## 概念

##### 1. 软件工程

1. 应用系统的、规范的、可量化的方法来开发、运行和维护软件，即将工程应用到软件。
2. 对应用系统的、规范的、可量化的各种方法的研究。

##### 2. 软件演化生命周期模型

1. **初始阶段**：第一个版本的软件产品开发；
2. **演化**：对演化增量进行处理以保持软件产品的持续增值；
3. **服务**：用户使用，开发者维护；
4. **逐步淘汰**：用户仍使用，开发者不维护；
5. **停止**：用户不使用，开发者不维护；

##### 3. 螺旋模型

按照风险解决的方式来组织软件开发活动；

##### 4. 软件验证与确认

在整个软件生命周期中对软件规范性评估活动，以保证软件开发各个环节的正确性；

##### 5. 增加开发模型和迭代开发模型

增量迭代模型：在项目早期界定项目的目标和范围，将后续开发活动组织为多个迭代、并行的瀑布式开发活动。

##### 6. 质量保障包括哪些活动

**需求开发阶段：**1.需求评审。2. 需求度量；

**建立体系结构阶段：**1.体系结构评审。2. 集成测试。

**详细设计阶段：**1. 详细设计评审。 2.设计度量。3.集成测试。

**实现（构造）阶段：**1.代码评审。2.代码度量。3.测试。

**测试阶段：**1.测试。2. 测试度量。

##### 7. 软件体系结构的分层风格

根据不同的抽象层次将系统组织为层次式 结构，每隔层次被建立为一个部件，不同部件之间通常用程序调用方式进行连接。

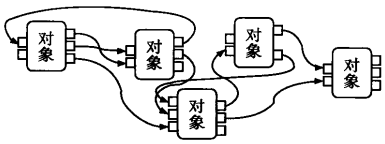
**优点：**设计机制清晰易于理解。支持并行开发。更好的可复用性和内容可修改性。

**缺点：**交互协议难以修改。性能损失，难以确定层次数量和粒度。

##### 8. 软件需求

1. 需求就是用户的一种期望，用户为了解决问题或达到某些目标所需要的条件或能力。
2. 系统或系统部件为了满足合同、标准、规则或其他正式文档所规定的要求而需要具备的条件或能力。
3. 对上述中的一个条件或一种能力的一种文档化表述。

##### 9. 图示解释面向对象式软件体系结构风格，说明其优缺点



该风格将系统组织为多个独立的对象，每个对象封装其内部的数据，并基于数据对外提供服务，不同对象之间通过协作机制共同完成系统任务。

**优点：**

1. 内部实现的可修改性。

2. 易开发、易理解、易服用的结构组织。

**缺点：**

1. 接口的耦合性。
2. 标识的耦合性。
3. 副作用。借鉴了面向对象的思想，也引入了了面向对象的副作用。A、B都使用对象C，B对C的修改可以会对A产生未预期的影响。

##### 10. 软件质量保障主要有哪三种手段

**评审：** 由作者之外的其他人来检查产品的问题。

**测试：**主要包括单元测试、集成测试、系统测试。

**质量度量：**用数字量化的方式描述软件产品。

##### 11. 持续集成

尽早集成和频繁集成，即在开发之初就利用stub开始集成和每次开发完成一些任务后就可以替换stub中的相应组件。

##### 12. 需求分为哪3个层次

**业务需求：**用于描述为什么开发系统；

**用户需求：**执行实际工作的用户对系统所能完成的具体任务的期望

**系统级需求：**用户对系统性行为的期望，每个系统级需求反应了一次外界与系统的交互行为，或者系统的一个实现细节。

##### 13. 从3个层次出发，各描述一条图书管理系统中的需求

**业务需求：**提供在线借阅、归还图书的服务。（抽象的描述一个目标，不指明一个用户，仅描述完成的功能，如系统应允许（用户）使用…服务…）

**用户需求：** 系统应该允许用户在线搜索馆内图书信息。（系统应允许（用户）使用…服务…）

**系统级需求：**用户在输入图书名或者其他检索信息后，系统应该提供所有可借阅书籍列表。（描述具体用户的一个具体任务，指明用户身份和完成的功能，如在（用户）做了什么事情之后，系统应该（做出什么样的响应））

##### 14. 黑盒测试

将测试对象看成一个黑盒子，完全基于输入和输出数据来判定测试对象的正确性。

##### 15. 设计黑盒测试用例有哪些方法

1. **等价类划分：**把所有可能的输入域划分成若干子集，然后从每一个子集中选取具有代表性的数据作为测试用例；
2. **边界值分析：**对等价类划分的补充，针对边界情况设计测试用例；
3. **决策表：**用于设计复杂逻辑的测试用例；
4. **状态转换：**用于处理输入输出与状态相关联的复杂测试对象；

