<https://blog.csdn.net/pangpang123654/article/details/50016045?ref=myread>

版权声明：本文为博主原创文章，未经博主允许不得转载。邮箱596878238@qq.com。 https://blog.csdn.net/pangpang123654/article/details/50016045

对于本篇文章，送给那些有部分代码经验的新手，因为我也是新手，如有错误，还请多多批评指正。

遥想码农当年，意气风华，神采飞扬，风华正茂，挥斥方遒，浮浮沉沉的码海中，来去自如。如今细细一想，已经踏入这条不归路两年有余，如今黯然失色，萎靡不振，执手相看泪眼，却无语凝咽。

言归正传，先把话题扯到command模式来，举一个很简单的例子，前几天去餐馆吃饭，对服务员点了一份红烧肉一份牛肉，此刻，服务员会在菜单上面把点的菜写上去，如果用代码去实现，只要有点代码经验的都可以搞定，代码量很简单。

public class Menu

{

public void Bouilli(){

// TODO Auto-generated method stub

System.out.println("您点了红烧肉");

}

public void beef()

{

// TODO Auto-generated method stub

System.out.println("您点了牛肉");

}

}

再写一个main函数调用一下

public class Client

{

public static void main(String[] args)

{

Menu menu = new Menu();

menu.Bouilli();

menu.beef();

}

}

运行了一下，console打印了   
您点了红烧肉   
您点了牛肉

嗯，是的，从功能角度来说，确实实现了我们所要的需求，即点了红烧肉又点了牛肉，可以嗨皮的吃一顿了，但是从代码的角度来说，这段代码不管从可重用性、可维护性、可扩展性还是灵活性都是很欠缺的，如果下次有别的客户点了鸡肉或者菠菜，我们既要改main函数还要改菜单，维护性带来了麻烦，代码的重用性也不好，扩展性也比较差，当然，这样的代码肯定也不够灵活，也许会有很少的部分人会思考，只要在菜单里面加上相应的菜不就行了吗？main函数加的代码也就加上需要的几行调用，为什么要搞得那么复杂，那么多类，看的都累，是的，最初刚入这条不归路的时候，看到设计模式我也这么想，但那时候我相信，它的存在必然有它的道理，一定是我做错了什么。

为什么不提倡这么写，抛开高内聚低耦合（以上代码中，“行为实现者”和“行为请求者”之间的紧耦合），先上第二部分代码再去分析。

public class Receiver

{

public void order(String str)

{

System.out.println("您点了"+str);

}

}

public interface Command

{

void execute();

}

public class BeefCommand implements Command

{

private Receiver receiver;

public BeefCommand(Receiver receiver)

{

this.receiver = receiver;

}

@Override

public void execute()

{

// TODO Auto-generated method stub

receiver.order("牛肉");

}

}

public class BouilliCommand implements Command

{

private Receiver receiver;

public BouilliCommand(Receiver receiver)

{

this.receiver = receiver;

}

@Override

public void execute()

{

// TODO Auto-generated method stub

receiver.order("红烧肉");

}

}

public class Client

{

public static void main(String[] args)

{

Receiver receiver = new Receiver();

BouilliCommand bouilliCommand = new BouilliCommand(receiver);

bouilliCommand.execute();

BeefCommand beefCommand = new BeefCommand(receiver);

beefCommand.execute();

}

}

运行了一下，和第一次的结果是一样的，现在的代码，从各角度来说，都得到了提升，如果需要点鸡肉或者菠菜，只要再建一个类去实现Command就好，但是又一个问题来了，我们平时去饭店点菜，你点了一份牛肉，此刻，你总不会把做牛肉的厨师喊过来和他说吧，command模式的核心也在这里，就像我们去点菜，我们只负责对服务员点菜，服务员把菜单递到厨房，哪位厨师做了什么菜，我们不需要去管，我们只是负责发送命令，服务员负责接受命令，厨师去执行命令。通过服务员，使客户与厨师之间的耦合度明显降低。

public class Waiter

{

private Command command;

/\*\*

\* @param command

\*/

public Waiter(Command command)

{

super();

this.command = command;

}

public void exe(){

command.execute();

}

}

main修改后

public class Client

{

public static void main(String[] args)

{

Waiter waiter;

Receiver receiver = new Receiver();

BouilliCommand bouilliCommand = new BouilliCommand(receiver);

waiter = new Waiter(bouilliCommand);

waiter.exe();

BeefCommand beefCommand = new BeefCommand(receiver);

waiter = new Waiter(beefCommand);

waiter.exe();

}

}

好了，现在写得差不多了，只剩下最后两个问题，一个是每次都要重新调用Waiter的exe()，看了都觉得烦，还有就是如果我点菜点多了，我想取消一些command，这个没法实现，所以，我们需要加上一次遍历，让我们代码中去一一执行exe()，在需要取消某个command的时候，我们就把不必要的菜取消掉，再去重新遍历菜单，好了，就说到这里，献上最后一次修改的代码

public class Waiter

{

private static List<Command> list;

public Waiter(Command command)

{

if(list==null){

list = new ArrayList<Command>();

}

list.add(command);

}

public void remove(Command command){

list.remove(command);

}

public void removeAll(Command command){

if(list!=null&&list.size()>0){

for (Command com : list)

{

list.remove(com);

}

}

}

public void exe(){

for (Command com : list)

{

com.execute();

}

}

}

public class Client

{

public static void main(String[] args)

{

Waiter waiter;

Receiver receiver = new Receiver();

BouilliCommand bouilliCommand = new BouilliCommand(receiver);

waiter = new Waiter(bouilliCommand);

BeefCommand beefCommand = new BeefCommand(receiver);

waiter = new Waiter(beefCommand);

waiter.exe();

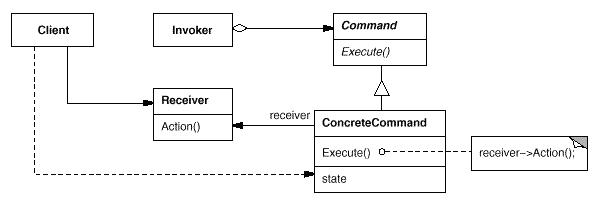
System.out.println("\n下面是新菜单\n");

waiter.remove(beefCommand);

waiter.exe();

}

}



好了，最后配上UML，今天这就写到这里了，要是有什么不妥，欢迎指正。