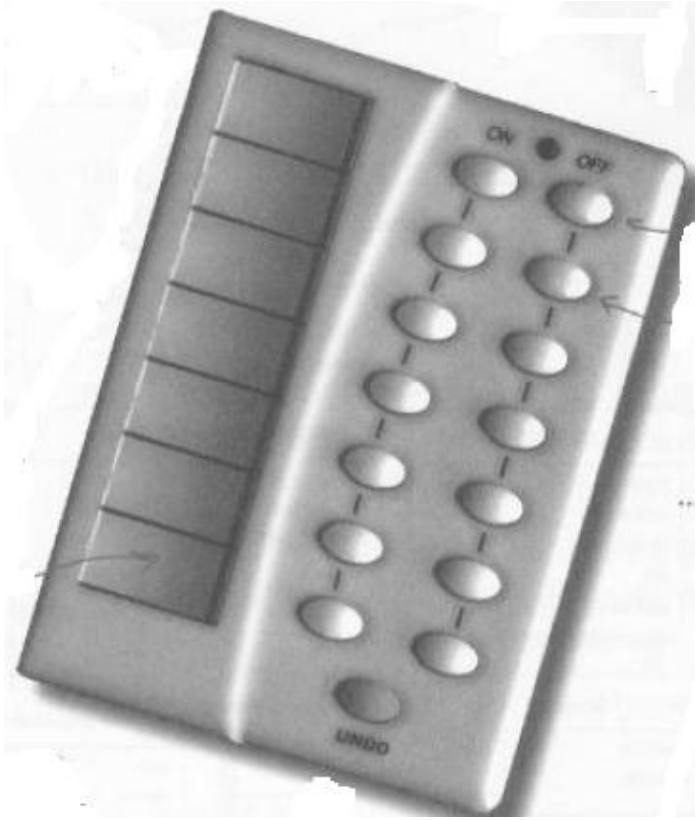
命令模式

孙熙然

例子：这次我们将设计一个家电自动化遥控器的API



- 1）、遥控器上具有七个可编程的插槽、七个开关按钮和一个整体撤销按钮。
- 2）、通过创建一组API，使插槽可以控制一个或一组家电装置，例如电灯、电风扇、热水器、音响设备和其他类似的可控制装置。
- 3）、插槽还可以控制未来可能出现的家电装置。
- 4）、整体撤销按钮具有撤销上一个命令的功能。

