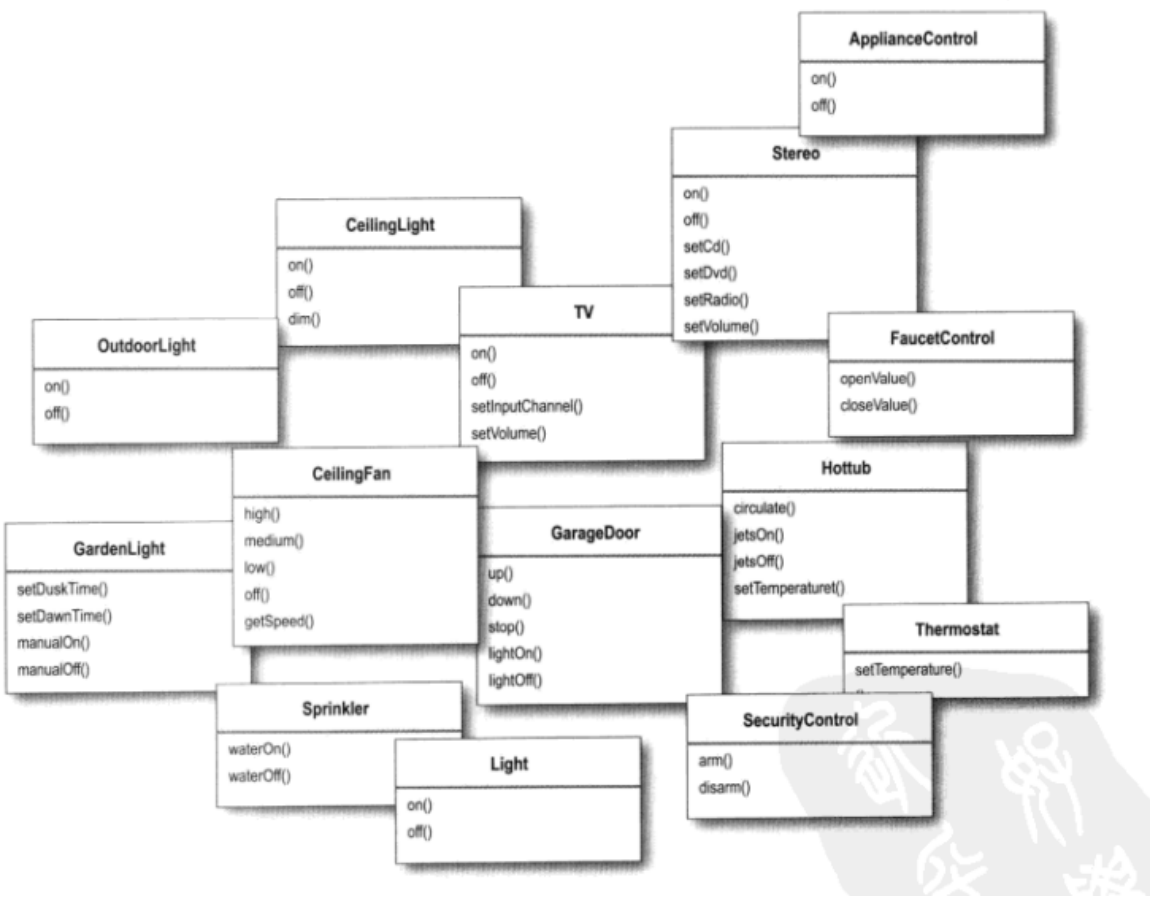
问题引入



- 遥控器有七个可编程的插槽，每个都可以指定到一个不同的家电装置
- 每个插槽都有对应的开关按钮
- 这个遥控器还具备一个整体的撤销按钮

- 希望可以创建一组控制遥控器的 API，让每个插槽都能够控制一个或一组装置

提供一系列厂商控制家电的类

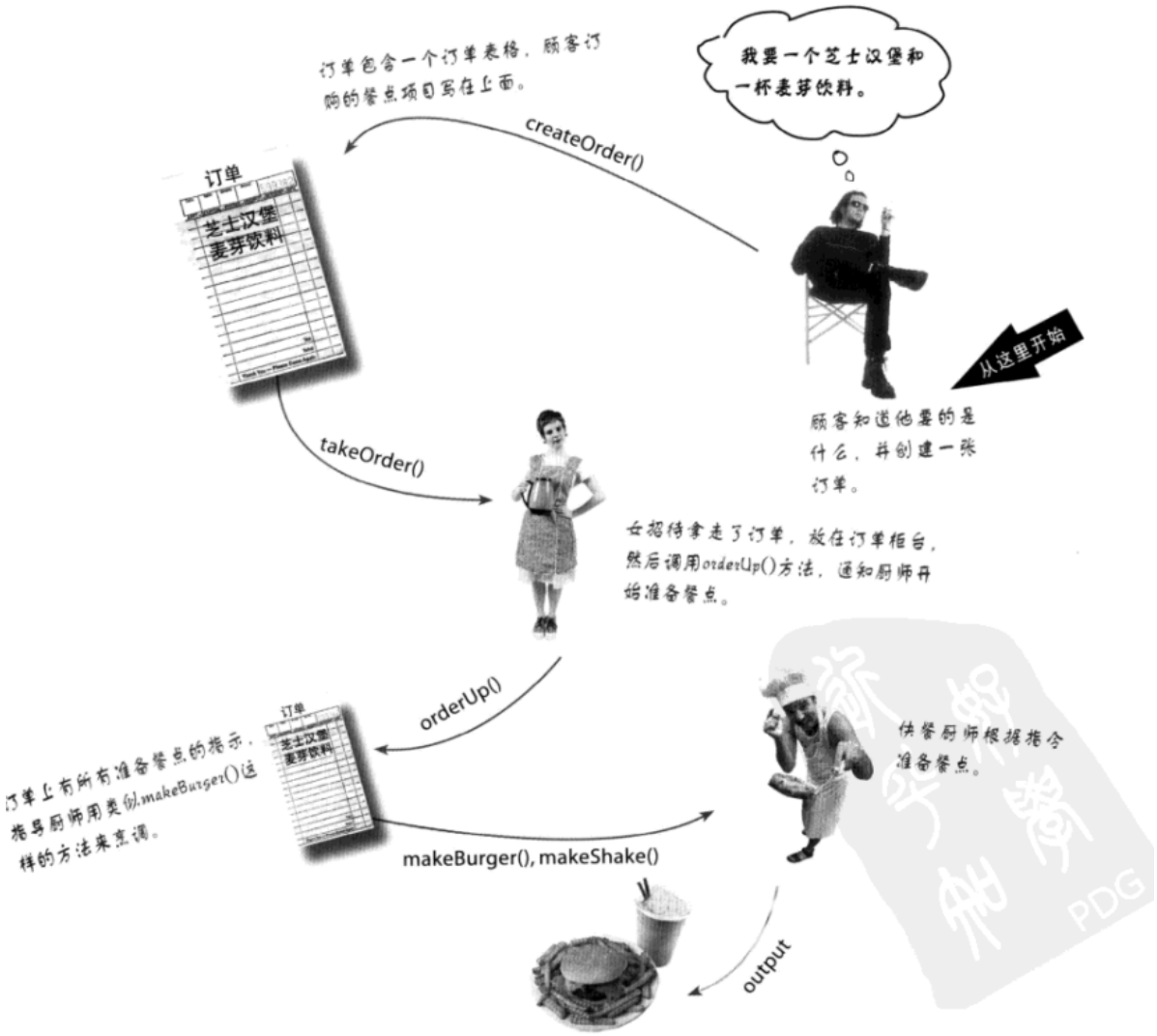


设计原则

命令模式适用于这种情况

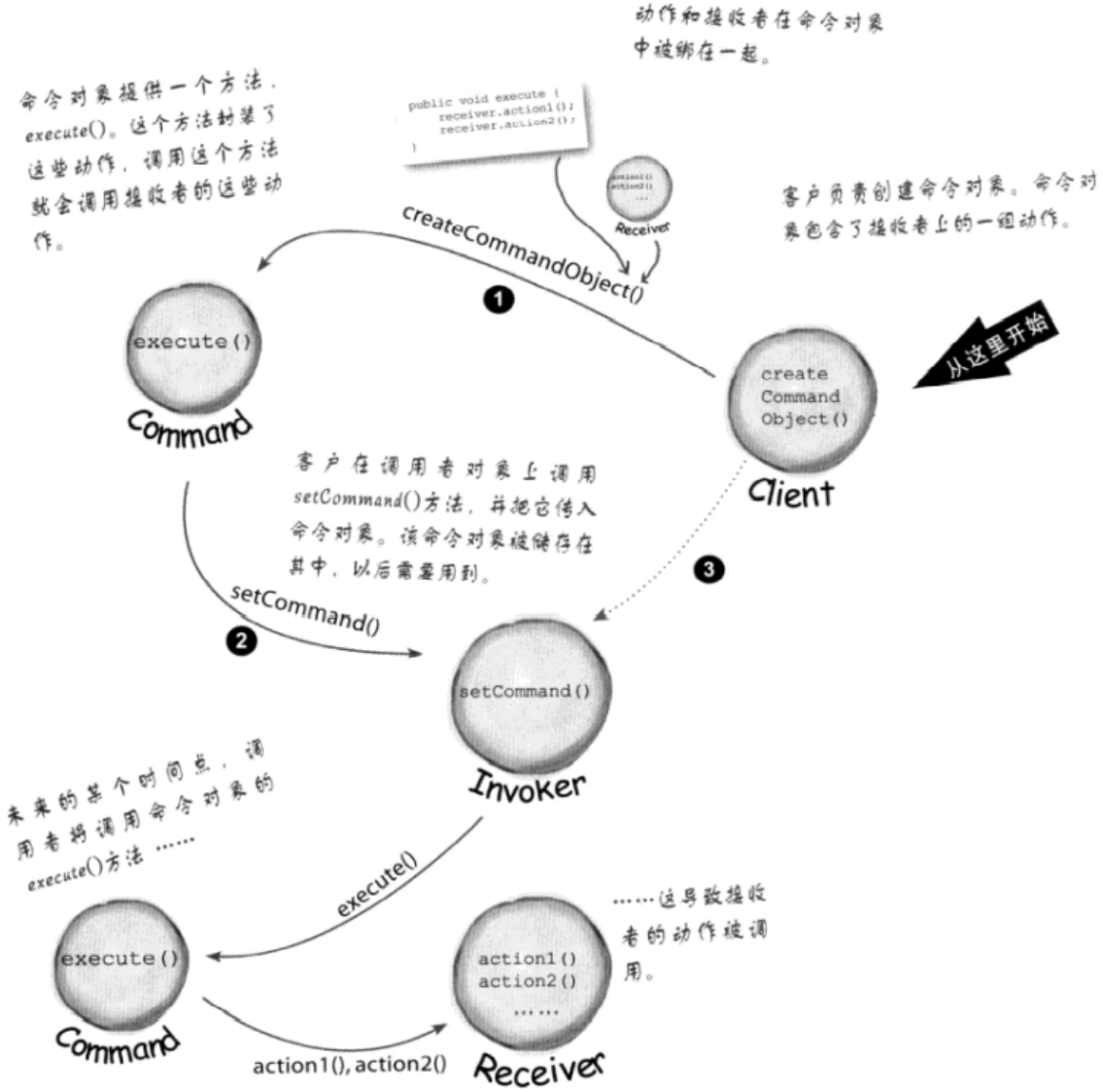
- 命令模式可以将“动作的请求者”从“动作的执行者”对象中解耦
- 利用命令对象，把请求（例如打开电灯）封装成一个特定的对象（客厅电灯对象），如果对每个按钮都存储一个命令对象，那么当按钮被按下的时候，就可以命令对象做相关的工作
