## 领域模型与对象范式（下）

### 抽象

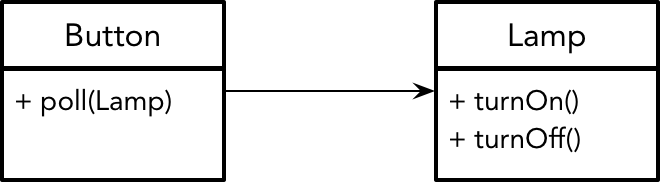
定义自治对象，进行合理的职责分配以实现对象的分治，是良好设计的基础。在应对需求变化时，分配职责采用了“分离”的方式，尽量完成对变化的隔离。然而，一旦分离了职责，就必然会产生对象之间的耦合，因为遵循自治与分治原则，设计的对象通常无法靠一个对象完成所有的任务。“高内聚松耦合”是软件设计的主旋律。自治对象做到了“高内聚”，通过良好的分治 **减少了耦合的数量** ，但要 **降低耦合的强度** ，则需要进行 **抽象** 才可。简言之，当依赖不可避免时，抽象可以把强依赖降低为弱依赖。

#### 抽象的意义

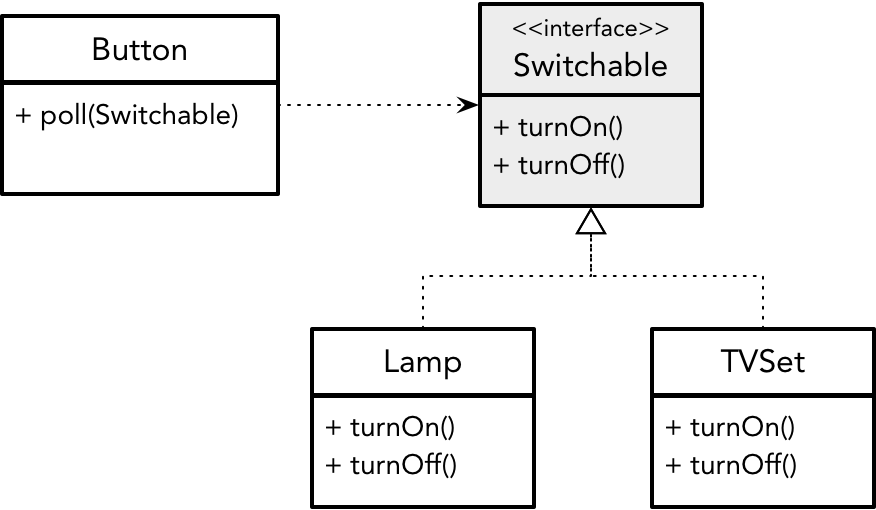
什么是设计的抽象呢？我们来看一则故事。

三个秀才到省城参加乡试，临行前三人都对自己能否中举惴惴不安，于是求教于街头的算命先生。算命老者的目光在这三人的脸上逡巡良久，最后徐徐伸出一个手指，就闭上眼睛不再言语，一副高深莫测的模样。三人纳闷，给了银子，带着疑惑到了省城参加考试。发榜之日，三人联袂去看成绩，得知结果后，三人齐叹，算命先生真乃神人矣！

抽象就是算命先生的“一指禅”，一个指头代表了四种完全不同的含义：是一切人高中，还是一个都不中？是一个人落榜，还是一个人高中？算命先生并不能未卜先知，因此只能给出一个包含了所有可能却没有实现的答案，至于是哪一种结果，就留给三个秀才去慢慢琢磨吧。这就是抽象，它意味着可以包容变化，也就意味着稳定。体现抽象价值的一个常见案例是按钮与灯泡之间的关系：



Button 依赖于具体的 Lamp 类，使得按钮只能控制灯泡，导致了二者之间的紧耦合。如果观察 Button 与 Lamp 的协作关系，我们发现按钮操作的是开关，而非灯泡，因而可以提炼出角色接口 Switchable。这个接口代表开和关的能力，只要具备这一能力的设备都可以被按钮控制，例如电视机：