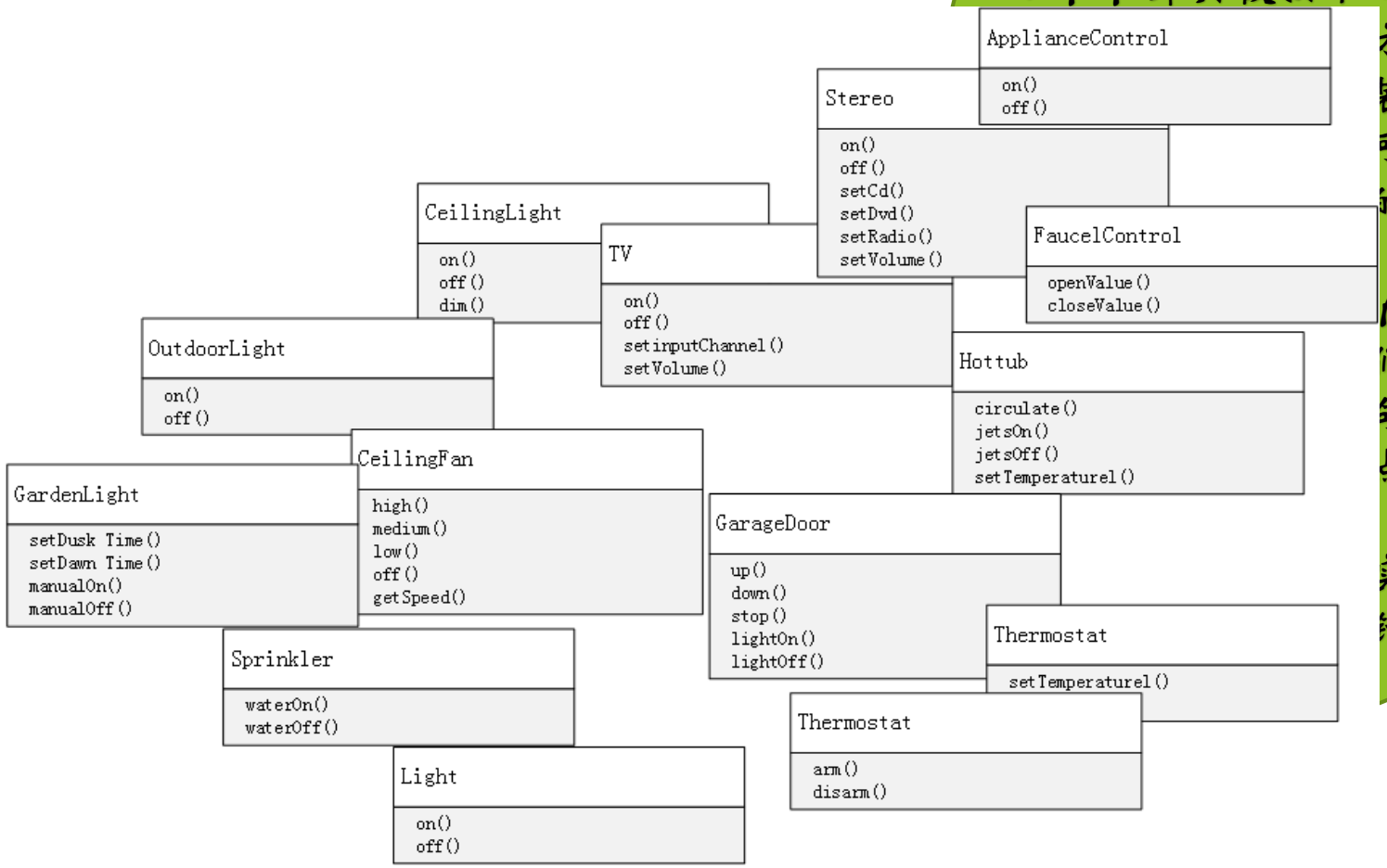
看一下厂商的类

1）、我们将设计一系列类，这些类都具有ON()和Off()方法。

2）、当遥控器上的on或off开关被按下时，某些方法被调装置，但可以被改的东西可

f f开关门通过i句加以选点需求我

实现，上针对的是

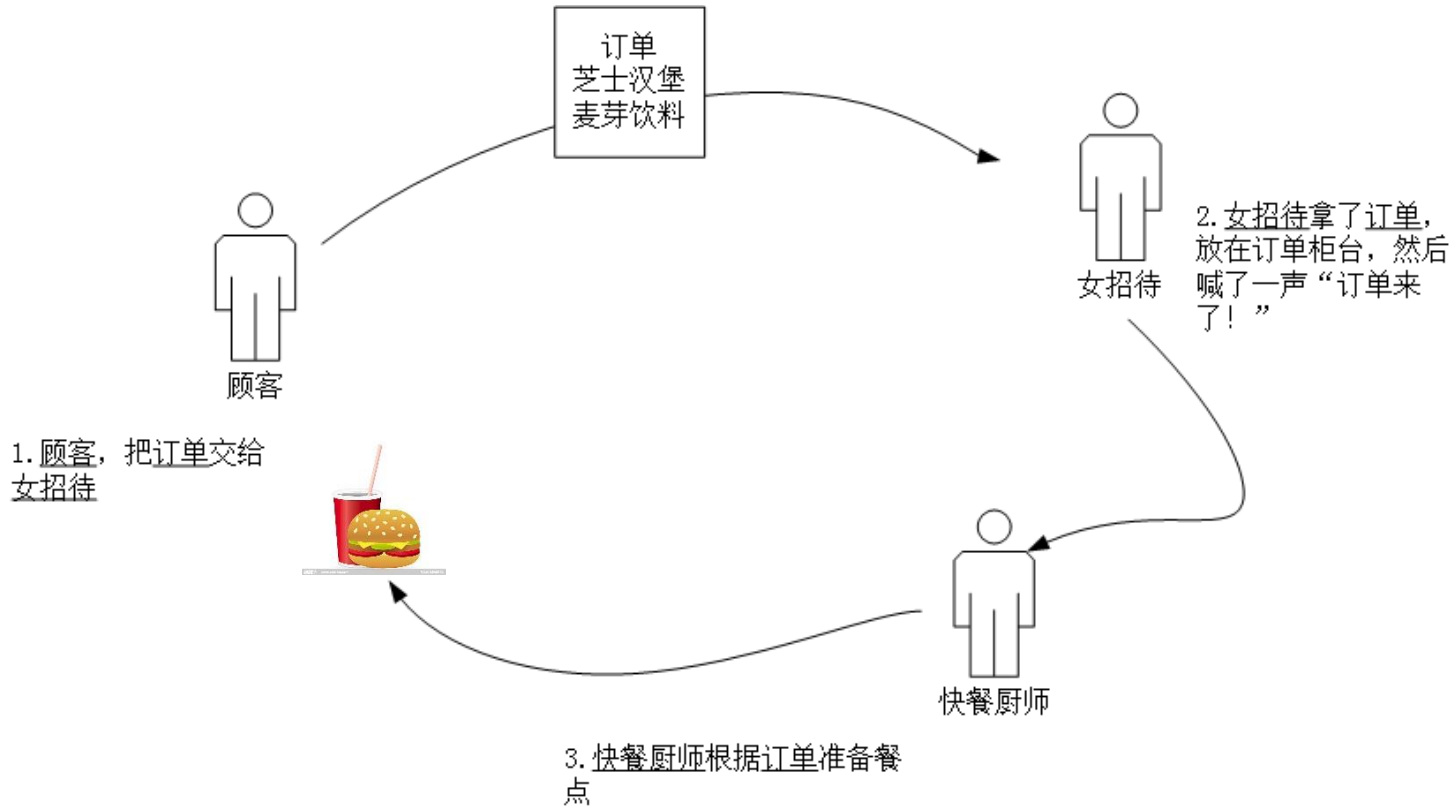


进一步思考：

- 这次设计中不变的是遥控器（或者说是遥控器的按钮），变化的是家电装置，例如第一排开关可以控制电灯，也可以控制电风扇，或者未来可能出现的家电。
- 所以我们需要对遥控器和家电装置进行解耦。
- 此时我们想到了命令模式：遥控器（或者说遥控器上的按钮）就是命令的请求者，家电装置就是命令的执行者，我们所要做的就是将命令的请求者和命令的执行者解耦。
- 利用命令对象，把请求(打开点灯)封装成一个特定对象(客厅点灯对象)，如果对每个按钮都存一个命令对象，那么当按钮被按下的时候，就可以请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，只要有个命令对象能和正确的对象沟通，把事情做好久可以了。

