



# 软件工程与方法

## 第5讲 面向对象方法

# 主要内容

1. 面向对象软件工程
2. 基本概念
3. 软件设计原则
4. 面向对象分析与设计
5. 面向对象测试





# 面向对象分析与设计

---

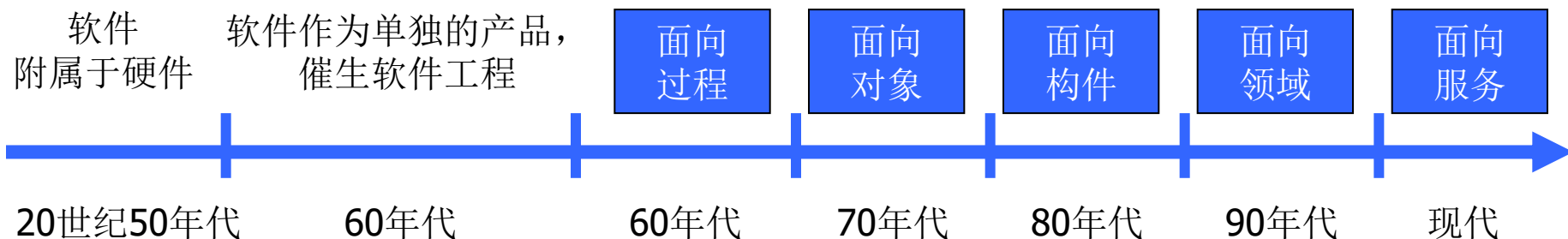
## 1 面向对象软件工程

# 面向对象软件工程—主要内容

1. 面向对象的发展过程
2. 面向对象的基本特征



# 软件的变化

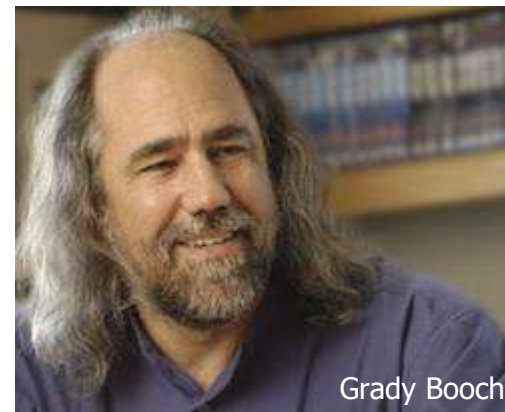


## 我们身边的“面向对象”

- 程序设计语言：C++，C#，Java
- 组件规范：DCOM，.net，J2EE
- 操作系统：Windows
- 开发工具：Eclipse，Microsoft Studio，Rational Rose，Xcode
- 数据库：Oracle，IBM DB2，SQL Server
- 办公软件：Microsoft Office

# 面向对象发展过程

- OO始于1966年，Kristen Nygaard和Ole-Johan Dahl开发了Simula语言，提供比子程序更高的抽象和封装。
- 同时期，Alan Kay开始图形化和仿真研究，1972年PARC发布Smalltalk的第一个版本，OO术语正式确定。
- Smalltalk-80是Smalltalk版本的总结，1981年发布，它导致一系列进展：Window、icon、mouse和下拉式菜单环境。Smalltalk语言还影响了80年代早期和中期的面向对象的语言，如：Object-C(1986)、C++(1986)、Self(1987)、Eiffel(1987)、Flavors(1986)。
- 1980年Grady Booch首先提出面向对象设计（OOD）的概念。然后其他人紧随其后，面向对象分析的技术开始公开发表。
- 1990年代，面向对象的分析、测试、度量和管理等研究都得到长足发展。
- 目前面向对象思想和对象技术已经渗透到软件领域的方方面面。

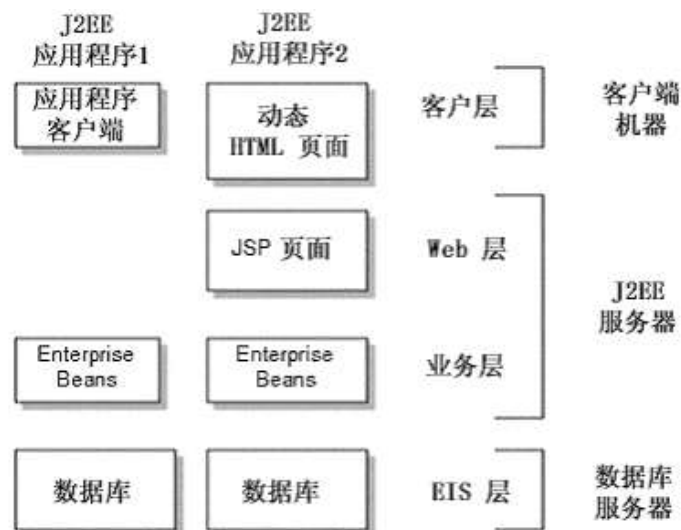


# 例：面向对象的架构

## Java EE（J2EE）的四层模型

### .Net框架分层结构

Web Services	
框架和库（ASP.NET, ADO.NET, Windows Forms）	
交互标准 （SOAP、WSDL）	开发工具 （Visual Studio.NET）
组件模型	
对象模型和公共语言规范	
公共语言运行时（Common language Runtime）	



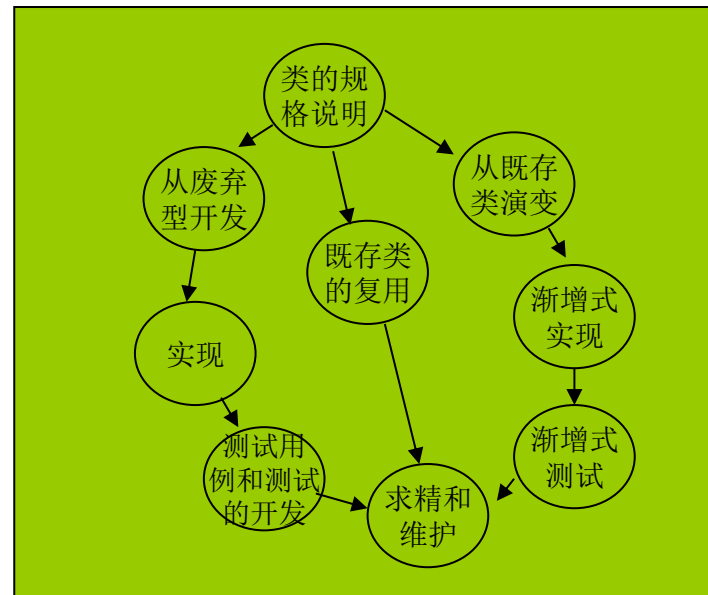
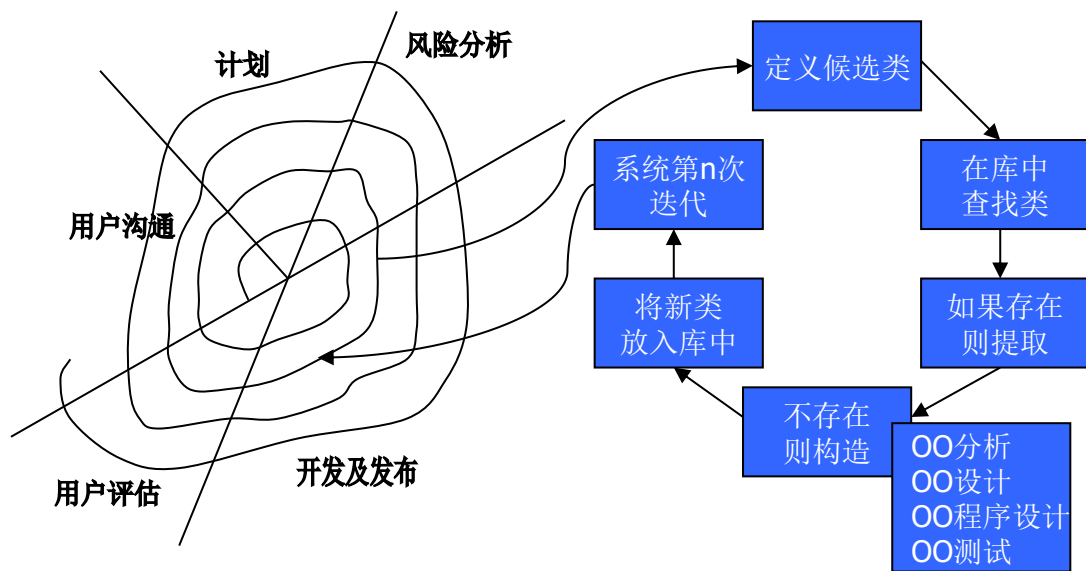


# OO方法兴起的原因

- 从认知学的角度来看，面向对象的方法符合人们对客观世界的认识规律。
  - 传统程序设计方法的解空间结构与问题空间结构不一致。
  - 面向对向方法的分析、设计、实现的结果能直接映射到客观世界中系统的实体上，即解空间的结构与问题空间的结构是一致的。
- 面向对象方法开发的软件系统易于维护，其体系结构易于理解、扩充和修改。
  - 对象的封装性很好地体现了抽象和信息隐蔽的特征。
- 面向对象方法中的继承机制有力地支持软件复用。



# 面向对象软件工程：演化模型



OO软件演化的多种途径

# 面向对象方法:



Peter Coad和Edward Yourdon提出用下列等式认识面向对象方法:

面向对象 (*Object-Oriented*)

= 对象 (*object*)

+ 分类 (*classification*)

+ 继承 (*inheritance*)

+ 通过消息的通信 (*communication with messages*)

## Key Characteristics of an OO Methodology

- |                                   |                          |
|-----------------------------------|--------------------------|
| 1. common methods of organization | 2. abstraction           |
| 3. Encapsulation                  | 4. inheritance           |
| 5. Polymorphism                   | 6. message communication |
| 7. Association                    | 8. reuse                 |



# 面向对象分析与设计

## 2 面向对象的基本概念

# 面向对象的基本概念—主要内容

1. 对象与类
2. 方法和消息



## Alan Kay总结了smalltalk中对象的5大基本特征:

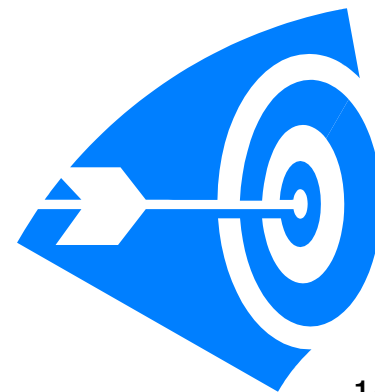


- ① 所有的东西都是对象。
- ② 程序是一大堆对象的集合，他们通过消息传递，各个对象之间知道要做些什么。
- ③ 每个对象都分配有自己的存储空间，可容纳其他对象。
- ④ 每个对象都有一个类型。
- ⑤ 同一类的所有对象能接收相同的消息。



# 对象（Object）

- 客观世界是由许多具体的事物或事件、抽象的概念、规则等组成的。因此，我们将任何感兴趣或要加以研究的事、物、概念都统称为对象。
- OO方法正是以对象作为最基本的元素，它也是分析问题、解决问题的核心。
- 对象是指一组属性以及这组属性上的专用操作的封装。属性（**attribute**）通常是一些数据，有时它也可以是另一个对象。





# 1) 对象是人們要進行研究的任何事物

- 从最简单的整数到极复杂的自动化工厂都可看作对象。
- 对象不仅能表示具体的实体，也能表示抽象的规则、计划或事件。
- 主要有如下对象类型：
  - 有形的实体：指一切看得见、摸得着的实物。
  - 作用：指人或组织所起的作用。
  - 事件：在特定的时间所发生的事。
  - 性能说明：如产品性能说明。



## 2)对象实现了数据与操作的结合

- **行为 (Behavior)** ——说明这个对象能做什么，就是对象能进行什么操作，由方法或函数描述。
- **状态 (State)** ——当对象施加方法时对象的反映，通常用数据描述。
- **标识 (Identity)** ——区别于其它对象标志，每一个对象有唯一的**ID**。在对象建立时，除有对象名外，还要有唯一且永久的标识符**OID (Object Identify)**。





### 3)对象必须参与一个或一个以上的对象类

- 类（**Class**）是对象（**Object**）的抽象，对象是类的实例（**instance**），对象与类实例是同义词。
- 在大多数OO方法中，类也可以看是对象，从而实现了类与对象的统一。
- 创建类的类称为元类（**Metaclass**），此时，类可以作为元类的实例。

例：用“飞机”类的实例来说明类及对象的关系。

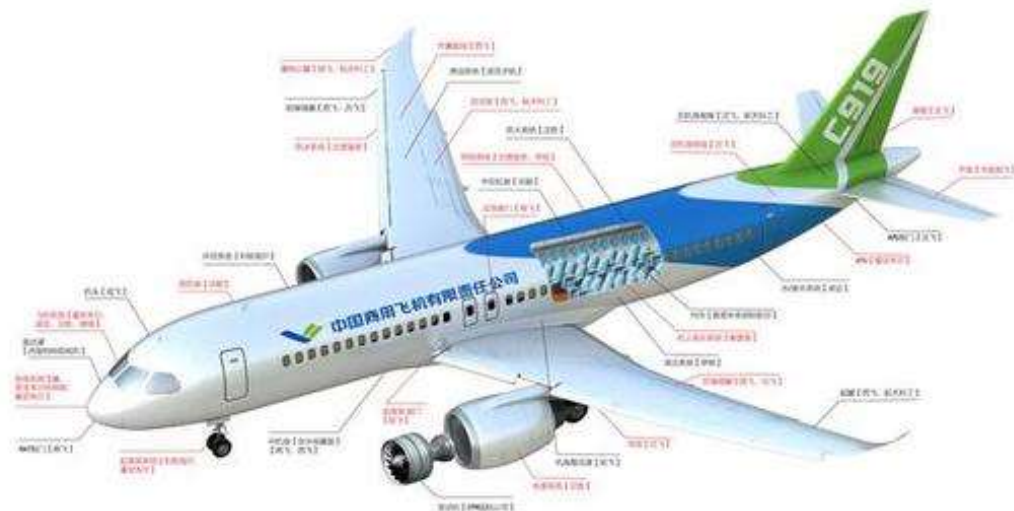
- 每架飞机都是一个具体的对象，如：歼10、歼15、歼20、轰6、运20。



(续)

## □抽取飞机共同的特性:

- 凡是飞机都能在空中飞行，具有改变飞行方向、控制飞行高度和飞行速度的操作特性；
- 凡是飞机都有机名、机型、飞行高度、飞行速度等数据，用以描述飞机的结构性能；
- 飞机还有严格的飞行安全约束规则，如飞行条件、起飞与降落条件等。





# 对象类（Object Class）

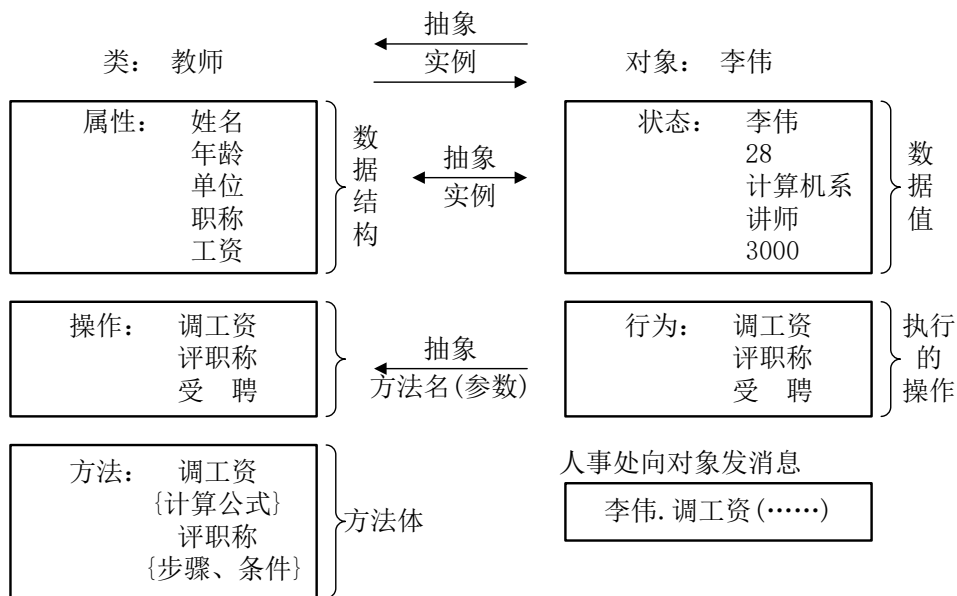
## 1)对象类的定义

- 将具有相同结构、操作，并遵守相同约束规则的对象聚合成一组，这组对象集合就称为对象类，简称为类（**class**）。
- 具体对类进行定义时，最低限度应包括如下内容：
  - 类名（**class name**）
  - 外部接口（**external interface**）
  - 内部表示（**internal interface**）
  - 接口的内部实现（**implementation**）

