

设计模式

>> Design Patterns

吴映波

wyb@cqu.edu.cn

虎溪Office: 虎溪学院楼410

A区Office: 九教205

Tel : 13594686661

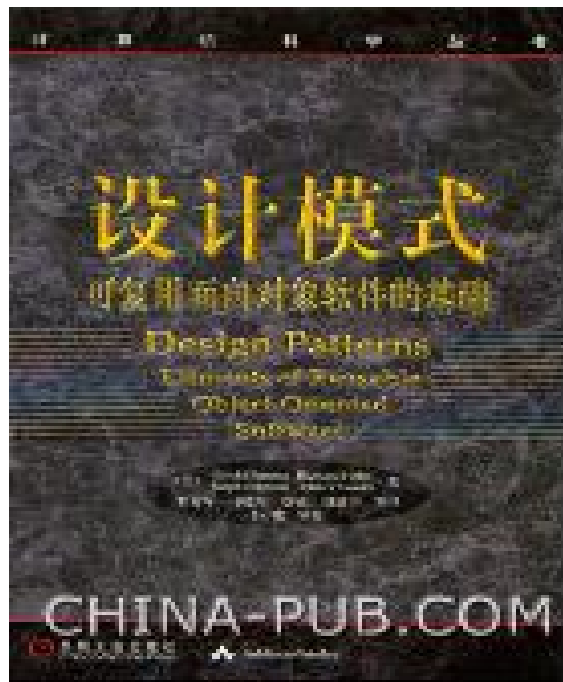
About the course

► Textbook

- 《设计模式：可复用面向对象软件的基础》，机械工业出版社。

► 其它

- 《Java设计模式》，电子工业出版社
- 《C# 3.0设计模式》
- 《设计模式之禅》
- 《Head First设计模式》



关于 “GoF Design Pattern”

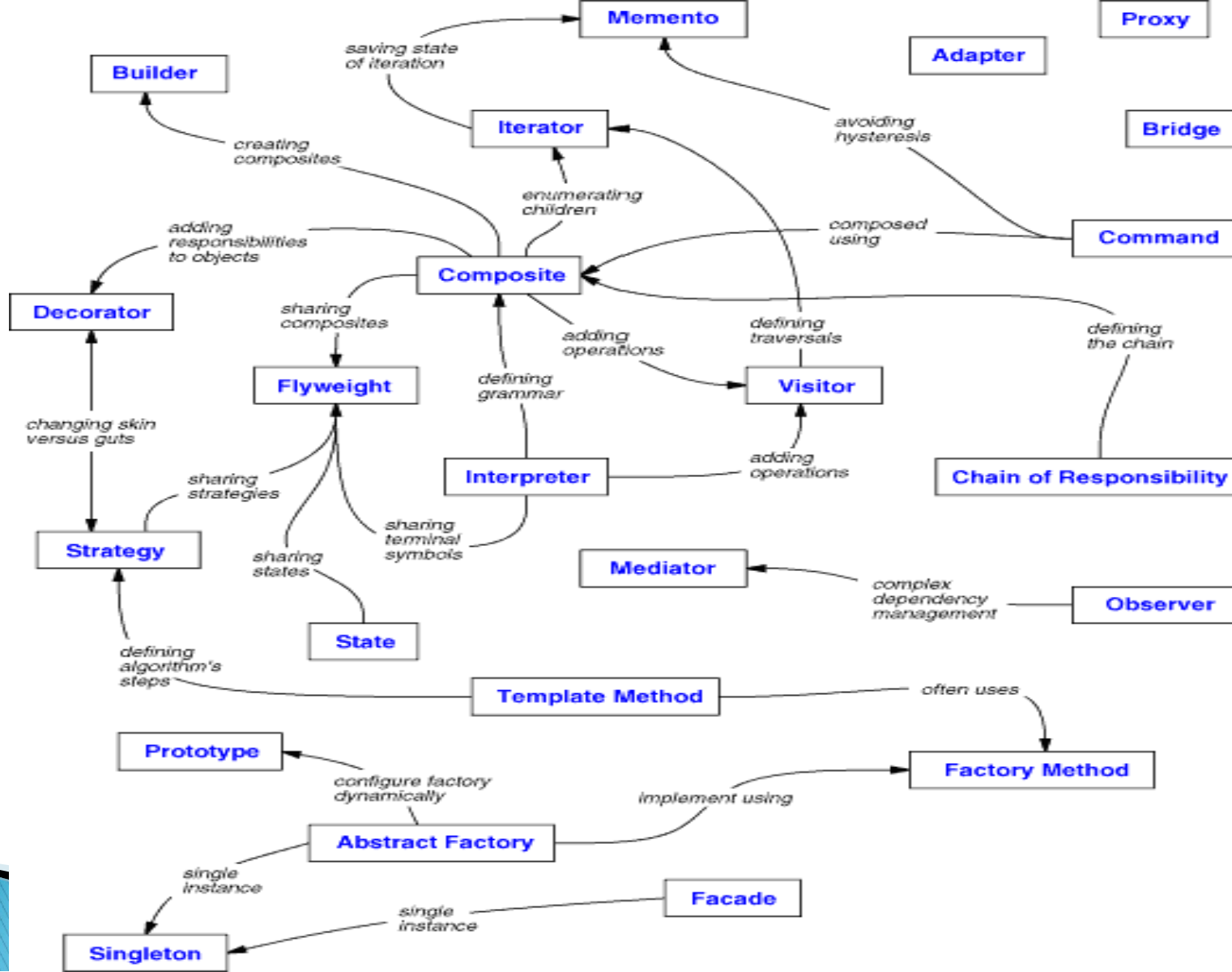
- ▶ 软件设计经验的总结，这些设计经验可以被重用，但不是简单的代码重用
 - They solve common problems in a “proven” way
 - They tend not to be implementation specific
 - They tend to be classified in a common way – context, forces, examples, etc
 - They embody good design principles.
- ▶ 一套描述模式的词汇，可用于交流和文档化

