Bridge Pattern

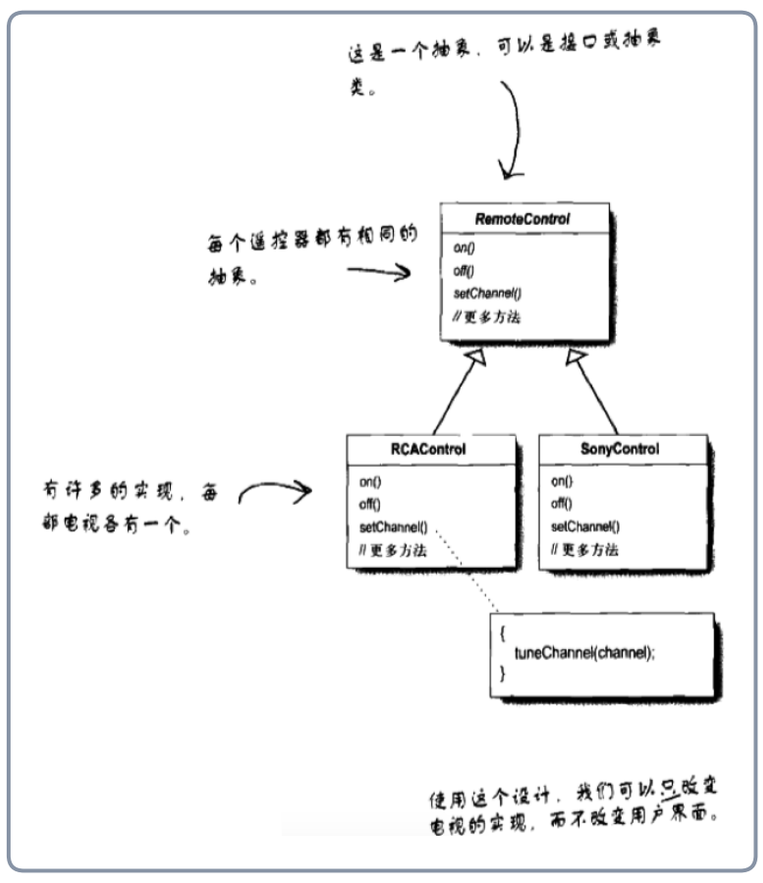
Decouple an abstraction from its implementation so that the two can vary independently.

