

第9章 面向对象设计

- 面向对象设计概述
- 精化类及类间关系
- 数据设计
- 设计模式简介
- 面向对象的测试

面向对象设计概述

面向对象分析（**OOA**）建立描述问题域的功能模型、静态模型和动态模型，刻画了“系统做什么”的问题。通过建立静态模型的**5**层结构来分解问题空间、抽象出类-对象，并分析类间关联、泛化、依赖和实现关系，建立问题域模型。

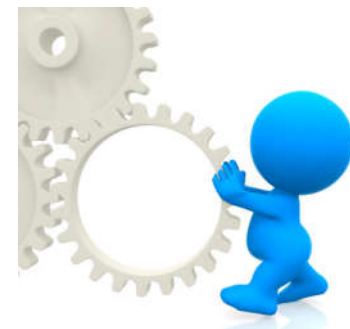
面向对象设计（**Object-Oriented Design, OOD**）是把**OOA**阶段得到的需求转换为符合用户功能、性能、领域等需求的设计方案。

面向对象设计概述

面向对象分析与设计的关系

OOD的特点主要体现在以下几个方面：

- (1) 与**OOA**和**OOP**共同构成面向对象开发的整个过程链，全面体现面向对象思想及特点。
- (2) **OO**强调**对象结构**而不是程序结构，增加了**信息共享**的机制，提高了信息共享的程度。
- (3) **OOD**的设计过程要与**OOP**所选用的编程语言相结合，因为**不同的面向对象编程语言对面向对象机制的支持程度不尽相同**。
- (4) 因为**OOA**和**OOD**的过程都使用**UML**语言来描述，因而**各阶段间的转换不需要任何映射方法和转换步骤**，更有利于各阶段间转换和分析结果的复用。



面向对象设计概述

面向对象分析与设计的原则

(1) 信息隐藏与模块化

- 类的属性，在类的内部被类的方法所共享，在类的外部被隐藏。
- 类的方法：统一实现对类的外部操作，并隐藏实现细节。
- 模块化的体现：类作为一个不可分割的整体，在系统中提供服务。



面向对象设计概述

面向对象分析与设计的原则

(2) 单一原则 (Simple Responsibility Principle, SRP原则)

一个接口、函数、类、界面等单元模块仅实现与它自身相关的功能，只包含实现功能所需的属性（数据结构）。

```
#include <string>
using namespace std;
// Define basic information associated with DB
class User
{
private:
    int ID;           // 系统识别号
    string Name;      // 登录名
    string Password;  // 密码

    // DB's statements with SQL
    string SQL;
```

数据操作

```
public:
    string GetName()           { return Name; }
    string GetPassword()       { return Password; }
    void SetName(string n)     { Name = n; }
    void SetPassword(string pwd) { Password = pwd; }

    int Insert(const User& user);
    int Delete(int ID);
    User Find(int ID);
};
```

数据库操作

面向对象设计概述

面向对象分析与设计的原则

(2) 单一原则 (Simple Responsibility Principle, SRP原则)

```
#include <string>
using namespace std;
// Define basic information associated with DB
class User
{
private:
    int ID;           // 系统识别号
    string Name;      // 登录名
    string Password;  // 密码
public:
    string GetName()      { return Name; }
    string GetPassword()  { return Password; }
    void SetName(string n) { Name = n; }
    void SetPassword(string pwd) { Password = pwd; }
};
```

```
// Get user's information in
// User DB by Data Access Object
class UserDAO
{
private:
    // Create SQL and excute
    // DB's statements with SQL
    string SQL;
public:
    UserDAO();
    int Insert(const User& user);
    bool Delete(int ID);
    User Find(int ID);
};
```

单一职责：数据操作与访问数据库的接口操作相分离。在新增类的同时，也增加了类间关系。

面向对象设计概述

面向对象分析与设计的原则

(3) 开放封闭原则 (Open Closed Principle, OCP原则)

开放：对系统功能扩展的完善性设计是开放的。

封闭：对系统内部代码、结构的修改是封闭的。

抽象是体现开放封闭原则的关键。



如何在不改动源码的情况下改变模块行为？

面向对象设计概述

面向对象分析与设计的原则

(3) 开放封闭原则 (Open Closed Principle, OCP原则)

例如：新增数据库类型 (1)

```
bool createDBConnect(const DBType& dbType)
{
    ... ..
    switch (dbType)
    {
        case ORACLE:
            Create Connection with Oracle;
            break;
        case SQLSERVER:
            Create Connection with SQLServer;
            break;
    }
    ... ..
}
```