- 遥控器并不需要知道工作内容是什么

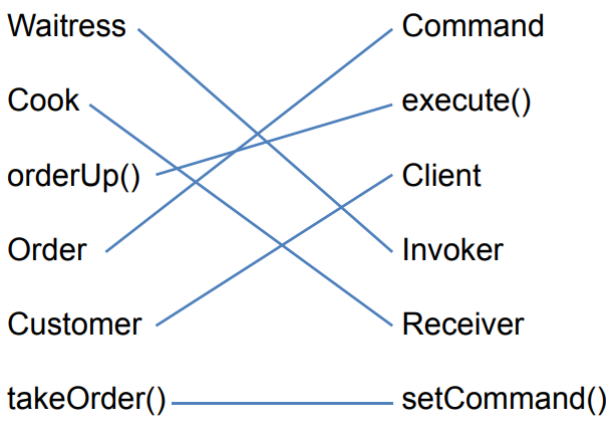
餐厅是怎么工作的



- **订单**
  - 封装了准备餐点的请求
  - 可以被传递，从女招待传递到订单柜台
  - 接口只包含一个方法 `orderUp()`，封装了准备餐点所需要的工作
  - 订单内有一个“需要进行准备工作的对象”（也就是厨师）的引用
  - 这一切都被封装起来
- **女招待**
  - 接受订单，调用订单的 `orderUp()` 方法
  - 女招待不担心订单的内容，或者谁来准备餐点
- **厨师**
  - 知道如何准备餐点
  - 一旦女招待调用 `orderUp()` 方法，快餐厨师就接手
  - 实现需要创建餐点的所有方法