##### 16. 工程和科学的区别

科学是关于事务的基本原理和事实的有组织、有系统的知识。科学的主要任务是研究世界万物变化的可观规律，他解决“为什么”的问题。

工程是自然科学或各种专门技术应用到生产部门中形成的各种学科的总称，其目的在于利用和改造自然科学来为人类服务。通过工程可以生产或开发出对社会有用的产品。科学可以作为工程的指导知识，例如，软件工程的指导知识是计算机科学。

##### 17. 什么是集成测试，什么是单元测试，单元测试用例和集成测试用例有什么区别

**单元测试：**对程序单元进行正确性检验的测试工作。

**集成测试：**通过大爆炸式的集成策略或者增量集成策略对系统的接口进行正确性检验的测试工作。

**区别：**单元测试主要关注代码段内部的具体逻辑，因此用例选择要尽可能的验证代码逻辑的完整性。集成测试则侧重于测试单元间的组合，因此用例选择也要侧重于选择能够使单元间交互的用例。

##### 18. 功能性需求和非功能性需求

**功能需求**是软件系统需求中最常见、最主要和最重要的需求，同时它也是最为复杂的需求，是一个软件产品能够解决用户问题和产生价值的基础。

**非功能需求**主要包括性能需求、质量属性、对外接口、约束、数据需求等。其中质量属性是非功能需求中影响最大的需求。

##### 19. 写出四个体系结构视角

组合视角：关注功能分解和运行时分解、子系统的构造，构件的服用。

逻辑视角：关注静态结构，类型与实现的服用。

依赖视角：关注互联分享。

信息视角：关注持久化信息。

接口视角：关注服务的定义、服务的访问。

模式视角：模式和框架的重用。

结构视角：设计主体的内部构造和组织。

交互视角：对象之间的消息通信。

##### 20. 质量模型的可用性

易学性：新手用户容易学习，能够很快使用系统。

效率：熟练用户使用系统完成任务的速度。

易记性：以前使用过软件系统的用户，能够有效记忆或者快速的重新学会使用该系统。

出错率：用户在使用系统时，会犯多少错，错误有多严重，以及是否能从错误中很容易地恢复。

主观满意度：让用户有良好的体验。

##### 21. 举例说明软件生命周期模型描述的是什么

软件生命周期模型是描述软件开发过程中各种活动如何执行的模型。有瀑布模型、原型模型、增量模型、迭代模型、螺旋模型等。以瀑布模型为例，它要求软件开发分为需求分析、软件设计、软件构造、软件测试、软件交付与维护阶段，每个阶段需要编写相应文档，且只有经过审核才能进入下一个阶段。

