

11 | 软件设计的开闭原则：如何不修改代码却能实现需求变更？

2019-12-16 李智慧

后端技术面试38讲

[进入课程 >](#)



讲述：李智慧

时长 12:21 大小 11.32M



我在上篇文章讲到，软件设计应该为需求变更而设计，应该能够灵活、快速地满足需求变更的要求。优秀的程序员也应该欢迎需求变更，因为持续的需求变更意味着自己开发的软件保持活力，同时也意味着自己为需求变更而进行的设计有了用武之地，这样的话，技术和业务都进入了良性循环。

但是需求变更就意味着原来开发的功能需要改变，也意味着程序需要改变。如果是通过修改程序代码实现需求变更，那么代码一定会在不断修改的过程中变得面目全非，这也意味着代码的腐坏。

有没有办法不修改代码却能实现需求变更呢？

这个要求听起来有点玄幻，事实上却是软件设计需要遵循的最基本的原则：开闭原则。

开闭原则

开闭原则说：**软件实体（模块、类、函数等等）应该对扩展是开放的，对修改是关闭的。**

对扩展是开放的，意味着软件实体的行为是可扩展的，当需求变更的时候，可以对模块进行扩展，使其满足需求变更的要求。

对修改是关闭的，意味着当对软件实体进行扩展的时候，不需要改动当前的软件实体；不需要修改代码；对于已经完成的类文件不需要重新编辑；对于已经编译打包好的模块，不需要再重新编译。

通俗的说就是，**软件功能可以扩展，但是软件实体不可以被修改。**

功能要扩展，软件又不能修改，似乎是自相矛盾的，怎样才能做到不修改代码和模块，却能实现需求变更呢？

一个违反开闭原则的例子


在开始讨论前，让我们先看一个反面的例子。

假设我们需要设计一个可以通过按钮拨号的电话，核心对象是按钮和拨号器。那么简单的设计可能是这样的：



按钮类关联一个拨号器类，当按钮按下的时候，调用拨号器相关的方法。代码是这样的：

```
1 public class Button {
2     public final static int SEND_BUTTON = -99;
3
4     private Dialer dialer;
```

 复制代码

```

5     private int         token;
6
7     public Button(int token, Dialer dialer) {
8         this.token = token;
9         this.dialer = dialer;
10    }
11
12    public void press() {
13        switch (token) {
14            case 0:
15            case 1:
16            case 2:
17            case 3:
18            case 4:
19            case 5:
20            case 6:
21            case 7:
22            case 8:
23            case 9:
24                dialer.enterDigit(token);
25                break;
26            case SEND_BUTTON:
27                dialer.dial();
28                break;
29            default:
30                throw new UnsupportedOperationException("unknown button pressed");
31        }
32    }
33 }

```

 复制代码

```

1 public class Dialer {
2     public void enterDigit(int digit) {
3         System.out.println("enter digit: " + digit);
4     }
5
6     public void dial() {
7         System.out.println("dialing...");
8     }
9 }

```

按钮在创建的时候可以创建数字按钮或者发送按钮，执行按钮的 `press()` 方法的时候，会调用拨号器 `Dialer` 的相关方法。这个代码能够正常运行，完成需求，设计似乎也没什么问题。

这样的代码我们司空见惯，但是它的设计违反了开闭原则：当我们想要增加按钮类型的时候，比如，当我们需要按钮支持星号（*）和井号（#）的时候，我们必须修改 Button 类代码；当我们想要用这个按钮控制一个密码锁而不是拨号器的时候，因为按钮关联了拨号器，所以依然要修改 Button 类代码；当我们想要按钮控制多个设备的时候，还是要修改 Button 类代码。

似乎对 Button 类做任何的功能扩展，都要修改 Button 类，这显然违反了开闭原则：对功能扩展是开放的，对代码修改是关闭的。

违反开闭原则的后果是，这个 Button 类非常僵硬，当我们想要进行任何需求变更的时候，都必须要修改代码。同时我们需要注意，大段的 switch/case 语句是非常脆弱的，当需要增加新的按钮类型的时候，需要非常谨慎地在这段代码中找到合适的位置，稍不小心就可能出现 bug。粗暴一点说，**当我们在代码中看到 else 或者 switch/case 关键字的时候，基本可以判断违反开闭原则了。**