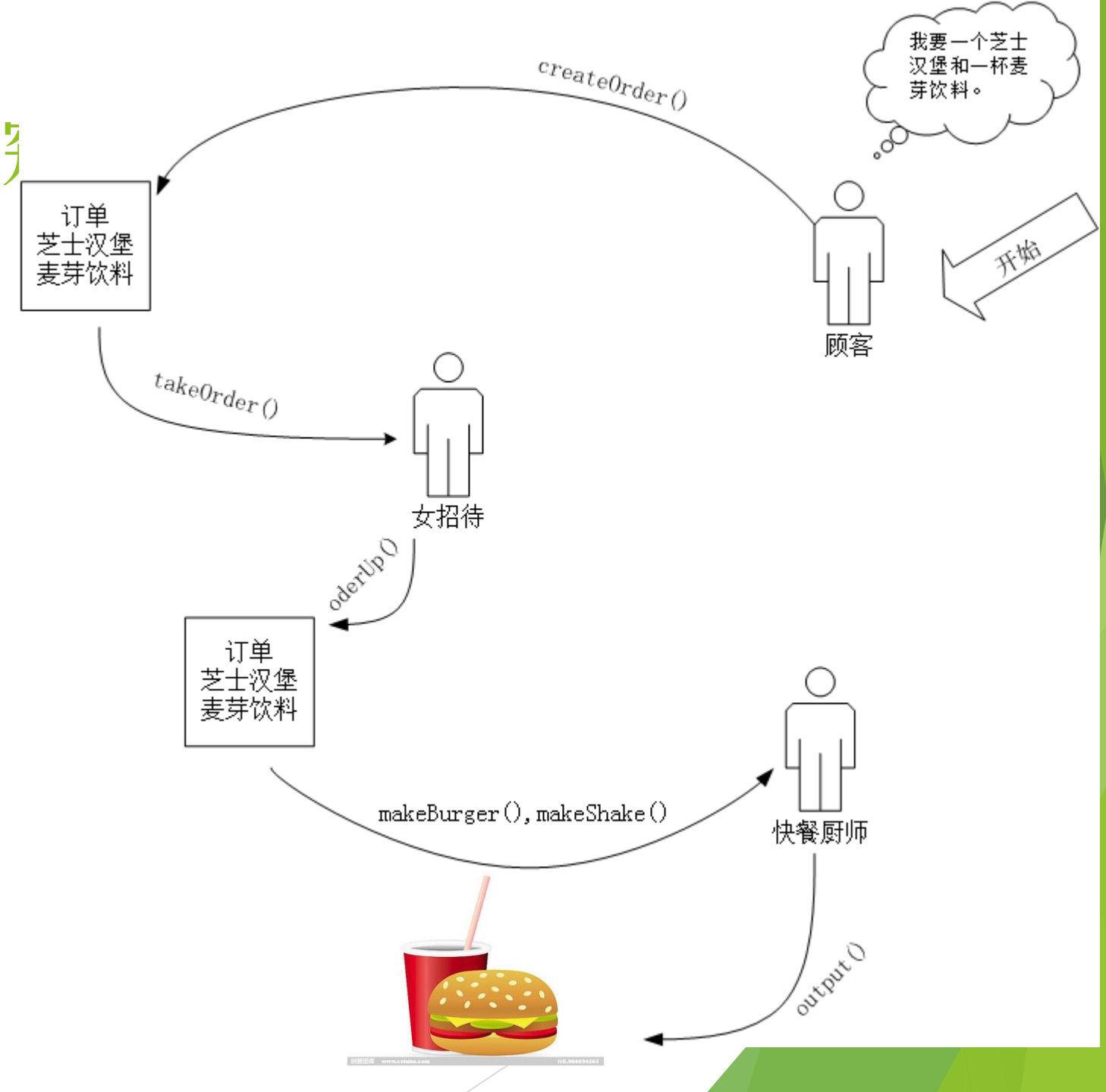
餐厅可以帮助我们了解命令模式

- 餐厅解析：研究顾客，女招待，订单，以及快餐厨师之间的交互。



让我们更详细地研究

- 思考对象和方法的调用关系



对象餐厅的角色和职责

- **一张订单封装了准备餐点的请求。**

把订单想象成一个用来请求准备餐点的对象，和一般的对象
从女招待传递到订单柜台，或者从女赵丹传递到阶梯下一班
含一个方法就是orderUp()。这个方法封装了准备从哪点所
“需要进行准备工作的对象”(也就是厨师)的引用。这一切
需要知道订单上有什么，也不需要知道是谁来准备餐点。

- **女招待的工作是接收订单，然后调用订单的ord**

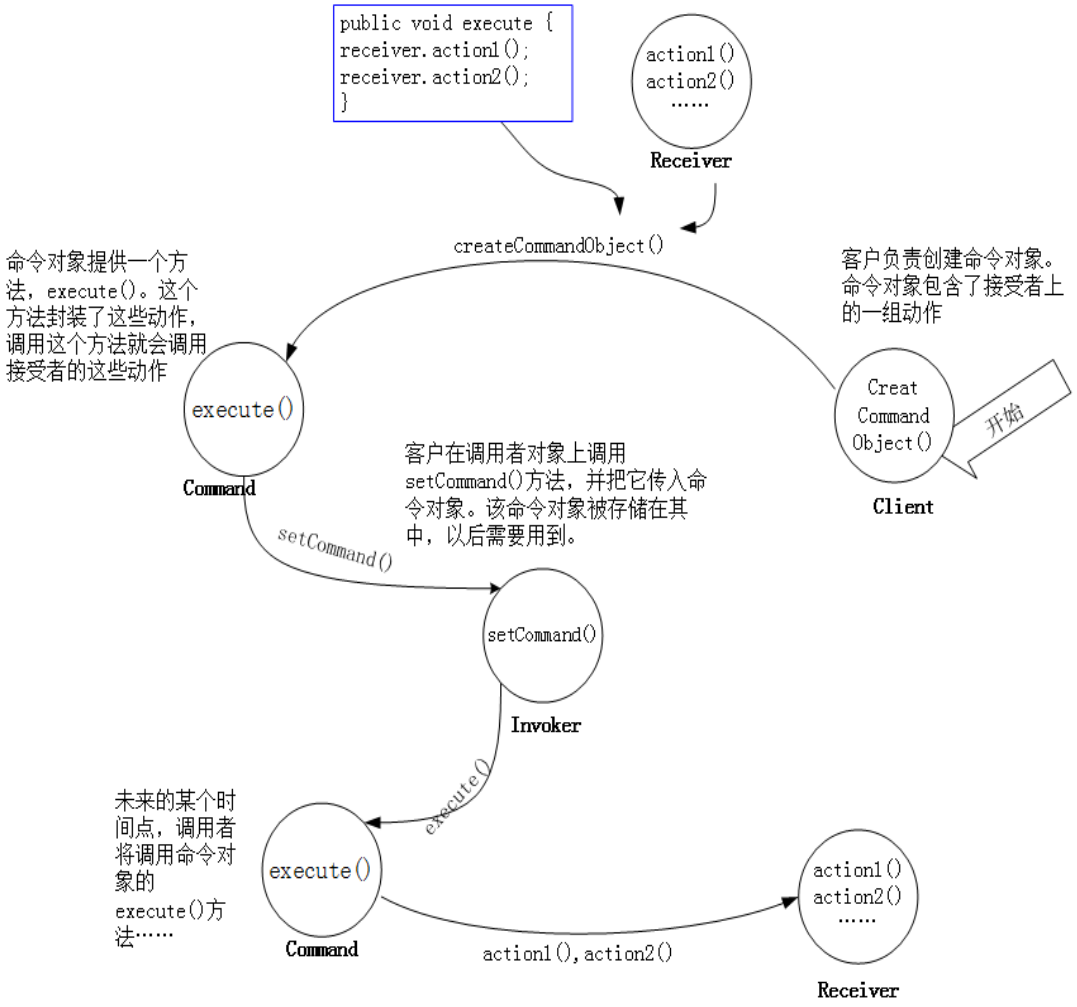
女招待其实不必担心订单的内容是什么，或者由谁来准
订单有一个orderUp()方法可以调用，这就够了。