##### 22. 黑盒测试和白盒测试的差别

从三个方面回答（黑白定义、黑白各自测试方法、黑白各自优缺点）

黑盒测试是基于规格的，将测试对象看成一个黑盒子，完全基于输入和输出数据来判定测试对象的正确性。

白盒测试是基于代码的：将测试对象看作透明，按照测试对象内部的程序结构来设计测试用例进行测试。

黑盒的测试方法有：1.等价类划分；2.边界值分析。3.决策表。

白盒测试方法有：1.语句覆盖。2.条件覆盖。3.路径覆盖。

白盒优点：1.覆盖率高。2.发现缺陷的数量较多。

白盒缺点：1.测试开销大。2.不能校验需求规格。

黑盒优点：1.测试效率高。2.可以校验需求规格。

黑盒缺点：1.覆盖率低。2.发现缺陷的数量较少。

## 详细设计中面向对象方法下的模块化

### 一、类之间的关系

**关联：**如果某个类关联另一个类，那么它就持有另一个类的应用，则这个类所有的对象都具有向另一个类的对象发送信息的能力。

**继承：**子类可以访问父类的成员方法和成员变量。

面向对象中的关联关系（访问耦合）和继承关系（继承耦合）

### 二、降低访问耦合

#### 1. 针对接口编程

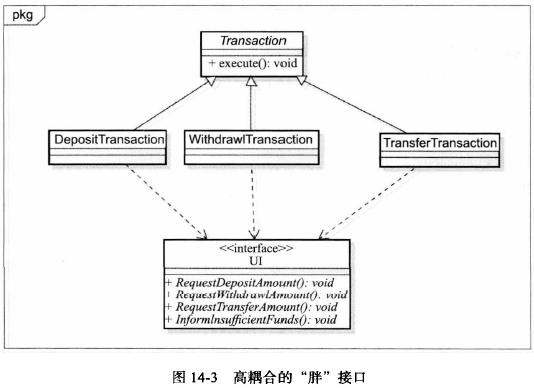
1. 只访问对象的接口（直接属性访问会导致公共耦合）
2. 避免隐式访问

如果为每个类都定义明确的契约（包括供接口和需接口），并按照契约组织和理解软件结构，那么就可以满足上述要求，这就是“针对接口编程”。

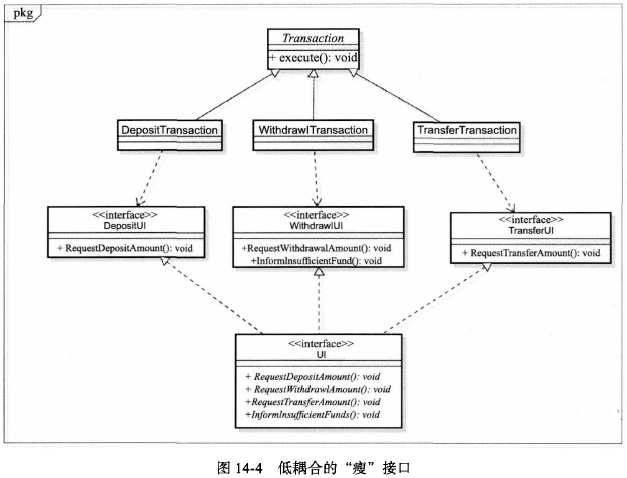
#### 2. 接口最小化/接口分离原则（Interface Segregation Principle/ISP）

将一个统一的接口匹配为多个更独立的接口，避免不必要的耦合，实现接口最小化。

如下所示，UI抽象接口的存在导致各个事物对象出现了不必要的耦合：DepositTransaction原本不需要依赖于除RequestDepositAmount() 之外的任何接口；WithdrawTransaction原本不需要依赖于除RequestWithdraw()和InformInsufficientFunds()之外的任何接口，等等。



UI抽象接口的存在合理性是很明显的，但是负责各种交互的UI接口的存在给Transaction的各个子类带来了不必要的耦合，需要将一个统一的接口匹配为多个更加独立的接口，这就是接口分离原则，可以避免不必要的耦合，实现接口最小化。