修改代码:

```
case MySQL:
    Create Connection with MySQL;
    break;
```

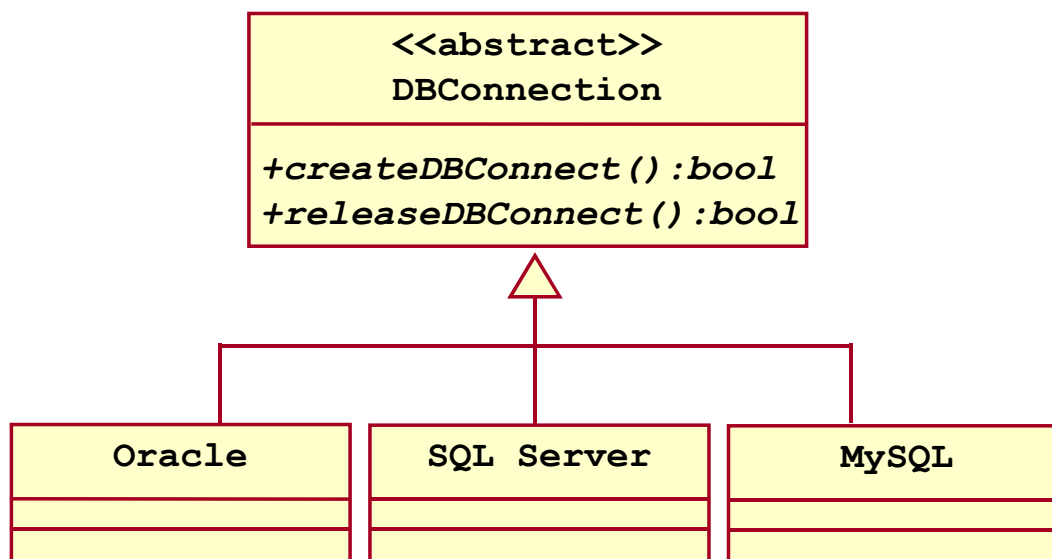


面向对象设计概述

面向对象分析与设计的原则

(3) 开放封闭原则 (Open Closed Principle, OCP原则)

例如：新增数据库类型 (2)：符合开放封闭原则



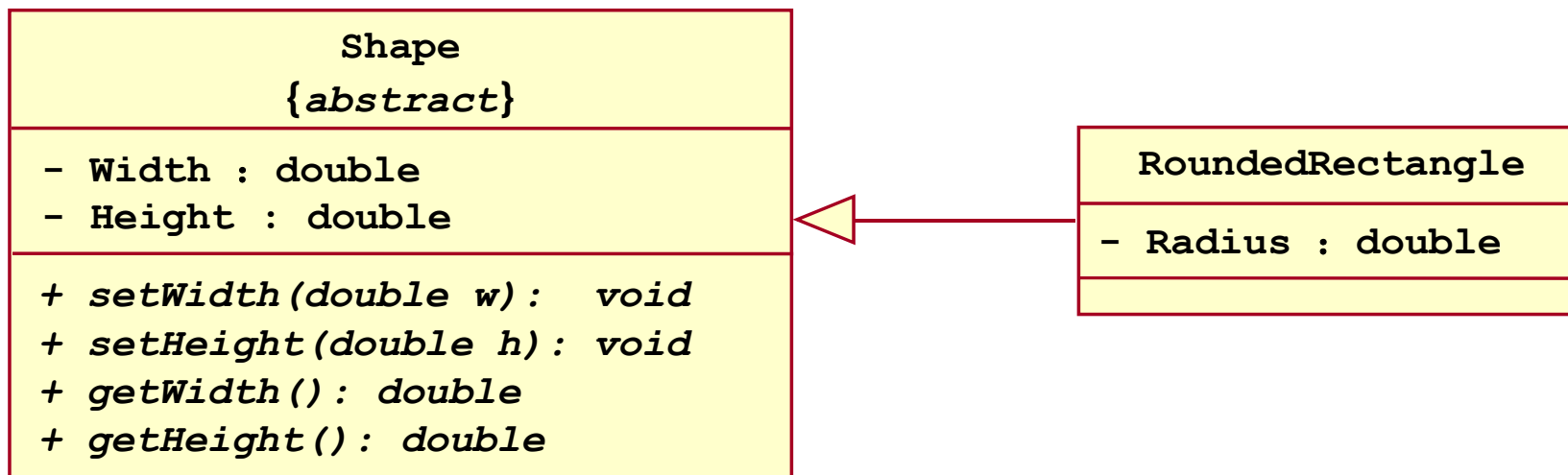
面向对象设计概述

面向对象分析与设计的原则

(4) 替换原则 (Liskov Substitution Principle)