Switchable 接口其实是面向调用者 Button 并根据它需要协作的需求提取的共同特征。只要 turnOn() 与 turnOff() 的共同特征不变，Button 就不会受到影响，二者形成了一种松散耦合的关系。抽象不提供具体实现，我们随时可以提供别的实现去替换它。如此的设计具有扩展性，满足 **开放封闭原则** ，即“对于扩展是开放的”。

TVSet 与 Lamp 的互替换性，就是面向对象设计的“多态（Polymorphism）”特征。所谓“多态”，是指对象在不同时刻体现为不同类型的能力。多态体现了角色的互换，就如生活中每个人在不同的场景会扮演不同的角色。以改进后的设计为例，Button 仅知道抽象的 Switchable 接口，poll() 方法操作该类型对象，至于具体类型是 Lamp 还是 TVSet，则取决于调用者究竟创建了哪一个具体类型的对象：

public class Client {  
 public static void final main(String[] args) {  
 Button button = new Button();  
  
 Switchable switchable = new Lamp();  
 // 开/关灯  
 button.poll(switchable);  
  
 switchable = new TVSet();  
 // 开/关电视  
 button.poll(switchable);  
 }  
}

当然，由于 Client 类的 main() 函数通过 new 关键字分别创建了 Lamp 与 TVSet 具体类的实例，使得 Client 并没有真正摆脱它对具体类的依赖。只要无法彻底绕开对具体对象的创建，抽象就不能完全解决耦合的问题。因此在面向对象设计中， **我们会尽量将产生依赖的对象创建工作往外推，直到调用者必须创建具体对象为止** 。这种把依赖往外推，直到在最外层不得不创建具体对象时，再将依赖从外部传递进来的方式，就是 Martin Fowler 所说的 **“依赖注入（Dependency Injection）”** 。

例如，下订单业务场景提供了两种插入订单的策略：同步和异步。在插入订单时，还需要用到事务，也有两种类型选择：本地事务和分布式事务。下订单方法的实现者并不知道调用者会选择哪种插入订单的策略，插入订单的实现者也不知道调用者会该选择哪种事务类型。要做到各自的实现者无需关心具体策略或类型的选择，就应该将这些决策向外推：

public interface TransactionScope {  
 void using(Command command);  
}  
public class LocalTransactionScope implements TransactionScope {}  
public class DistributedTransactionScope implements TransactionScope {}  
  
public interface InsertingOrderStrategy {  
 void insert(Order order);  
}  
public class SyncInsertingOrderStrategy implements InsertingOrderStrategy {  
 // 把对 TransactionScope 的具体依赖往外推  
 private TransactionScope ts;  
 // 通过构造函数允许调用者从外边注入依赖  
 public SyncInsertingOrderStrategy(TransactionScope ts) {  
 this.ts = ts;  
 }  
  
 public void insert(Order order) {  
 ts.using(() -> {  
 // 同步插入订单，实现略  
 return;  
 });  
 }  
}  
  
public class AsyncInsertingOrderStrategy implements InsertingOrderStrategy {  
 // 把对 TransactionScope 的具体依赖往外推  
 private TransactionScope ts;  
 // 通过构造函数允许调用者从外边注入依赖  
 public AsyncInsertingOrderStrategy(TransactionScope ts) {  
 this.ts = ts;  
 }  
  
 public void insert(Order order) {  
 ts.using(() -> {  
 // 异步插入订单，实现略  
 return;  
 });  
 }  
}  
  
public class PlacingOrderService {  
 // 把对 InsertingOrderStrategy 的具体依赖往外推  
 private InsertingOrderStrategy insertingStrategy;  
 // 通过构造函数允许调用者从外边注入依赖  
 public PlacingOrderService(InsertingOrderStrategy insertingStrategy) {  
 this.insertingStrategy = insertingStrategy;  
 }  
  