#### 3. 迪米特法则/访问耦合的合理范围（Demeter Law）

1. 每个单元对于其他的单元只能拥有有限的知识，只是与当前单元紧密联系。
2. 每个单元只能和它的朋友交谈，不能和陌生单元交谈。
3. 只和自己直接的朋友交谈。

如果一个对象O有一个方法M，那么M只能调用下列对象的方法：

O自己；

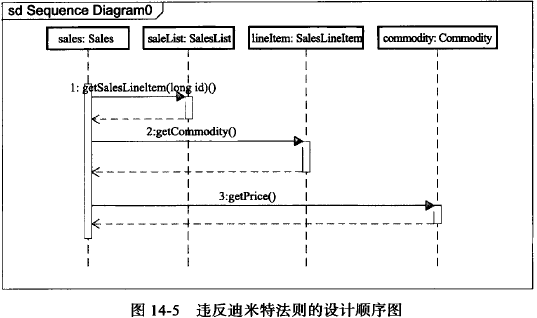
M中的参数对象；

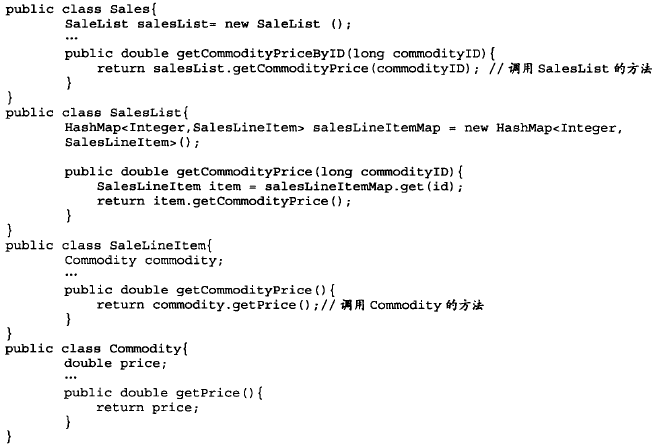
任何在M中创建的对象；

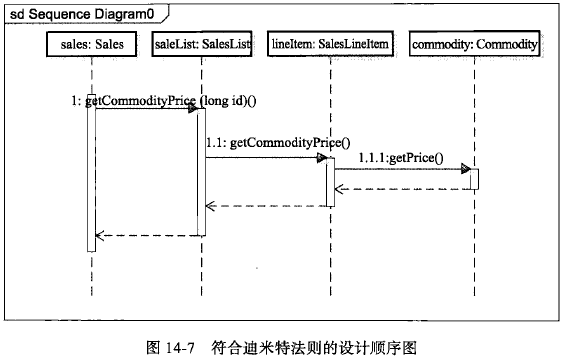
O的人员变量；

Sales类与Saleslist、SalesLineItem、commodity都有访问耦合。









### 三、降低继承耦合

#### 1. Liskov替换原则（里氏替换原则LSP）

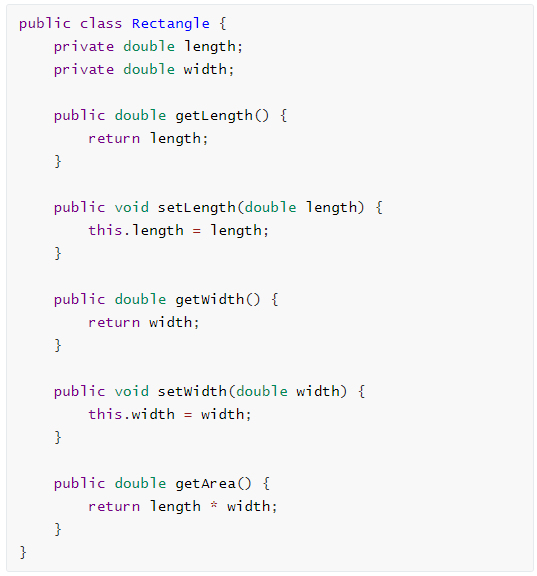
子类型必须能够替换掉基类型而起同样的作用（只要有父类出现的地方，都可以用子类替代）。

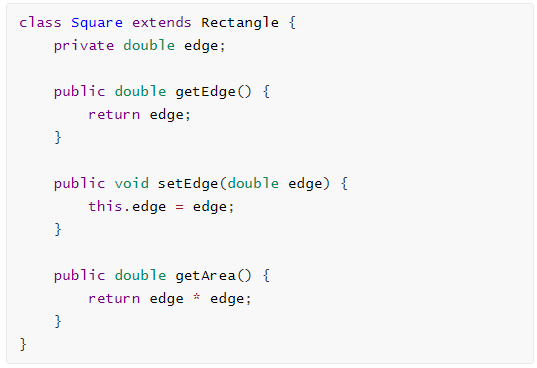
**对继承规则的四层约束**

1. 子类必须实现父类的抽象方法，但不得重写（覆盖）父类的非抽象方法；
2. 子类中可以增加自己特有的方法；
3. 当子类覆盖或实现父类的方法时，方法的前置条件（方法形参）要比父类方法的输入参数要宽松。
4. 当子类的方法实现父类抽象方法时，方法的后置条件（返回值）要比父类更严格。

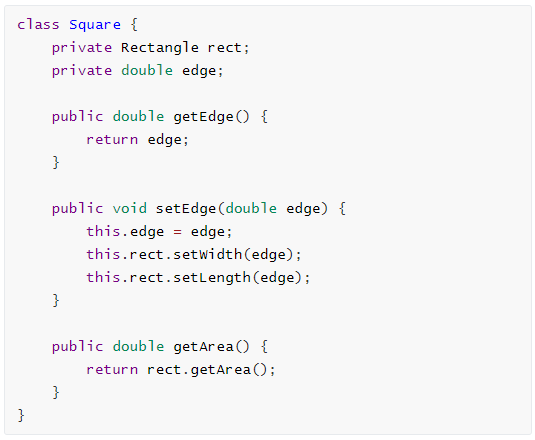
#### 2. 使用组合替代继承

在希望复用代码又不能满足LSP时，往往会用组合来代替继承。





Square类型的对象所调用的getArea()方法不能替代Rectangle中的getArea()方法起同样的作用，违反LSP里氏替代原则。如下使用组合代替继承来改进。