在面向对象技术中，用子类可以替代父类出现的任何位置，而系统不会出现问题。

例：圆角矩形可以从图形继承吗？



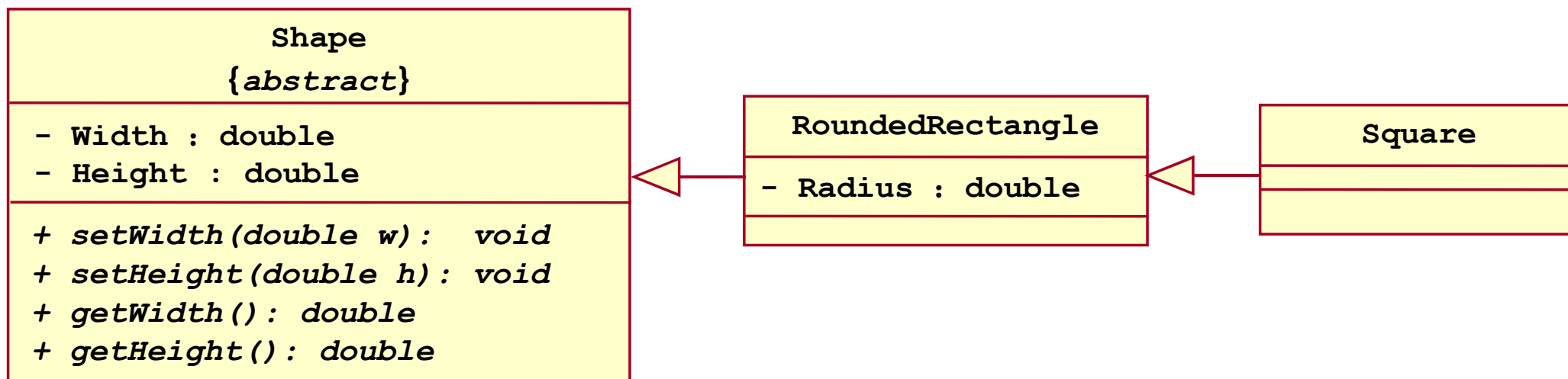
面向对象设计概述

面向对象分析与设计的原则

(4) 替换原则 (Liskov Substitution Principle)

在面向对象技术中，用子类可以替代父类出现的任何位置，而系统不会出现问题。

例：正方形可以从圆角矩形中继承吗？



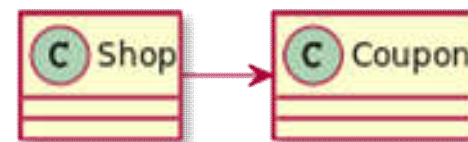
面向对象设计概述

面向对象分析与设计的原则

(5) 依赖倒置原则 (Dependence Inversion Principle)

- 系统高层接口不依赖于底层模块，抽象不依赖于实现细节。
- 一般情况下，系统在最初创建依赖关系时，采取的策略是直接依赖底层细节的实现。这样的方式实现快速、简单；但这一策略会因底层实现细节的改变而受到影响。

假定最初的外卖平台如下：当用户因为收不到所订的外卖，平台商户为了商铺的信誉，会采取赠送代金券的形式补偿用户。这个需求可以这样来设计：



需求变更：外卖平台为了吸引更多的用户到平台订餐，会不定时发放通用红包，用于平台的不同商铺。

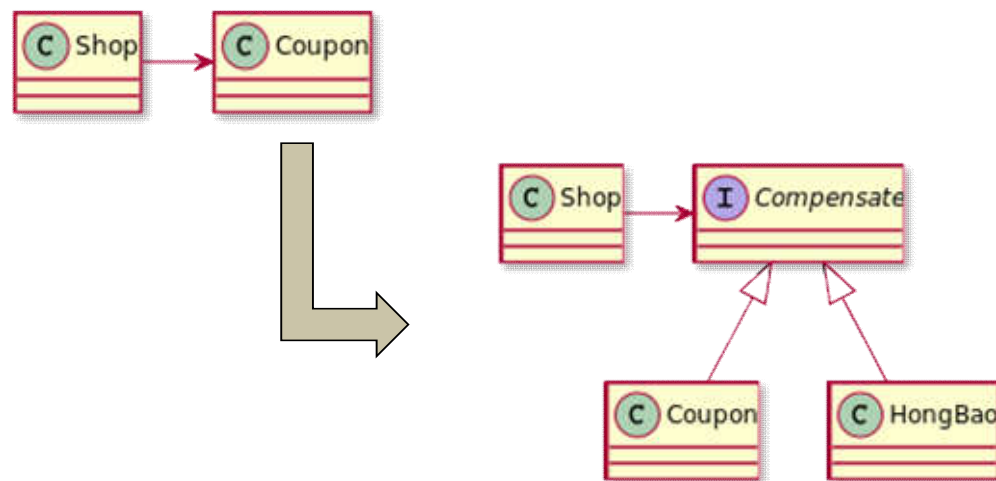
面向对象设计概述

面向对象分析与设计的原则

(5) 依赖倒置原则 (Dependence Inversion Principle)

- 系统高层接口不依赖于底层模块，抽象不依赖于实现细节。
- 一般情况下，系统在最初创建依赖关系时，采取的策略是直接依赖底层细节的实现。这样的方式实现快速、简单；但这一策略会因底层实现细节的改变而受到影响。