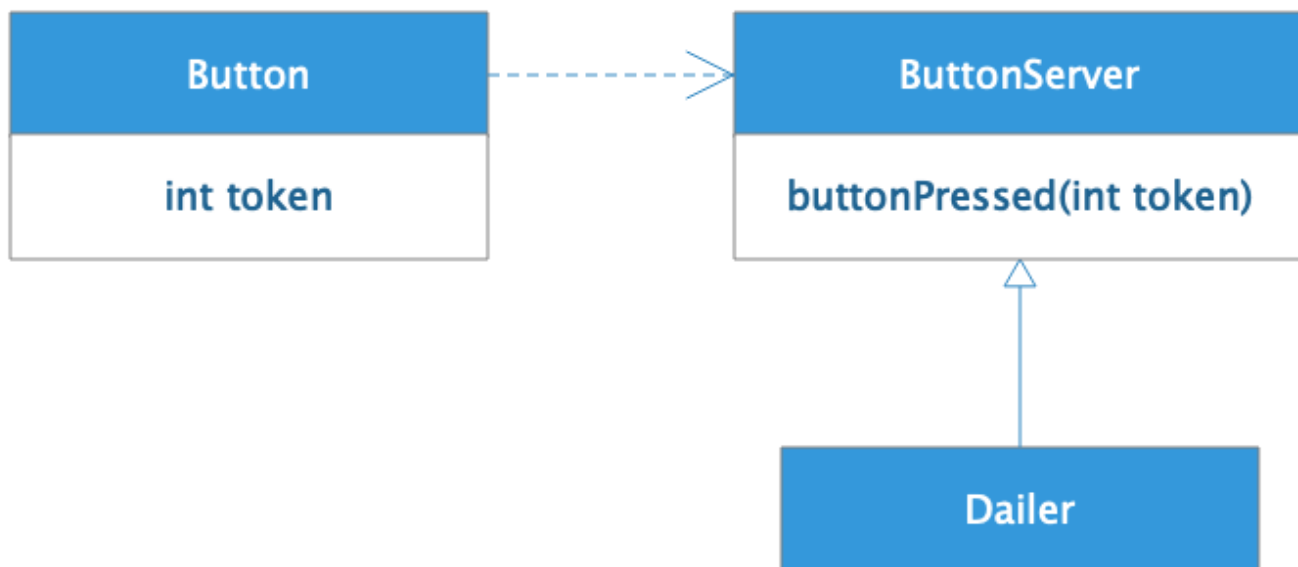
而且，这个 Button 类也是难以复用的，Button 类强耦合了一个 Dailer 类，在脆弱的 switch/case 代码段耦合调用了 Dailer 的方法，即使 Button 类自身也将各种按钮类型耦合在一起，当我想复用这个 Button 类的时候，不管我需不需要一个 Send 按钮，Button 类都自带了这个功能。

所以，这样的设计不要说不修改代码就能实现功能扩展，即使我们想修改代码进行功能扩展，里面也很脆弱，稍不留心就掉到坑里了。这个时候你再回头审视 Button 的设计，是不是就感觉到了代码里面腐坏的味道，如果让你接手维护这些代码实现需求变更，是不是头疼难受？

很多设计开始看并没有什么问题，如果软件开发出来永远也不需要修改，也许怎么设计都可以，但是当需求变更来的时候，就会发现各种僵硬、脆弱。所以设计的优劣需要放入需求变更的场景中考察。当需求变更时发现当前设计的腐坏，就要及时进行重构，保持设计的强壮和代码的干净。

使用策略模式实现开闭原则

设计模式中很多模式其实都是用来解决软件的扩展性问题的，也是符合开闭原则的。我们用**策略模式**对上面的例子重新进行设计。



我们在 Button 和 Dailer 之间增加了一个抽象接口 ButtonServer，Button 依赖 ButtonServer，而 Dailer 实现 ButtonServer。

当 Button 按下的时候，就调用 ButtonServer 的 buttonPressed 方法，事实上是调用 Dailer 实现的 buttonPressed 方法，这样既完成了 Button 按下的时候执行 Dailer 方法的需求，又不会使 Button 依赖 Dailer。Button 可以扩展复用到其他需要使用 Button 的场景，任何实现 ButtonServer 的类，比如密码锁，都可以使用 Button，而不需要对 Button 代码进行任何修改。