从餐厅到命令模式





实现

Command 接口

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

LightOnCommand 类

```
public class LightOnCommand implements Command{ // 这是一个命令，实现 Command 接口
    Light light;
    // 构造器传入了某个电灯（Reveiver），以便让命令控制
    public LightOnCommand(Light light){
        this.light = light;
    }

    // execute 调用接受对象（电灯）的，让电灯接管执行命令 light.on()
    @Override
    public void execute(){
        light.on();
    }
}
```

使用命令对象

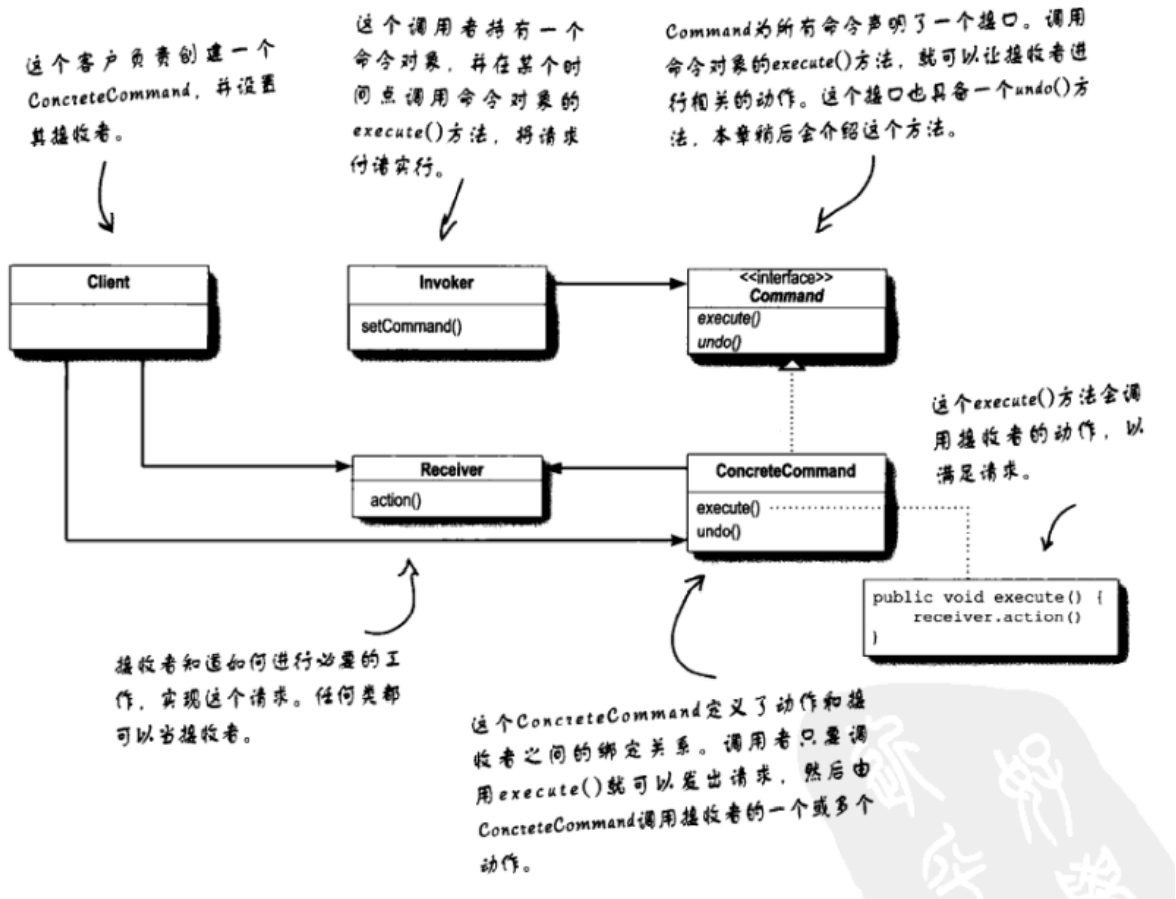
```
public class SimpleRemoteControl {
    Command slot; // 有一个插槽持有命令，这个命令控制着一个装置
    public SimpleRemoteControl(){

    }
    // 插槽可以改变命令的行为
    public void setCommand(Command command){
        slot = command;
    }
    public void onPressBtn(){
        slot.execute();
    }
}
```

遥控器的简单测试

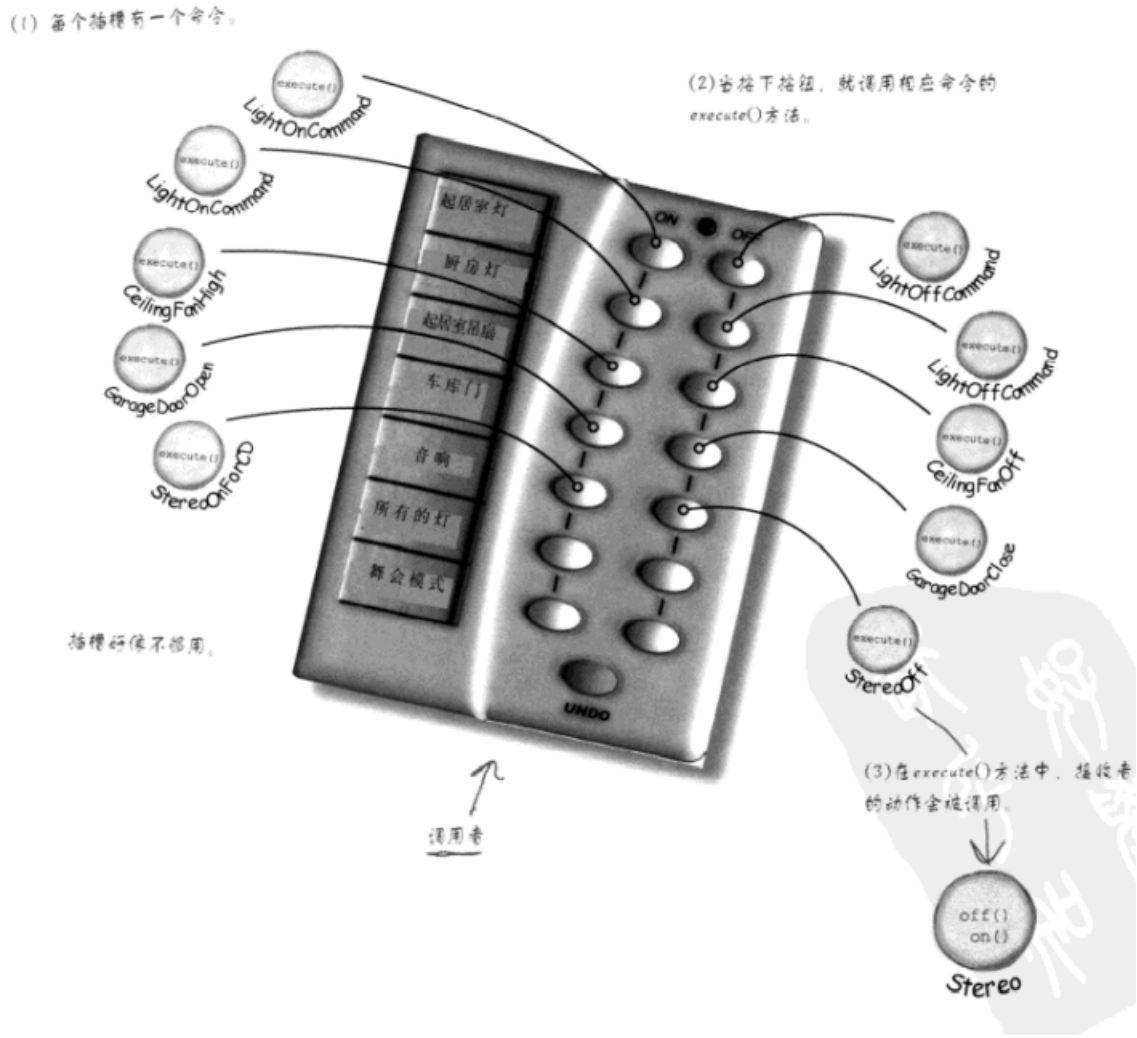
```java
public class RemoteControlTest {

    @Test
    public static void main(String[] args) {
        SimpleRemoteControl remote = new SimpleRemoteControl();
        Light light = new Light();
        LightOnCommand lightOnCommand = new LightOnCommand(light);
        remote.setCommand(lightOnCommand); // 将命令传给调用者
        // 客户端与实现解耦了
        remote.onPressBtn();
    }
```