采用 DIP 的解决方案：



面向对象设计概述

面向对象分析与设计的原则

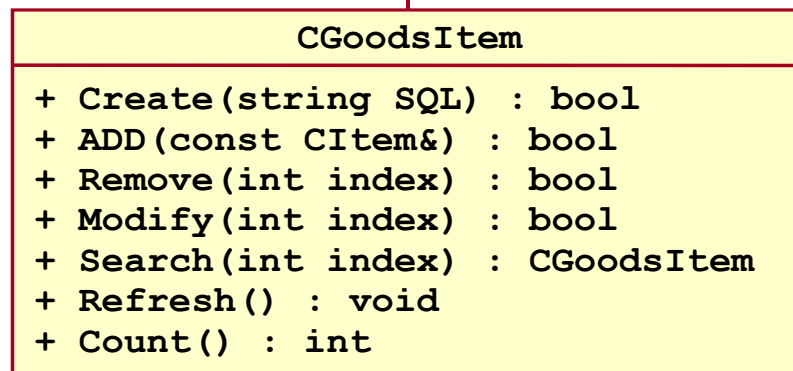
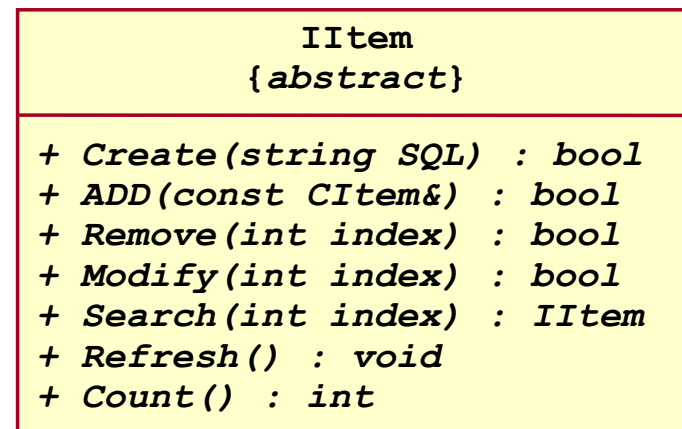
(6) ISP原则 (Interface Segregation Principle)

一个对象应该对其他对象有最少的了解，一个类对自己依赖的类知道得越少越好。

当系统运行一段时间后，由于数据库访问压力增大，需要增加在缓存。因此希望在查询数据时，先查缓存再查询数据库。**问题：**该如何修改设计方案？

方法一：修改Search函数的接口。

```
+ Search(int index, bool isCache) : IItem
```

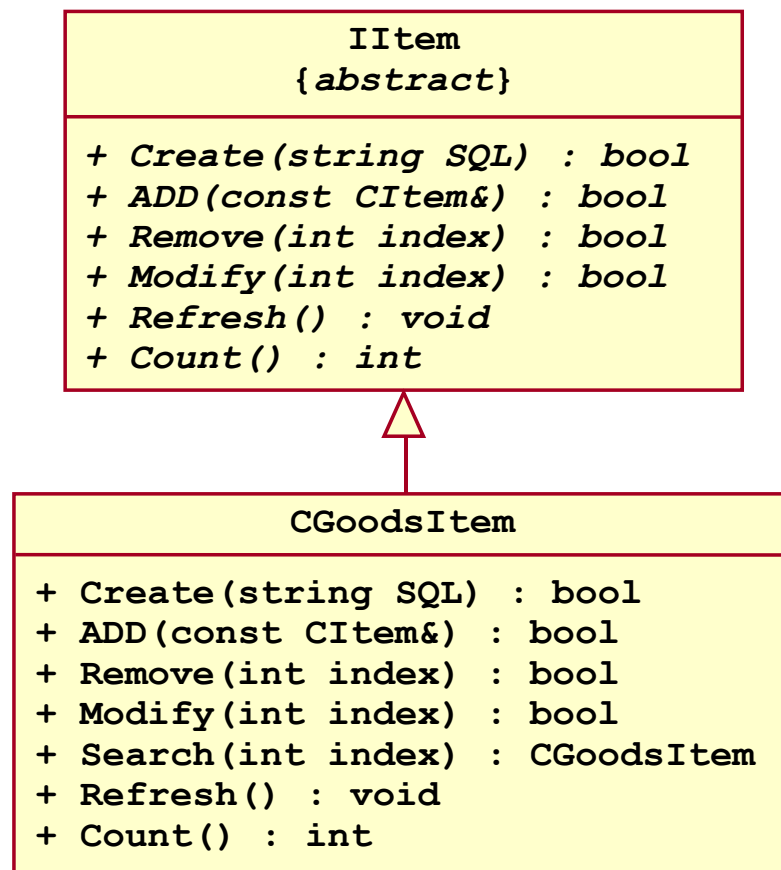
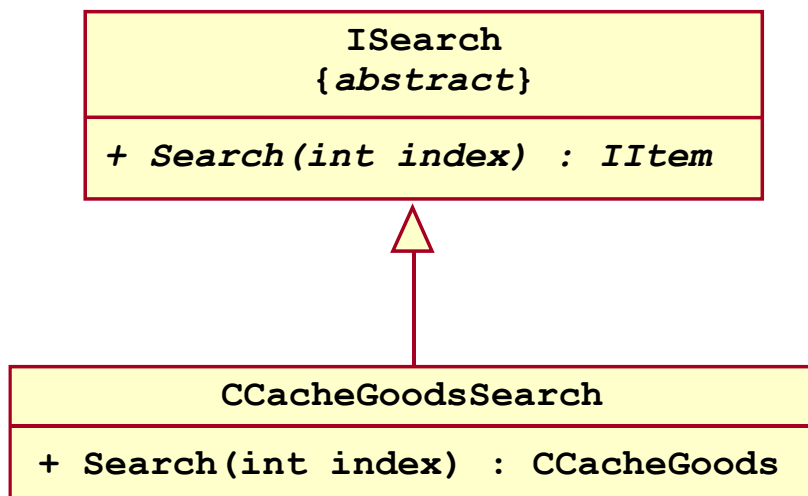