而且 Button 也不需要 switch/case 代码段去判断当前按钮类型，只需要将按钮类型 token 传递给 ButtonServer 就可以了，这样增加新的按钮类型的时候就不需要修改 Button 代码了。

策略模式是一种行为模式，多个策略实现同一个策略接口，编程的时候 client 程序依赖策略接口，运行期根据不同上下文向 client 程序传入不同的策略实现。

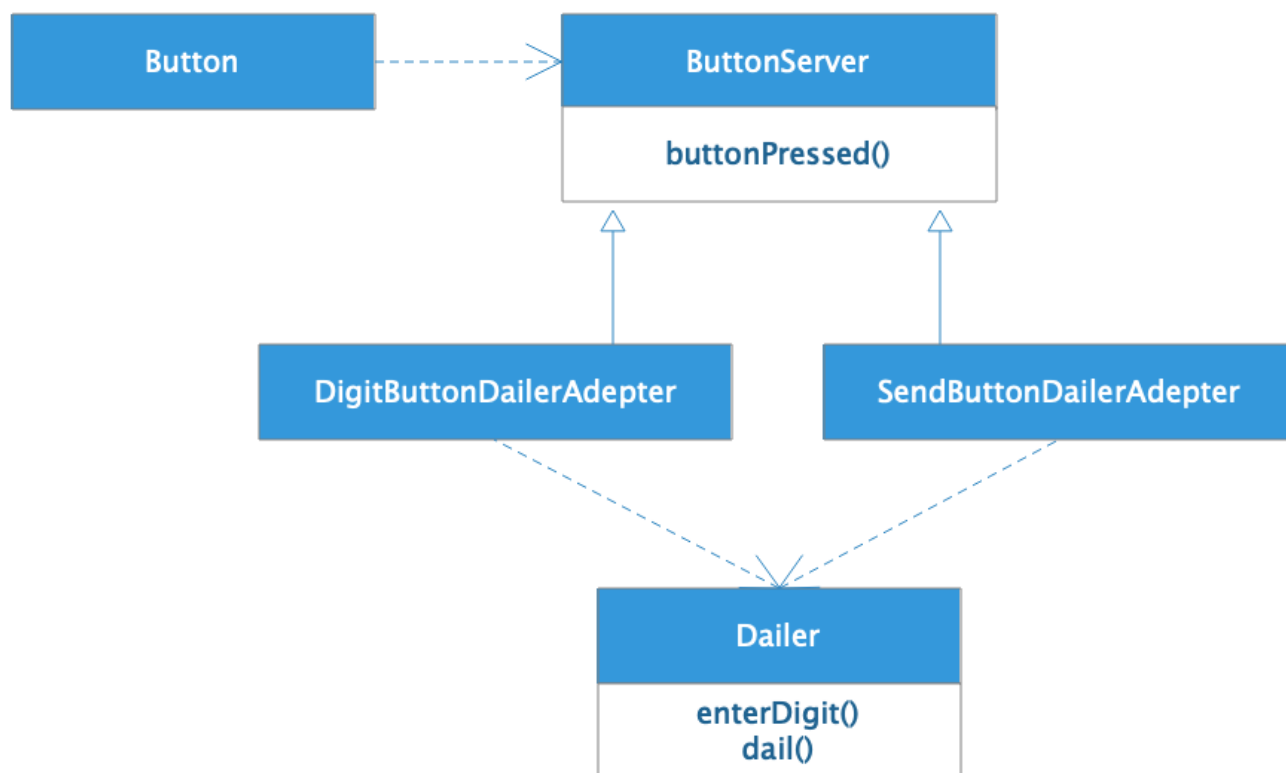
在我们这个场景中，client 程序就是 Button，策略就是需要用 Button 控制的目标设备，拨号器、密码锁等等，ButtonServer 就是策略接口。通过使用策略模式，我们使 Button 类实现了开闭原则。

使用适配器模式实现开闭原则

Button 符合开闭原则了，但是 Dailer 又不符合开闭原则了，因为 Dailer 要实现 ButtonServer 接口，根据参数 token 决定执行 enterDigit 方法还是 dail 方法，又需要 if/else 或者 switch/case，不符合开闭原则。

