依赖之殇

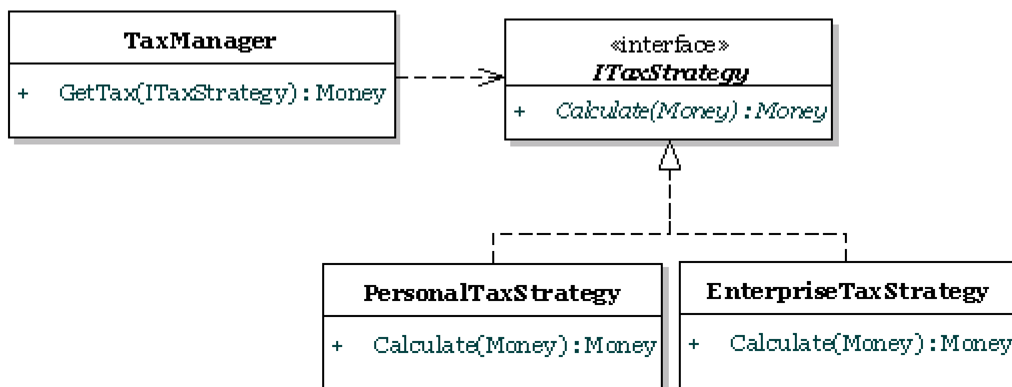
张逸
www.agiledon.com

没有对象协作的系统是不可想象的，因为此时的系统就是一个庞大的类，一个无所不知的"上帝类"。每个对象都有自己的自治领域，"各人自扫门前雪"，对象定义的法则就是这么自私。单一职责原则（SRP）[1]体现的正是这样的道理。对象的职责越少，则对象之间的依赖就越少。这一前提就是对象具有足够的高内聚与细粒度。这样的对象一方面有利于对象的重用，另一方面也保证了对象的稳定性。

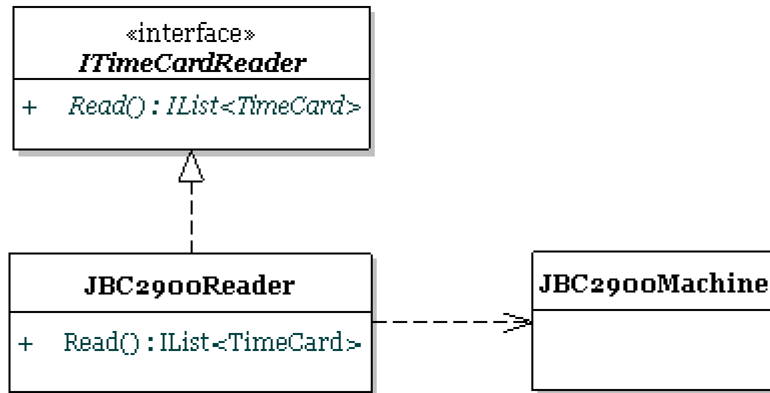
对象的职责可以是自己承担，也可以委派给其他对象。因此，有对象就必然有依赖，正如有人就有江湖。那么，我们该如何降低对象之间的依赖？第一要则是依赖于抽象，如依赖倒置原则（DIP）[2]所云。如果无法依赖于抽象，则至少应该保证你所依赖的对象是足够稳定的。事实上，最稳定的对象就是抽象对象，所以万法归一，稳定才是降低依赖的基础。

依赖之殇的源头是"变化"。变化与稳定显然是矛盾的，软件设计的最大问题就是如何协调这两者之间的矛盾。我们需要像高明的杂技师，要学会掌握平衡，能够在钢丝绳上无碍的行走。那么，如何解决变化带来的影响呢？答案是利用封装来隔离变化。