关于 “GoF Design Pattern”

▶ 分类：

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

GoF 设计模式全图



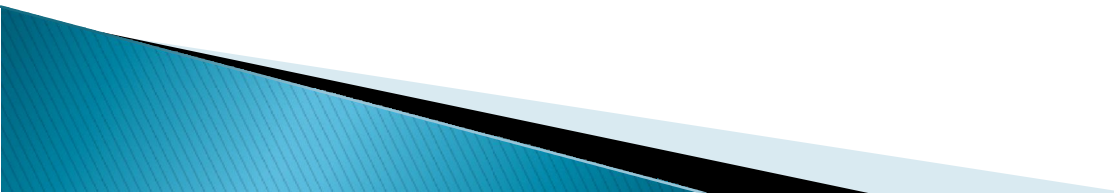
About the course

► Schedule

时间	内容安排
13周	<ul style="list-style-type: none">● 设计模式基础● 创建型设计模式
14周	<ul style="list-style-type: none">● 结构型设计模式● 行为型设计模式
15周	<ul style="list-style-type: none">● 讨论● 复习&答疑

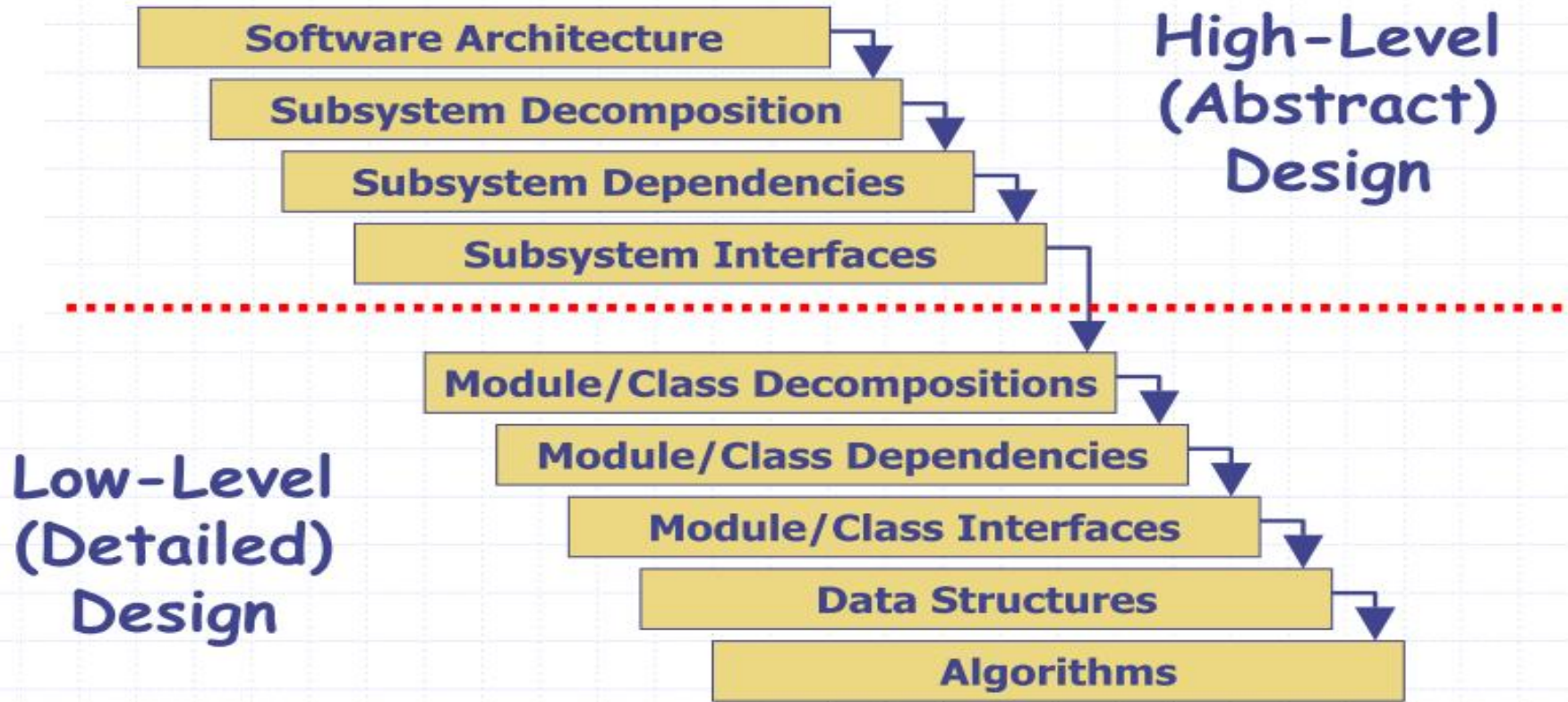
About the course

- ▶ OO fundamentals and UML is essential to achieving success in this course.
- ▶ Exercise sets and tutorials are important.
 - Give you a chance to exercise your skills.
 - If you do not understand, try it out, post to the mailing list, ask your classmates or instructor, ...
- ▶ Lectures
 - If you do not understand, ask!



什么是“软件设计”

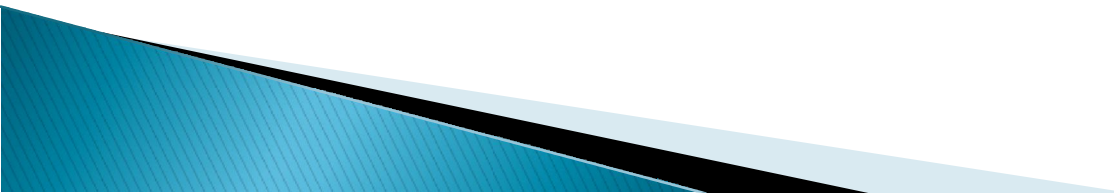
► 设计层次 (Design Levels)



Why do we design...

- ▶ A software design is not necessary for trivial systems, but for large systems a design is essential
 - Manage complexity
 - Validation of delivered software
 - Simplify future maintenance
 - A mechanism for communication between domain experts and technical professionals
 - Enables Visualization
 - Enables project team members to work concurrently
 - Partitioning the work effort with limited overlap
 - Example: Concurrently developing test cases while the code is being development

What is a good design?

- ▶ Satisfies user needs.
 - ▶ Is as simple as possible. (Kent Beck)
 - Fewest number of classes or methods
 - Simple Interfaces: Endless flexibility adds complexity. Complex interfaces mean:
 - hard to understand by users and developers
 - many possible variations of use
 - inconvenient to change interface in order to eliminate “bad options”.
- 

GRASP模式概念

- ▶ GRASP是General Responsibility Assignment Software Patterns（通用职责分配软件模式）的简称，它的核心思想“职责分配”，即Responsibility Assignment。
- ▶ GRASP的主要特征：
 - 对象职责分配的基本原则。
 - 主要应用在分析和建模上。
- ▶ GRASP的核心思想的理解：
 - 自己干自己的事（职责的分配）
 - 自己干自己的能干的事（职责的分配）
 - 自己只干自己的事（职责的内聚）
- ▶ 也就是说，如何把现实世界的业务功能抽象成对象，如何决定一个系统有多少对象，每个对象都包括什么职责，GRASP模式给出了最基本的指导原则。