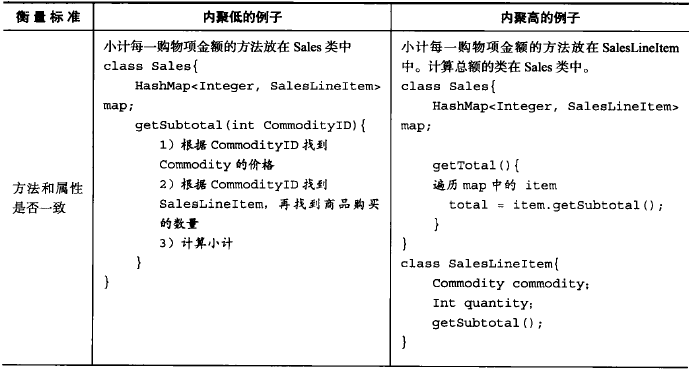
### 四、面向对象中的内聚类型

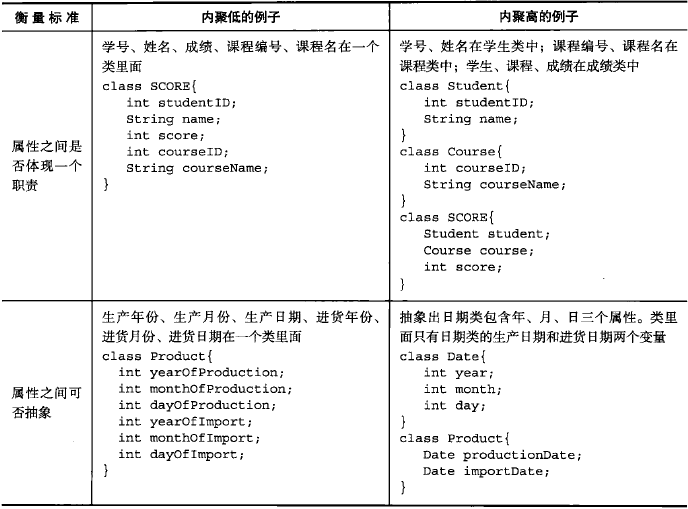
#### 1. 方法的内聚

和结构化中的函数内聚一致，主要是体现方法实现时语句之间的内聚性。内聚由低到高为：功能内聚、通信内聚、过程内聚、时间内聚、逻辑内聚、偶然内聚。

#### 2. 类的内聚

1. 类的内聚主要是衡量类的成员变量和方法之间的内聚。
2. 类既应该是**信息内聚**的，又应该是**功能内聚**的。



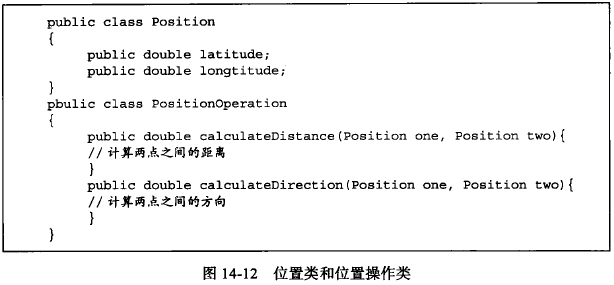


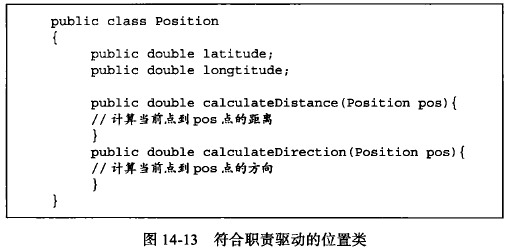
#### 3. 提高内聚的方法

##### (1) 集中信息行为

一个高内聚的类应该应该是信息内聚的，也就是说类的信息应该和访问这些信息的行为放在一个类中，即集中信息与行为。

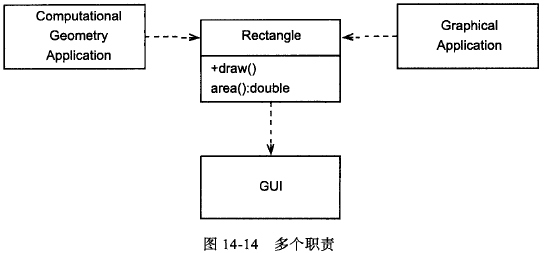
例：Position类和PositionOperation类放在一个类中。图14-12设计中两个类，一个是位置类，一个是位置操作类，分别用来保存信息和提供操作，这是典型的结构化思想（分离数据与函数）；更好的设计是Position对象本身拥有位置信息，可以据此计算与另一点的距离和相对方向，体现出一个整体职责，而不是简单的数据和方法的集中存放。