 public void execute(Order order) {  
 insertingStrategy.insert(order);  
 }  
}

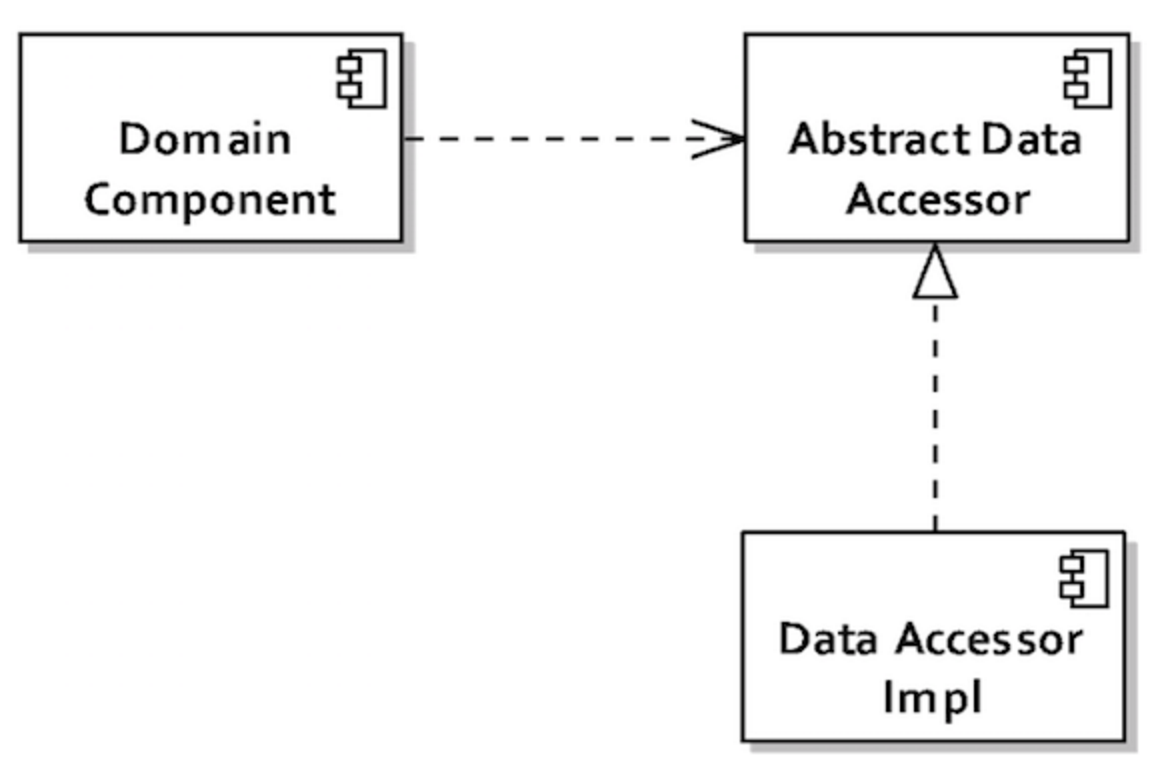
从内到外，在 SyncInsertingOrderStrategy 和 AsyncInsertingOrderStrategy 类的实现中，把具体的 TransactionScope 依赖向外推给 PlacingOrderService；在 PlacingOrderService 类中，又把具体的 InsertingOrderStrategy 依赖向外推给潜在的调用者。因此，到底使用何种插入策略和事务类型，与 PlacingOrderService 等提供服务行为的类无关，选择权被交给了最终的调用者。如果使用类似 Spring 这样的依赖注入框架，则可以通过配置或者注解等方式完成依赖的注入。

利用抽象，就可以降低对象之间的耦合度。这就要求我们在设计时，应考虑对外交互的接口而非实现，这就是“面向接口设计”原则。因为客户程序关心的仅仅是对象提供什么功能，而不是功能如何实现，甚至不关心对象的具体类型。这就好比我们连接电源，只需要根据电源插头确定需要什么样的插座即可，不用关心插头与插座内的电线是如何连接的。试想，每次连接电源时都要使用电焊来接通电线，那未免太可怕了：



图片截取自：[jrebel.com](https://jrebel.com/rebellabs/object-oriented-design-%20principles-and-the-5-ways-of-creating-solid-applications/)

面向接口的设计思想与 **依赖倒置原则** 不谋而合。该原则要求：高层模块不应该依赖于低层模块，二者都应该依赖于抽象。这一原则在分层架构模式中，得到了淋漓尽致地运用。例如，业务逻辑层的对象就不应该直接依赖于数据访问层的具体实现对象，而应该通过数据访问层的抽象接口进行访问，如下图所示：