面向对象设计概述

面向对象分析与设计的原则

(6) ISP原则 (Interface Segregation Principle)

方法二：拆分IItem接口类。



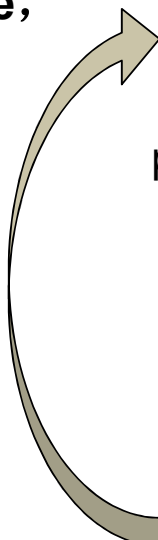
面向对象设计概述

面向对象分析与设计的原则

(7) 基米特法则 (Law of Demeter)

也被称为最小知识原则 (Least Knowledge Principle, **LKP**)，是指一个对象对其它对象尽可能少的理解。

```
class CStash
{
private:
    int size;      // Size of each space
    int quantity; // Number of storage spaces
    int next;      // Next empty space
    // Dynamically allocated array of bytes
    unsigned char* storage;
public:
    void initialize(int size);
    void cleanup();
    int add(const void* element);
    void* fetch(int index);
    int count();
    void inflate(int increase);
    void shrink();
};
```

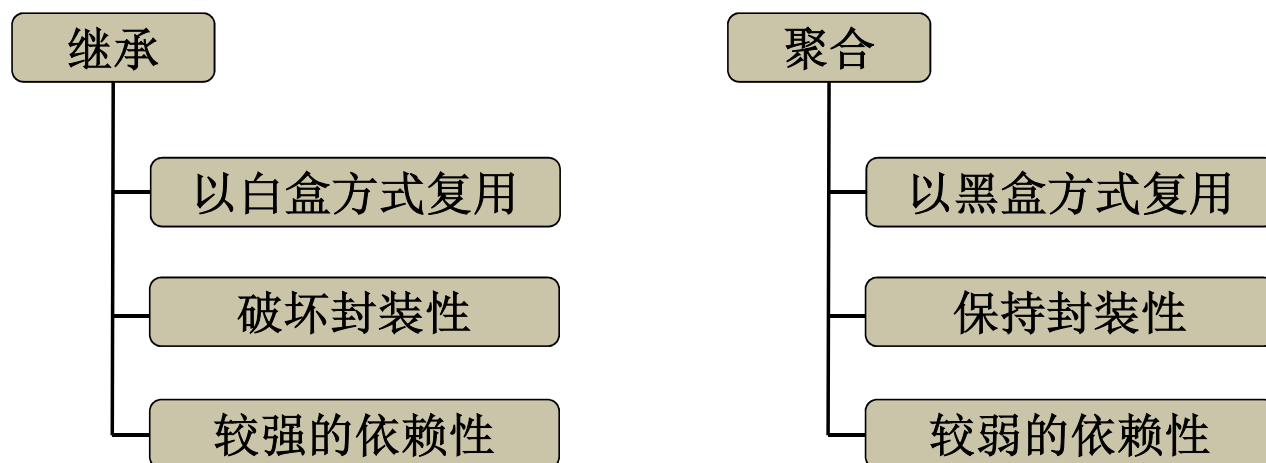


面向对象设计概述

面向对象分析与设计的原则

(8) 聚合优先原则 (Prefer Component to Inheritance Principle)