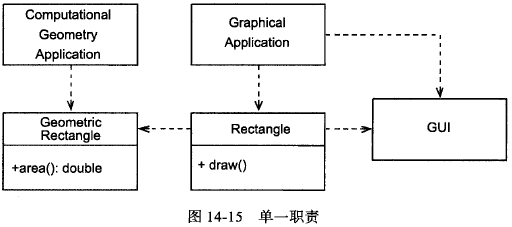




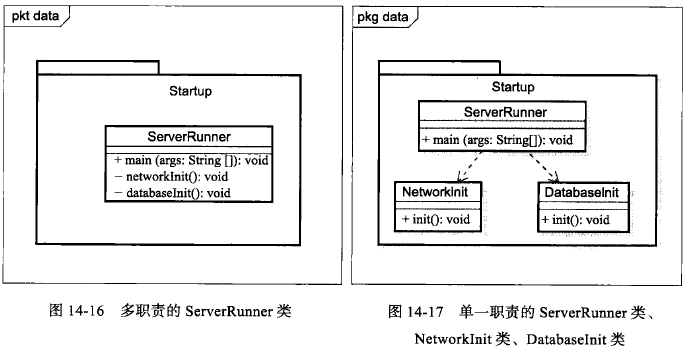
##### (2) 单一职责原则（SRP）

一个高内聚的类不仅要是信息内聚的，还应该是功能内聚的，也就是说，信息与行为除了要集中之外，还要联合起来表达一个内聚的概念，而不是单纯的堆砌。





ServerRunner类有三个职责：程序的入口、网络的初始化、数据库初始化。根据单一职责原则，为了增加类的内聚，应该将这几个职责分配给三个不同的类去处理。



## 面向对象方法下的信息隐藏

模块需要隐藏的决策主要有“职责的实现”和“实现的变更”两类。在面向对象中，需要做到的就是：

Ⅰ. 封装类的职责，隐藏职责的实现；

Ⅱ. 预计将会发生的变更，抽象它的接口，隐藏它的内部机制；

### 封装——分离接口与实现

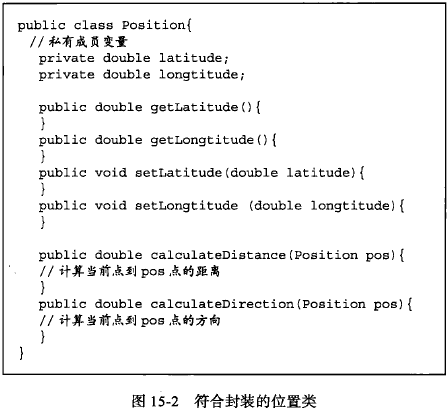
封装有两方面的含义：

1. 将数据和行为包含在类中；集中数据与行为，数据与行为需要相互支撑、紧密联系。
2. 分离对外接口与内部实现（接口描述了类的职责，须要对外公布；实现是类的内部实现机制，不需要对外公开，外界也不应该知道它的具体细节）；

### 封装实现细节

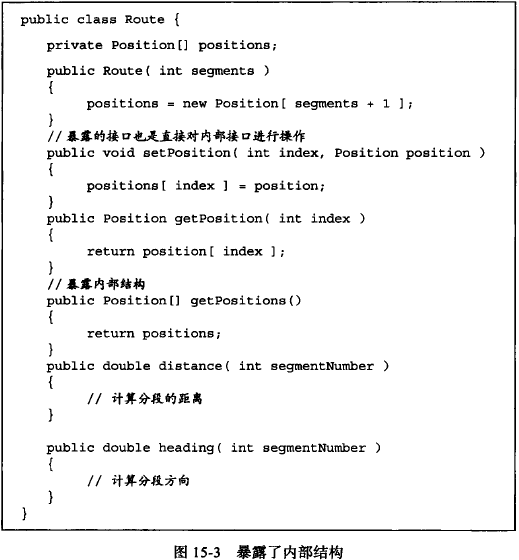
#### 封装数据和行为；

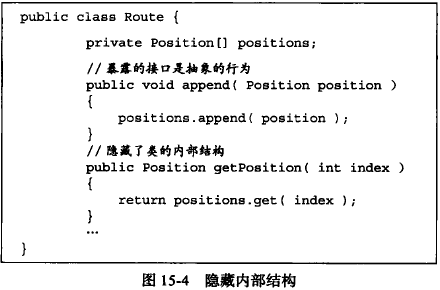
面向对象方法中，对成员变量和成员方法都设置不同的可见性，封装需要根据声明的可见性保护类的数据与行为。



#### 封装内部结构

信息隐藏要求我们不能暴露数据结构的实现决策；**迭代器模式是封装内部结构的典型案例；**



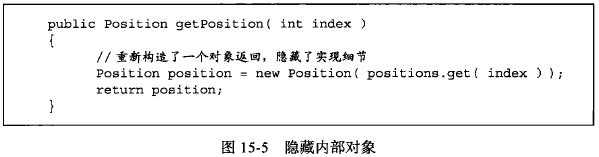


#### 封装其他对象的引用

一个对象所持有的其他对象的引用也是需要隐藏起来的。

重新构造新的对象，将新的对象的引用返回，这样无论外界如何操作这个引用，都不会影响到内部实现List。

设计模式中的**代理模式**、**中介模式**等都是封装其他对象引用的典型案例。



#### 封装类型信息

在多种子类型因为具备一些共性而被视作一种类型加以使用时，应该隐藏其具体子类型的类别，而只需要知道其共性类别。

LSP就是封装类型信息的典型方法；

#### 封装潜在变更

如果预计类的实现中有特定地方会发生变更，就应该将其独立为单独的类或方法，然后为单独的类或方法抽象建立稳定接口，并在原类中使用该稳定接口以屏蔽潜在变更的影响。这样，即便将来某个方法真的发生变更，也不会影响到具体的那个类。