那怎么办？

这种情况可以使用**适配器模式**进行设计。适配器模式是一种结构模式，用于将两个不匹配的接口适配起来，使其能够正常工作。



不要由 Dailer 类直接实现 ButtonServer 接口，而是增加两个适配器 DigitButtonDailerAdepter、SendButtonDailerAdepter，由适配器实现 ButtonServer 接口，在适配器的 buttonPressed 方法中调用 Dailer 的 enterDigit 方法和 dail 方法，而 Dailer 类保持不变，Dailer 类实现开闭原则。

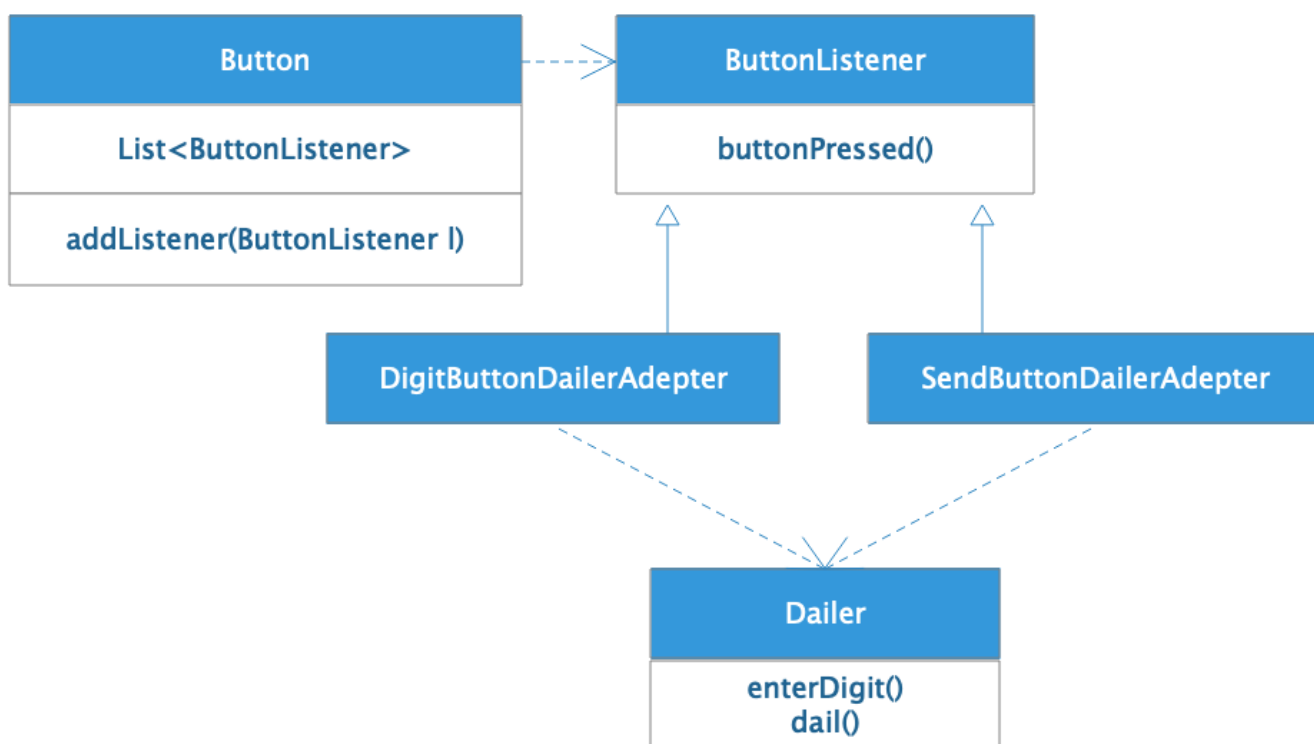
在我们这个场景中，Button 需要调用的接口是 buttonPressed，和 Dailer 的方法不匹配，如何在不修改 Dailer 代码的前提下，使 Button 能够调用 Dailer 代码？就是靠适配器，适配器 DigitButtonDailerAdepter 和 SendButtonDailerAdepter 实现了 ButtonServer 接口，使 Button 能够调用自己，并在自己的 buttonPressed 方法中调用 Dailer 的方法，适配了 Dailer。