在类的重用过程中，继承与聚合是两类常用的技术。



精化类及类间关系

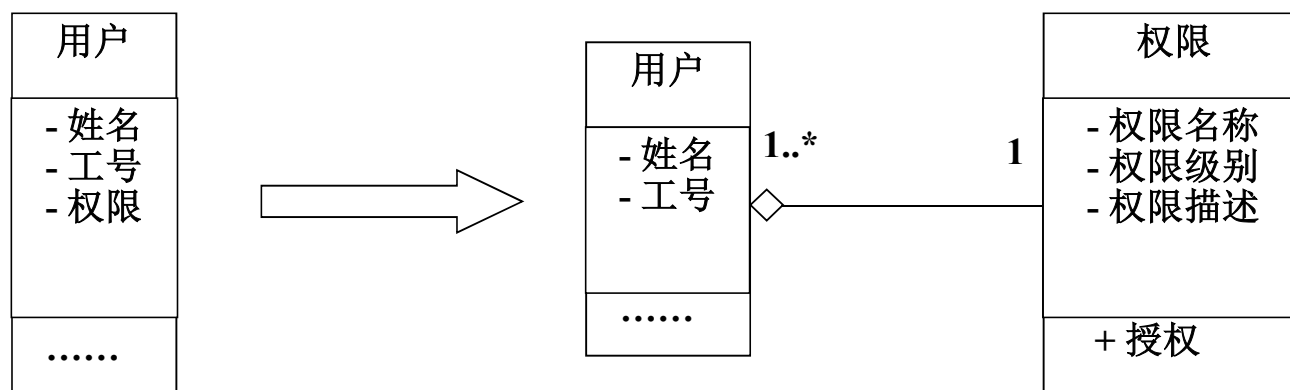
- 设计类的属性
- 设计类的方法
- 类间泛化 / 聚合关系
- 类间关联关系

精化类及类间关系

1. 设计类的属性

在**OOD**过程中，对类的属性设计需要补充和完善下面的相关工作。

(1) 复杂属性的分离和描述。



精化类及类间关系

1. 设计类的属性

在**OOD**过程中，对类的属性设计需要补充和完善下面的相关工作。

(2) 类间重数的属性表示。

- 类中定义指针，它指向另一个关联类的对象列表。这样，通过指针访问多个对象，实现一对多或多对一的关联。多对多的关联通过相互定义关联类的对象指针来实现。
- 如果编程语言不支持指针，通过定义关联类的对象数组来实现。由于一对多的映射是动态变化，因而还需要对对象数组进行约束，以形成对属性的约束。

精化类及类间关系

1. 设计类的属性

在**OOD**过程中，对类的属性设计需要补充和完善下面的相关工作。

- (3) **对属性的约束**。类的封装性约束了类的外部对属性和方法的存取权限。
- (4) **对属性的初始化**。属性的初始化设计，确保了对象在启动时处于正常初始状态。
- (5) **导出新“属性”**。注意，“属性”用引号括起来，是因为这里“属性”并不是类真正定义的属性实体，而是通过方法计算出的具有属性特征的结果。

精化类及类间关系

2. 设计类的方法

在**OOA**过程中，主要明确类所提供的方法和分析类间关系；而在**OOD**过程中，需要细化类的方法，并希望通过类方法的识别，体现类间的动态连接。

- (1) 具有公共服务性质的方法，应该放在继承结构的高层类中，以使得方法重用达到最大化。
- (2) 尽量在已有类中定义新方法，或重用已有代码——尽量避免引入新类，破坏原有类结构（系统结构）。
- (3) 反映类间的动态关系，即类间的每个消息都要有相应的操作。

精化类及类间关系

3. 设计类间泛化 / 聚合关系

- 类的泛化关系分为单继承和多继承两种形式。
- 在单继承的设计中，可以比较聚合方式与单继承对类的组织结构的利弊。
- 在多继承的设计中，由于多继承带来的二义性，需要考虑将其进行转换。

精化类及类间关系

3. 设计类间泛化 / 聚合关系

●单继承与聚合

类间关系是定义为泛化关系还是聚合关系？

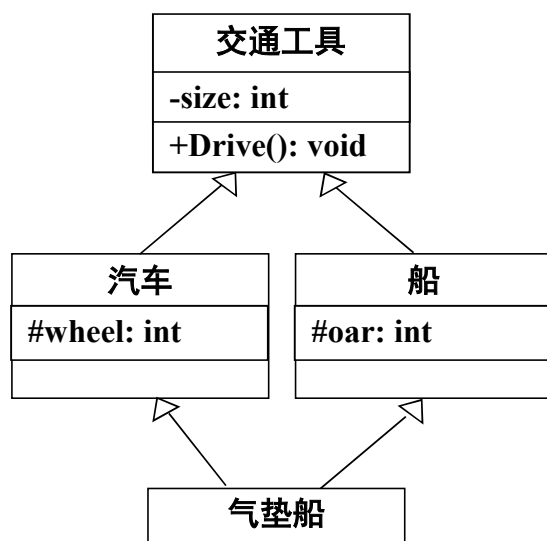
- **从泛化关系看**，派生类将直接得到基类的属性和方法，并根据基类虚函数定义和多态性来修改虚函数的局部内容，以适应派生类的特殊需求。受保护类型（**protected**）提供了派生类内部访问基类的属性和方法，同时又不破坏类的封装性，便于派生类和基类消息间的消息传递。**泛化的不足在于对基类的任何修改，都将影响到派生类。**
- **聚合关系**在一定程度上也与继承类似，通过在类（例如类**A**）中定义另一个类的子对象（类**B**的子对象**Obj**）来访问该类。这样，在类**A**的外部，看不到子对象**Obj**的存在，但在类**A**中的方法中，能通过子对象**Obj**访问类**B**的公有部分，达到扩展类**A**功能的目的。此外，如果对类**B**进行修改，在接口不变的情况下，将不会影响到类**A**的设计和实现。**因而在一般情况下，如果只是使用类**B**提供的方法，聚合要比继承方式好。**