场景

对电视，开发遥控器

所有遥控器基于相同抽象

而对此抽象又做出许多不同实现



两难

都会改变

电视机（实现）

抽象出界面

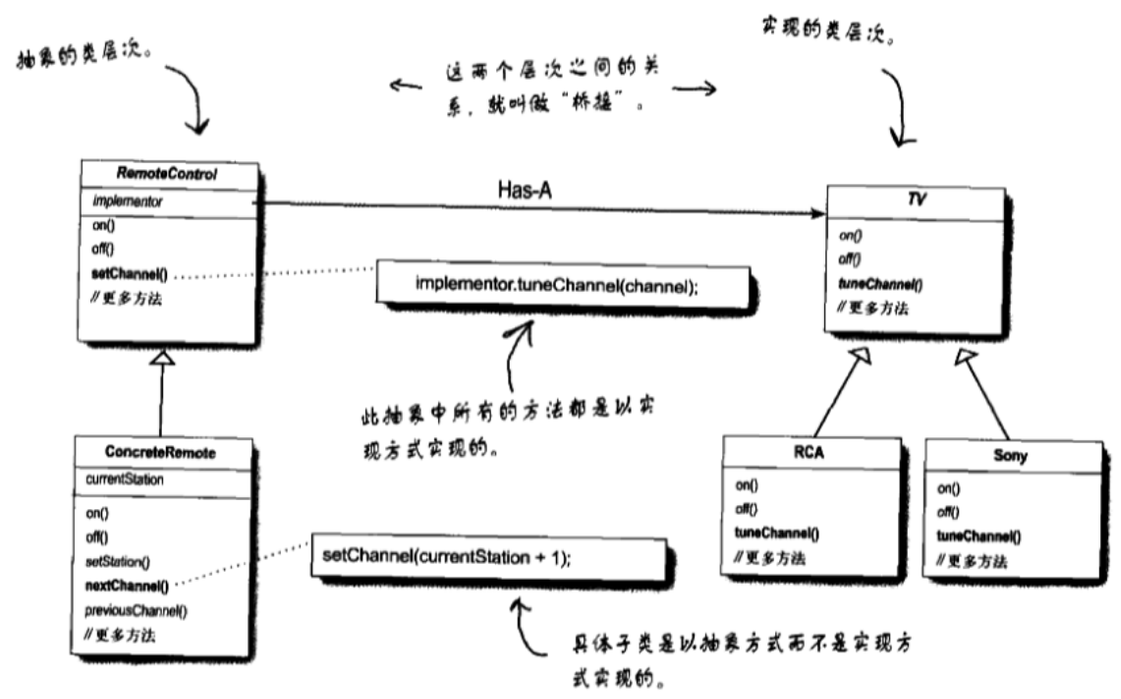
对不同对电视机，做不同的实现

遥控器（抽象）

用户对界面提出想法

应对用户反馈，对抽象做出改变

实现和抽象放在两个不同的类层次中，使它们可以独立变化



优点用途

优点

抽象和实现分离 — 解决继承的缺点

抽象和实现分离 — 实现可以不受抽象的约束

优秀的扩充能力 — 实现 抽象 都可以增加

实现细节对客户透明 — 由抽象层通过聚合关系完成了封装

用途

不希望或不适用使用继承的场景 — 继承层次过多、无法更细化设计颗粒

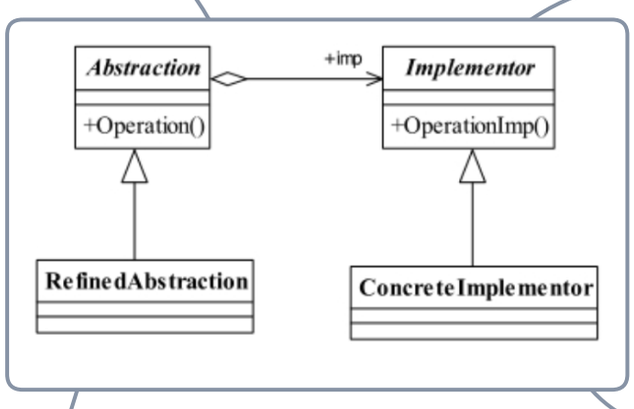
接口或抽象类不稳定的场景 — 明知道接口不稳定还想通过实现或继承来实现业务需求，那是得不偿失的，也是比较失败的做法。

重用性要求较高的场景 — 设计的颗粒度越细，则被重用的可能性就越大，而采用继承则受父类的限制，不可能出现太细的颗粒度

```
public abstract class Abstraction{
    //定义对实现化角色的引用
    private Implementor imp;
    //约束子类必须实现该构造函数
    public Abstraction(Implementor _imp) {
      this.imp=_imp;
    }
    //自身的行为和属性
    public void request() {
      this.imp.doSomething();
    }
    //获得实现化角色
    public Implementor getImp() {
      return imp;
    }
}
```

```
public interface Implementor {
  public void doSomething();
  public void doAnything();
}
```

```
public class RefinedAbstraction extends
Abstraction{
  //覆盖构造函数
  public RefinedAbstraction(Implementor _imp){
   super(_imp);
  }
  //修改父类的行为
  @Override
  public void request(){
    //业务处理...
    super.request();
    super.getImp().doAnything();
  }
}
```




```
public class ConcreteImplementor1 implements Implementor{
  public void doSomething(){// 业务逻辑处理}
  public void doAnything() {// 业务逻辑处理}
}
public class ConcreteImplementor2 implements Implementor{
  public void doSomething(){// 业务逻辑处理}
  public void doAnything() {// 业务逻辑处理}
}
```

```
public class Client{
  public static void main(String[] args) {
    //定义一个实现化角色
    Implementor imp = new ConcreteImplementor1();
    //定义一个抽象化角色
    Abstraction abs = new RefinedAbstraction(imp);
    //执行行为
    abs.request();
  }
}
```