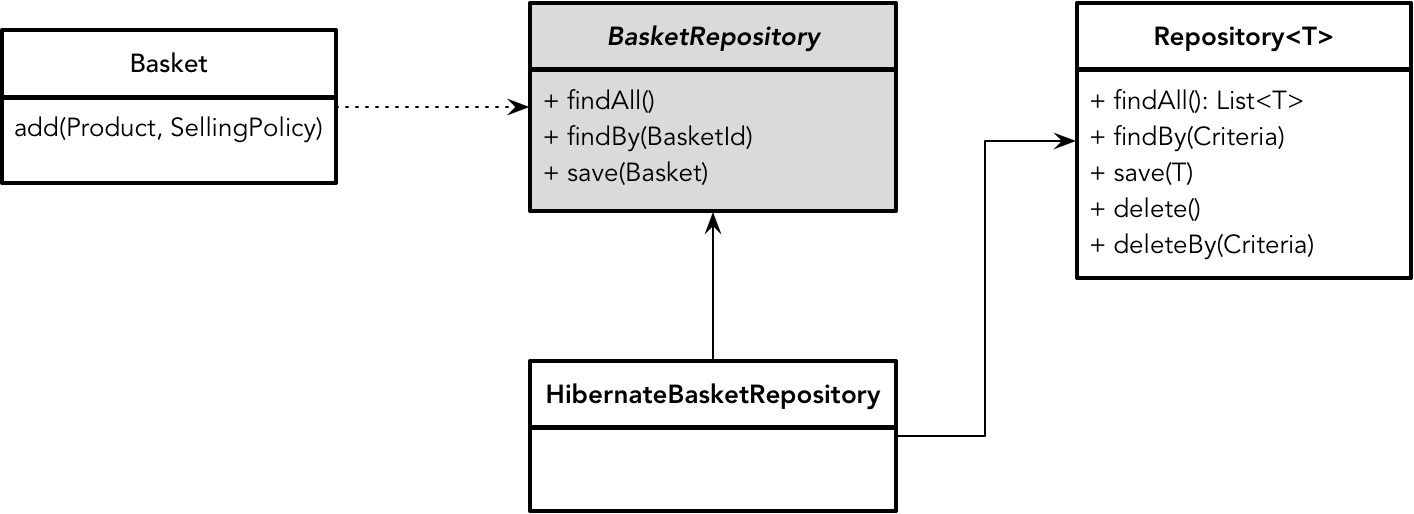


如果高层模块直接依赖于低层模块，一旦低层模块发生变化，就会影响到高层模块。高层模块是低层模块的调用者，通过引入抽象，就使得低层模块的实现是可替换的，保证了设计的可扩展性。

依赖倒置原则还要求：抽象不应该依赖于细节，细节应该依赖于抽象。依赖的关系与影响的方向相反，被依赖方处于上游，当上游发生变化时，下游的依赖方就会受到影响。要让下游不受影响，就需要保持上游不变。寄希望于需求的稳定是不现实的，在软件的世界中，唯一不变的就是变化。这就需要引入抽象来封装变化。抽象相对实现细节更加稳定，依赖于抽象可以让整个系统变得更加稳定。故而依赖于抽象，实则是依赖于稳定。这就是所谓的“稳定依赖原则”。城堡是不能修建于沙滩之上的，参与协作的对象、组件或模块也当如此。