GRASP基本原则

- ▶ GRASP提出了九个基本原则，用来解决面向对象设计的一些问题。
 - 信息专家(Information expert)
 - 创建者(Creator)
 - 高内聚(High Cohesion)
 - 低耦合(Low Coupling)
 - 控制者(Controller)
 - 多态 (Polymorphism)
 - 纯虚构 (Pure Fabrication)
 - 间接性 (Indirection)
 - 变化预防 (Protected Variations)

信息专家(Information expert)

- ▶ 将职责分配给具有履行职责所需要的信息的类
- ▶ 通俗点就是：该干嘛干嘛去，别管别人的闲事或者我的职责就是搞这个，别的事不管。
- ▶ 举个简单的例子，如果有一个类是专门处理字符串相关的类，那么这个类只能有字符串处理相关的方法，而不要将日期处理的方法加进来。也就是提高软件高内聚一种原则。

例子：

- 为了说明问题，我们使用“学生成绩管理系统”来说明。

用例1：

- 管理员创建题库（把题条加入题库）

再细化一下：

- 管理员创建题库（把题条加入题库）：如果题库中已经存在所给的题条，则退出，否则加入题条。

这样就存在3个对象：管理员用户User，题条SubjectItem，题库SubjectLibrary

2个职责：判断（新加入的题条是否与题库某题条相等），加入（题条的加入）

- 这2个职责究竟应该由哪个对象执行？

我们使用Information Expert模式来分析。

1. 判断2个题条是否相等，只要判断题条的ID属性（或其它属性）是否相等就可以了。题条的ID是属于题条对象的，所以对它的操作应该放在题条SubjectItem里。

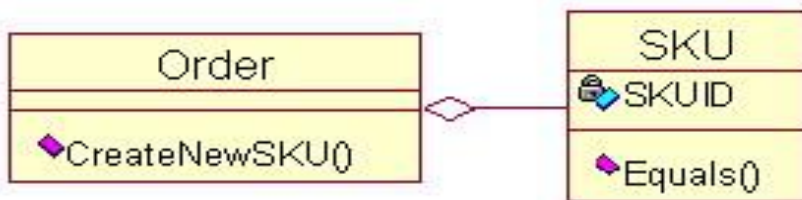
2. 题条的加入需要操作的数据有2部分，一部分是新加入的题条本身，另一部分是题库（加入到题库），题条是题库的一部分，所以题条的加入应该放在题库SubjectLibrary里完成。

创建者(Creator)

- ▶ 将创建一个类A的实例的职责指派给类B的实例，如果下列条件满足：
 - a) B聚合了A对象
 - b) B包含了A对象
 - c) B记录了A对象的实例
 - d) B要经常使用A对象
 - e) 当A的实例被创建时，B具有要传递给A的初始化数据(也就是说B是创建A的实例这项任务的信息专家)
 - f) B是A对象的创建者
- ▶ 如果以上条件中不止一条成立的话，那么最好让B聚集或包含A。
- ▶ 通俗点就是：我要用你所以我来创建你，请不要让别人创建你
- ▶ 这个模式是支持低耦合度原则的一个体现

例子：

- ▶ 例如：因为订单 (Order) 是商品 (SKU) 的容器，所以应该由订单来创建商品。 如下图：



创建者图

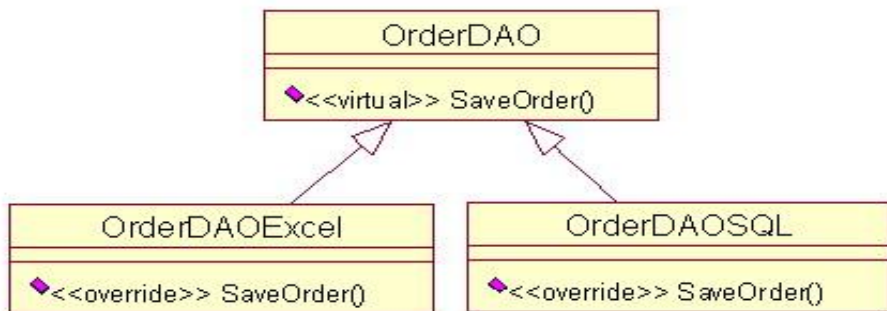
- ▶ 这里因为订单是商品的容器，也只有订单持有初始化商品的信息，所以这个耦合关系是正确的且没办法避免的，所以由订单来创建商品。