- **快餐厨师具备准备餐点的知识。**

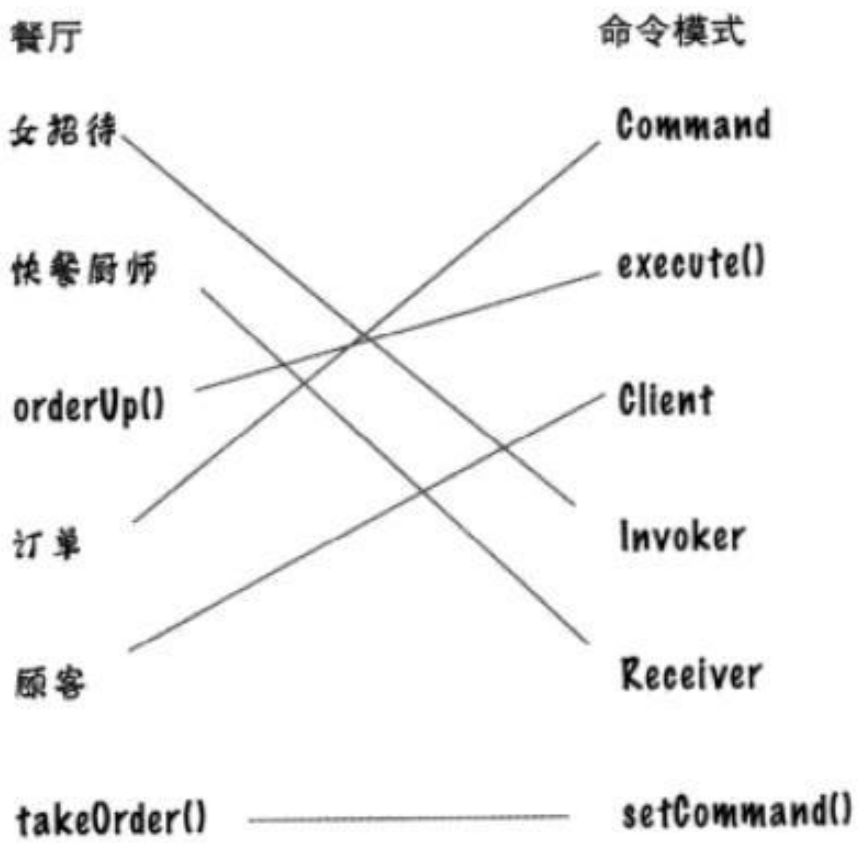
快餐厨师是一种对象，他真正知道如何准备餐点，一旦女找到调
法，快餐厨师就接手，实现需要创建餐点的所有方法，女找到和
底的解耦，女招待的订单封装了餐点的细节，厨师只要调用每个订 方法即
可。

把餐厅想成是OO设计模式的一种模型，而这个模型允许将“发出请求的对象”和“接收与执行这些请求的对象”分隔开来。对于遥控器API，我们需要分隔开“发送请求的按钮代码”和“执行请求的厂商特定对象”。

从餐厅到命令模式


public void execute {
receiver.action1();
receiver.action2();
}
action1()
action2()
……
Receiver
createCommandObject()
命令对象提供一个方法，execute()。这个方法封装了这些动作，调用这个方法就会调用接受者的这些动作
客户负责创建命令对象。命令对象包含了接受者上的一组动作
execute()
Command
Creat Command Object()
开始
Client
客户在调用者对象上调用setCommand()方法，并把它传入命令对象。该命令对象被存储在其中，以后需要用到。
setCommand()
setCommand()
Invoker
execute()
未来的某个时间点，调用者将调用命令对象的execute()方法……
execute()
Command
action1(),action2()
action1()
action2()
……
Receiver