使用观察者模式实现开闭原则

通过策略模式和适配器模式，我们使 Button 和 Dailer 都符合了开闭原则。但是如果要求能够用一个按钮控制多个设备，比如按钮按下进行拨号的同时，还需要扬声器根据不同按钮发出不同声音，将来还需要根据不同按钮点亮不同颜色的灯。按照当前设计，可能需要在适配器中调用多个设备，增加设备要修改适配器代码，又不符合开闭原则了。

怎么办？

这种情况可以用**观察者模式**进行设计：



这里，ButtonServer 被改名为 ButtonListener，表示这是一个监听者接口，其实这个改名不重要，仅仅是为了便于识别。因为接口方法 `buttonPressed` 不变，ButtonListener 和 ButtonServer 本质上是一样的。

重要的是在 Button 类里增加了成员变量 List 和成员方法 `addListener`。通过 `addListener`，我们可以添加多个需要观察按钮按下事件的监听者实现，当按钮需要控制新设备的时候，只需要将实现了 ButtonListener 的设备实现添加到 Button 的 List 列表就可以了。

Button 代码：


```
1 public class Button {
2     private List<ButtonListener> listeners;
3
4     public Button() {
5         this.listeners = new LinkedList<ButtonListener>();
6     }
7
8     public void addListener(ButtonListener listener) {
9         assert listener != null;
10        listeners.add(listener);
11    }
12
13    public void press() {
14        for (ButtonListener listener : listeners) {
15            listener.buttonPressed();
16        }
17    }
18 }
```

Dailer 代码和原始设计一样，如果我们需要将 Button 和 Dailer 组合成一个电话，Phone 代码如下：

```
1 public class Phone {
2     private Dialer dialer;
3     private Button[] digitButtons;
4     private Button sendButton;
5
6     public Phone() {
7         dialer = new Dialer();
8         digitButtons = new Button[10];
9         for (int i = 0; i < digitButtons.length; i++) {
10            digitButtons[i] = new Button();
11            final int digit = i;
12            digitButtons[i].addListener(new ButtonListener() {
13                public void buttonPressed() {
14                    dialer.enterDigit(digit);
15                }
16            });
17        }
18        sendButton = new Button();
19        sendButton.addListener(new ButtonListener() {
20            public void buttonPressed() {
21                dialer.dial();
22            }
23        });
24    }
25 }
```



```

24     }
25
26     public static void main(String[] args) {
27         Phone phone = new Phone();
28         phone.digitButtons[9].press();
29         phone.digitButtons[1].press();
30         phone.digitButtons[1].press();
31         phone.sendButton.press();
32     }
33 }

```

观察者模式是一种行为模式，解决一对多的对象依赖关系，将被观察者对象的行为通知到多个观察者，也就是监听者对象。

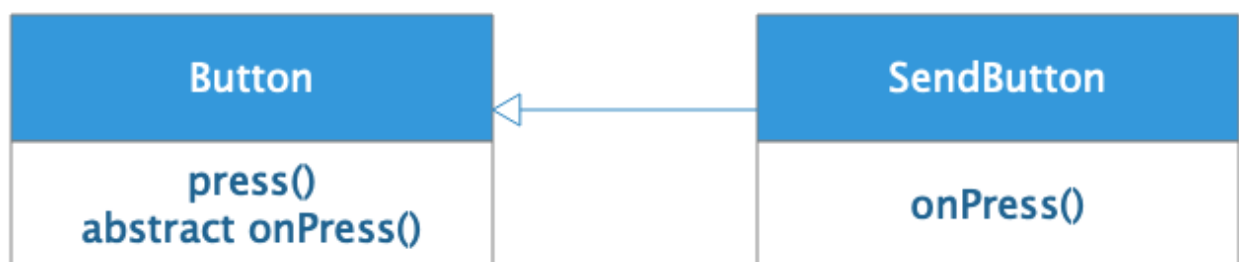
在我们这个场景中，Button 是被观察者，目标设备拨号器、密码锁等是观察者。被观察者和观察者通过 Listener 接口解耦合，观察者（的适配器）通过调用被观察者的 addListener 方法将自己添加到观察列表，当观察行为发生时，被观察者会逐个遍历 Listener List，通知观察者。