高内聚(High Cohesion)

- ▶ 分配一个职责的时候要保持类的高聚合度。
- ▶ 内聚度(cohesion)是一个类中的各个职责之间相关程度和集中程度的度量。一个具有高度相关职责的类并且这个类所能完成的工作量不是特别巨大，那么他就是具有高聚合度。

例子：

- ▶ 例如：一个订单数据存取类 (OrderDAO)，如果订单既可以保存为 Excel 模式，也可以保存到数据库中；那么，不同的职责最好由不同的类来实现，这样才是高内聚的设计，如下图：



高内聚图

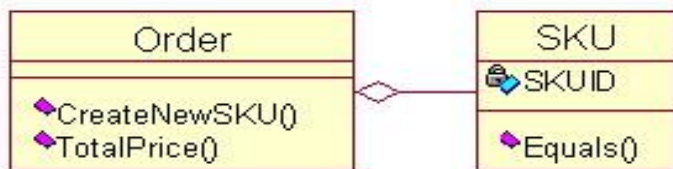
- ▶ 这里把两种不同的数据存储功能分别放在了两个类里来实现，这样如果未来保存到 Excel 的功能发生错误，那么就去检查 OrderDAOExcel 类就可以了，这样也使系统更模块化，方便划分任务，比如这两个类就可以分配个不同的人同时进行开发，这样也提高了团队协作和开发进度。

低耦合(Low Coupling)

- ▶ 在分配一个职责时要使保持低耦合度。
- ▶ 耦合度(coupling)是一个类与其它类关联、知道其他类的信息或者依赖其他类的强弱程度的度量。一个具有低(弱)耦合度的类不依赖于太多的其他类。

例子：

- 例如：Creator 模式的例子里，实际业务中需要另一个出货人来清点订单 (Order) 上的商品 (SKU)，并计算出商品的总价，但是由于订单和商品之间的耦合已经存在了，那么把这个职责分配给订单更合适，这样可以降低耦合，以便降低系统的复杂性。如下图：



低耦合图

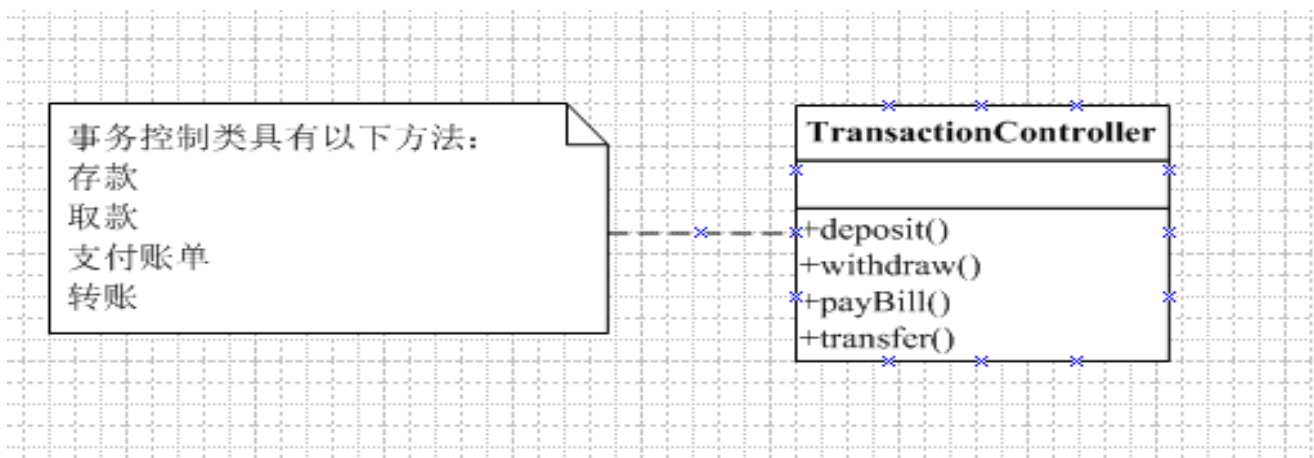
- 这里我们在订单类里增加了一个 `TotalPrice()` 方法来执行计算总价的职责，没有增加不必要的耦合。