精化类及类间关系

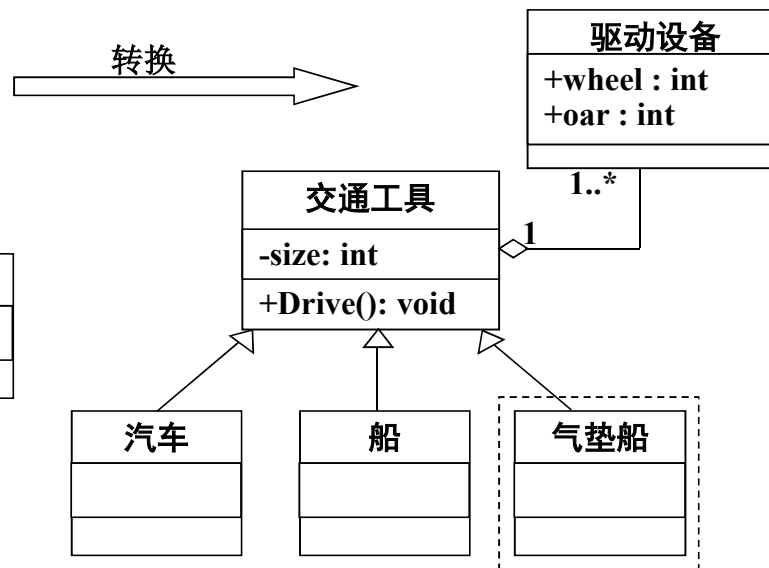
3. 设计类间泛化 / 聚合关系

● 多继承与转换

(1) 将多继承转换为单继承



图a



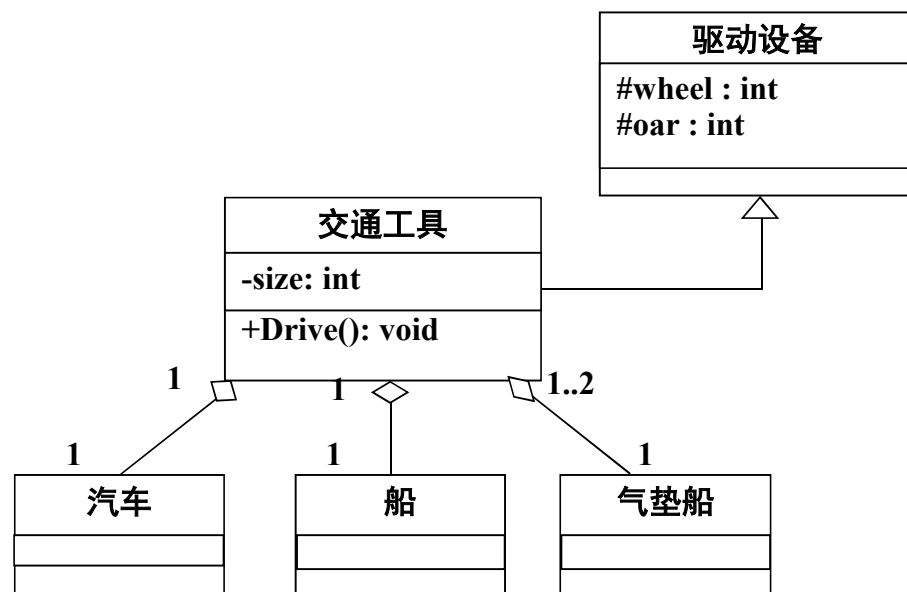
图b

精化类及类间关系

3. 设计类间泛化 / 聚合关系

● 多继承与转换

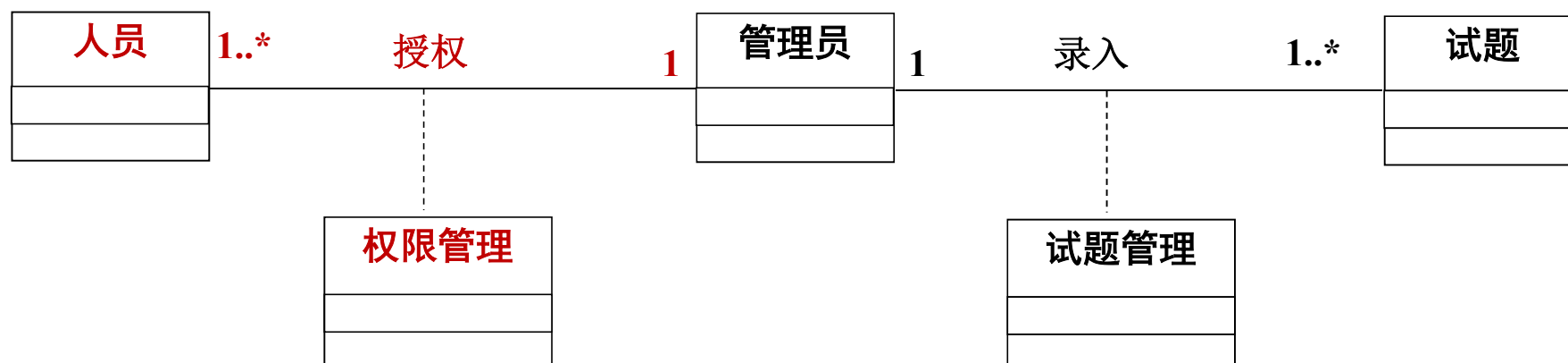
(2) 将多继承转换为聚合方式



精化类及类间关系

4. 设计关联类

- 属性设计中分析了多对多、一对多关系的设计方式。
- 对于多对多关系的转换，还能通过定义关联类来实现。



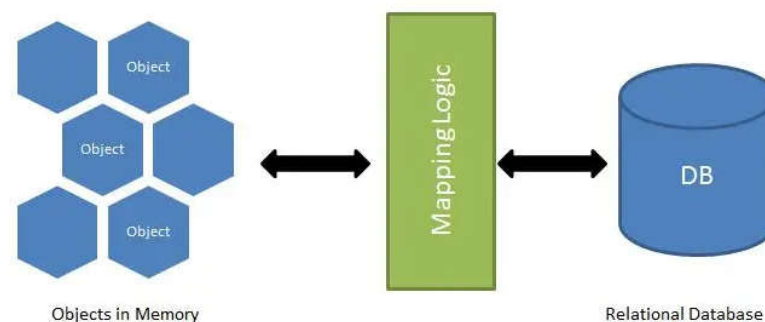
数据设计

- 数据设计是**OOD**模型中的主要部分之一，负责对永久对象（**Persistent Object**）的读取、存储等过程进行管理。
 - 类到关系数据库的映射——概念模型
 - 类到关系数据库的映射——重数
 - 类间关系的映射——泛化关系

数据设计

类到关系数据库的映射

基于关系数据库的设计，就是将类图作为关系数据库的**概念模型**，并兼顾类间的关联关系和泛化关系在数据库中的表示。



对UML的类图，通常只考虑转换类中的属性而不考虑类的方法。因为对关系数据库中表（属性集）的操作，通过关系数据库系统的接口，或在系统中提供统一的方法对数据进行操作，在这些方法中就包括了原有类中的方法。

数据设计

类到关系数据库的映射

对UML的类图，通常只考虑转换类中的属性而不考虑类的方法。因为对关系数据库中表（属性集）的操作，通过关系数据库系统的接口，或在系统中提供统一的方法对数据进行操作，在这些方法中就包括了原有类中的方法。

```
#include <string>
using namespace std;
// Define basic information associated with DB
class User
{
private:
    int ID;           // 系统识别号
    string Name;      // 登录名