使用模板方法模式实现开闭原则

如果业务要求按下按钮的时候，除了控制设备，按钮本身还需要执行一些操作，完成一些成员变量的状态更改，不同按钮类型进行的操作和记录状态各不相同。按照当前设计可能又要在 Button 的 press 方法中增加 switch/case 了。

怎么办？

这种情况可以用**模板方法模式**进行设计：



在 Button 类中定义抽象方法 onPress，具体类型的按钮，比如 SendButton 实现这个方法。Button 类中增加抽象方法 onPress，并在 press 方法中调用 onPress 方法：

```
1  abstract void onPress();  
2  
3  public void press() {  
4      onPress();  
5      for (ButtonListener listener : listeners) {  
6          listener.buttonPressed();  
7      }  
8  }
```

所谓模板方法模式，就是在父类中用抽象方法定义计算的骨架和过程，而抽象方法的实现则留在子类中。

在我们这个例子中，press 方法就是模板，press 方法除了调用抽象方法 onPress，还执行通知监听者列表的操作，这些抽象方法和具体操作共同构成了模板。而在子类 SendButton 中实现这个抽象方法，在这个方法中修改状态，完成自己类型特有的操作，这就是模板方法模式。

通过模板方法模式，每个子类可以定义自己在 press 执行时的状态操作，无需修改 Button 类，实现了开闭原则。

小结

实现开闭原则的关键是抽象。当一个模块依赖的是一个抽象接口的时候，就可以随意对这个抽象接口进行扩展，这个时候，不需要对现有代码进行任何修改，利用接口的多态性，通过增加一个新实现该接口的实现类，就能完成需求变更。不同场景进行扩展的方式是不同的，这时候就会产生不同的设计模式，大部分的设计模式都是用来解决扩展的灵活性问题的。

开闭原则可以说是软件设计原则的原则，是软件设计的核心原则，其他的设计原则更偏向技术性，具有技术性的指导意义，而开闭原则是方向性的，在软件设计的过程中，应该时刻以开闭原则指导、审视自己的设计：当需求变更的时候，现在的设计能否不修改代码就可以实现功能的扩展？如果不是，那么就应该进一步使用其他的设计原则和设计模式去重新设计。

更多的设计原则和设计模式，我将在后面陆续讲解。

思考题

我在观察者模式小节展示的 Phone 代码示例中，并没有显式定义 DigitButtonDailerAdepter 和 SendButtonDailerAdepter 这两个适配器类，但它们是存在的。在哪里呢？

欢迎在评论区写下你的思考，我会和你一起交流，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

点击参加 21 天打卡计划 

搞定后端技术基础



扫一扫参与小程序话题



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | 软件设计的目的：糟糕的程序员比优秀的程序员差在哪里？

下一篇 12 | 软件设计的依赖倒置原则：如何不依赖代码却可以复用它的功能？

精选留言 (13)

 写留言



山猫

2019-12-16

我同意老师通过这个例子简单的描述开闭原则。但如果项目初始就对button按钮需要进行这么复杂的设计，那么这个项目后期的维护成本也是相当之高。

展开 

作者回复: 是否要使用各种设计模式设计一个非常灵活的程序, 主要是看你的需求场景, 而不是看项目的阶段。

如果你的场景就是需要这么灵活, 就是要各种复用, 应对各种变更, 那么从一开始就应该这样设计。

如果你的场景根本不需要一个可复用的button, 那么就不需要这样设计。

关键还是看场景。

但是场景也会变化, 一开始不需要复用, 但是后来又需要复用了, 那么在在需要复用的第一个场景, 就重构代码, 而不是等将来维护困难局面hold不住了再重构。

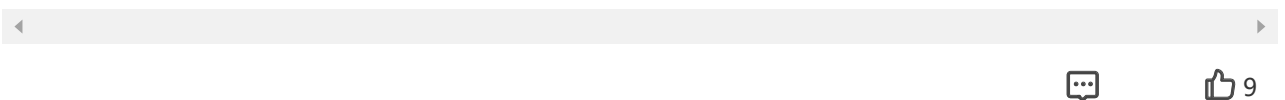
ps 如果你习惯了这种灵活的设计, 你会觉得这种设计并不复杂。对于软件开发而言, 复杂的永远是业务逻辑, 而不是设计模式。设计模式是可重复的, 可重复的东西即使看起来复杂, 熟悉了就会觉得很简单。