请将餐厅的对象和方法对应到命令模式的相应名称。

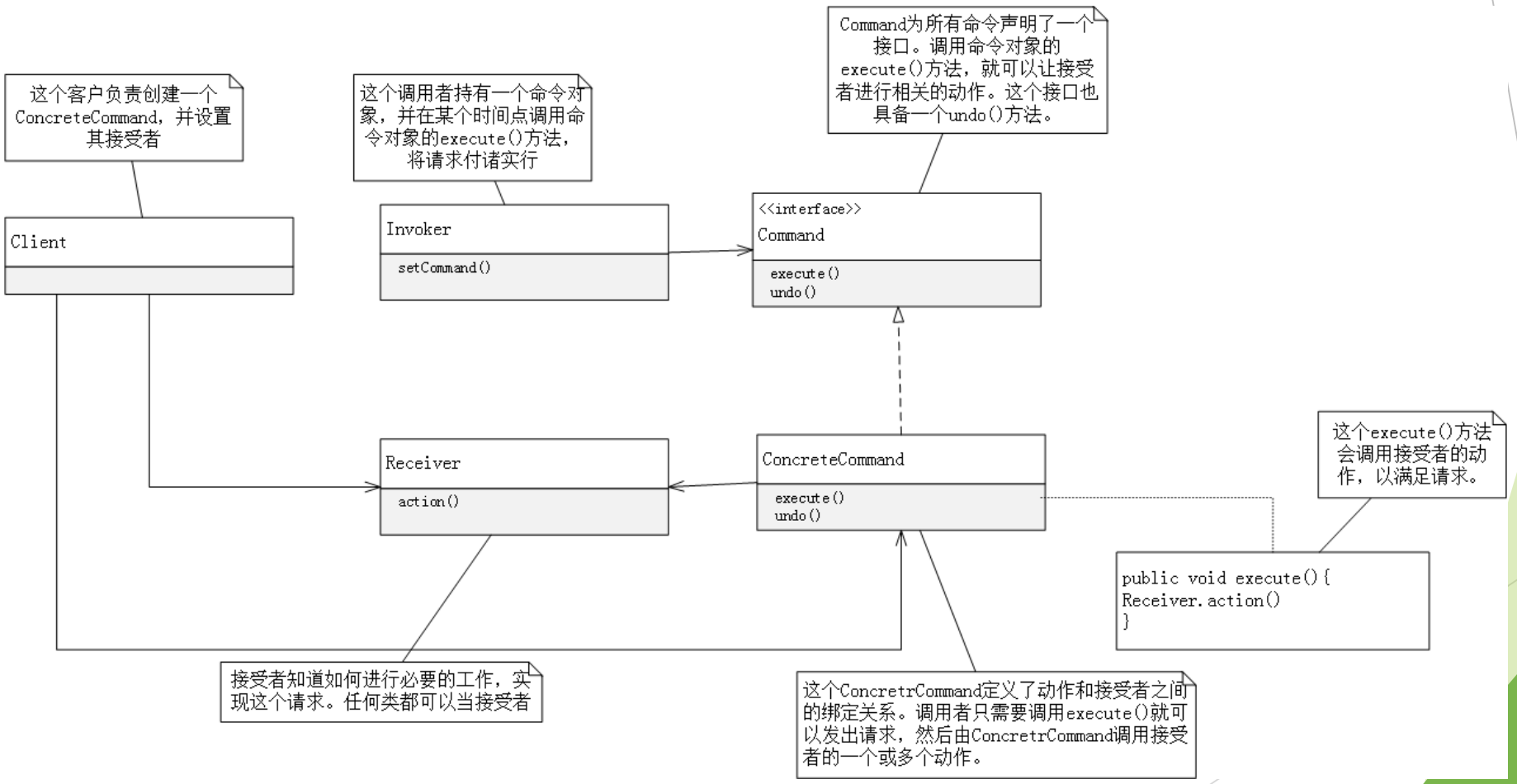


定义命令模式

- 命令模式：将“请求”封装成对象，一边使用不同的请求、队列或者日志来来参数化其他对象。命令模式也支持可撤销的操作。
- 我们知道一个命令对象通过在特定接收者上绑定一组动作来封装一个请求。要达到这点，命令对象间动作和接收者包进对象中，这个对象只暴露出一个execute()方法，当此方法被调用的时候，接收者就会进行这些动作，从外面来看，其他对象不知道究竟哪个接收者进行了那些动作，只知道如果调动execute()方法，请求的目的就能达到。

定义命令模式

- 类图


这个客户负责创建一个ConcreteCommand，并设置其接受者
这个调用者持有一个命令对象，并在某个时间点调用命令对象的execute()方法，将请求付诸实行
Command为所有命令声明了一个接口。调用命令对象的execute()方法，就可以让接受者进行相关的动作。这个接口也具备一个undo()方法。
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
Receiver
action()
ConcreteCommand
execute()
undo()
这个execute()方法会调用接受者的动作，以满足请求。
public void execute(){
Receiver.action()
}
接受者知道如何进行必要的工作，实现这个请求。任何类都可以当接受者
这个ConcretrCommand定义了动作和接受者之间的绑定关系。调用者只需要调用execute()就可以发出请求，然后由ConcretrCommand调用接受者的一个或多个动作。