## 2)类说明 (Class Specification)

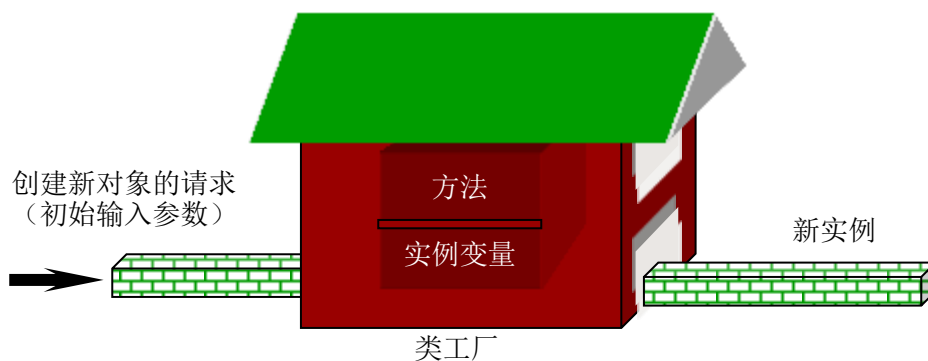
- 类说明也称为类的外部特性或称外部接口。
- 类说明是让用户了解对象是什么和能够做什么事，因而也是与用户沟通的“外部接口”或界面。
- 通常包括：
  - 公共成员 (public member) 变量
  - 公共成员函数

### 例：类与对象



### 3)类实现 (Class Implementation)

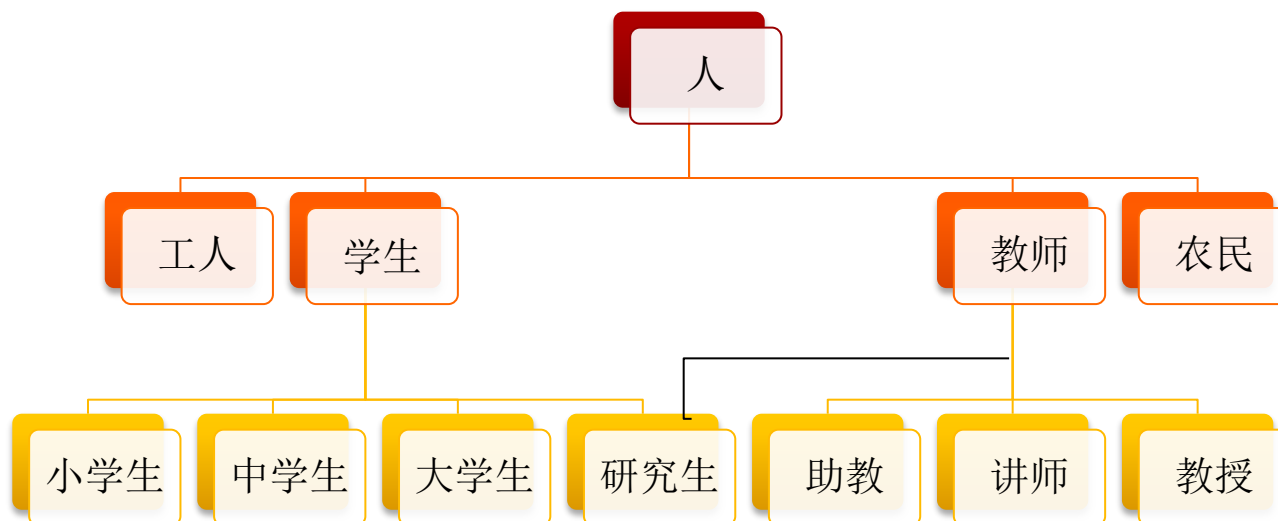
- 类实现是类的内部表示及类说明的具体实现方法。
- 类实现对用户隐藏。
- 对应于每一个操作符，都应在内部实现中找出具体的实现这一操作的程序模块。



## 4) 类层次与类格

### (Class hierarchy and Class lattice)

- 类继承导致类之间可能存在层次结构或格结构。
- 类层次结构特点是每个子类只有一个超类。
- 类格结构的特点是一个类至少存在一个子类有一个以上的超类。





## 5)对象类概念的精髓

- 对象类最鲜明的特色是将数据的结构与数据的操作者封装在对象类中。
- 实现了类的外部特性与类实现的隔离。
- 实现了对象使用者与开发者的分离。
- OO方法具有良好的模块化特性，进而为复杂的系统的分析、设计、实现提供有效的方法。





# 类实例及实例变量

## 1)类实例（Class Instance）

- 类所描述的对象就称为类的实例。如公司类，就可以有Lenovo、Hauwei等实例。
- 对象可同时参与到多个类并作为这些类的实例。如张三既可以作为某个工厂的职工，同时又可作为武术协会的会员。
- 凡参与某个类的实例，必然共享该类的全部语义特性；如果还要加入实例自身的特性时，则应另外加补充说明。

## 2)类实例变量（Instance Variable）

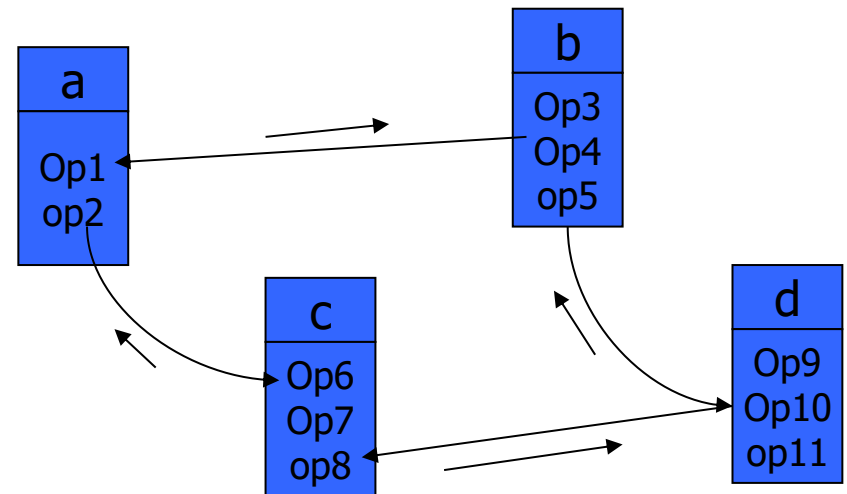
- 类实例变量为描述类的结构特性提供了统一的框架或模式，而每个实例都有自身的实例变量值。

# 方法和消息

- 对象之间进行通信的结构叫做消息（**Message**），它包含接收对象去执行某种操作的信息。
- 类中操作的实现过程叫做方法（**Method**），一个方法有方法名、返回值、参数、方法体。
- 消息是对象之间相互联系和通信的唯一途径。消息由方法来处理。

- 消息内容包括：

- ① 对象名：接受消息的对象名
- ② 消息名：发送给对象的消息名（即对象名、方法名）
- ③ 消息参数：消息内容

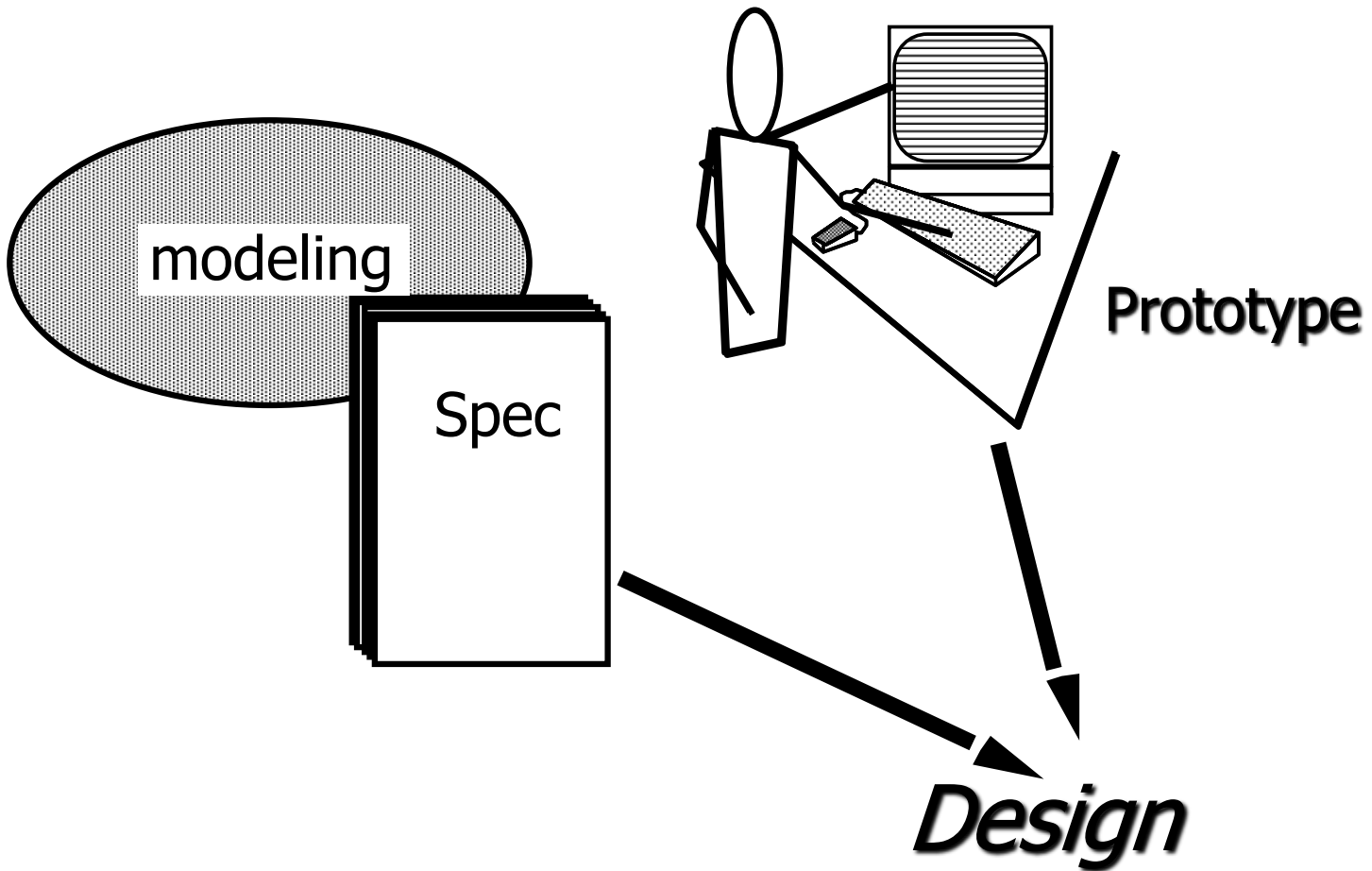




# 面向对象分析与设计

## 3 面向对象设计原则

# Where Do We Begin?

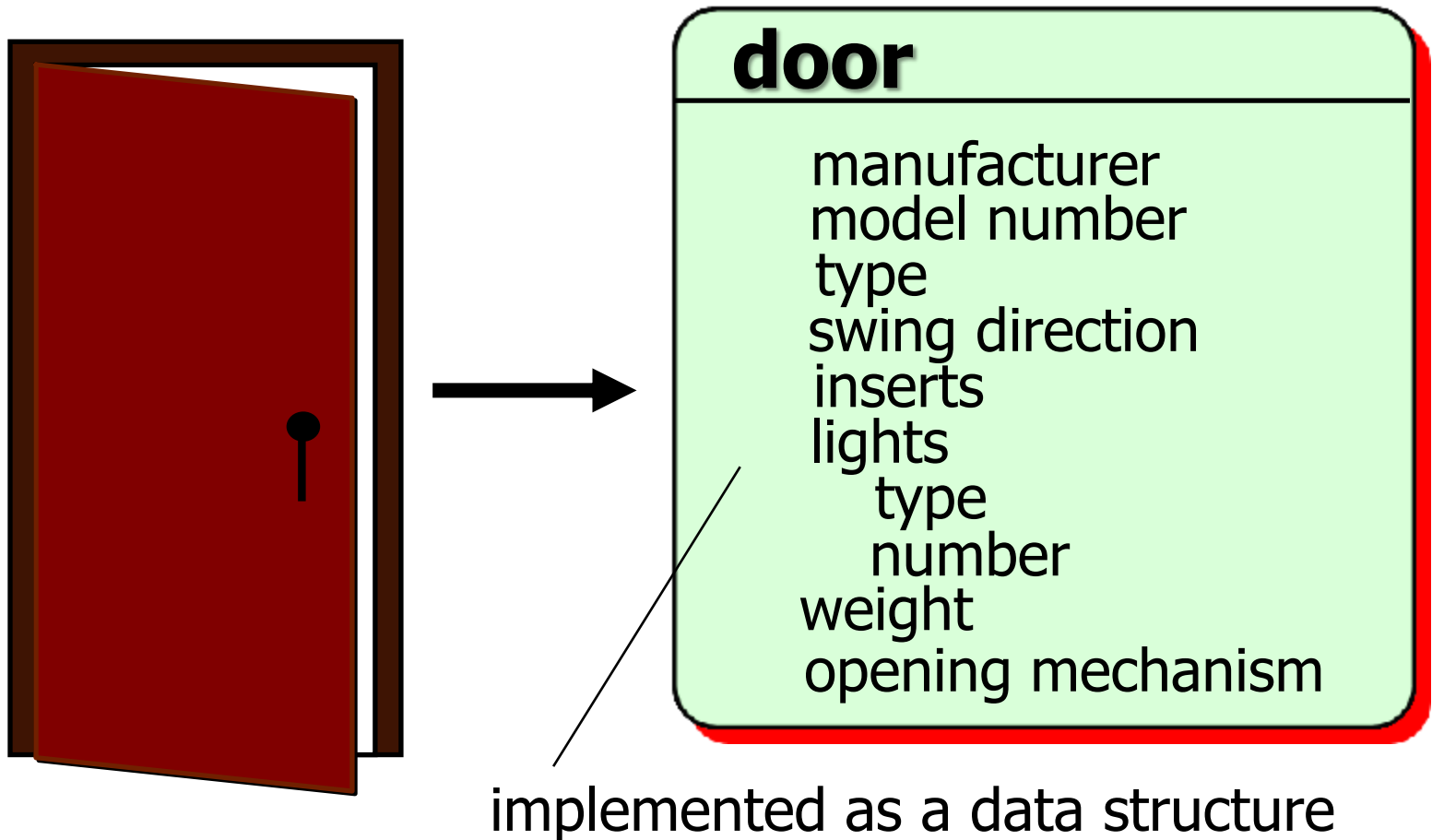




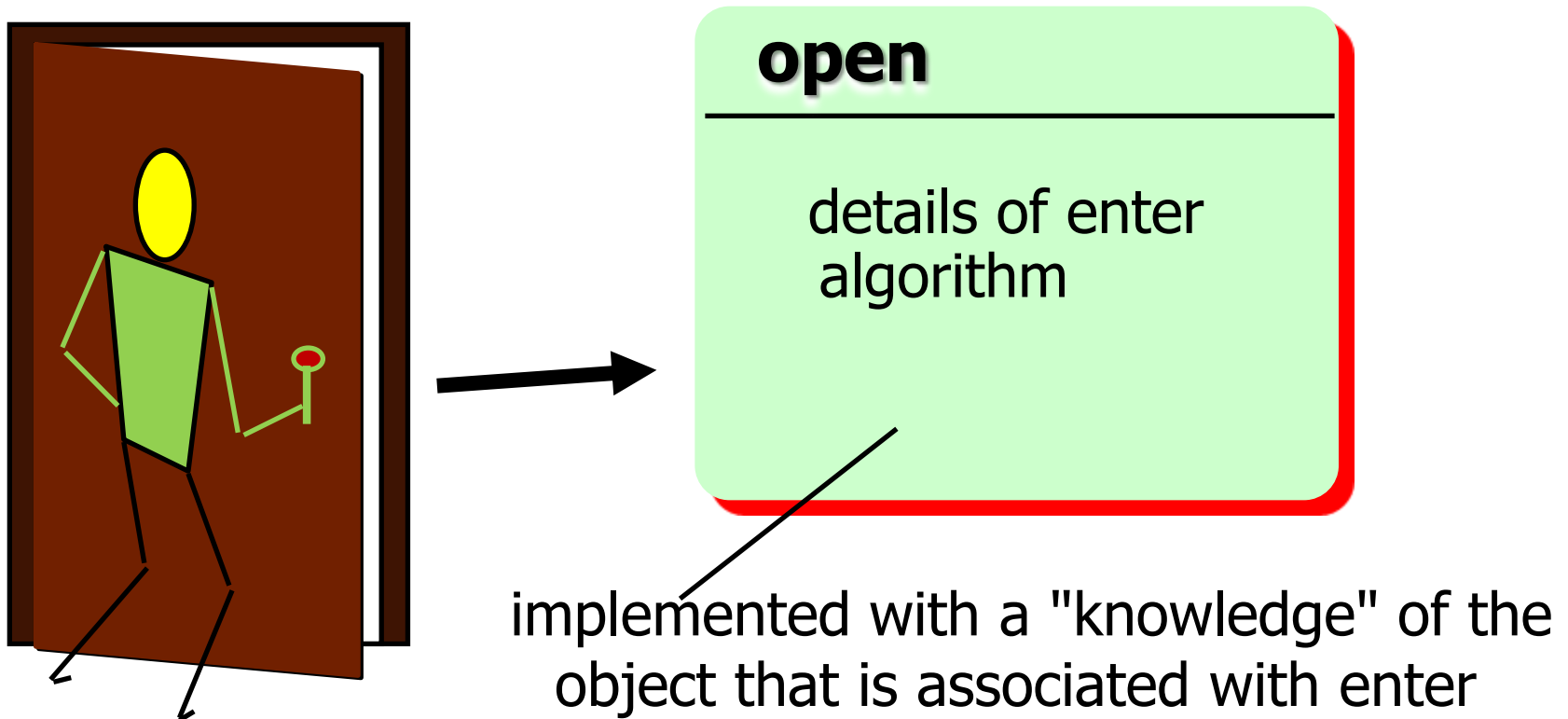
# Fundamental Concepts

- **abstraction** - data, procedure, control
- **refinement** - elaboration of detail for all abstractions
- **modularity** - compartmentalization of data and function
- **architecture** - overall structure of the software
  - Structural properties
  - Extra-structural properties
  - Styles and patterns
- **procedure** - the algorithms that achieve function
- **hiding** - controlled interfaces