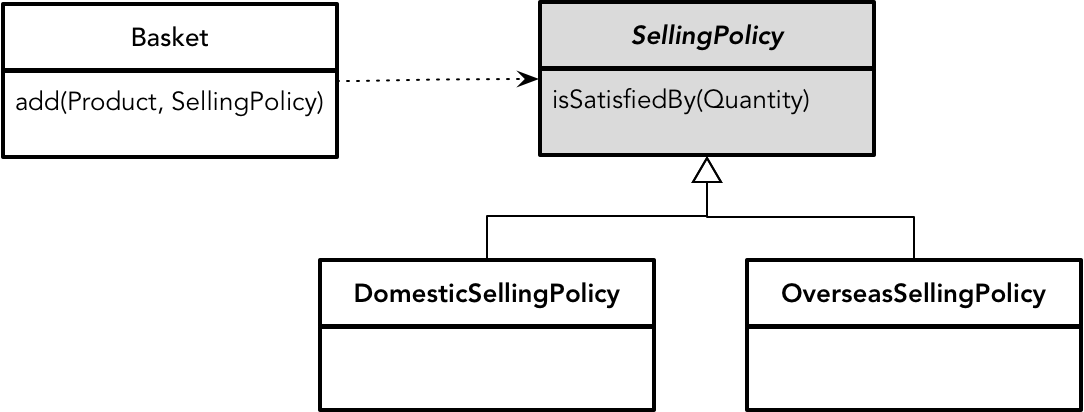
抽象不能无的放矢，关键在于识别变化点，只有对可能发生变化的功能进行抽象才是合理的设计。通常，我们将可能发生变化的功能点称之为热点（Hot Spot）。常见的热点包括业务规则、算法策略、外部服务、硬件支持、命令请求、协议标准、数据格式、业务流程、系统配置、界面表现。在领域建模特别是建立领域设计模型时，寻找这些热点是可扩展设计的关键。事实上，Robert Martin 提出的整洁架构就是为外部易变的部分与相对稳定的领域模型划分了清晰的边界，这个边界实则是通过抽象来隔离的。我在《领域驱动战略设计实践》中提到的南向网关，目的就是建立这样的抽象。

领域驱动设计的资源库（Repository）模式，就体现了封装变化的思想。资源库为聚合提供访问数据库的操作，而数据库访问的实现逻辑常常会发生变化，因而属于系统的热点。该热点自身遵循了自治对象的“最小完备”原则，即从职责来看，已经不可细分。为了降低耦合，就应该提取热点的共同特征，建立抽象接口：



32506101.png

如果热点还与其他职责黏合在一个对象中，需遵循自治对象的“稳定空间”原则，分离热点，然后再引入抽象，形成基于接口行为的对象协作方式。领域驱动设计的规格（Specification）模式体现了这一设计思路。业务规则是我们无法控制的，只要外部的需求发生变化，就可能调整业务规则。业务规则又是领域知识的重要组成部分，例如在电商领域，商品促销规则、支付规则、订单有效性验证规则随时都可能调整。这时，就需要将业务规则从领域模型对象中单独分离出来，识别规则的共同特征，为其建立抽象接口。例如电商网站的购物车验证规则，针对国内顾客和国外顾客的购买行为提供了不同的限制：



31951655.png

当然，我们也需要克制设计的过度抽象，不要考虑太多不切实际的扩展性与灵活性，避免引入过度设计，毕竟未来是不可预测的。Raphael Malveau 和 Thomas Mowbray 在 Software Architect Bootcamp 中警告了“弹性的弊端”，其症状包括如下几点：

* 过度复杂的工程。如果扩展过程异常复杂，那么实现了弹性的过程则是艰难而容易出错的。
* 许多明文的惯例。有时候，弹性设计具有很多的编码惯例，它们以恼人的细节来阻止你对体系进行破坏。
* 额外的编码。为了使用一种配置性服务，客户端必须参数化其请求。而服务提供者为了处理所有的选项，可能会更加复杂。额外的复杂性可能会堆积在一个弹性接口的两端。

引入抽象的可扩展设计需要结合具体的业务场景做出判断。我们应该首先考虑职责的合理分配，从自治对象的角度保证对象的稳定性，寻找到变化的可能，然后再进行合理的抽象。抽象时，又要从对象分治的角度确保对象之间的协作是不同角色行为之间的协作。此时的角色，就是抽象的潜在目标，即定义可能的角色接口。抽象应保持足够的前瞻性，又必须恰如其分，最好是水到渠成的设计决策。

### 分享交流

我们为本课程付费读者创建了微信交流群，以方便更有针对性地讨论课程相关问题。入群方式请到第 6-1 课末尾添加小编的微信号，并注明「DDD」。

阅读文章过程中有任何疑问随时可以跟其他小伙伴讨论，或者直接向作者提问（作者看到后抽空回复）。 **你的分享不仅帮助他人，更会提升自己** 。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持