pps 看起来复杂的设计模式就是用来解决维护困难问题的, 正确使用设计模式, 看起来复杂了, 其实维护简单了, 因为关系和边界更清晰了, 你不需要在一堆强耦合的代码里搅来搅去。真正维护成本高的其实是你所谓的简单的设计, 牵一发动全身, 稍不注意就是各种bug。

ppps 重要的话再说一次:

关键还是看场景。

没有银弹, 没有一种必然就是好的设计方案, 能理解场景的才是真·高手。



Jesse

2019-12-16

思考题

匿名内部类, 已经数字按钮注册的listener其实就是DigitButtonDailerAdepter适配器的实现, sendButton中注册的listener其实就是SendButtonDailerAdepter适配器的实现。

展开 ∨



Roy Liang

2019-12-17

/*适配器模式*/

```
public class Button {  
    private ButtonServer server;  
    private int token;
```

...

展开 ▾



Jonathan Chan

2019-12-16

求老师后续给出完整代码学习！

展开 ▾



Roy Liang

2019-12-18

依葫芦画瓢，应用观察者模式写了一个密码箱类

```
public class CodeCase {  
    private Dialer dialer;  
    private Button[] digitButtons;  
    private Button lockButton;...
```

展开 ▾



陈小龙 Cheney

2019-12-17

希望老师给出几个阶段的代码. 方便对着代码对比学习. 直接看文字感觉抽象模糊了.



草原上的奔跑

2019-12-17

想问下老师，在写设计文档的时候，系统，子系统，组件，分别映射到代码的什么层面(类很好理解)；还有老师的设计文档中没有业务架构图和技术架构图，老师对这两种架构图是怎么理解的，设计文档中需要加入吗？希望老师给予解答

展开 ▾

作者回复: 组件就是jar，dll这些。子系统和系统就是可运行的完整程序，一个或者多个war。

不管是什么架构图，只要你觉得有助于你描述你的架构设计，都可以加入设计文档中。



鹏酱 □□

2019-12-17

Phone的构造函数里了

展开 ▾



一步

2019-12-16

结合王争老师的设计模式课，会理解的更透彻

展开 ▾



Geek_d048e4

2019-12-16

老师，Dailer 实现 ButtonServer你画的UML图用的是实线三角形图，接口实现应该是虚线三角头

作者回复: 谢谢指正，我用的UML工具没有虚线三角 😊

我在第9篇专栏提到，UML用来交流和思考，如果不影响交流，那么UML画法可以简化，也就是所谓的UML方言。



龙龙first

2019-12-16

我觉得还是需要业务和技术经验才能知道怎样模块解耦。如果不清楚，就不知道是对button的实现做预留扩展还是对action做预留扩展，因为也有可能后续需求不是点击按钮操作，而是动作识别出发操作



Paul Shan

2019-12-16

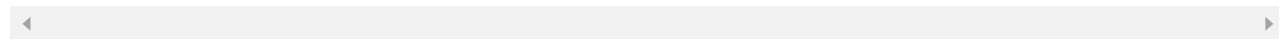
开闭原则是移除底层的if else，取而代之的是上层的类结构。不过，我个人以为一开始的if else, 甚至switch 也没什么不妥的，毕竟代码简单直接。引入了很多类，读代码也是负担，

而且也很难预料到哪些修改是必要的。当if else数量多于一定的数目，再开始重构。不知道李老师如何看待这种观点。

展开 ∨

作者回复: 当你准备写第一个else的时候，就说明你的代码即将陷入僵化、牢固和脆弱，而且为将来的需求变更引入了一个糟糕的“设计模式”。

如果其他人接手你的代码，他有两个选择，要么继续写更多的else以应对需求变更；要么心理暗骂一声然后重构你的代码。你希望他选择哪个？



💬 3



Paul Shan

2019-12-16

思考题

Adapter 是在继承接口的时候调用了dialer不同的函数实现了，没有显式的Adapter。

展开 ∨