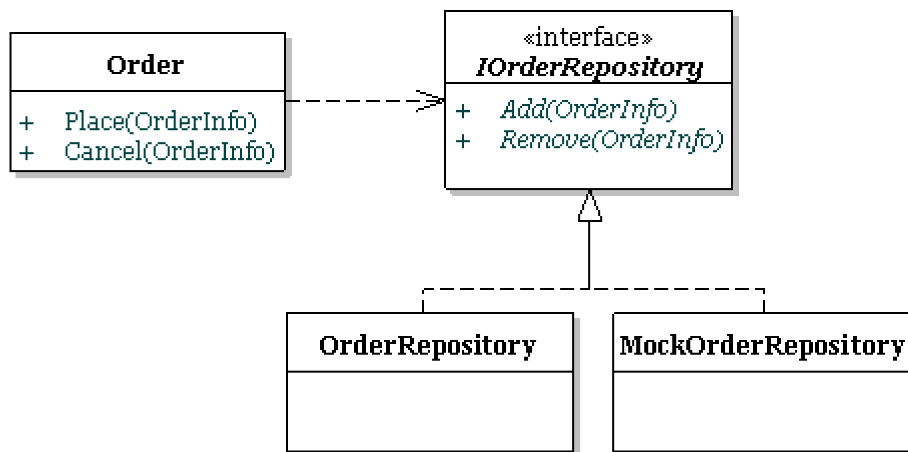
封装的一种方式是抽象，因为相对于实现而言，接口总能保持一定的稳定性。例如税收策略。对于调用方而言，只是希望能够得到准确的税值，至于如何计算，则不是他关心的内容。抽象出计算税值的接口，就能够隔离调用方与可能变化的税收策略之间的依赖关系，如下图所示：



利用抽象还可以解除对特定实现环境例如外部资源、硬件或数据库的依赖。此时抽象隔离的变化可能是外部环境提供的 API。例如，在考勤系统中，利用抽象隔离不同型号考勤机的变化。



利用抽象解除对象之间的依赖，还可以保证系统具有良好的可测试性。因为调用者依赖于抽象接口，就为我们引入 Mock 对象（当然也可以是 Fake 对象）执行单元测试提供了方便。尤其是当我们对领域对象进行测试时，如果领域对象需要对数据库操作，可以通过依赖抽象的持久对象（或仓储对象）实现职责的委派。此时，我们可以引入持久对象的 Mock 对象，模拟领域对象持久化的职责，既分离了领域对象与数据库资源的依赖关系，又能够提高单元测试的效率。




```
public interface IOrderRepository {
    void Add(OrderInfo order);
    void Remove(OrderInfo order);
}
public class Order {
    private IOrderRepository m_repository;
    public Order(IOrderRepository repository) {
        m_repository = repository;
    }
    public void Place(OrderInfo order) {
        if (order.Validate()) {
```

```
            m_repository.Add(order);
        }
    }
}
```

利用封装隔离变化，并非必须依赖于抽象，根据不同的场景，降低要求，依赖于较为稳定的具体类对象也是可行的。这是一种降低复杂度的设计方式。例如，我们可以引入一个 Helper 类来封装第三方 API 的调用，从而实现调用方与第三方 API 的隔离。例如为 SQL Server 数据库操作定义一个 Helper 类：

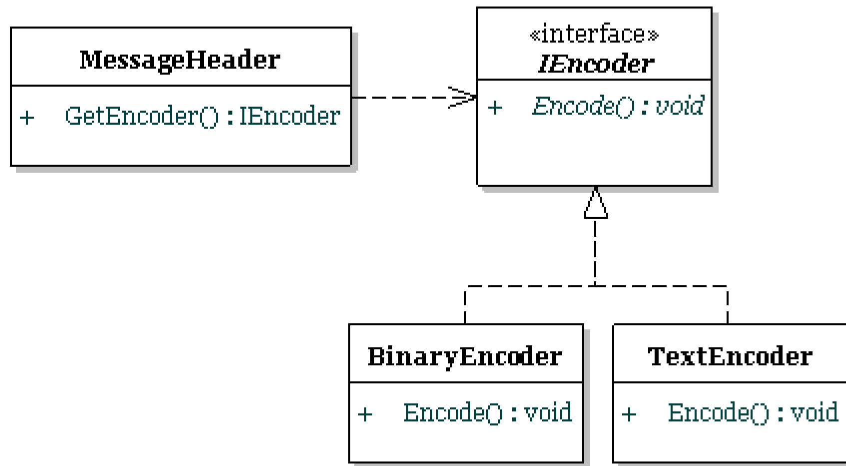
```
public static class SQLHelper {
    public int ExecuteNonQuery() {}
    public DataSet ExecuteQuery() {}
}
```