# Data Abstraction



# Procedural Abstraction



# Stepwise Refinement

**open**

walk to door;  
reach for knob;  
open door; →  
walk through;  
close door.

repeat until door opens  
  turn knob clockwise;  
  if knob doesn't turn, then  
    take key out;  
    find correct key;  
    insert in lock;  
  endif  
  pull/push door  
  move out of way;  
end repeat





# 面向对象的设计原则

## ■ 模块化：

- 对象就是模块

## ■ 抽象：

- 面向对象支持过程和数据抽象，类是一种抽象数据类型，即规格说明抽象。

## ■ 信息隐蔽：

- 对象封闭支持信息隐蔽。

## ■ 弱耦合：

- 不同对象之间相互关联的紧密程度。对象之间的两类耦合：
  - ① 交互耦合（交换消息），使交互耦合尽可能松散，即减少消息中包含的参数个数，降低参数的复杂程度，减少消息数。
  - ② 继承耦合。应该提高继承耦合程度。通过继承关系结合起来的基类和派生类构成了更大的模块。



# 面向对象的设计原则（续）

## ■ 强内聚：

### ■ 共有3种内聚：

- ① 方法内聚，一个方法应该完成一个且仅完成一个功能。
- ② 类内聚，一个类应该只有一个用途，它的属性和方法应该是高内聚的。类的属性和方法应该全都是完成该类对象的任务所必需的。
- ③ 一般-特殊（继承）内聚，这种结构应该是对相应领域知识的正确抽取。紧密的继承耦合与高度的一般-特殊内聚是一致的。

## ■ 可重用：

### ■ 有两方面的含义：

- ① 尽可能使用已有的类
- ② 在设计新类时考虑送来的可重用性。

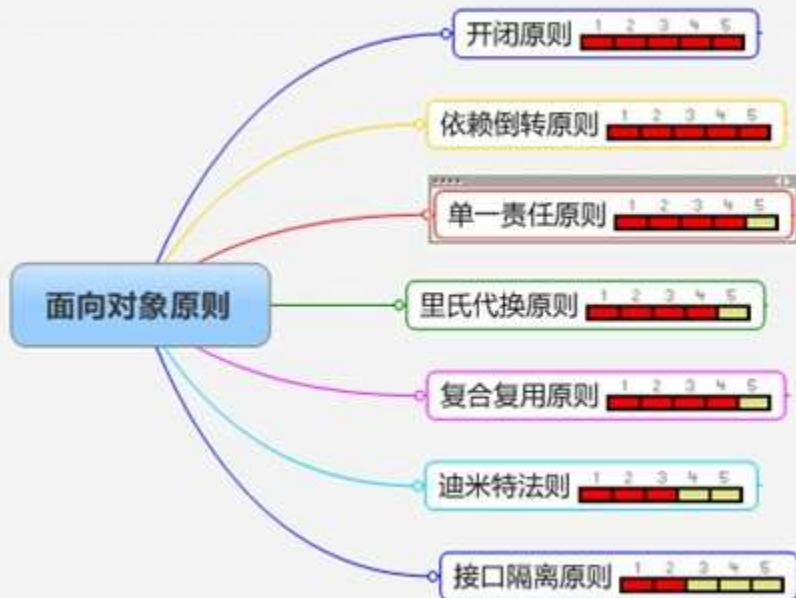


# 什么是好的设计？

Peter Code提出的一个好的系统设计应该有如下特性：

- 可扩展性（**extensibility**）：容易添加新功能。
- 灵活性（**flexibility**）：代码修改平稳地发生。
- 可插拔性（**pluggability**）：容易将一个类抽出去，同时将另一具有同样接口的类加入进来。

# 面向对象设计原则



## 目标与原则的关系

可扩展性

灵活性

可插拔性

开-闭原则

里氏替换原则

依赖倒置原则

合成/聚合复用原则

Demeter法则

接口隔离原则

设计原则名称	设计原则简介	重要性
开闭原则 (Open-Closed Principle, OCP)	软件实体对扩展是开放的，但对修改是关闭的，即在不修改一个软件实体的基础上去扩展其功能	★★★★★
依赖倒转原则 (Dependency Inversion Principle, DIP)	要针对抽象层编程，而不要针对具体类编程	★★★★★
里氏代换原则 (Liskov Substitution Principle, LSP)	在软件系统中，一个可以接受基类对象的地方必然可以接受一个子类对象	★★★★☆
单一职责原则 (Single Responsibility Principle, SRP)	类的职责要单一，不能将太多的职责放在一个类中	★★★★☆
接口隔离原则 (Interface Segregation Principle, ISP)	使用多个专门的接口来取代一个统一的接口	★★★☆☆
合成复用原则 (Composite Reuse Principle, CRP)	在系统中应该尽量多使用组合和聚合关联关系，尽量少使用甚至不使用继承关系	★★★★☆
迪米特法则 (Law of Demeter, LoD)	一个软件实体对其他实体的引用越少越好，或者说如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，而是通过引入一个第三者发生间接交互	★★★☆☆



# 面向对象的设计原则

- 开-闭原则（**OCP**）：对可变性封装
- 里氏替换原则（**LSP**）：如何进行继承
- 依赖倒转原则（**DIP**）：针对接口编程
- 接口隔离原则（**ISP**）：恰当的划分角色和接口
- 合成复用原则（**CRP**）：尽量使用合成/聚合，尽量不使用继承
- 迪米特原则（**LoD**）：不要跟陌生人说话
- 单一职责原则（**SRP**）：一个类有且只有一个改变的理由。

# 开闭原则

- 开闭原则定义：一个软件实体应当对扩展开放，对修改关闭。
- **Ivar Jacobson**：任何系统在其生命周期中都会发生变化。如果我们希望开发出的系统不会在第一版后就被抛弃，那么我们就必须牢牢记住这一点。
- **Bertrand Meyer(1988)**：软件组成实体（类、模块、函数等等）应该是可扩展的，但是不可修改。





# 开闭原则分析

- ① 在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即实现在不修改源代码的情况下改变这个模块的行为。
- ② 抽象化是开闭原则的关键。通过从抽象体派生，也可以扩展模块的行为功能。
- ③ 还可以通过一个更加具体的对可变性封装原则（**EVP, Principle of Encapsulation of Variation**）来描述，这一原则要求找到系统的可变因素并将其封装起来。
- ④ 开闭原则总结：**面对需求，对程序的改动是通过增加新代码进行的，而不是改变原来的代码。**

# 开闭原则实例

- 某图形界面系统提供了各种不同形状的按钮，客户端代码可针对这些按钮进行编程，用户可能会改变需求要求使用不同的按钮。



图1

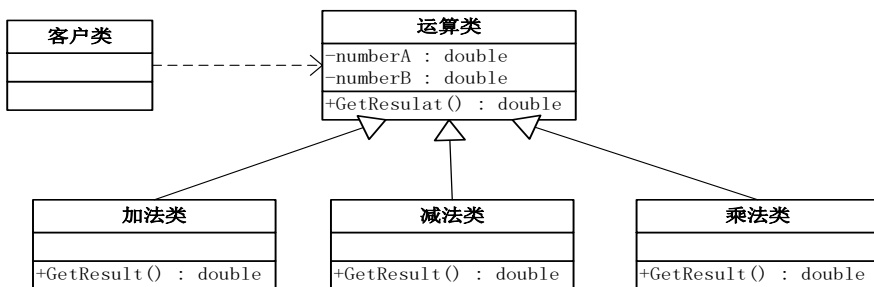
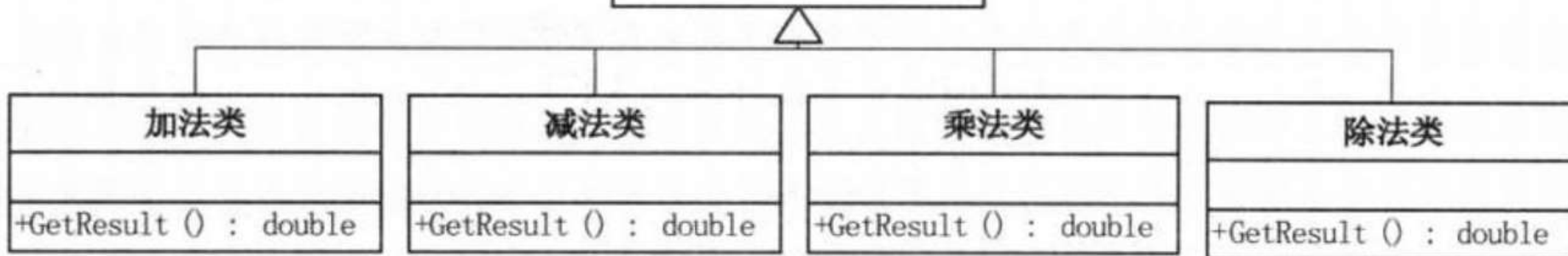
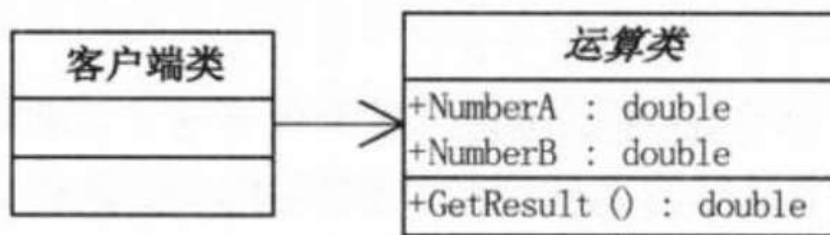
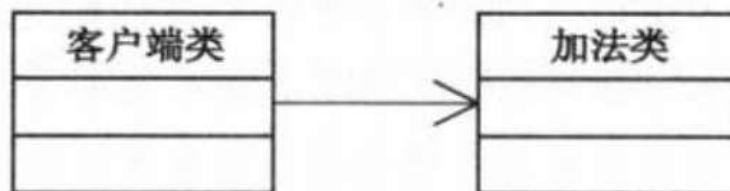


图2

- 图1：客户端的一个方法直接调用加法类。如果想添加一个减法类，就得改变加法类中代码（用switch语句实现），这就违背了“开闭原则”。
- 图2：重构图1，添加一个运算类（加法类的父类），这样再添加减法类的时候就不用修改客户端类。

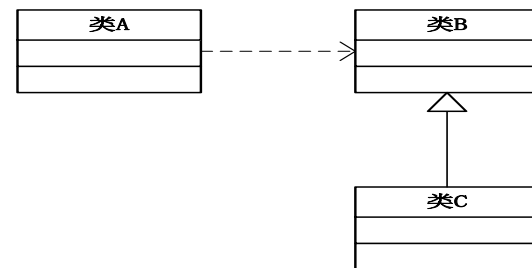


(续)



# 替换原则

- 子类应当可以替换父类并出现在父类能够出现的任何地方。这个原则是Liskov于1987年提出的设计原则。它同样可以从Bertrand Meyer 的DbC (Design by Contract) 的概念推出。



- 运用替换原则时，尽量把类B设计为抽象类或者接口，让C类继承类B（接口B）并实现操作A和操作B，运行时，类C实例替换B，这样我们即可进行新类的扩展（继承类B或接口B），同时无须对类A进行修改。



# 替换原则的定义

Liskov (1987) 的定义:

- I. 如果对每一个类型为S的对象o1, 都有类型为T的对象o2, 使得以T定义的所有程序P在所有的对象o1都代换成o2时, 程序P的行为没有变化, 那么类型S是类型T的子类型。
- II. 所有引用基类(父类)的地方必须能透明地使用其子类的对象。



# 替换原则分析

- 如果能够使用基类对象，那么一定能够使用其子类对象。把基类都替换成它的子类，程序将不会产生任何错误和异常，反过来则不成立；如果一个软件实体使用的是一个子类的话，那么它不一定能够使用基类。
- 里氏替换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。
- 总结：子类型必须能够替换掉它们的父类型。

# 替换原则实例

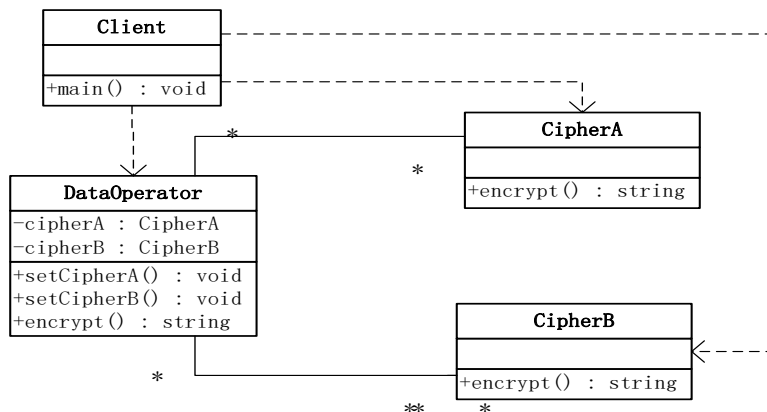


图1

- 某系统需要实现对重要数据的加密处理，在数据操作类DataOperator中需要调用加密类中定义的加密算法，系统提供了两个不同的加密类CipherA和CipherB，它们实现不同加密方法，在DataOperator中可以选择其中的一个实现加密操作。

- 图1为什到图2？因为如果需要更换一个加密算法类或者增加一个新的加密算法类，如将CipherA改为CipherB，则需要修改客户类Client和DataOperator的源代码，违背了开闭原则。现使用里氏代换原则对其进行重构，使得系统可以灵活扩展，符合开闭原则。

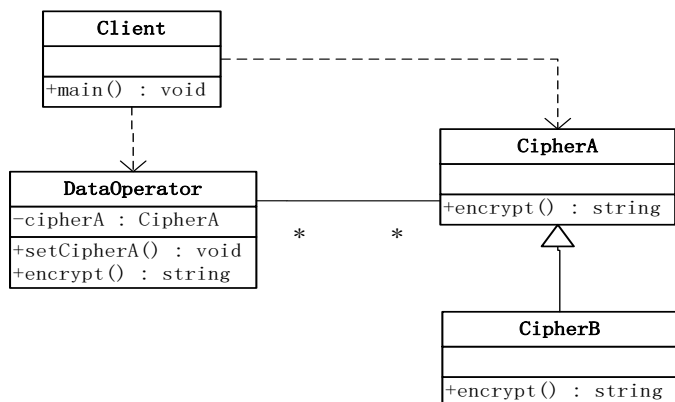


图2



# 接口隔离原则

- 接口污染（**interface contamination**）：
  - 一个没有经验的设计师往往节省接口的数目，将一些功能相近或功能相关的接口合并，并将这看成是代码优化的一部分。
- 定义：从一个客户类的角度来讲：
  - I. 一个类对另外一个类的依赖性应当是建立在最小的接口上的，客户端不应该依赖那些它不需要的接口。
  - II. 使用多个专门的接口比使用单一的总接口要好。一旦一个接口太大，则需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。