控制者(Controller)

- ▶ 将处理系统事件消息的职责分派给代表下列事物的类：
 - a) 代表整个“系统”的类
 - b) 代表整个企业或组织的类
 - c) 代表真实世界中参与职责的主动对象类（例，一个人的角色）
 - d) 代表一个用况中所有事件的人工处理者类
- ▶ 这是一个控制者角色职责分配的原则，就是哪些控制应该分派给哪个角色。

例子：

- ▶ 例如，在银行系统中，我们会会有一个控制器类用于处理所有银行事务，即类TransactionController。

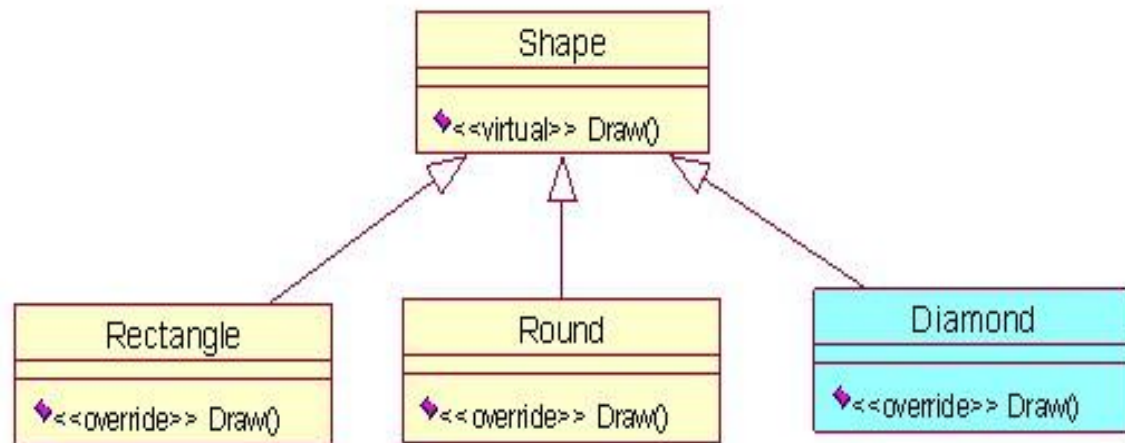


- ▶ 当出纳点击用户界面中用于执行支付账单的按钮时，这个事务控制类TransactionController就会取得并且处理这个事件。

多态 (Polymorphism)

- ▶ 当相关的可选择的方法或行为随着类型变化时，将行为的职责分配给那些行为变化的类型。
- ▶ 也就是说尽量对抽象层编程，用多态的方法来判断具体应该使用那个类，而不是用if 来判断该类是什么接来执行什么。
- ▶ 提倡通过多态操作把基于类型的可变行为的定义职责分配给行为发生的类。所谓多态性，简单地说，就是具有同一接口的不同对象对相同的消息具有不同的行为。或者说同一消息作用于不同的对象，而产生不同的结果。