这样的设计类似于 Gateway 模式[3]，利用一个 Gateway 对象来封装外部系统或资源访问。具体类对象显然不如抽象接口稳定，因此在设计时，我们需要遵循单一职责原则。这样的设计体现了 DRY[4]原则，利用封装避免代码的重复，避免解决方案蔓延的坏味道[5]。合理的封装可以将变化点集中或限制到一处，以应对变化。一个常见的例子是利用简单工厂模式，将所有对象的创建集中在一个类中（当然也可以按模块创建不同的静态工厂）。即使创建的产品对象发生了变化，我们也可以只修改静态工厂类一处的实现。简单工厂模式常常可以应用在领域层中，通过工厂对象创建持久层对象（或所谓的数据访问对象）。

依赖源于对象的协作。传递依赖的方式可以通过属性，构造函数或方法的参数。若要保证对象间的松散耦合，构造函数或方法的参数以及属性的类型就应定义为抽象类型，如前面例子中的 Order 类。这是依赖解耦的关键方式，完全符合“面向接口设计”的编程思想，同时，它也有利于我们在后期实现“依赖注入”。

然而，产生依赖的方式绝不仅限于上述三种情形。例如，方法的返回值以及方法体中局部对象的创建，同样可能产生依赖。比较而言，这种依赖关系更难解除，因为它与具体的实现紧密相关。换句话说，因为这两种情形的依赖都涉及到具体对象的创建，且由实现者完成，而不能转交给调用方。例如，在如下的设计中，消息头会决定消息编码的方式。

捷道咨询 版权所有©2010 http://www.agiledon.com



MessageHeader 的 GetEncoder()方法需要返回一个 IEncoder 对象，这就要求在方法体中创建一个具体的 IEncoder 对象。要解除这样的依赖关系非常困难，如需彻底解除，一种可能是利用反射技术，通过具体类的类型来创建。还有一种可能是利用"惯例优于配置"实现解耦。如果不需要彻底解除依赖，也可以利用"表驱动法"，或者直接将条件分支语句封装到方法中。

如果在方法实现中需要创建一个局部对象，我们可以考虑简单工厂模式或 Registry 模式 [3]。例如，在 Role 对象的 IsAuthorized()方法中，需要创建一个 PriviledgeFinder 对象，通过调用它的 FindPriviledges()方法获得角色对应的权限集。此时，我们可以在 Registry 对象中提供 PriviledgeFinder 对象：

```
interface IPriviledgeFinder {
    IList<Priviledge> GetPriviledges(int roleID);
}
public class PriviledgeFinder:IPriviledgeFinder {}
public class Registry {
    private Registry() { }
    private static Registry Instance = new Registry();
    protected virtual IPriviledgeFinder m_priviledge =
        new PriviledgeFinder();
    public static IPriviledgeFinder PriviledgeFinder() {
        return Instance.m_priviledge;
    }
}
public class Role {
    public bool IsAuthorized() {
        IList<Priviledge> priviledges =
```