### 为变更而设计

变更是软件开发面临的最大挑战，在开发阶段就为将来可能的变更进行预设计以减少维护成本就成为设计师必须考虑的问题。

信息隐藏是为了变更而设计的主要思想，封装潜在变更方法是应对变更的必要手段，但是仅仅封装变更是不够的，还应该有更高的要求—开闭原则。

##### 封装变更/开闭原则

开闭原则是对面向对象设计的一个指导性、方针性原则；好的设计只需要添加新的代码而不需要修改原有的代码，就能够实现变更。

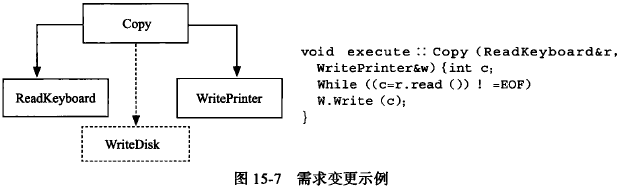
1. 好的设计应该对“扩展”开放；
2. 好的设计应该对“修改”关闭；

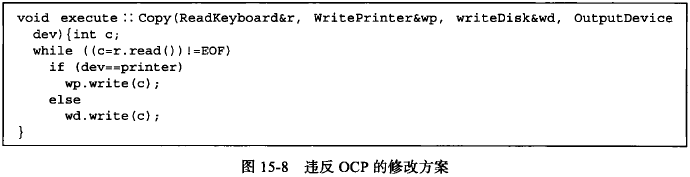
要实现开闭原则仅仅依靠封装潜在变更是不够的，还需要充分利用抽象与多态机制；

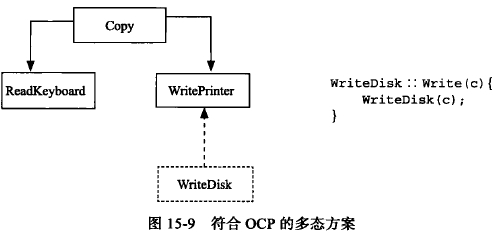
1. **通过多态实现OCP**

通过父类表现统一的接口，通过子类表现不同的行为；

C++使用模版机制，java使用泛化机制；







##### (2) 依赖倒置原则

很多时候耦合的方向是很重要的（比如，A不稳定、B稳定，那么A发生修改可能会给B带来影响），这就是依赖倒置原则。

**依赖倒置是指：**

1. 抽象不应该依赖于细节，细节应该依赖于抽象。因为抽象稳定，细节是不稳定的。
2. 高层模块不应该依赖于底层模块，而是双方都依赖于抽象。因为抽象是稳定的，而高层模块和低层模块都可能是不稳定的。