实现遥控器

```
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 onButtowWasPushed(int slot){
        onCommands[slot].execute();
    }
    public void offButtowWasPushed(int slot){
        onCommands[slot].execute();
    }

    public String toString() {
        StringBuffer stringBuff=new StringBuffer();
        stringBuff.append("\n------Remote Control------\n");
        for(int i=0;i<onCommands.length;i++){
            stringBuff.append("[slot"+i+"]"+onCommands[i].getClass().getName()
                    +"    "+offCommands[i].getClass().getName()+"\n");
        }
        return stringBuff.toString();
    }
}
```

实现命令

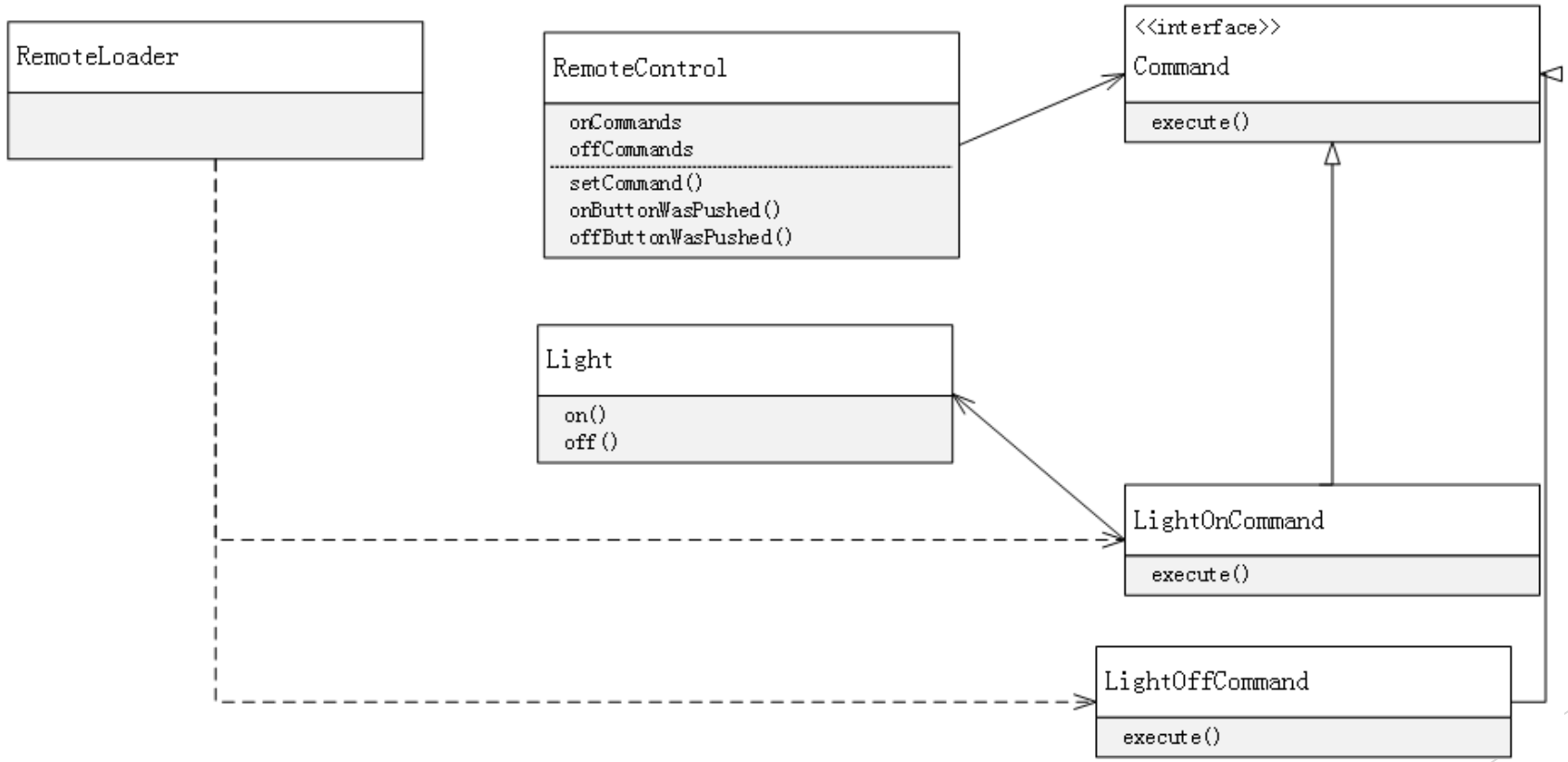
```
class LightOffCommand implements Command{
    Light light;
    public LightOffCommand(Light light){
        this.light=light;
    }
    @Override
    public void execute() {
        light.off();
    }
}
```