# 接口隔离原则分析

- **ISP**指使用多个专门的接口，而不使用单一的总接口。每一个接口应该承担一种相对独立的角色，不多不少，不干不该干的事，该干的事都要干。
- 使用**ISP**拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好。
- 可以在进行系统设计时采用定制服务的方式，即为不同的客户端提供宽窄不同的接口，只提供用户需要的行为，而隐藏用户不需要的行为
- 总结：类应该完全依赖相应的专门的接口。

# 接口隔离原则实例

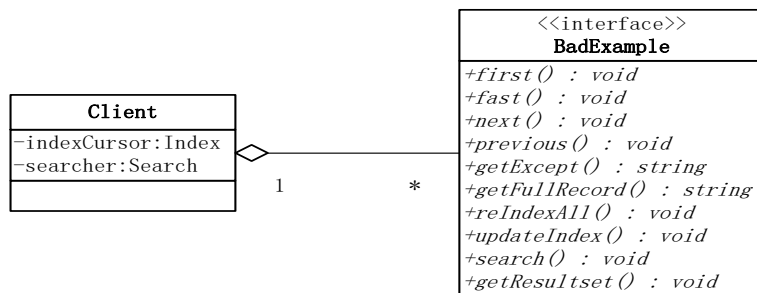


图1

■ 图1为坏的例子

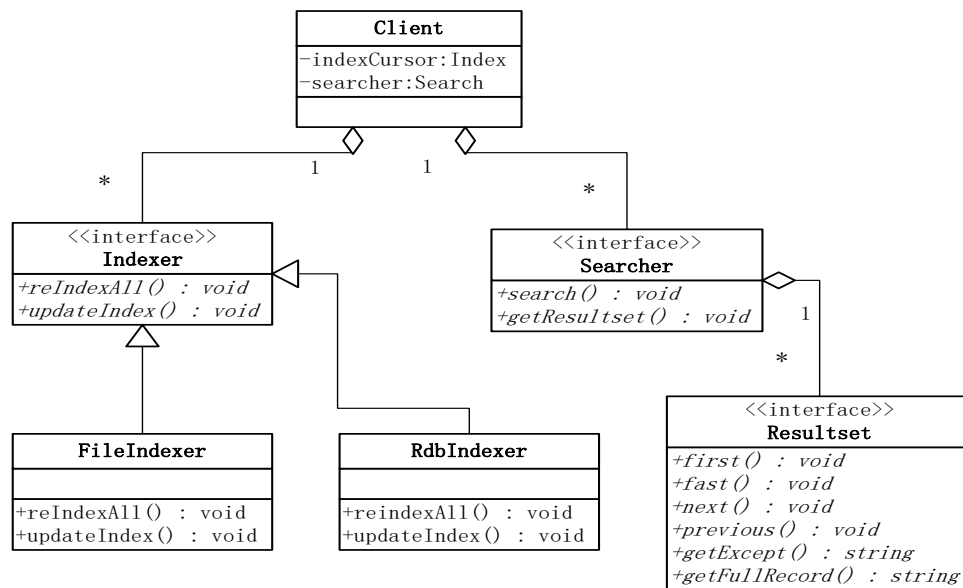


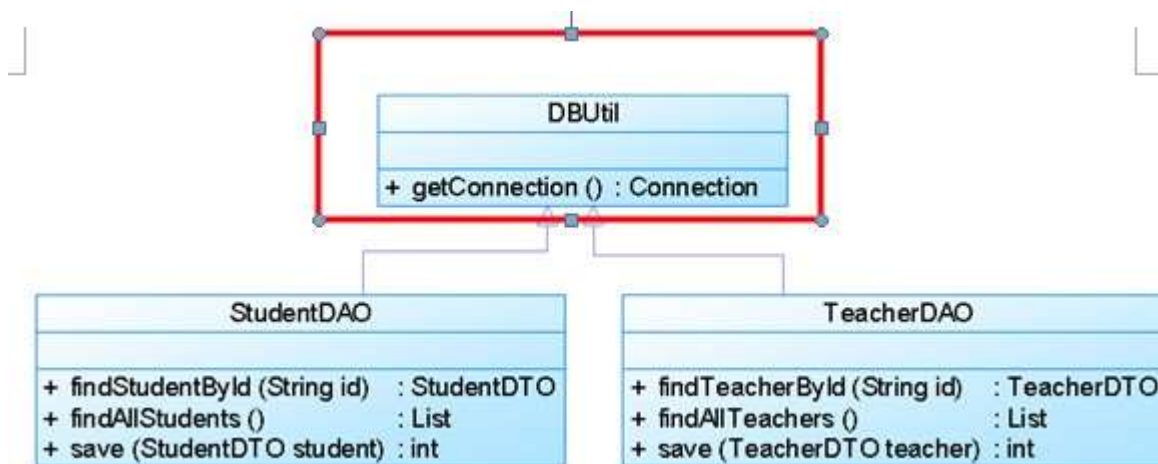
图2

■ 图2为对图1使用接口隔离原则



# 合成复用原则

- 定义：尽量使用对象组合，而不是继承来达到复用的目的。





# 合成复用原则分析

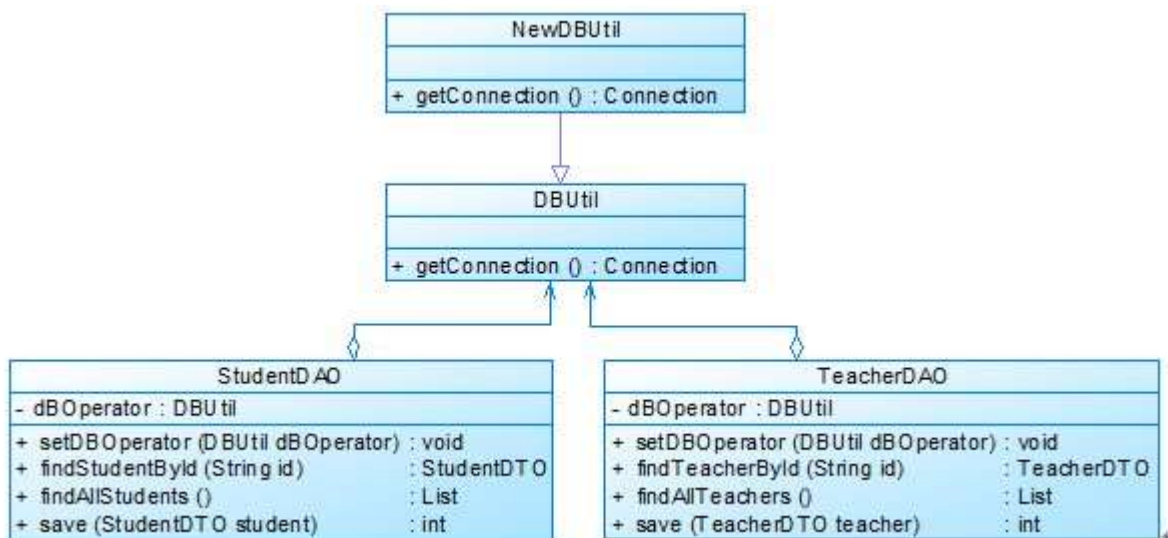
- **CRP**是指在一个新对象里通过关联关系（包括组合关系和聚合关系）来使用已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。简言之：要尽量使用组合/聚合关系，少用继承。
- 在**OOD**中，可以通过组合/聚合关系或通过继承在不同的环境中复用已有的设计和实现：
  - a. 继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用）
  - b. 组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用）



## (续)

- 组合/聚合可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。
- 总结：类中应用，尽量使用对象组合而不是用继承来达到复用的目的。

# 合成复用原则实例



- 因为如果需要更换数据库连接方式，如原来采用JDBC连接数据库，现在采用数据库连接池连接，则需要修改DBUtil类源代码。如果StudentDAO采用JDBC连接，但是TeacherDAO采用连接池连接，则需要增加一个新的DBUtil类，并修改StudentDAO或TeacherDAO的源代码，使之继承新的数据库连接类，这将违背开闭原则，系统扩展性较差。



# 迪米特法则

- 定义：每个软件单元对其他的单元都只有最少的知识，而且局限于那些与本单位密切相关的软件单元。
- Lod法则：
  - 最少知识原则
  - 只与你直接的朋友们通信
  - 不要跟“陌生人”说话
- 朋友的条件：
  - 当前对象本身（**this**）
  - 以参量形式传入到当前对象方法中的对象
  - 当前对象的实例变量直接引用的对象
  - 当前对象的实例变量如果是一个聚集，那么聚集集中的元素都是朋友
  - 当前对象所创建的对象



# 迪米特法则分析

- I. **LoD**指一个软件实体应当尽可能少的与其他实体发生相互作用。这样，当一个模块修改时，就会尽量少的影响其他的模块，扩展会相对容易，这是对软件实体之间通信的限制，它要求限制软件实体之间通信的宽度和深度。
- II. 狭义的**LoD**：可以降低类之间的耦合，但是会在系统中增加大量的小方法并散落在系统的各个角落，它可以使一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联，但是也会造成系统的不同模块之间的通信效率降低，使得系统的不同模块之间不容易协调。
- III. 广义的**LoD**：指对对象之间的信息流量、流向以及信息的影响的控制，主要是对信息隐藏的控制。信息的隐藏可以使各个子系统之间脱耦，从而允许它们独立地被开发、优化、使用和修改，同时可以促进软件的复用，由于每一个模块都不依赖于其他模块而存在，因此每一个模块都可以独立地在其他的地方使用。一个系统的规模越大，信息的隐藏就越重要，而信息隐藏的重要性也就越明显。



(续)

#### IV. 迪米特法则的主要用途在于控制信息的过载。

- ① 在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及
- ② 在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限
- ③ 在类的设计上，只要有可能，一个类型应当设计成不变类
- ④ 在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

■ 总结：一个软件实体应当尽可能少的与其他实体发生相互作用。

# 迪米特法则实例

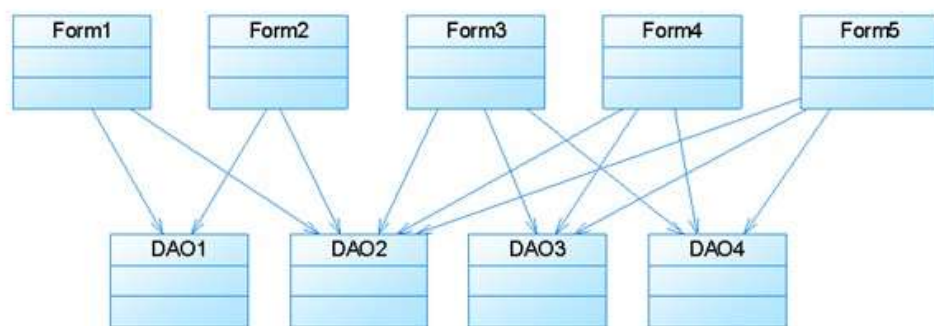


图1

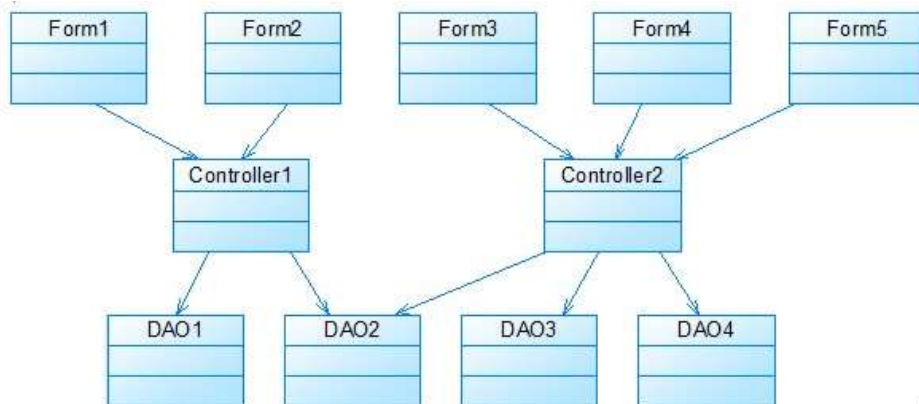


图2

- 某系统界面类（如Form1、Form2等类）与数据访问类（如DAO1、DAO2等类）之间的调用关系较为复杂。

- 图1为什么到图2哪？因为这样就可以降低类的耦合性，是类中功能更加单一，相当于外观模式。





# 单一职责原则

## ■ 定义

- I. 一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。
- II. 就一个类而言，应该仅有一个引起它变化的原因。

## ■ 总结：就一个类而言，应该仅有一个引起它变化的原因。



# 单一职责原则分析

- 一个类（或者大到模块，小到方法）承担的职责越多，它被复用的可能性越小，而且如果一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作。
- 类的职责主要包括两个方面：数据职责和行为职责，数据职责通过其属性来体现，而行为职责通过其方法来体现。
- 单一职责原则是实现高内聚、低耦合的指导方针，在很多代码重构手法中都能找到它的存在，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

# 单一职责原则实例

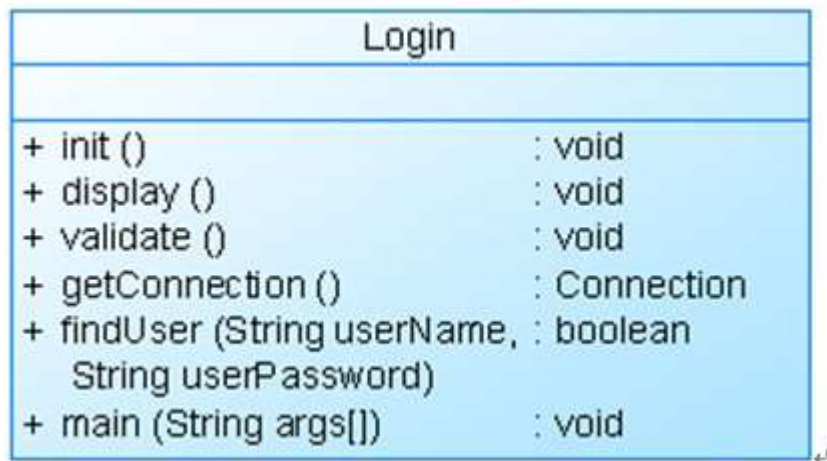


图1

- 某基于Java的C/S系统的“登录功能”通过登录类（**Login**）实现，见图1。

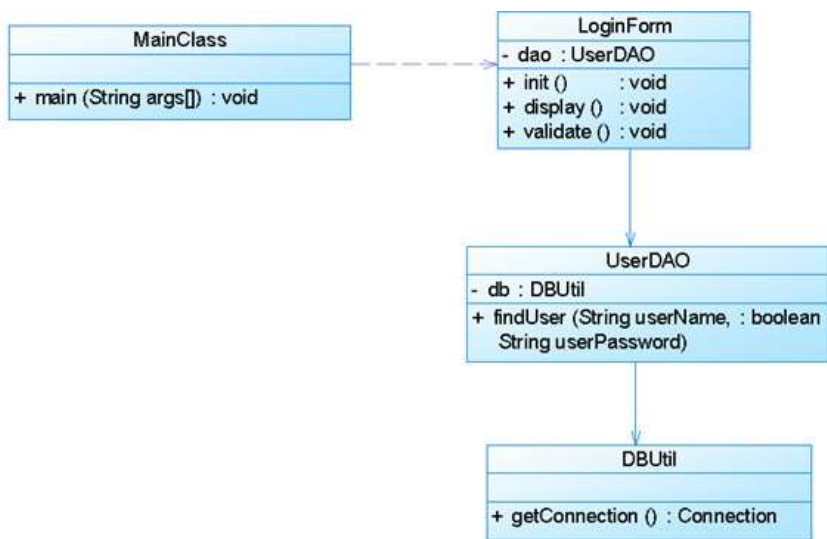


图2

- 图1功能太过于集成，严重违反类的单一原则。图2为按单一原则重新设计的类。



# 依赖倒置原则

## ■ 定义：

- I. 高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。
- II. 要针对接口编程，不要针对实现编程。

## ■ 总结：高层模块不应该依赖底层模块，两个都应该依赖与抽象；抽象不应该依赖于细节，细节应该依赖于抽象。



# 依赖倒置原则分析

- **DIP**简单说：代码要依赖于抽象的类，而不要依赖于具体的类；要针对接口或抽象类编程，而不是针对具体类编程。
- 实现开闭原则的关键是抽象化，并且从抽象化导出具体化实现，如果说**OCP**是面向对象设计的目标的话，那么**DIP**就是面向对象设计的主要手段。
- **DIP**的常用实现方式之一是在代码中使用抽象类，而将具体类放在配置文件中。
- **DIP**要求客户端依赖于抽象耦合，以抽象方式耦合是依赖倒转原则的关键。