完成遥控器的设计

定义命令模式



遥控器的实现




```java
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand(int slot, Command onCommand,Command offCommand){
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

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

```

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

- `NoCommand` 对象是一个**空对象**的例子，当你不想返回一个有意义的对象时，空对象就很有用
- 客户端也可以将处理 null 的责任转移给空对象

完成命令撤销的设计

因为我们采用命令模式，所以很容易地加上撤销的功能

Command 接口

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

LightOnCommand 类

如果 `LightOnCommand` 的 `execute()` 方法被调用，那么最后被调用的是 `on()` 的方法，我们知道 `undo()` 需要调用 `off()` 方法进行相反的动作

```
public class LightOnCommand implements Command{
    Light light;
    // 构造器传入了某个电灯（Reveiver），以便让命令控制
    public LightOnCommand(Light light){
        this.light = light;
    }

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

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

RemoteControl 类

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;
```

```
    Command undoCommand; // 前一个命令将被记录在这里

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
        undoCommand = noCommand; // 开始没有所谓的“前一个命令，所以它被空对象替代
    }

    public void setCommand(int slot, Command onCommand,Command offCommand){
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    // 按下按钮时，我们记录 undoCommand 为当前按下的，为了以后能够撤销
    public void onButtonWasPushed(int slot){
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void offButtonWasPushed(int slot){
        offCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    // 当按下撤销按钮，调用 undoCommand 的 undo 撤销上一步动作
    public void undoButtonWasPushed(){
        undoCommand.undo();
    }
}
```

宏命令

如果拥有一个遥控器，光凭按下一个按钮，就能同时调暗灯光，打开音响和电视，设置好 DVD，并让热水器开始加温

PartyCommand 类

```
public class PartyOnCommand implements Command{
    Command[] commands;
    public MacroCommand(Command[] commands){
        this.commands = commands;
    }
    // 宏命令被遥控器执行时，依次执行数组里的所有命令
    public void execute(){
        for(Command cmd : commands){
            cmd.execute();
        }
    }
}
```

最后将宏命令指定给我们希望的按钮即可

```
remoteControl.setCommand(0, partyOnCommand, partyOffCommand);
```

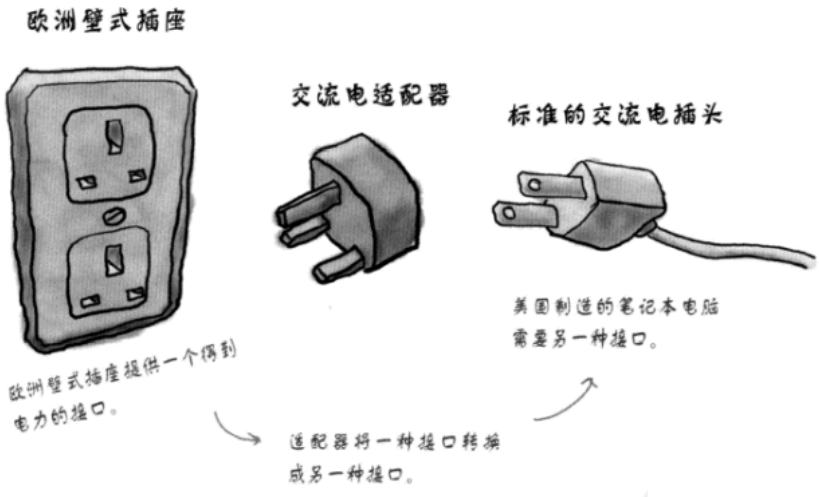
3. 适配器模式 Adapter Pattern

定义

> The Adapter Pattern converts the interface of a class into another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

问题引入

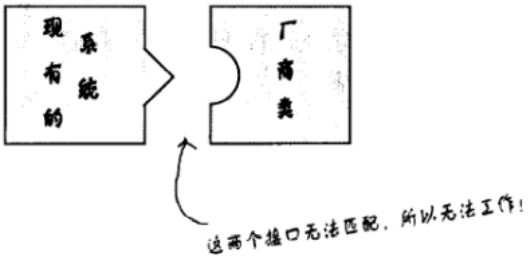
如果你去欧洲国家旅行，使用美国制造的笔记本电脑，可能需要使用一个交流电适配器



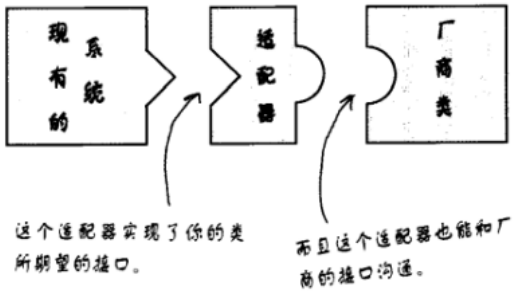
- 适配器的作用就是改变了插座的接口，以符合不同插头的笔记本的需求

面向对象适配器

假设已经有一个软件系统，你希望它能和一个新的厂商类库搭配使用，但是这个新厂商所设计出来的接口，不同于旧厂商的接口



如果不想改变现有的代码，为了解决这个问题（你也不能改变厂商的代码），此时就应该写个类，将新厂商的接口转换成你所期望的接口



实现

还记得策略模式的 Duck 吗

Duck 接口

```
public interface Duck{
    public void quack();
    public void fly();
}
```