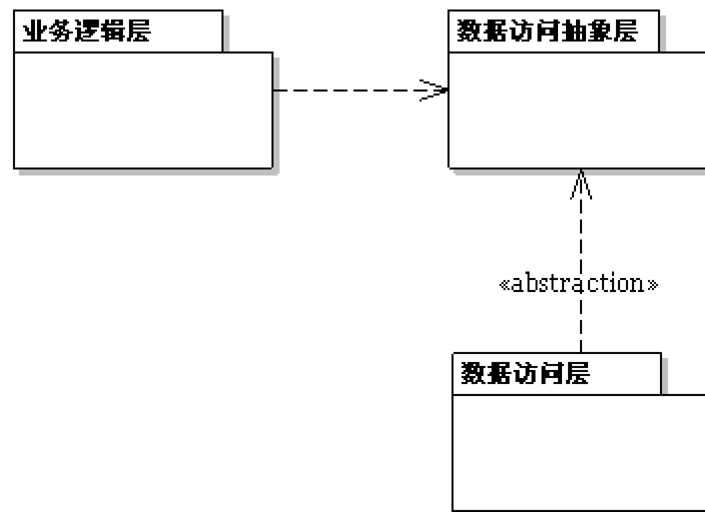
```
            Registry.PriviledgeFinder().GetPriviledges(this.ID);
    }
}
```

上述实现实际上仍然利用了“将变化集中在一处”的设计原则。注意 Registry 类中的 m_priviledge 属性是 virtual 的受保护属性，它提供了一种变化的可能，可以交由子类去实现。

如何知道一个类是否过多的依赖其他类？一个办法就是创建这个类，并保证创建的对象能够正常使用。如果创建的过程非常复杂，就说明该类的依赖过多。此时，可以考虑分解该类的职责。如果这些依赖是必须的，则可以考虑利用封装，例如将对外部对象的调用修改为在内部创建（应用 builder 模式）；也可以考虑使用 Factory Method 模式或者利用简单工厂。

依赖关系不仅仅只限于类与类之间，包（组件、模块、层）与包（组件、模块、层）之间同样存在依赖关系。良好的设计需要包之间保持松散耦合。大体上讲，包之间的依赖解除与类之间的依赖解除方式是一致的。即：要求一个包尽量依赖于一个稳定的包。注意，一个包依赖于另一个包，就代表着它依赖于这个包的每一个类。Robert C. Martin 说：“我放入一个包中的所有类是不可分开的，仅仅依赖于其中一部分的情况是不可能的。”[6]因此，我们可以将一个包看做是一个类，它仍然要求职责的高内聚。在包中对类的分配，就相当于是对类进行一次分类。共同封闭原则[6]要求：“包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包产生影响，则将对该包中的所有类产生影响，而对于其他的包不造成任何影响。”简言之，我们在对包进行设计时，需要避免将不同的职责耦合在一个包中，它会造成变化点的扩散。

解除包之间依赖关系的一个重要方法仍然是抽象。使用 Seperated Interface 模式[3]，在一个包中定义接口，而在另一个与这个包分离的包中实现这个接口。例如在分层架构模式中，我们常常对数据访问层进行抽象，使得业务逻辑层依赖于该抽象层，而不是它的低层模块。



上图的设计实际上是依赖倒置原则的体现。在项目开发中，这种将抽象与实现分别放在不同的包中，是系统设计中常见的方式。这样的设计也能够更好地应用在分布式开发场景中。

[1]单一职责原则（Single Responsibility Principle）：就一个类而言，应该只专注于做一件事和仅有一个引起变化的原因;

[2]依赖倒置原则（Dependency Inversion Principle）：高层模块不应该依赖于低层模块，二者都应该依赖于抽象；抽象不应该依赖于细节，细节应该依赖于抽象;

[3]Martin Fowler, Patterns of Enterprise Application Architecture;

[4]DRY 原则，即“不要重复你自己（Don't Repeat Yourself）”它要求“系统中的每项知识只应该在一个地方描述。”

[5]Joshua Kerievsky, Refactoring to Patterns;

[6]Robert C. Martin Agile Software Development:Principles,Patterns and Practices