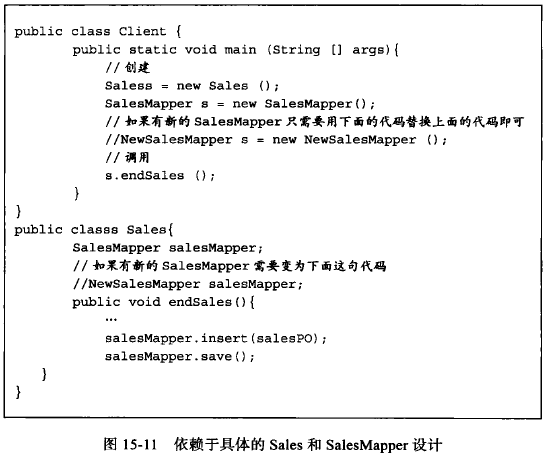
为具体类建立抽象接口并分离该接口是实现DIP的基本手段。

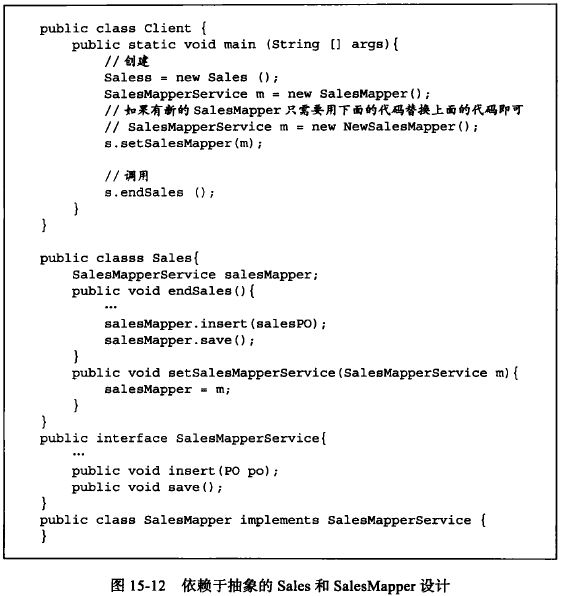
如果A是抽象的，B依赖于A是符合DIP；

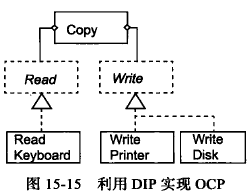
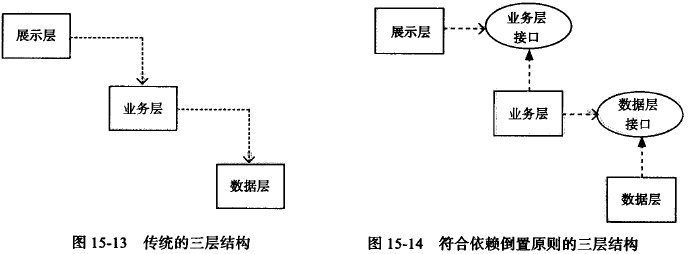
如果A是具体的，B依赖于A就不符合DIP。此时应该是为A建立抽象接口IA，然后B依赖IA、A实现IA。

如下所示：Sales直接依赖于具体的实现类SalesMapper，如果SalesMapper类 发生变化，会直接影响Sales类。

符合DIP设计方案的是，让Sales持有一个SalesMapperService接口，然后用SalesMapper实现SalesMapperService接口，那么当发生变化的时候，Sales并不需要修改，也不需要重新编译。只需要修改Client类并重新编译即可。







## 设计模式

### 策略模式

**策略模式所体现的设计原则**

**减少耦合：**减少策略的使用类和策略的实现类直接耦合；

**依赖倒置：**策略的使用类依赖的是策略的接口，而非策略的实现类；

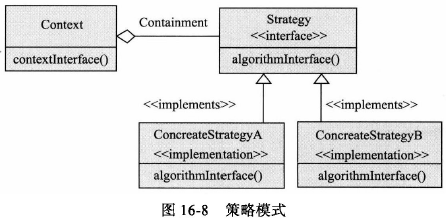
**注意点**

策略模式消除switch语句；

可以动态选择不同的策略；

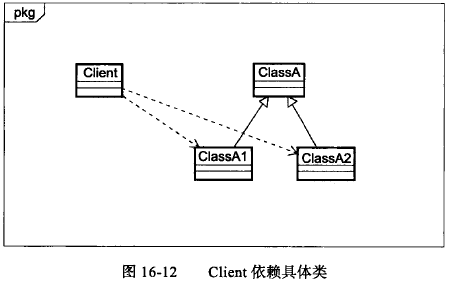
客户必须提前知晓各自不同的策略；

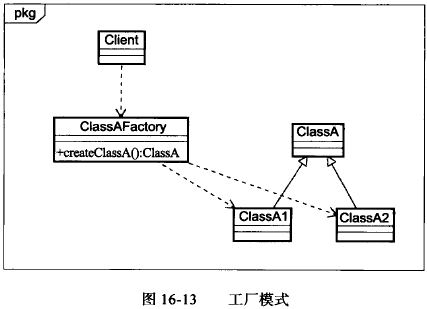
**设计分析**

****

### 工厂方法

**工厂模式就是为对象的创建提供一个接口，将具体创建的实现封装在接口之下，这样具体创建地实现的改变就不会对客户代码Client类产生影响。**从而降低了Client类和ClassA1等具体类的耦合。

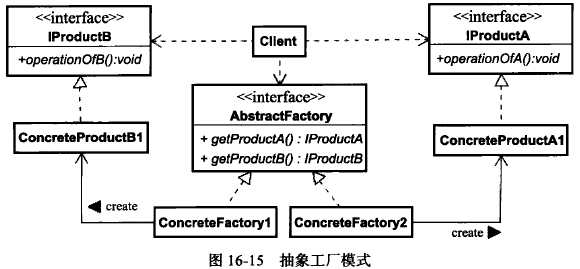




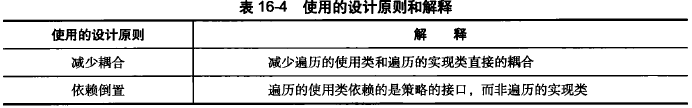
在此基础之上，如果根据前面讲述的DIP，要依赖于抽象，而不是依赖于具体，我们可以将ClassAFactory变为接口或者是父类，从而达到一个共享的接口契约。

工厂接口既使得原本分布于代码各处的多处对象的实例化集中到具体的工厂内部，又隔离了“对象实例化的组合”的变化。





### 迭代器模式



**应用场景**

访问一个聚合对象的内容而无需暴露它的内部实现；

支持对聚合对象的多种遍历；

为遍历不同的聚合结构提供一个统一的接口；