MallardDuck 类

```
public class MallardDuck implements Duck{
    @Override
    public void quack(){
        System.out.println("Quack");
    }
    @Override
    public void fly(){
        System.out.println("I'm flying")
    }
}
```

Turkey 接口

现在出现了最新的火鸡接口

```
public interface Turkey{
    public void gobble(); // 火鸡不会呱呱叫，只会咯咯叫
    public void fly(); // 火鸡会飞，虽然飞不远
}
```

WileTurkey 类

```
public class WileTurkey implements Turkey{
    @Override
    public void gobble(){
        System.out.println("gobble gobble");
    }
    @Override
    public void fly(){
        System.out.println("I'm flying a short distance")
    }
}
```

现在，假设你缺鸭子对象，想要用一些火鸡来冒充

显然，因为火鸡的接口不同，不能公然拿来用

TurkeyAdapter 类

```
public class TurkeyAdapter implements Duck{
    Turkey turkey;
    public TurkeyAdapter(Turkey turkey){
        this.turkey = turkey;
    }
    // quack() 在类之间的转换很简单，只需要调用gobble()就可以了
    @Override
    public void quack(){
        turkey.gobble();
    }
    // 虽然两个接口都具备flu()方法
    // 但是火鸡的飞行距离很短，不能像鸭子一样长途飞行
    // 要想让火鸡飞行与鸭子对应，必须连续五次调用火鸡的fly()
    @Override
    public void fly(){
        for(int i = 0; i < 5; i++){
            turkey.fly();
        }
    }
```

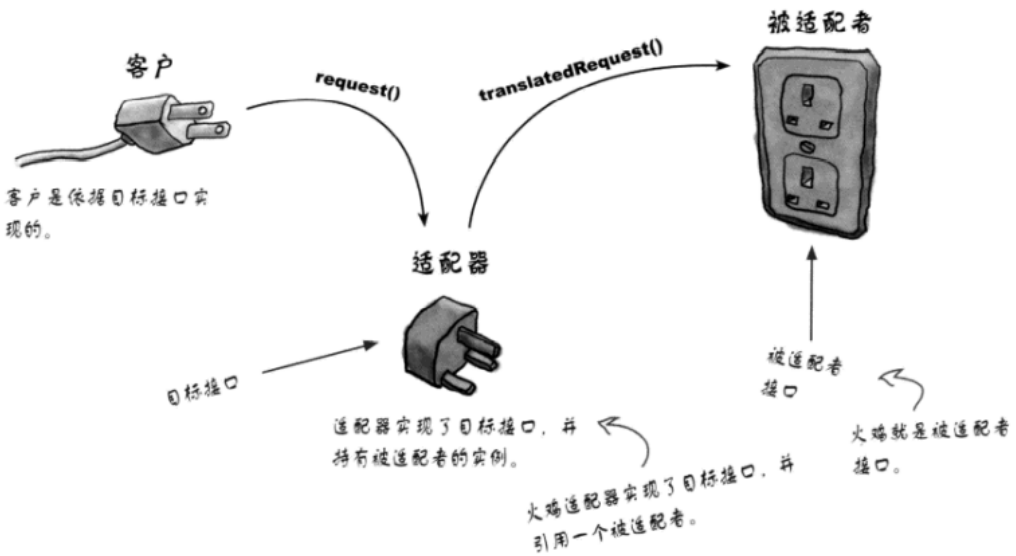
```
20  }
```

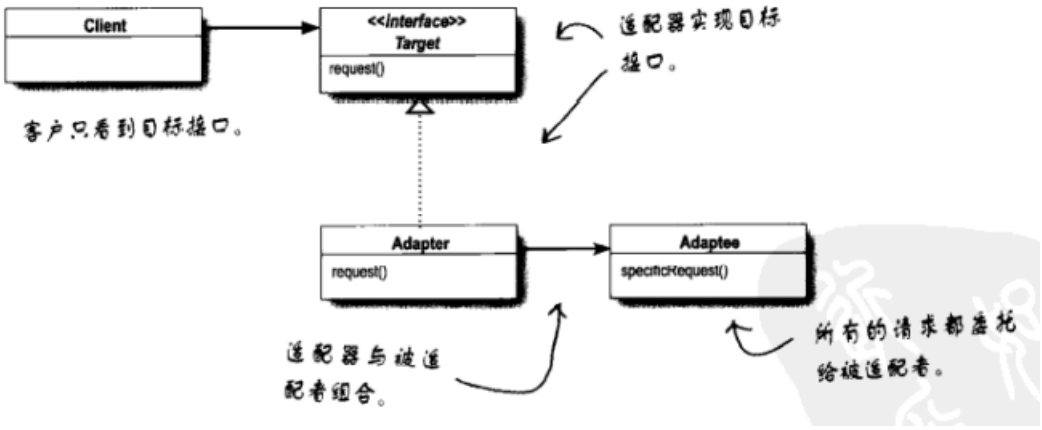
测试类

```
1   public class DuckTestDrive{
2       public static void main(String[] args){
3           MallardDuck duck = new MallardDuck();
4           WildTurkey turkey = new WileTurkey();
5           Duck turkeyAdapter = new TurkeyAdapter(turkey);
6
7           System.out.println("The Turkey says...");
8           turkey.gobble();
9           turkey.fly();
10
11          System.out.println("\nThe Duck says...");
12          testDuck(duck);
13
14          System.out.println("\nThe TurkeyAdapter says...");
15          testDuck(turkeyAdapter);
16      }
17
18      static void testDuck(Duck duck){
19          duck.qack();
20          duck.fly();
21      }
22  }
```

适配器模式解析

我们来看一下各个部分的关系





客户使用适配器的过程如下

- 客户通过目标接口调用适配器的方法对适配器发出请求
- 适配器使用被适配者接口把请求转换成被适配者的一个或者多个调用接口
- 客户收到调用的结果，但未察觉这一切都是适配器在起转换作用

一个适配器需要做多少“适配”的工作?

- 这取决于特定的情况和特定的接口，它可能只是基本的转换或大量的工作

一个适配器只能够封装一个类吗?