# 类之间的耦合关系

- 三种耦合关系：

- ① 零耦合关系
- ② 具体耦合关系
- ③ 抽象耦合关系。

- 依赖注入：

- 构造注入：通过构造函数注入实例变量
- 设值注入：通过**Setter**方法注入实例变量
- 接口注入：通过接口方法注入实例变量

# 依赖倒转原则实例

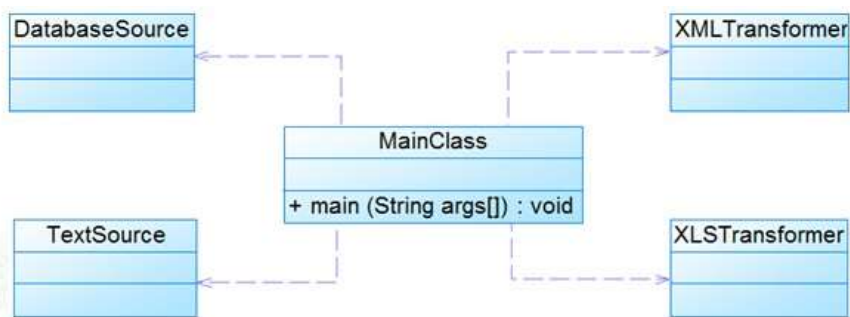


图1

- 某系统提供一个数据转换模块，可以将来自不同数据源的数据转换成多种格式，如可以转换来自数据库的数据(**DataSource**)、也可以转换来自文本文件的数据(**TextSource**)，转换后的格式可以是XML文件(**XMLTransformer**)、也可以是XLS文件

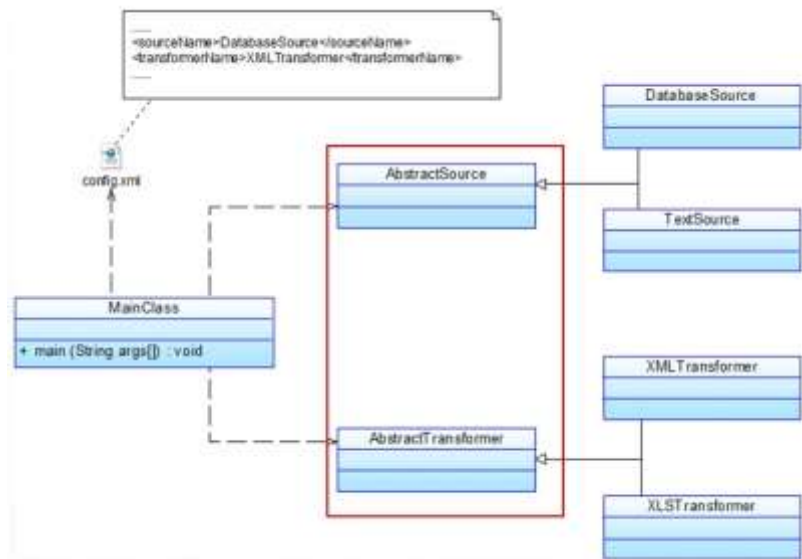


图2

- 图1需要增加新的数据源或者新的文件格式时，客户类**MainClass**都需要修改源代码，以便使用新的类，但违背了开闭原则。现使用依赖倒转原则对其进行重构。



# 面向对象分析与设计

---

## 4 面向对象分析



# 面向对象分析—主要内容

1. 面向对象分析OOA（Object-Oriented Analysis）
2. 领域分析
3. OOA的一般步骤



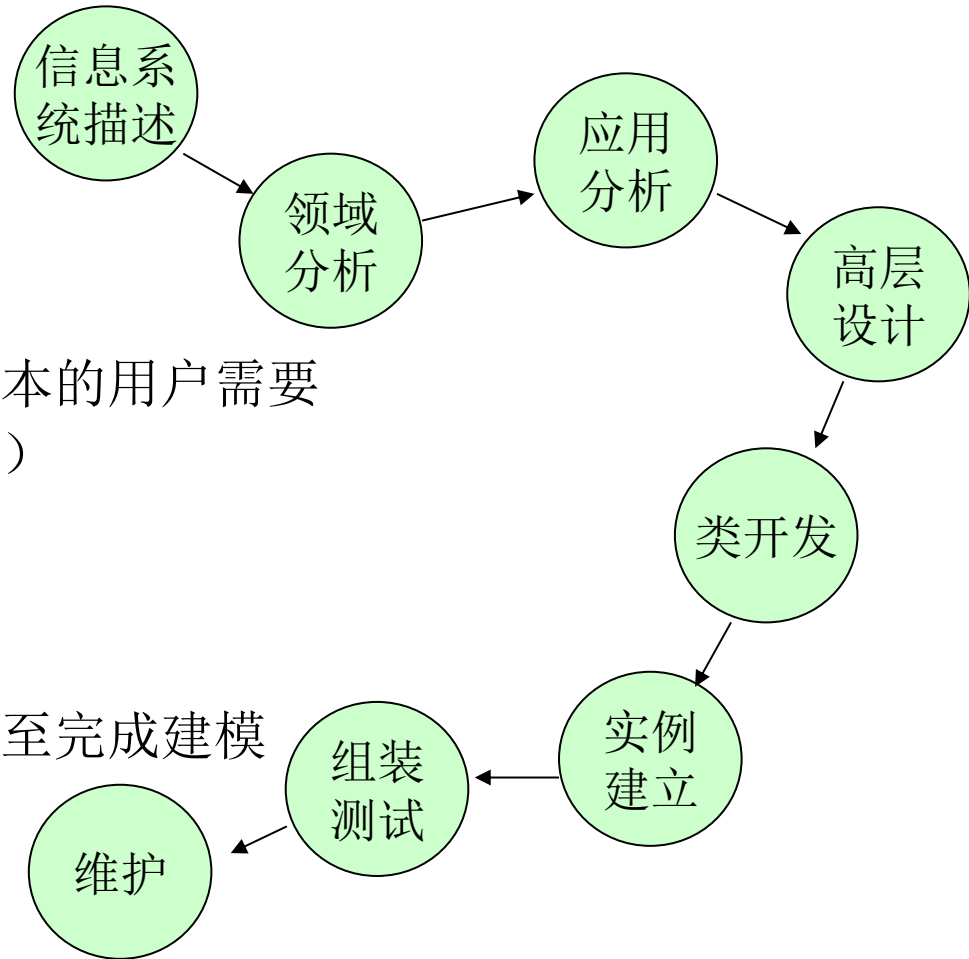
# OOA的目标和任务

## ■ OOA目标

- 完成对所求解问题的分析，确定待建系统要做什么
- 建立系统的模型

## ■ OOA的任务

- ① 在客户和软件工程师之间沟通基本的用户需要
- ② 标识类（包括定义其属性和操作）
- ③ 刻画类的层次结构
- ④ 表示类（对象）之间的关系
- ⑤ 为对象行为建模
- ⑥ 递进地重复任务①至任务⑤，直至完成建模



# 4.1 面向对象分析

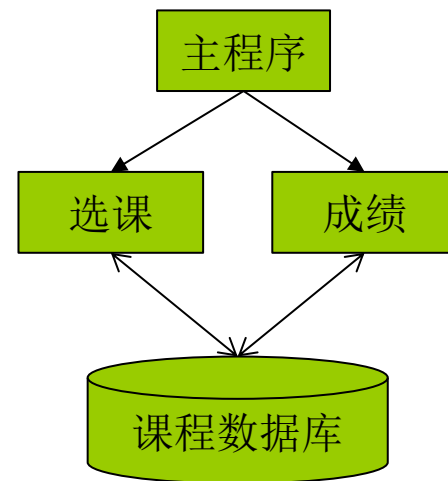
- 传统方法和面向对象方法
- 面向对象分析概述



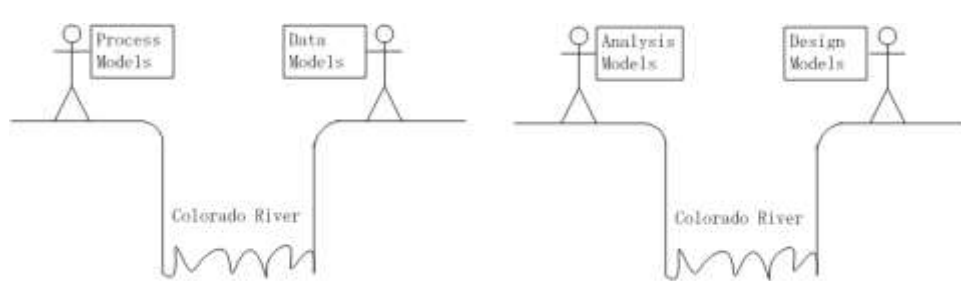
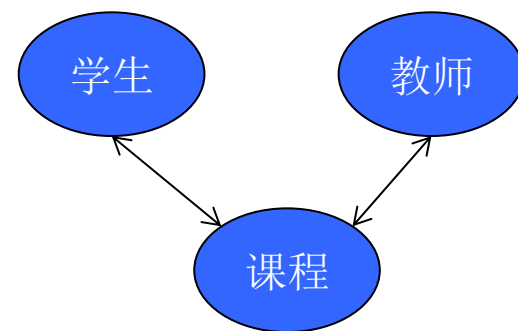
# 传统方法与面向对象方法

需求分析产生描述系统功能和**问题域**基本特征的综合文档。

- **传统文档**：面向功能，把系统看成一组功能，围绕输入—加工—输出的需求视角。
- **OOA文档**：把问题当作一组相互作用的实体，并确定实体间关系



传统方法



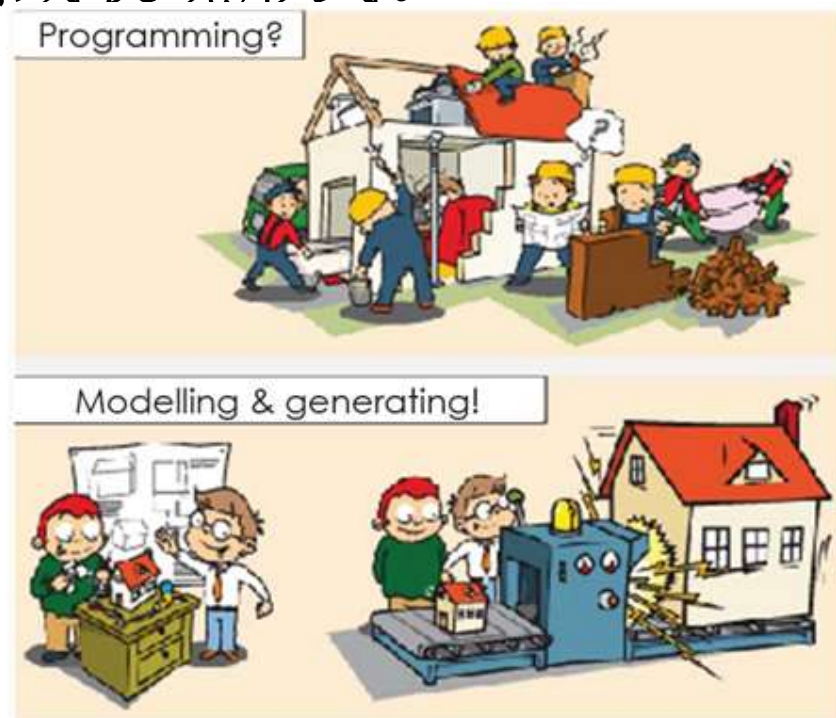
Two Classic Systems Development Problem

# 模型 (model)

- 模型是现实的抽象。
- 对同一个事物，可以从不同的视角描述这个事物，从而得到不同的模型。
- 模型可以有不同的描述方法或表现形式。

## Restraining factors to construct a model

- *Assumptions* (假设)
- *Simplifications* (简化)
- *Limitations* (限制)
- *Constrains* (约束)
- *Preferences* (偏爱)

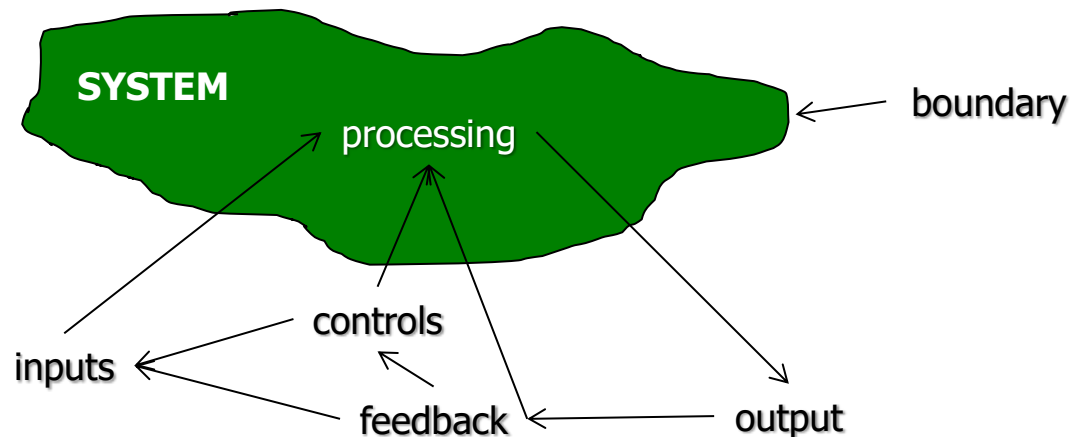


# Systems Model with 6 Components

## Model

- *Define the processes that serve the needs of the view under consideration.*
- *Represent the behavior of the processes and the assumptions on which the behavior is based.*
- *Explicitly define both exogenous and endogenous input to the model.*
- *Represent all linkage (including output) that will enable the engineer to better understand the view.*

-- Mortamarrin, 1992



## 4.2 领域分析

- 领域分析与复用
- 领域分析过程



# 什么是领域？

- 领域（Domain）是指一组具有相似或相似软件需求的应用系统所覆盖的功能区域。一个领域是一组相关的系统，相关的方面是公共设计、公共服务、公共技术和公共信息体系结构。领域内的不同系统，相同系统的不同版本之间由于共享领域知识，因此具有相似的设计、体系结构、服务和技术。
- 领域工程是为领域产品族中相似应用系统建立核心资产的过程。这些核心资产包括领域需求、领域框架、以及领域可复用构件，领域工程覆盖了开发可利用软件构件的所有活动。
- 应用工程则是根据领域工程中所生产的核心可复用软件构件进行基于可复用软件构件的应用系统开发的过程和活动。

## 软件复用的效益

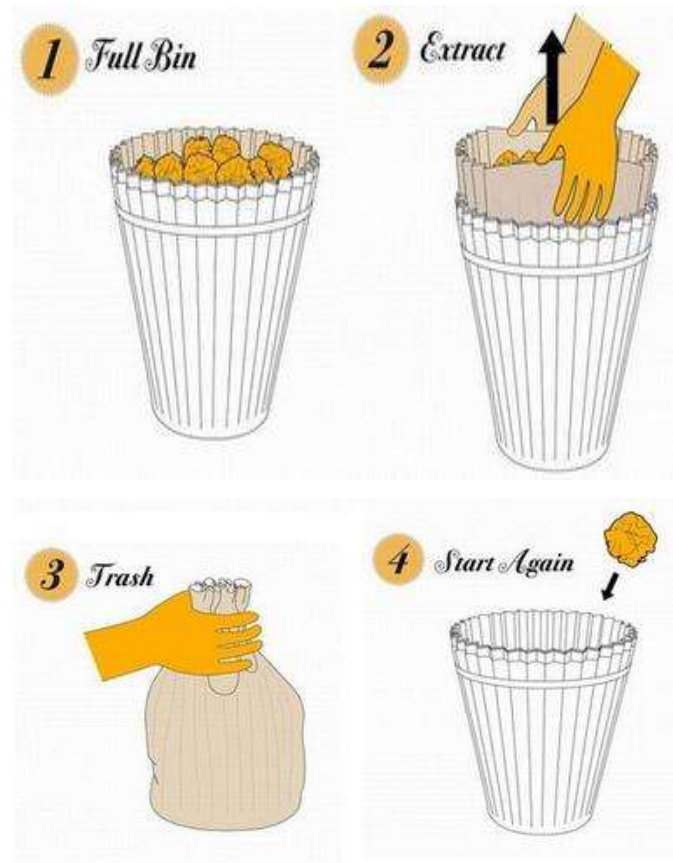
- 质量：复用构件错误较少
- 生产率：30-50%的复用导致生产提高25-40%
- 成本节省