例子：



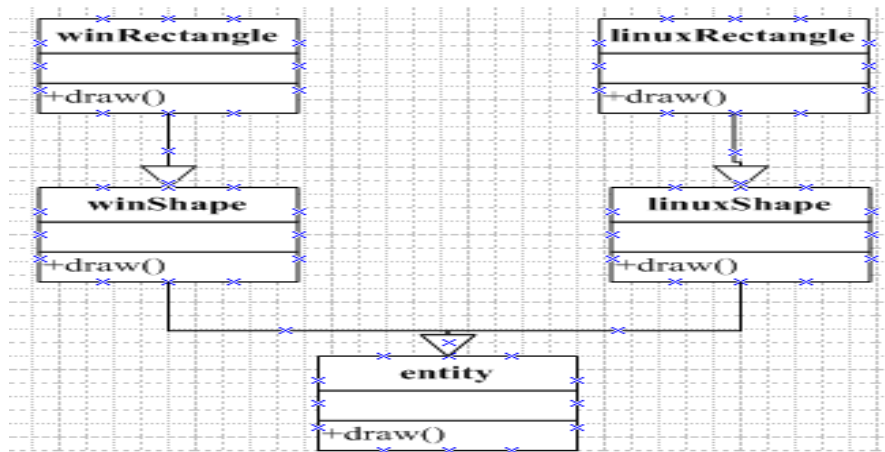
多态图

纯虚构（Pure Farication）

- ▶ 把那些非问题领域的职责分配给那些人工生成的或者容易此类职责的领域类（Domain Class）。
 - **Domain Class**：设计对象的时候应该尽量保持与现实世界里的对象一致。这种与现实世界里的对象保持一致的从业务分析中抽象出来的类叫做“Domain Class”。
 - ▶ 比如一个简单的用例：用户注册。用户就是一个“Domain Class”，它是现实世界里的业务对象。
 - ▶ 用户注册需要操作数据库，[数据库操作]是系统功能实现的一个必需功能，它不是现实世界里存在的业务对象，它是一个非Domain Class。如果把[数据库操作]看作一个行为职责，它就相当于这里所说的“非问题领域里的职责”。
- 一般来说，Domain Class与非Domain Class的功能如果聚集在一个类里，就破坏了“高内聚”原则。

例子：

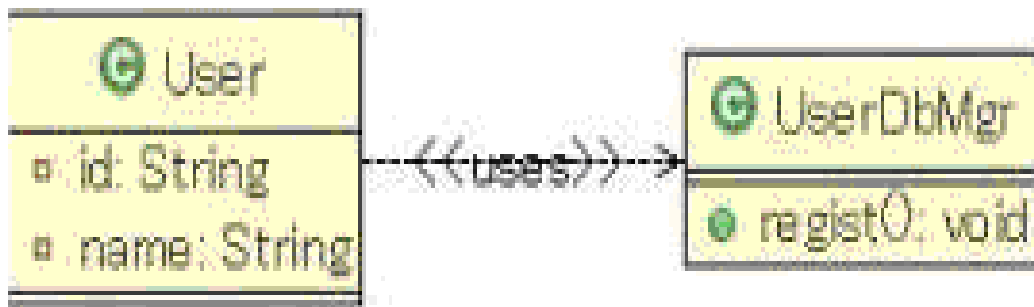
- ▶ 如上节的绘画例子，如果我们的系统要考虑在不同的系统，如Linux及Windows下绘画，该如何设计？如何使我们设计的类因系统的不同而独立？我们应该设计一个高层次的类，用于分配高内聚的责任，如图：



- ▶ 上面的entity类就是纯虚构的设计，无论哪一个系统(Windows或Linux)，都需要实体来完成绘画。所以纯虚构来表现为一种概念。

例子：

- ▶ 以上述“用户注册”的用例为例，对于Domain类“用户（User）”，如果把“数据库操作的职责”分配给“用户（User）”，那么User类的内聚性大大降低。
- ▶ 应用Pure Fabrication模式，应该人工定义一个数据库管理的非Domain类UserDbMgr，把数据库操作的功能分配给它完成。



- ▶ 如图，分离Domain类User与非Domain类UserDbMgr，User类只保持问题领域中的信息。保证了高内聚性，和易重用性。

中介者（Indirection）

- ▶ 将职责分配给一个中间对象以便在其他对象之间仲裁，这样这些对象没有被直接耦合。这个中间对象 (intermediary) 在其他对象间创建一个中介者 (Indirection)。
- ▶ 当多个类之间存在复杂的消息交互（关联）时，Indirection 模式提倡类之间不直接进行消息交互处理（非直接），而是导入第三方类，把责任（多个类之间的关联责任）分配给第三方类，降低类之间的耦合程度。

保护变化（Protected Variations）

- ▶ 提倡在可预测的变化或不安定因素的周围，用稳定的接口来承担职责。
 - 变化点：现有、当前系统或需求中的变化；
 - 演化点：预测将来可能会发生的变化点，但并不存在于现有需求中。在面向对象设计中，面向接口编程便符合Protected Variations模式的概念。
- ▶ 预先找出不稳定的变化点，使用统一的接口封装起来，如果未来发生变化的时候，可以通过接口扩展新的功能，而不需要去修改原来旧的实现。也可以把这个模式理解为 **OCP(开闭原则) 原则**，就是说一个软件实体应当**对扩展开放，对修改关闭**。在设计一个模块的时候，要保证这个模块可以在不需要被修改的前提下可以得到扩展。这样做的好处就是通过扩展给系统提供了新的职责，以满足新的需求，同时又没有改变系统原来的功能。

例子：

- ▶ 例：把一段字符串保存到文件，打印机等输出设备。