- 虽然大部分适配器的工作是将一个接口转换成另一个，但是真实情况确实很复杂
- 这涉及到另一个模式，被称为**外观模式 Façade Pattern**

4. 外观模式 Façade Pattern

定义

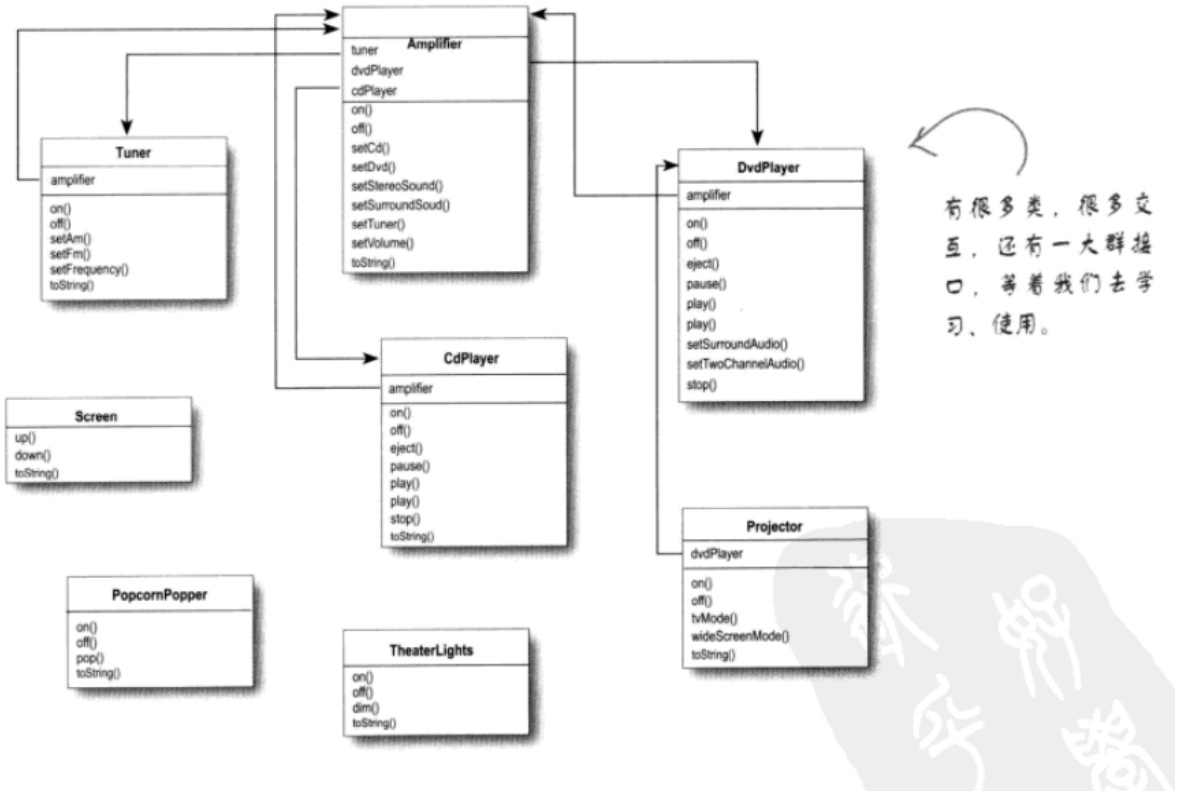
> The Façade Pattern provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

背景

跟适配器不同，我们现在要看到的改变接口的新模式，它改变接口的原因是为了简化接口，它将**一个或者数个类的复杂的一切都隐藏在背后**，只露出一个干净美好的外观

家庭影院

假设我们要建立一个自己的家庭影院

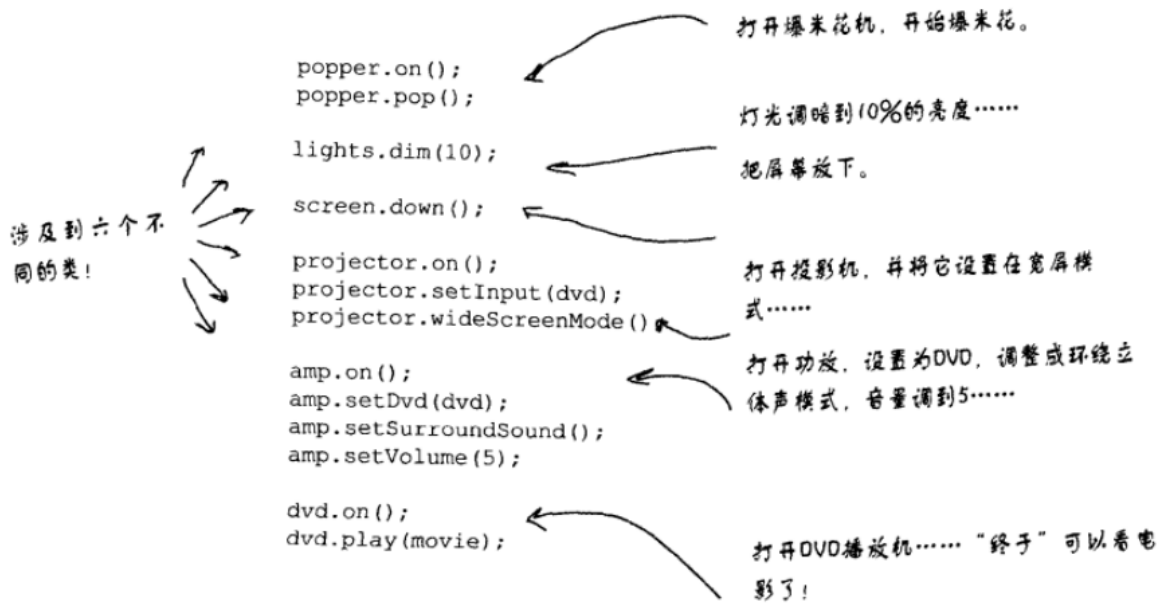


你花费了很久布置，连接所有装置进行微调，此时终于准备观赏电影

在观赏电影前，你还必须执行一系列行动

1. 打开爆米花机
2. 开始爆米花
3. 将灯光调暗
4. 放下屏幕
5. 打开投影机
6. 将投影机的输入切换到DVD
7. 将投影机设置在宽屏模式
8. 打开功放
9. 将功放的输入设置为DVD
10. 将功放设置为环绕立体声
11. 将功放音量调到中（5）
12. 打开DVD播放器
13. 开始播放DVD