NoCommand对象

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

- NoCommand对象是一个空对象（null object），当你不想返回一个有意义的对象时，空对象就很有用，客户也可以将处理null的责任转移给空对象。
- 遥控器不可能一出厂就设置了有意义的命令对象，所以提供了NoCommand对象作为代用品，当调用它的execute()方法时，这种对象什么事情都不做。

为家电自动化公司设计的遥控器API



在遥控器上加上撤销的功能

- 当命令支持撤销时，该命令就必须提供和execute()方法相反的undo()方法，不管execute()刚才做什么，undo()都会倒转过来，这么一来，在各个命令中加入undo()之前，我们必须先在Command接口中加入undo()方法。

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

深入电灯的命令，从LightOnCommand开始

Undo（）需要调用off（）进行相反的动作

```
class LightOnCommand implements Command{
    Light light;
    public LightOnCommand(Light light){
        this.light=light;
    }
    @Override
    public void execute() {
        light.on();
    }

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

处理LightOFFCommand，undo（）需要调用电灯的on（）

```
class LightOffCommand implements Command{
    Light light;
    public LightOffCommand(Light light){
        this.light=light;
    }
    @Override
    public void execute() {
        light.off();
    }

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

要加上对撤销按钮的支持，我们必须对遥控器类做一些小修改，加入一个新的实例变量，用来追踪最后被调用的命令，然后，不管何时撤销按钮被按下，我们可以取出这个命令并调用它的undo()方法

```
class RemoteControl{
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;
    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;
    }
    public void setCommand(int slot,Command onCommand,Command offCommand){
        onCommands[slot]=onCommand;
        offCommands[slot]=offCommand;
    }
    public void onButtowWasPushed(int slot){
        onCommands[slot].execute();
        undoCommand=onCommands[slot];
    }
    public void offButtowWasPushed(int slot){
        onCommands[slot].execute();
        undoCommand=offCommands[slot];
    }
    public void undoButtowWasPushed(){
        undoCommand.undo();
    }

    @Override
    public String toString() {
        StringBuffer stringBuff=new StringBuffer();
        stringBuff.append("\n------Remote Control------\n");
        for(int i=0;i<onCommands.length;i++){
            stringBuff.append("[slot"+i+"]"+onCommands[i].getClass().getName
                    +"    "+offCommands[i].getClass().getName()+"\n");
        }
        return stringBuff.toString();
    }
}
```

宏命令----每个遥控器都具备“party模式”

- 打开或关闭所有的电器

```
class MacroCommand implements Command{
    Command[] commands;
    public MacroCommand(Command[] commands){
        this.commands=commands;
    }
    @Override
    public void execute() {
        for(Command command:commands){
            command.execute();
        }
    }
    @Override
    public void undo() {
        for(Command command:commands){
            command.undo();
        }
    }
}
```

命令模式的更多用途

队列请求

- 命令可以将运算块打包(一个接受者和一组动作)，然后将它传来传去，就像是一般的对象一样，现在，即使在命令对象被创建许久之后，运算依然可以被调用，事实上，它甚至可以在不同的线程中被调用，我们可以利用这样的特性衍生一些应用，例如：日程安排，线程池，工作队列等。
  想象有一个工作队列：你在某一端添加命令，然后另一端则是线程，线程进行下面的动作：从队列中取出一个命令，调用它的execute()方法，等待这个调用完成，然后将此命令对象丢弃，再取出下一个命令.....

日志请求

- 更多应用需要我们将所有的动作都记录在日志中，并能在系统死机后，重新调用这些动作恢复到之前的状态。当我们执行命令的时候，将历时记录存储在磁盘中。一旦系统死机，我们就可以将命令对象重新加载，并成批地依次调用这些对象的execute()方法。

总结

- 优点：解除了请求者与实现者之间的耦合，降低了系统的耦合度。

  对请求排队或记录请求日志，支持撤销操作。

  可以容易地设计一个组合命令。

  新命令可以容易地加入到系统中。

- 缺点：因为针对每一个命令都需要设计一个具体命令类，使用命令模式可能会导致系统有过多的具体命令类。

- 适用场景：

  当需要对行为进行“记录、撤销/重做”等处理时。

  系统需要将请求者和接收者解耦，使得调用者和接收者不直接交互。

  系统需要在不同时间指定请求、请求排队和执行请求。

  系统需要将一组操作组合在一起，即支持宏命令。