    string Password;  // 密码
public:
    string GetName()      { return Name; }
    string GetPassword()  { return Password; }
    void SetName(string n) { Name = n; }
    void SetPassword(string pwd) { Password = pwd; }
};
```

```
// Get user's information in
// User DB by Data Access Object
class UserDAO
{
private:
    // Create SQL and excute
    // DB's statements with SQL
    string SQL;
public:
    UserDAO();
    int Insert(User user);
    bool Delete(int ID);
    int Find(int ID);
};
```

数据设计

类到关系数据库的映射

在将持久对象转换为关系数据时，类和对象与关系数据库的表之间有如下的基本对应关系。

OOD	关系数据库	描述
类	表	类中关于属性的定义，就是关系数据库中表的结构。
对象	行	对象是类的实例，即对类的属性有具体的值，对应表中的行。
属性	列	类中的一个属性，对应关系数据库中表的一列。
关系	表间连接	通过关系数据库中表间连接来设计类间关系。

数据设计

类到关系数据库的映射

类间关系在关系数据库中的表示主要涉及**关联关系**和**泛化关系**。

类间关联关系的数据设计主要涉及类间重数的描述。

类间重数的关联包括：

- (1) 一对一的关联
- (2) 一对多的关联
- (3) 多对多的关联

数据设计

类间关系的映射

由于基类和派生类之间的泛化关系，使得派生类具有基类的属性和方法。泛化关系的数据设计主要为：

- (1) 可以仅将派生类映射为表，而将基类中的属性直接定义在派生类的映射表中。
- (2) 对于基类和派生类各自定义对应的表，同时把基类的表中的主键定义为派生类表中的外键，以实现基类和派生类的泛化关系。