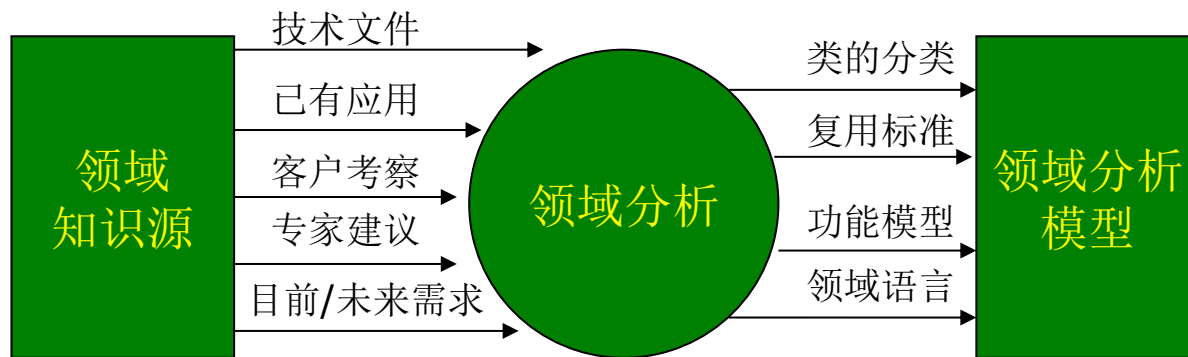
# 什么是领域分析？

- 领域分析指对特定问题空间内的知识进行提取、整理和建模，它为以后涉及该问题空间的系统开发提供**复用基础**。
- 领域分析的目标是为领域产品族中相似应用系统建立核心资产的过程，用于应用工程的复用。
- 领域工程可以分为以下三个主要阶段：**领域分析、领域设计和领域实现**。



# 领域分析与复用

- 领域分析是为了建立复用的类库。
- 以公共对象、类、子集合和框架等形式，在特定的应用领域中标识、分析和规约公共的可复用的能力。
- 领域分析是软件过程的一个全程活动，不以任何软件项目联系，它类似于制造环境中的工具制造者。



领域分析过程

- 定义领域范围
- 领域中项目的分类
- 收集有代表性的应用



# 面向对象复用

- 软件成分的复用级别
  - 代码复用
  - 设计复用
  - 分析复用
- 可复用构件的特点：
  - 模块独立性强
  - 具有高度可塑性：提供扩充和修改机制
  - 接口清晰、简明、可靠，详尽的文档说明
- 类构件的重用方式
  - 实例复用
  - 继承复用
  - 多态复用

## 4.3 OOA的一般步骤

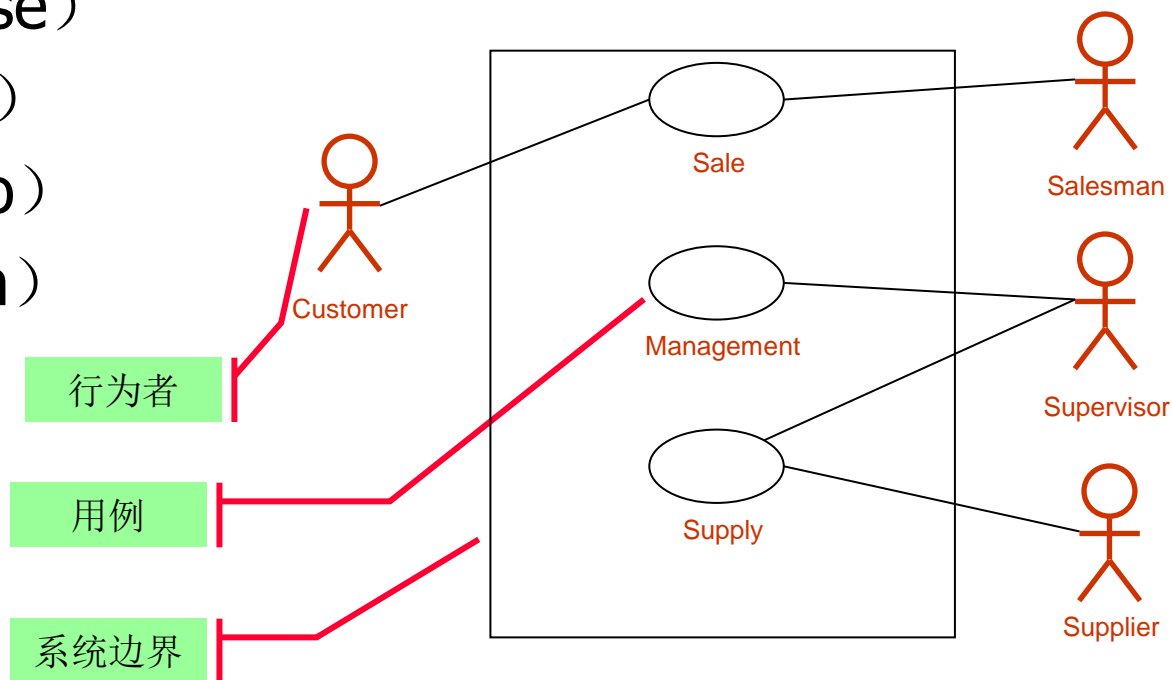
- ① 获取客户对系统的需求：包括标识场景（**scenario**）和用例（**use case**），以及建造需求模型。
- ② 用基本的需求为指南来选择类和对象
- ③ 定义类的结构和层次
- ④ 建造对象-关系模型
- ⑤ 建造对象-行为模型
- ⑥ 利用用例/场景来复审分析模型



# 1) 获取客户对系统的需求

必须让客户与开发者充分地交流。

- 用例 (user case)
- 行为者 (actor)
- 场景 (scenario)
- 功能 (function)



Use-case图



## 2) 标识类和对象

- CRC索引卡，Class-Responsibility-Collaborator，类-责任-协作者。

类名：

类的类型：（如：设备、角色，场所，……）

类的特征：（如：有形的、原子的，并发的，……）

责任：

协作者：



# 识别类对象

- 外部实体
  - 他们生产或消费计算机系统所使用的信息，
  - 如：其它系统、设备、人员
- 角色
  - 他们由与系统交互的人扮演，
  - 如：管理者、工程师、销售员
  - 也可以是其它的系统。
- 构造物
  - 它们定义一类对象，或者定义对象的相关类
  - 如：交通工具、计算机
- 发生的事件
  - 它们出现在系统运行的环境中
  - 如，完成一系列遥控动作
- 组织单位
  - 他们与一个应用有关
  - 如：部门、小组、小队

- 场所
  - 它们建立问题和系统所有功能的环境
  - 如：制造场所、装载码头
- 事物
  - 它们是问题信息域的一部分
  - 如：报告、显示、信函、信号

潜在对象成为最终的对象：

- 保留的信息
- 需要的服务
- 多个属性
- 公共属性
- 公共操作
- 必需的需求



# 责任与协作者

## 责任 (Responsibility)

- 是类所知道的或要做的任何事情。
- 是与类相关的属性和操作。
  - 属性定义了对象，阐明了问题空间中对象的作用和地位。
  - 操作定义了对象的行为并以某种方式修改对象的属性值。操作分为三类：
    - 操纵数据的操作
    - 计算的操作
    - 为控制事件的发生而监控对象的操作

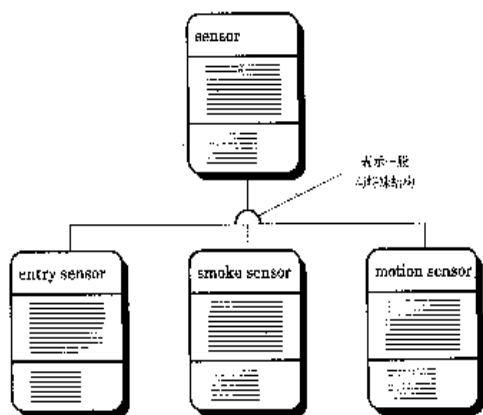
## 协作者 (Collaborator)

- 如果一个对象为完成某个责任需要向其它对象发送消息，则该对象和另一个对象协作。
- 协作反映了类之间的关系。
- 一个类的协作可以通过确定该类是否能独立完成每个责任来标识，如果不能，则它需要和另一个类交互，从而可识别一个协作。

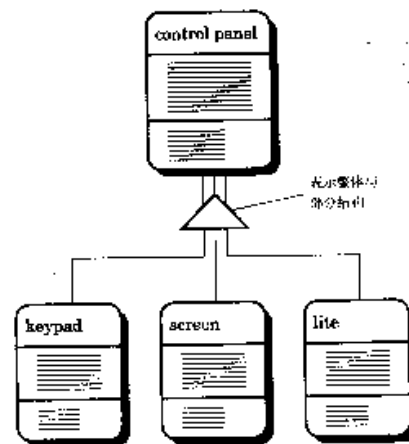


### 3) 定义类的结构和层次

- 类的结构主要有两种：
  - 一般-特殊（generalization-specialization）：“is a”的关系
  - 整体-部分（whole-part）：“has a”的关系
- 可以将互相协作的类的集合定义为主题（subject）或子系统（subsystem）。
- 类结构和层次图(Coad & Yourdon)：



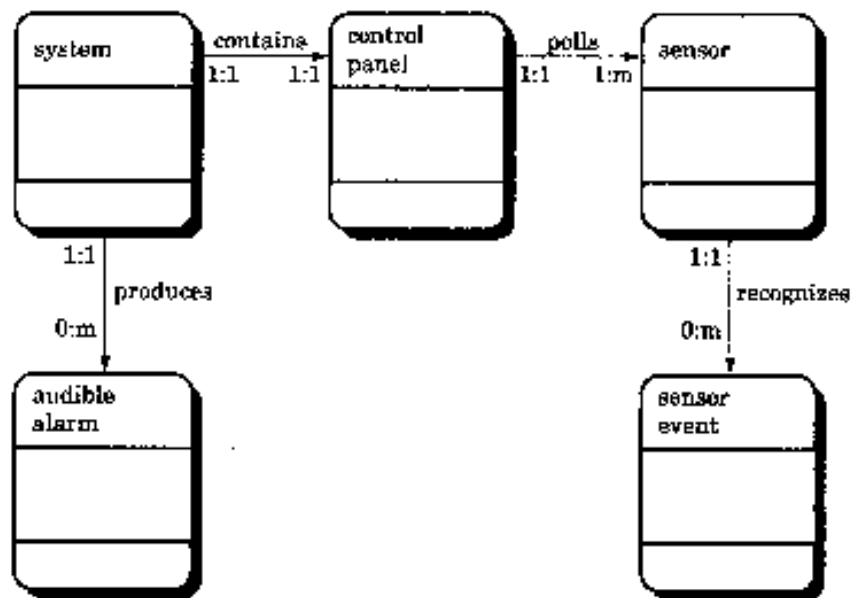
一般与特殊结构



整体与部分关系

## 4) 建造对象-关系模型

- 对象-关系模型描述了系统的静态结构，它指出了类间的关系（**relationship**）及对象之间的消息路径。
- 三个步骤：
  - ① 用CRC画协作者对象网络
  - ② 复审CRC，标出箭头
  - ③ 确定关系基数（1对1，1对多，多对1，多对多）



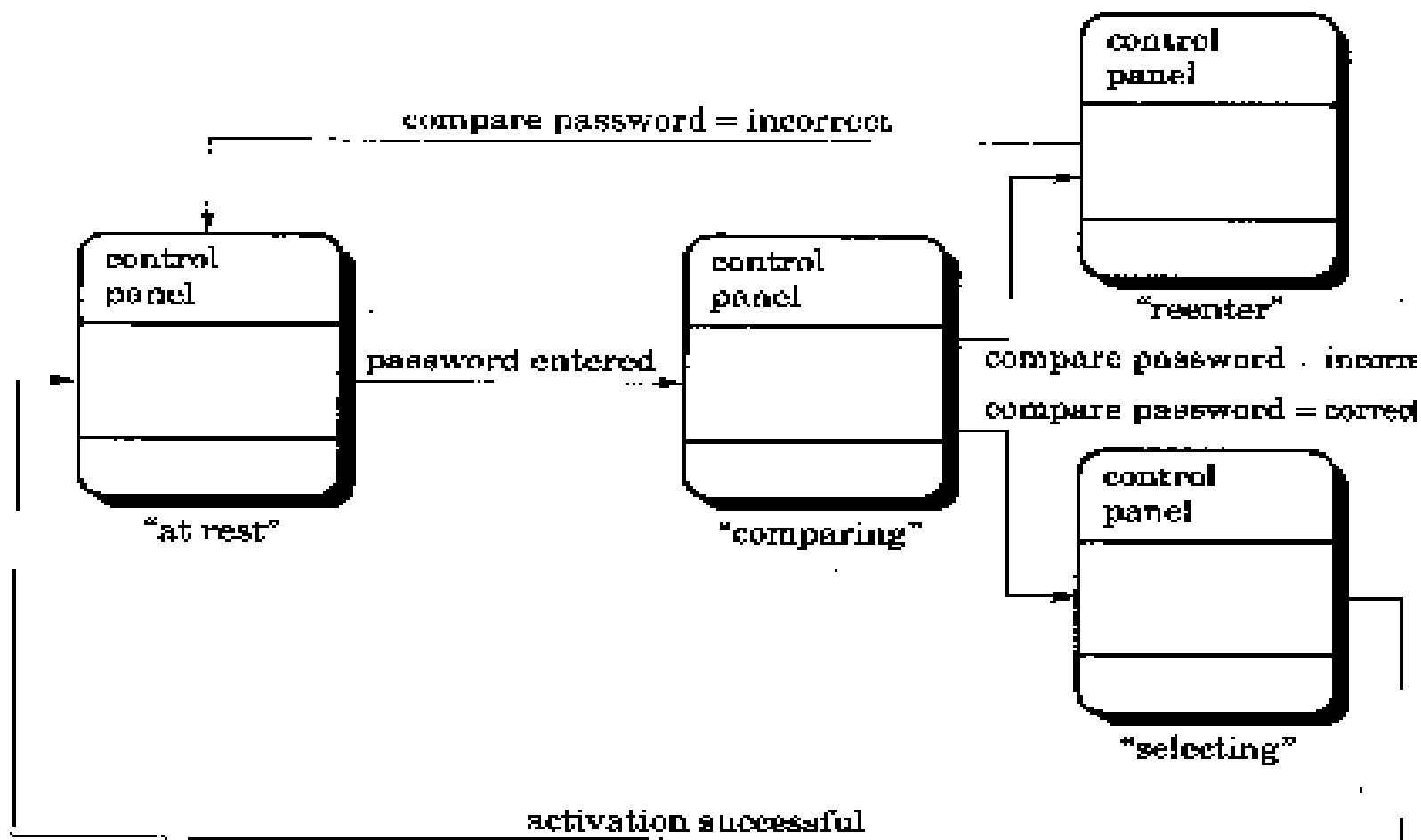
类间的连接（完成责任的协作者类），最常见是二元关系如：**1对1**，**1对多**，**多对1**，**多对多**。