如果将这些任务写成类和方法调用的话

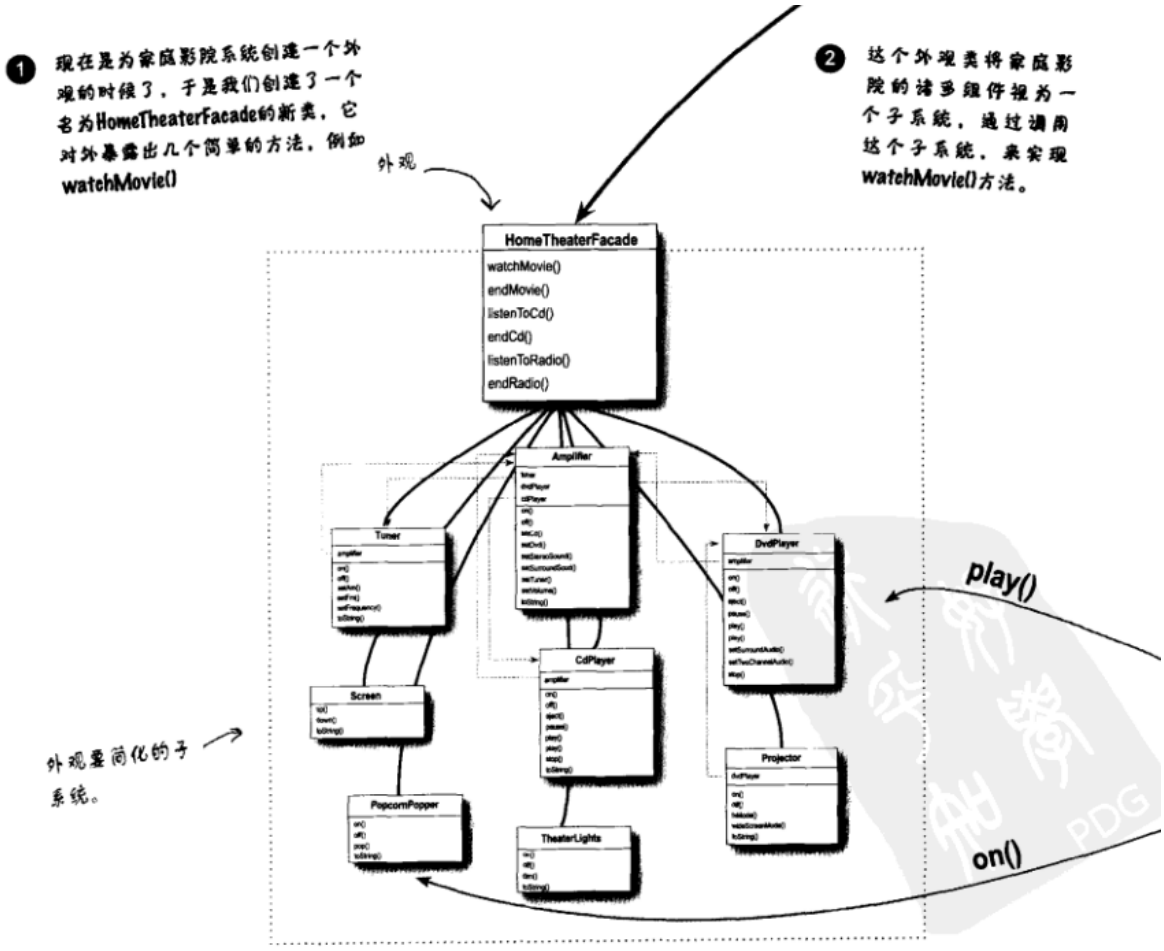


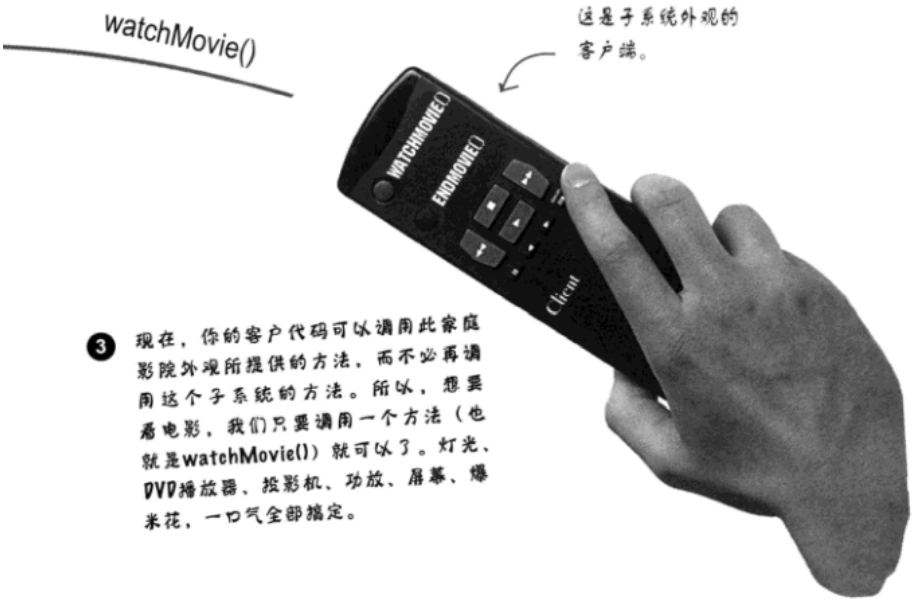
不只是这样，如果你还要做

- 看完电影后，把一切都关掉?
- 如果要听 CD 或者广播?
- 如果决定升级你的系统?

实现

你真正需要的是一个外观，通过实现一个提供更合理接口的外观类，使一个复杂的子系统变得容易使用





- 外观模式只是提供更直接的操作，并未将原来的子系统阻隔起来
- 如果需要用子系统类更高层的功能，仍然可以使用原来的子系统

适配器模式和外观模式的区别

两种模式的差异，不在于它们“包装”了几个类 / 接口，它们都可以只包装一个，或者多个

- 适配器模式：“改变”接口符合客户的期望
- 外观模式：提供子系统的一个简化的接口

HomeTheaterFacade 类

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                Tuner tuner,
                DvdPlayer dvd,
                CdPlayer cd,
                Projector projector,
                Screen screen,
                TheaterLights lights,
                PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    // 其他的方法

}
```

这就是组合：我们会用到的子系统组件全部都在这里。

外观将子系统中每一个组件的引用都传入它的构造器中。然后外观把它们赋值给相应的实例变量。

这部分的代码，等一下就会填进去……

实现简化的接口

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}

public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

watchMovie()将我们之前手动进行的每项任务依次处理。请注意，每项任务都是委托子系统中相应的组件处理的。

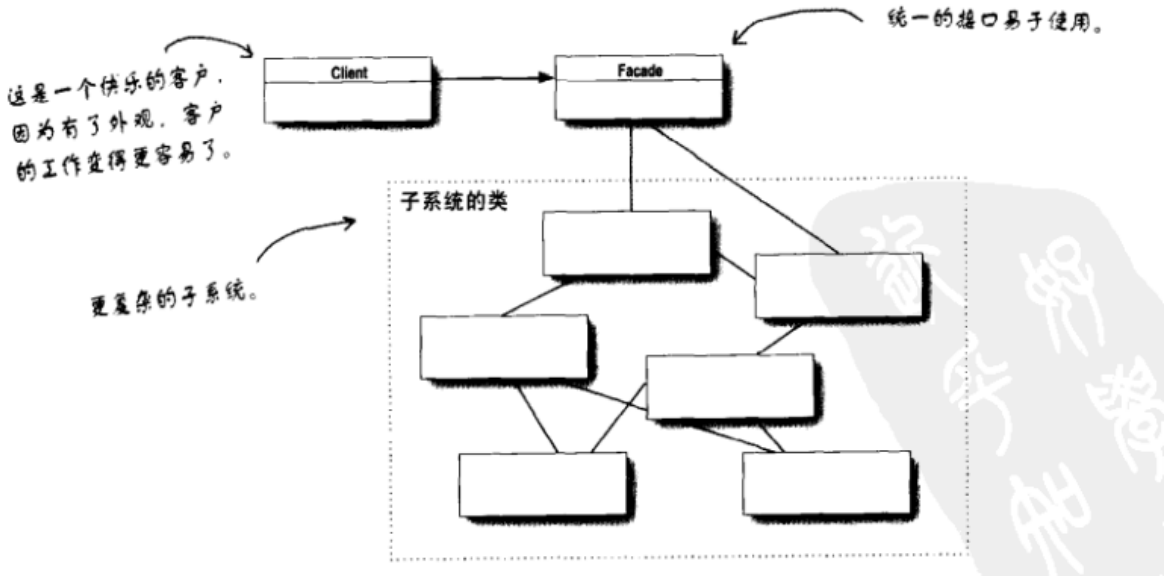
而endMovie()负责关闭一切。每项任务也都是委托子系统中合适的组件处理的。

测试类

```java
public class HomeTheaterTestDrive{
    public static void main(String[] args){
        // ... 在这里实例化组件
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(amp, tuner, dvd, cd, projector, screen, lights, popper);
        // 使用简化的接口
        homeTheater.watchMovie("Radiers of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

设计原则

整体来看，外观模式的意图是要提供一个简单的接口，好让一个子系统更加易于使用



> Principle of Least Knowledge: Only talk to your closest friends.
> 最少知识原则：只和你的密友谈话

- 当你设计一个系统，不管是任何对象，都要注意它所交互的类有哪些，并且注意这些类是如何交互的
- 这个原则希望我们在设计中，不要让太多的类耦合在一起，避免修改系统的一部分，会影响到另外一部分

如何满足最少知识原则，就任何对象而言，在该对象的方法内，我们应该只调用属于以下范围的方法

- 该对象本身
- 被当作方法的参数而传递进来的对象
    - 如果某个对象是调用其它方法的返回结果，不要调用该对象的方法
- 此方法所创建或实例化的任何对象
- 对象的任何组件
    - Has-A 关系的组件

5. 桥接模式 Bridge Pattern

定义

> Bridge Pattern are used to decouple abstractions from implementations so that they can vary independently. This type of design pattern is structural in that it decouples abstraction and implementation by providing a bridge between them.

问题引入

在画图中考虑两个维度

Color/Shape	圆形	方形
红色	红色圆形	红色方形
绿色	绿色圆形	绿色方形

设计原则

还记得**多用组合，少用继承**的设计原则吗？

- 现在你有了两个独立的类继承层次结构：一个用于抽象，一个用于实现
  - 两者之间的 `has-a` 关系就是是 “桥”
- 优点
  - 解耦一个实现，这样它就不会永久地绑定到一个不变的接口
  - 抽象和实现可以独立扩展
  - 对具体抽象类的更改不会影响客户端
- 使用
  - 适用于需要在多个平台上运行的图形和窗口系统
  - 在需要以不同的方式改变接口和实现时非常有用
- 缺点
  - 增加复杂性

6. 享元模式 Flyweight Pattern

定义

> The Flyweight Pattern is mainly used to reduce the number of objects created to reduce memory footprint and improve performance. This type of design pattern is a structural pattern that provides a way to reduce the number of objects to improve the object structure required by the application.

问题引入

我们创建了一个画圆的工厂，它可能会产生不同颜色的圆形

我们对不同颜色的圆形应用可能不止一次，如果绘制了 10000 个圆形，那么可能会创建大量的对象

为了避免内存溢出，可以把其中功能共同的部分抽离出来，比如说指定颜色的圆形本身，下次再有相同的请求，直接返回内存中已有的对象，避免重复创建

实现

Shape 接口

```
1  public interface Shape {
2     void draw();
3  }
```

Circle 实现类

```
public class Circle implements Shape {
   private String color;
   private int x;
   private int y;
   private int radius;

   public Circle(String color){
      this.color = color;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
   }

   public void setRadius(int radius) {
      this.radius = radius;
   }

   @Override
   public void draw() {
      System.out.println("Circle: Draw() [Color : " + color
         +", x : " + x +", y :" + y +", radius :" + radius);
   }
}
```

ShapeFactory 类

```
import java.util.HashMap;

public class ShapeFactory {
    // 用来保存可以复用的对象
   private static final HashMap<String, Shape> circleMap = new HashMap<>();

   public static Shape getCircle(String color) {
       Circle circle = (Circle)circleMap.get(color);
       // 如果内存中没有，才创建
       if(circle == null) {
           circle = new Circle(color);
           circleMap.put(color, circle);
           System.out.println("Creating circle of color : " + color);
       }
       // 如果有，复用
       return circle;
```

```
    }
}
```

测试类

```
public class FlyweightPatternDemo {
   private static final String colors[] =
      { "Red", "Green", "Blue", "White", "Black" };
   public static void main(String[] args) {

      for(int i=0; i < 20; ++i) {
         Circle circle =
            (Circle)ShapeFactory.getCircle(getRandomColor());
         circle.setX(getRandomX());
         circle.setY(getRandomY());
         circle.setRadius(100);
         circle.draw();
      }
   }
   private static String getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   private static int getRandomX() {
      return (int)(Math.random()*100 );
   }
   private static int getRandomY() {
      return (int)(Math.random()*100);
   }
}
```

应用实例

- Java 中的 `String`，字符串常量池即为：如果有则返回，如果没有则创建一个字符串保存在字符串缓冲池里面
- 数据库中的数据池
- 当系统中有大量**相似对象**，或者需要**缓冲池**的的时候