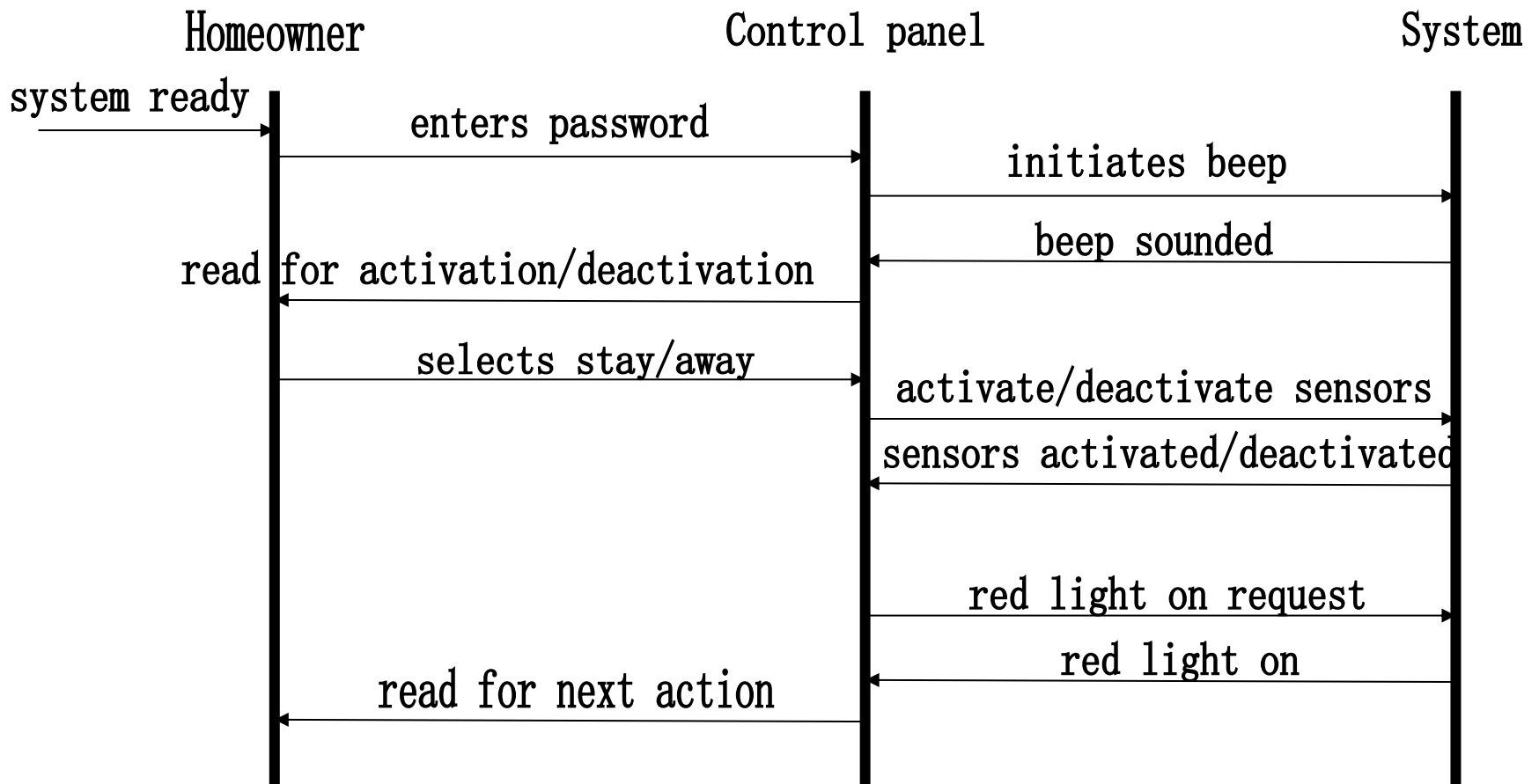
## 5) 建立对象-行为模型

- 对象-行为模型描述了系统的动态行为，它们指明系统如何响应外部的事件或激励（**stimulus**）。
- 目的是描述系统的动态行为。
- 对所有实例要完全理解系统中交互的序列
- 找出驱动交互序列的事件
- 创建事件轨迹
- 创建状态-变迁（转换）图
- 检查模型的正确
- 状态的表示要考虑两个方面：
  - 每个对象的状态变化（状态图）；
  - 系统中各对象的状态关系（轨迹、时序图）。

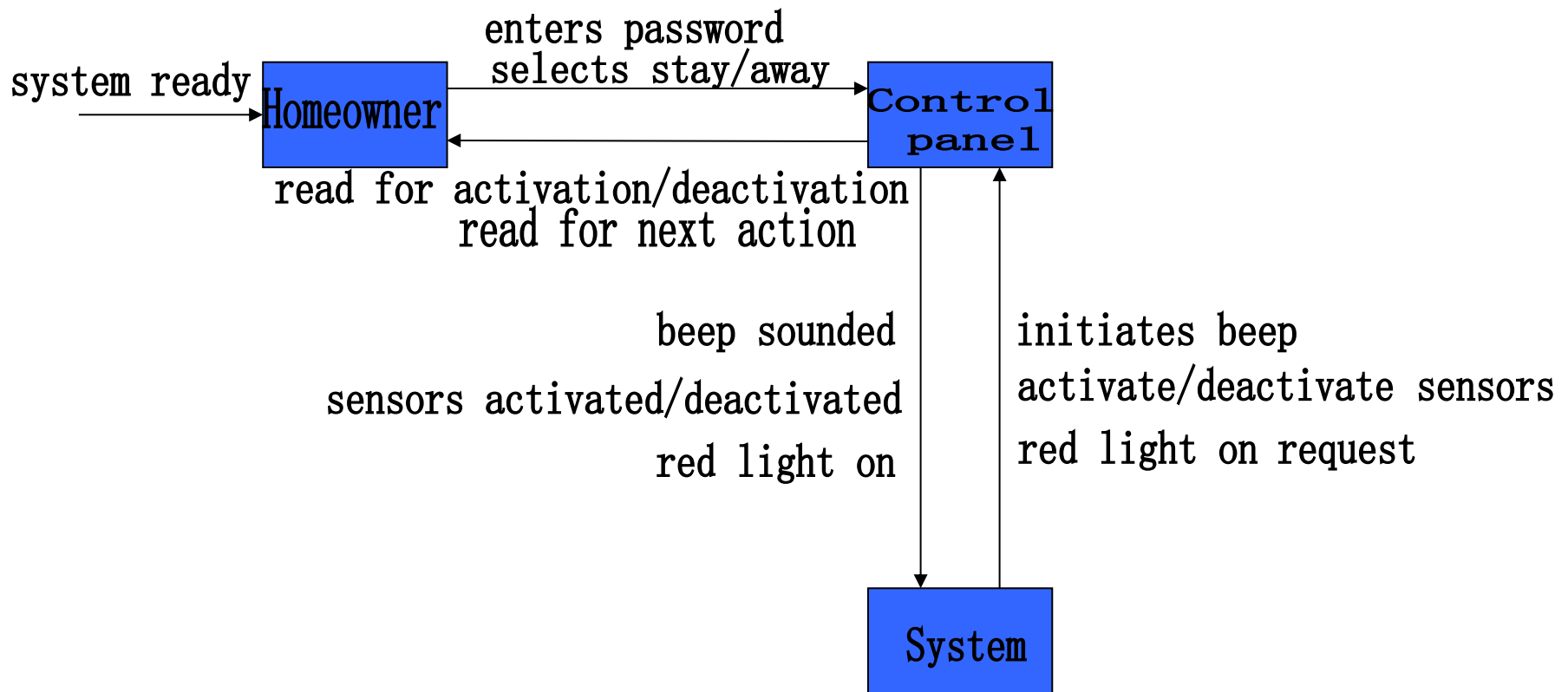
# 对象行为模型（状态迁移）



# 对象行为模型（轨迹图）



# 对象行为模型（事件流图）





# 面向对象分析与设计

---

## 5 面向对象设计

# 面向对象的设计——主要内容

1. 传统与OOD方法比较
2. OOD模型的体系成分
3. OOD设计方法
4. OOD步骤







## 5.1 传统与OOD方法比较

	传统方法	<b>OOD方法</b>
相同点	数据设计	属性设计
	接口设计	消息模型
	过程设计	操作设计
区别 (总体结构设计)	层次化	对象间的协作

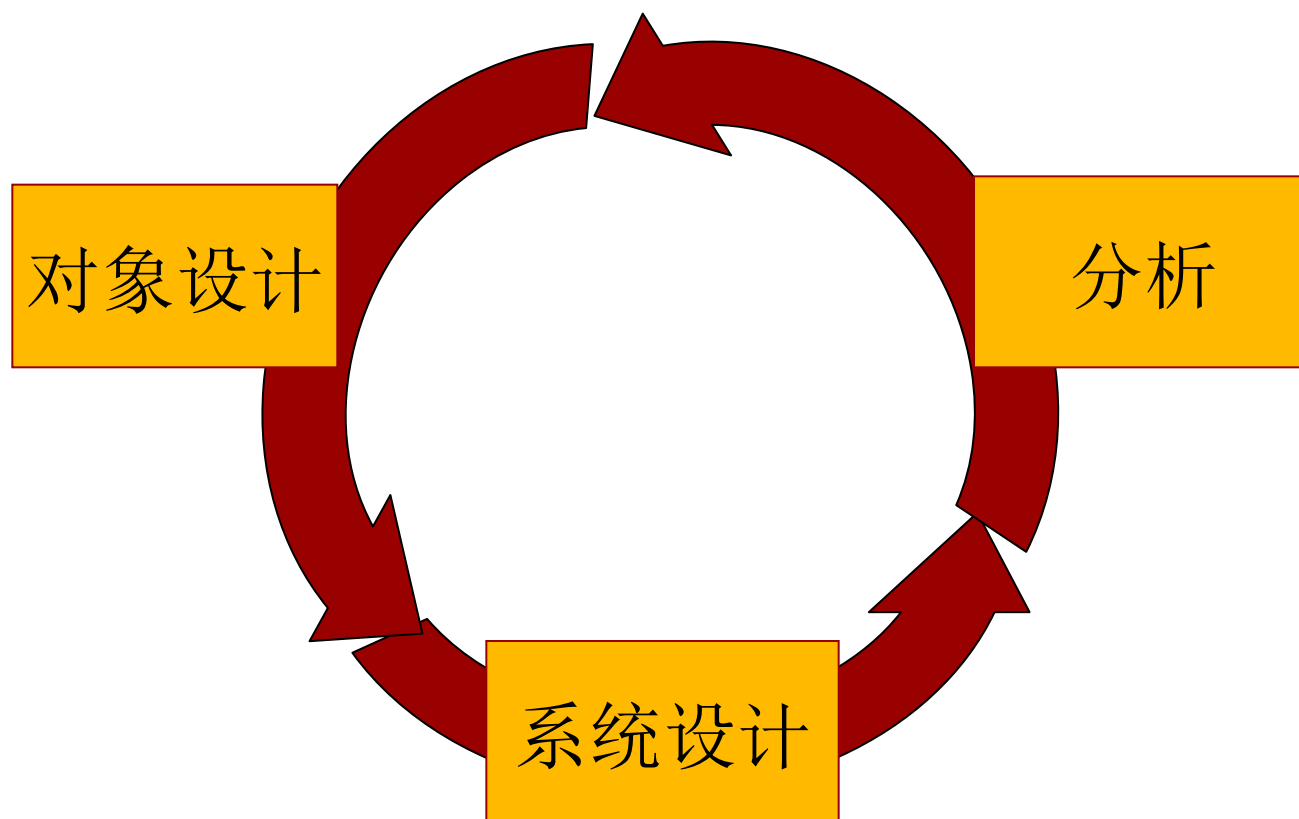
# 系统设计的十个方面

1. 模块层次的表示
2. 数据定义的规约
3. 过程逻辑的规约
4. 端点间处理序列的指明
5. 对象状态和转变的表示
6. 类及层次的定义
7. 类的操作
8. 详细的操作定义
9. 消息连接的规约
10. 独有的服务标识



结构化方法不支持5~10条

## 5.2 OOD模型的体系成分



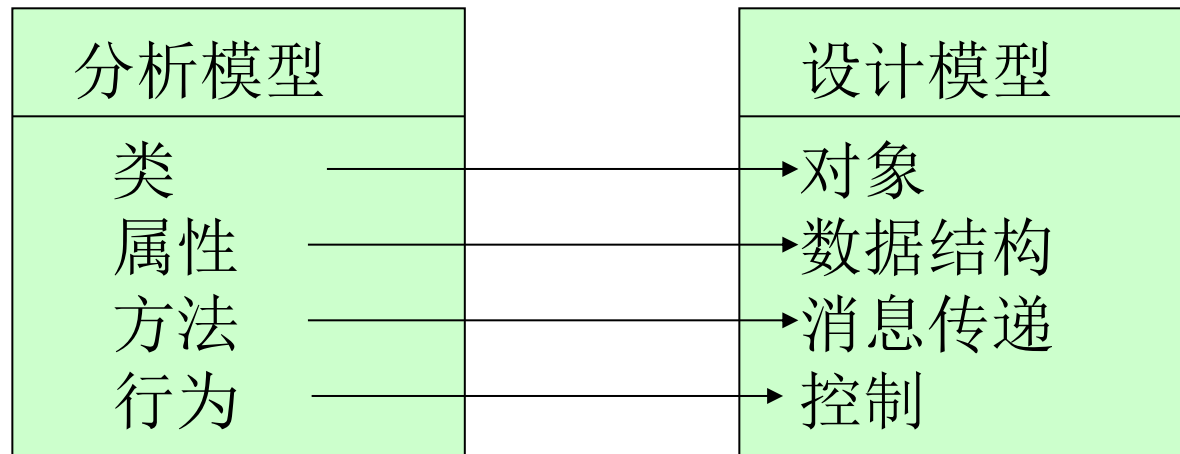


# 面向对象的设计（OOD）

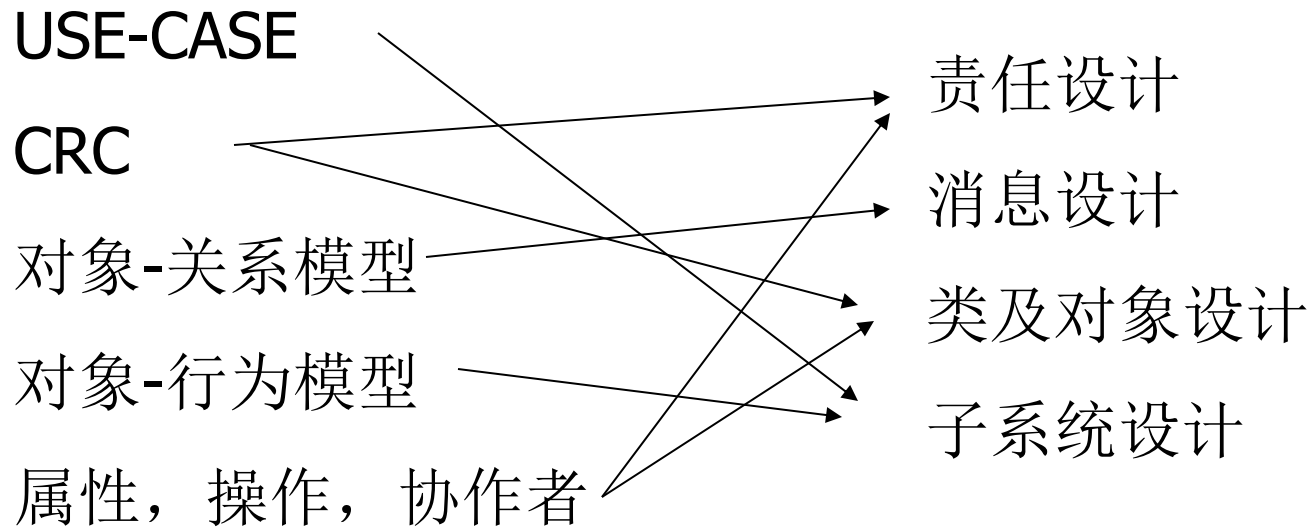
- 面向对象设计（Object-oriented Design, OOD）是将OOA所创建的分析模型转化为设计模型。
- OOD与OOA之间没有明显的分界线，两阶段采用相同的符号表示。
- 在**OOA**时，主要考虑系统做什么，而不关心系统如何实现；在**OOD**时，主要解决系统如何做。
- OOD要在OOA的模型中为系统补充一些新的类，或在原有的类中补充一些属性和操作。
- OOD时应能从类中导出对象，以及对象的关联，描述对象的关系、行为以及对象通讯等。


# OO设计模型的体系成分

- 系统设计要定义四种重要构件：
  - 问题域：负责实现需求的子系统
  - 人机交互：实现用户界面子系统
  - 任务管理：负责控制和协调并发任务的子系统
  - 数据管理：负责对象的存储和检索子系统



## 5.3 OO设计方法





# 设计中的问题

Bertrand Meyer的设计方法模块化能力评判：

- 分解性（方法能提供帮助设计者分解大型问题）
- 组装性（方法能支持设计的构件可复用）
- 易理解性（使模块可独立被理解）
- 连贯性（使修改后对系统影响小）
- 保护性（出现问题时对系统的影响小）

从而对设计提出基本设计原则：语义模块单元；很少接口；接口简单；显式接口；信息隐蔽等

## 5.4 OOD的步骤

### ① 系统设计

- 将子系统分配到处理器；
- 选择实现数据管理、界面支持和任务管理的设计策略；
- 为系统设计合适的控制机制
- 复审并考虑权衡

### ② 对象设计

- 在过程级别设计每个操作
- 定义内部类
- 为类属性设计内部数据结构

### ③ 消息设计

- 使用对象间的协作和对象-关系模型，设计消息模型。

### ④ 复审

- 复审设计模型，并在需要时迭代。







# 系统设计

## 1) 将分析模型划分成子系统

- 子系统的设计准则是：

- 子系统应该具有定义良好的接口，通过接口和系统的其它部分通信；
- 除了少数的通信类外，子系统内的类应只和该子系统内的其它类协作；
- 子系统的数量不宜太多；
- 可以在子系统内部再次划分，以降低复杂度。

## 2) 标识问题本身的并发性，并为子系统分配处理器

- 通过对对象-行为模型的分析，找出系统的并发性。
- 并发的子系统可以分配在不同的处理器上，也可以分配在同一个处理器上，由操作系统提供并发处理。



# (续)

## 3) 任务管理设计

- 设计策略：
  - 确定任务的特征
  - 定义协调者任务和关联的对象
  - 集成协调者任务和其它任务
- 可以通过任务如何被启动的，来确定任务的特征。

## 4) 数据管理设计

- 数据管理设计包括：
  - 系统中各种存储对象的存储方式，如：内存数据结构、文件、数据库
  - 设计相应的服务，即为要存储的对象增加所需的属性和操作。

### 基本任务模板：

- 任务名
- 描述
- 优先级
- 服务
- 由...协调
- 通过...通信。



## (续)

### 5) 资源管理设计

- 设计一套控制机制和安全机制，以控制对资源的访问，避免对资源使用的冲突。

### 6) 人机界面的设计

### 7) 子系统间的通信

- 子系统之间可以通过建议C/S连接进行通信，也可以通过端到端（peer to peer）连接进行通信。



# 对象设计


- 对象设计是为每个类的属性和操作作出的详细设计，并设计连接类与它的协作者之间的消息规约。

## 1) 对象描述

- 对象描述的形式有：
  - 协议描述：描述对象的接口
  - 实现描述：每个操作的实现细节

## 2) 设计算法和数据结构

- 为对象中的属性和操作设计数据结构。



# 对象描述—协议描述

- 协议描述

- 通过接收的消息和完成的操作来建立对象的接口

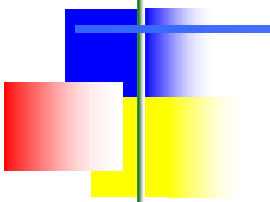
- 例:

- 读传感器时所需消息:

Message(motion sensor) read: RETURNS sensor ID, sensor status;

- 设置和复位传感器:

Message(motion sensor) set: SENDS sensor ID, sensor status;



# 对象描述—实现描述

- 实现描述
  - 从消息蕴涵的操作来表明对象数据结构的内部细节和操作过程。
- 实现描述如下信息：
  - 对象名字的定义和类的引用；
  - 表明数据项和类型的私有数据结构定义；
  - 每个操作的过程描述或指针。
- 例：
  - 对象名： smoke sensor
  - 类名： sensor
  - 私有数据结构： ID: string;
  - 操作过程： set/reset/read/get...



# 算法设计

- 系统中类的操作通常有三类：
  - 操纵数据的操作；
  - 执行计算的操作；
  - 监控对象的操作。



# 设计模式（pattern）

- 在许多面向对象的系统中，存在一些类和通信对象的重复出现模式。
- 一个设计模式通常用四类信息描述：
  - 模式名
  - 应用模式时必须存在的环境条件
  - 设计模式的特征
  - 应用设计模式的结果





# 面向对象分析与设计

## 6 面向对象测试

# 面向对象测试—主要内容

1. OO单元测试
2. OO集成测试
3. 确认测试与系统测试
4. OO软件的测试用例





# OO概念对测试的影响

- 封装性和继承性给OO软件开发带来很多好处，同时它又对OO软件测试带来负面影响。
- 类的属性和操作是被封装的，而测试需要了解对象的详细状态。同时，需要考虑当改变数据成员的结构时，是否影响了类的对外接口，是否导致相应外界必须改动。
- 继承不会减少对子类的测试，相反，会使测试过程更加复杂化。当父类与子类的环境不同时，父类的测试用例对子类没有什么使用价值，必须为子类设计新的测试用例。



# 6.1 OO软件的单元测试

- 封装起来的类和对象是最小的可测试单元。
- 一个类可以包含一组不同的操作，而一个特定的操作也可能定义在一组不同的类中。
- 全面地测试类和对象所封装的属性和操纵这些属性的操作整体。
- 在OO的单元测试中不仅要发现类的所有操作中存在的问题，还要考查一个类与其他的类协同工作时可能出现的错误。



## 6.2 OO软件的集成测试

- OOP没有层次的控制结构，传统的集成测试不能用于OOP中。
- 由于OOP具有动态性，程序的控制流往往难以确定，因此只能做基于黑盒方法的集成测试。
- OO集成测试主要关注于系统的结构和内部的相互作用。



# OO集成测试的两种方法:

## ■ 基于线程的测试(**Thread-based Testing**)

- 基于线程的测试是指把响应系统一个输入或一个事件所需要的一组类集成起来进行测试。
- 应当分别集成并测试每个线程，同时为了避免产生副作用再进行回归测试。

## ■ 基于使用的测试(**Use-based Testing**)

- 首先，测试独立类——几乎不使用服务器类的那些类，把独立类都测试完；
- 接着，测试依赖类——使用独立类的最下层的类。
- 然后，根据依赖类的使用关系，从下到上一个层次一个层次地持续进行测试，直至把整个软件系统测试完为止。



## 6.3 OO软件的确认测试与系统测试

- 通过对OO软件单元测试和集成测试，还必须经过规范的确认测试和系统测试。
- OO软件的确认测试或系统测试，与传统的确认测试一样，通过设计测试用例，主要检查用户界面和用户可识别的输出，不再考虑类之间相互连接的细节。



## 6.4 设计测试用例

- 1993年，Berard提出了指导OO软件测试用例设计的方法，其要点如下：
  - 每一个测试用例都要有一个惟一的标识，并与被测试的一个或几个类相关联起来；
  - 每个测试用例都要陈述测试目的；
  - 对每个测试用例要有相应的测试步骤，包括被测对象的特定状态，所使用的消息和操作，可能产生的错误及测试需要的外部环境。
  - 与传统软件测试（测试用例的设计由软件的输入—处理—输出或单个模块的算法细节驱动）不同，面向对象测试关注于设计适当的操作序列以检查类的状态。





# 设计OO测试用例应注意：

- ① 继承的成员函数需要测试。一般在下述两种情况下要对成员函数重新进行测试：
  - 继承的成员函数在子类中有所改动；
  - 成员函数调用了改动过的成员函数。
- ② 子类的测试用例可以参照父类。
- ③ 设计测试用例时，不但要设计确认类功能满足的输入，而且还应有意识地设计一些被禁止的例子，确认类是否有不合法的行为产生。



# 类测试用例设计

一般来说，在设计测试用例时，可参照下列步骤：

- ① 根据**OOD**分析结果，选定检测的类，并仔细分出类的状态和相应的行为，以及成员函数间传递的消息和输入输出的界定。
- ② 确定覆盖标准。
- ③ 利用结构关系图确定待测试类的所有关联。
- ④ 构造测试用例，确认使用什么输入来激发类的状态，使用类的什么服务，期望产生什么行为。



# 基于故障的测试用例设计

- 基于故障的测试（**Fault-based Testing**）与传统的错误推测法类似，通过对OOA/OOD模型的分析，首先推测软件中可能有的错误，然后设计出最可能发现这些错误的测试用例。
- 基于故障的测试用例有一个很突出的缺点：当功能描述是错误的或子系统间交互存在错误时，基于故障的测试用例就无法发现错误。



# 基于用例的测试用例设计

- 基于用例（**Use-case-based Testing**）的测试用例更关心的是用户想做什么而不是软件想做什么，通过用例获取用户要完成的功能，并以此为依据设计所涉及的各个类的测试用例。
- 更具体地说，先搞清楚用户想实现哪些功能？然后去寻找要完成的这些功能，需要哪些类参与，从功能出发，对所确定的这些类及其子类分别设计类测试用例。



# 类间测试用例设计

- 在集成面向对象系统阶段，必须对类间协作进行测试，因此测试用例的设计变得更加复杂。
- 通常可以从**OOA**的对象-关系模型和对象-行为模型导出类间测试用例。
- 测试类协作可以使用随机测试方法和划分测试方法，以及基于情景的测试和行为测试来完成。



# 多个类测试

- 多个类的划分测试方法有三种：
  - 基于状态的划分：根据类操作改变类状态的能力来划分类操作；
  - 基于属性的划分：根据类操作使用的属性来划分类操作；
  - 基于功能的划分：根据类操作所完成的功能来划分类操作。
- 另外，多类测试还应该包括那些通过发送给协作类的消息而被调用的操作。根据与特定类的接口来划分类操作又是一种划分测试方法。



# 从行为模型导出测试用例。

- 动态行为模型是由几个状态转换图组成的。根据类的状态图，可以设计出测试该类以及类间的动态行为的测试用例。



# 面向对象分析与设计

## 7 OOA与OOD方法介绍





# OOA与OOD方法—主要内容

- Coad & Yourdon方法
- Rumbaugh的OMT方法
- Booch方法
- OOSE方法



## 7.1 Coad & Yourdon方法

- Coad & Yourdon方法比较简单，也是最容易学习的一种OO方法。
- Coad & Yourdon的OOA模型由五个层次和五个活动组成。
- Coad & Yourdon的OOD模型由四个部件和四个活动组成。



# OOA的五个层次和五个活动

## OOA的五个层次：

- ① 主题层：控制一次分析所考虑的范围，即对相关的类进行归并。
- ② 类及对象层：在分析范围内找出全部的对象。
- ③ 结构层：分析对象的分类结构和组装结构。
- ④ 属性层：描述每个对象的状态特征。
- ⑤ 服务层：描述每个对象所具有的操作。

## OOA五个活动：

- ① 确定主题
- ② 识别类及对象
- ③ 识别类的结构
- ④ 定义属性
- ⑤ 定义服务

# OOD模型及组成



Coad & Yourdon OOD模型

## ■ 四个部件

- ① 问题域部件
- ② 人机交互部件
- ③ 任务管理部件
- ④ 数据管理部件

## ■ 四个活动

- ① 设计问题域部件
- ② 设计人机界面部件
- ③ 设计任务管理部件
- ④ 设计数据管理部件



# 问题域部件 (problem domain component)

- 组合所有的领域特定类
- 为应用类设计适当的类层次
- 必要时简化继承
- 细化设计以改善性能
- 开发与数据管理部件的接口
- 按要求细化并加入低层次的对象
- 复审设计并审查对分析模型的增补



# 人机交互部件 (human interface component)

- 定义参与人员
- 开发任务场景
- 设计用户命令的层次
- 细化用户交互序列
- 设计相关的类和类层次
- 必要时集成图形用户界面



# 任务管理部件 (task management component)

- 标识任务的类型
- 建立优先级
- 标识作为其它任务协作者的任务
- 为每任务设计合适的对象



# 数据管理部件 (data management component)

- 设计数据结构
- 设计管理数据结构所需要的服务
- 标识可以协助实现数据管理工具
- 设计适当的类和类层次





## 7.2 OMT方法

- OMT（Object Modeling Technique）方法是由Rumbaugh等人提出的。
- OMT的三种模型
  - 对象模型
    - 描述一个系统中对象的结构，它表示静态的、结构上的、系统的数据特征。
  - 动态模型
    - 描述与时间和操作顺序有关的系统特征，它表示瞬时的、行为的、系统的控制特征。
  - 功能模型
    - 描述与值的变换有关的系统特征，它表示变换、系统的功能特征。



# OMT的OOA过程

## ① 问题陈述

- 问题陈述为记下或获取对问题的初步描述。

## ② 构造对象模型

- 确定对象类。
- 编制描述类、属性及关联的数据词典。
- 在类之间加入关联。
- 给对象和链加入属性。
- 使用继承来构造和简化对象类。
- 将类组合成模块，这种组合在紧耦合和相关功能上进行。
- 最后得到：对象模型=对象模型图+数据词典。



# (续)

## ③ 构造动态模型

- 准备典型交互序列的脚本。
- 确定对象间的事件并为各脚本安排事件跟踪。
- 准备系统的事件流图。
- 开发具有重要的动态行为的各个类的状态图。
- 检查状态图中共享事件的一致性和完整性。
- 最后得到： 动态模型=状态图+全局事件流图。

## ④ 构造功能模型

- 确定输入、 输出值。
- 需要时使用数据流图来表示功能依赖关系。
- 描述各功能“干什么”。
- 确定约束。
- 详细说明优化标准。
- 最后得到： 功能模型=数据流图+约束。



# OMT的OOD过程

## ① 系统设计

- 将系统分解为各子系统。
- 确定问题中固有的并发性。
- 将各子系统分配给处理器及任务。
- 根据数据结构、文件及数据库来选择实现存储的基本策略。
- 确定全局资源和制定控制资源访问的机制。
- 选择实现软件控制的方法。
- 考虑边界条件。
- 最后得到：系统设计文档=系统的基本结构+高层次决策策略。



# (续)

## ② 对象设计

- 从其他模型获取对象模型上的操作：在功能模型中寻找各个操作，为动态模型中的各个事件定义一个操作，与控制的实现有关。
- 设计实现操作的算法：指选择开销最小的算法，选择适合于算法的数据结构，定义新的内部类和操作。给那些与单个类联系不太清楚的操作分配内容。
- 优化数据的访问路径：指增加冗余联系以减少访问开销，提高方便性，重新排列运算以获得更高效率。为防止重复计算复杂表达式，保留有关派生值。
- 实现系统设计中的软件控制。



## (续)

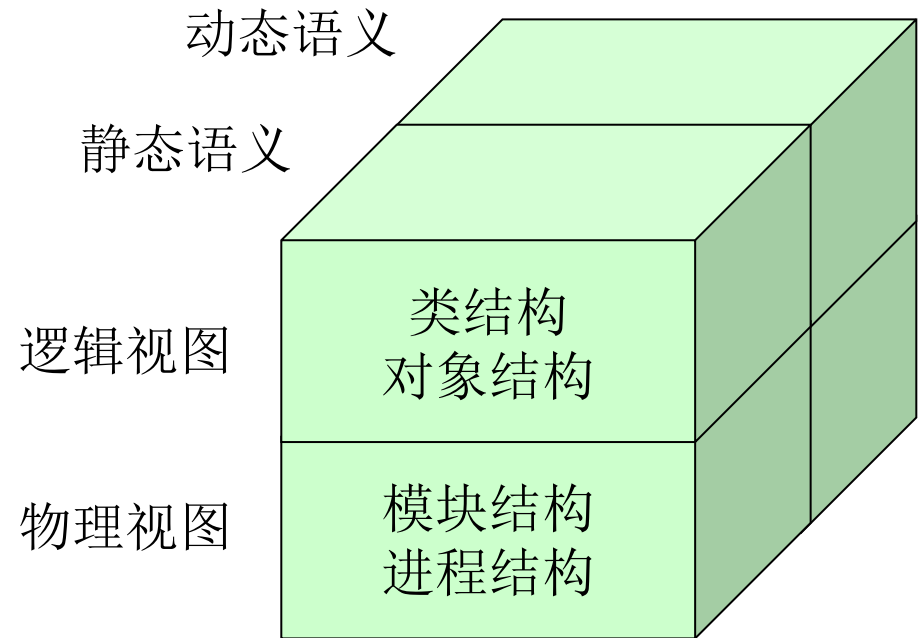
- 为提高继承而调整类体系：是为提高继承而调整和重新安排类和操作，从多组类中把共同行为抽取出来。
- 设计关联的实现。分析关联的遍历，使用对象来实现关联或者对关联中的 1、2 个类增加值对象的属性。
- 确定对象属性的明确表示：是将类、关联封装成模块。
- 最后得到：对象设计文档=细化的对象模型+细化的动态模型+细化的功能模型。

## 7.3 Booch方法

### ■ Booch方法的一般过程:

- ① 在一定的抽象层次上标识类和对象
- ② 标识类和对象的语义
- ③ 标识类和对象间的关系
- ④ 实现类和对象

■ Booch认为软件开发是一个螺旋上升过程，他强调过程的多次重复。



Booch方法的开发模型图



## 7.4 OOSE方法

- OOSE（Object-Oriented Software Engineering）是Jacobson提出的一种用例（use case）驱动的OO方法。







# OOSE的OOA过程

- ① 标识对象的用户和他们的责任
- ② 建造需求模型
  - 定义行为者（**actor**）及其责任
  - 为每个行为者标识用例
  - 制定系统对象及其关系的初始视图
  - 利用用例去复审模型以确定其有效性
- ③ 建造分析模型
  - 用行为者交互的信息来标识界面对象
  - 创建界面对象的结构视图
  - 表示对象的行为
  - 分离出子系统和模型
  - 利用用例作为场景去复审模型 以确定其合法性



# OOSE的OOD过程

- ① 修改理想化的分析模型以适合现实世界环境
- ② 创建块作为主要的设计对象
- ③ 创建一个显示激励如何在块间传送的交互图
- ④ 把块组织成子系统
- ⑤ 复审设计工作



# 课堂讨论：

- (1) 使用自己的话和例子，定义术语“类”、“封装”、“继承”和“多态”。
- (2) 设计者如何识别必须并发的任务？
- (3) 论述面向对象方法对传统方法的优势和问题。
- (4) 如何从需求文档中识别类和对象？