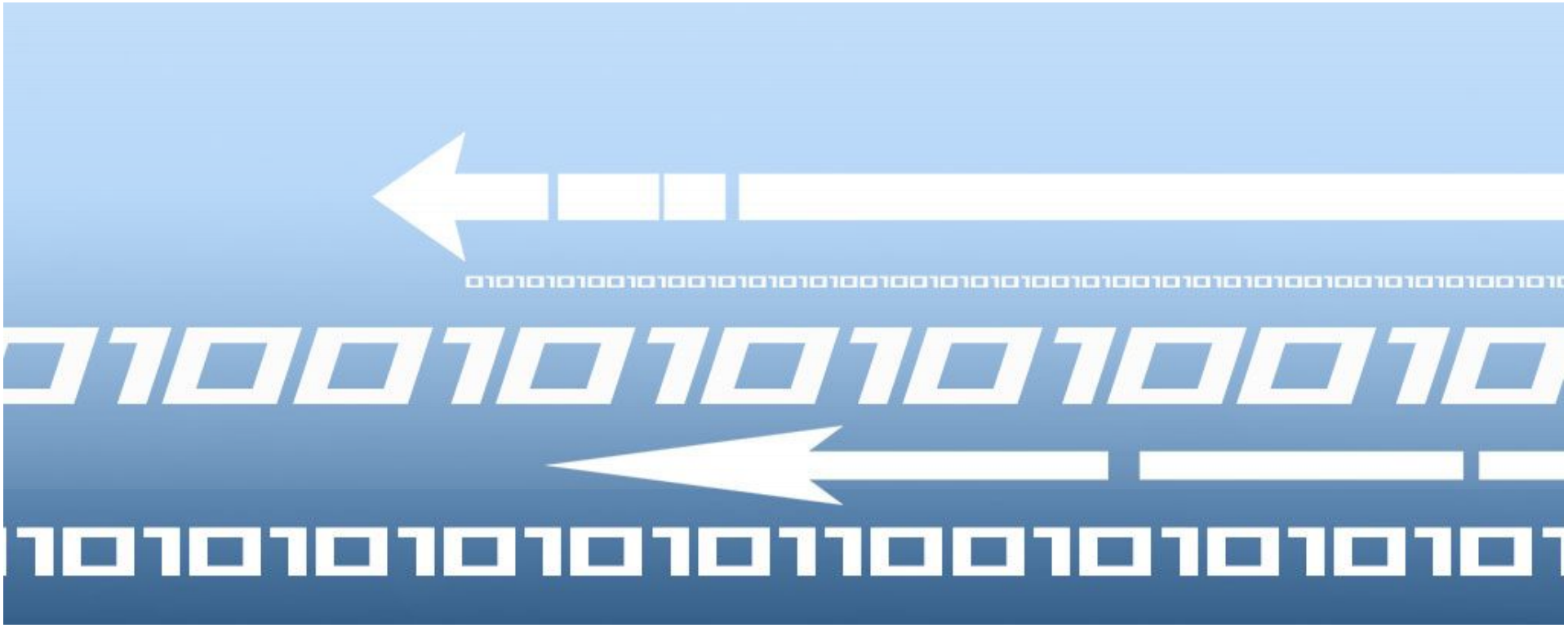


设计模式公开课

包子

2014-07

天群

1. 引言
2. 设计模式概述
3. 面向对象的设计原则
4. UML概述

弗谷

✓模式，我们并不陌生……

足球：4-4-2、4-3-3战术模式；

篮球：普林斯顿、三角进攻战术；

LOL：全球流、换线流、四一分推；

建筑：地中海风格、美国西海岸风格；

……

几乎所有的团队活动都讲究模式（战术）

弗谷


计 算 机 科 学 丛 书
设计模式
可复用面向对象软件的基础
Design Patterns
Elements of Reusable Object-Oriented Software
经 典 原 版 书 库
设 计 模 式
可复用面向对象软件的基础
（英文版）
Design Patterns
Elements of Reusable Object-Oriented Software
Design Patterns Explained
设计模式解析
（第2版）
CHINA-PUB.COM
软件工程研究院
Java
模
设计模式之禅
The Zen of Design Patterns
重视大脑的学习指南
Head First
设计模式
（中文版）
Design Patterns in Java
Java设计模式
C# 3.0 Design Patterns
C# 3.0
设计模式
O'REILLY
设计模式Java手册
Design Patterns Java workbook
Design Patterns in C#
C#设计模式
（影印版）

弗谷

✓从三个实例说起……

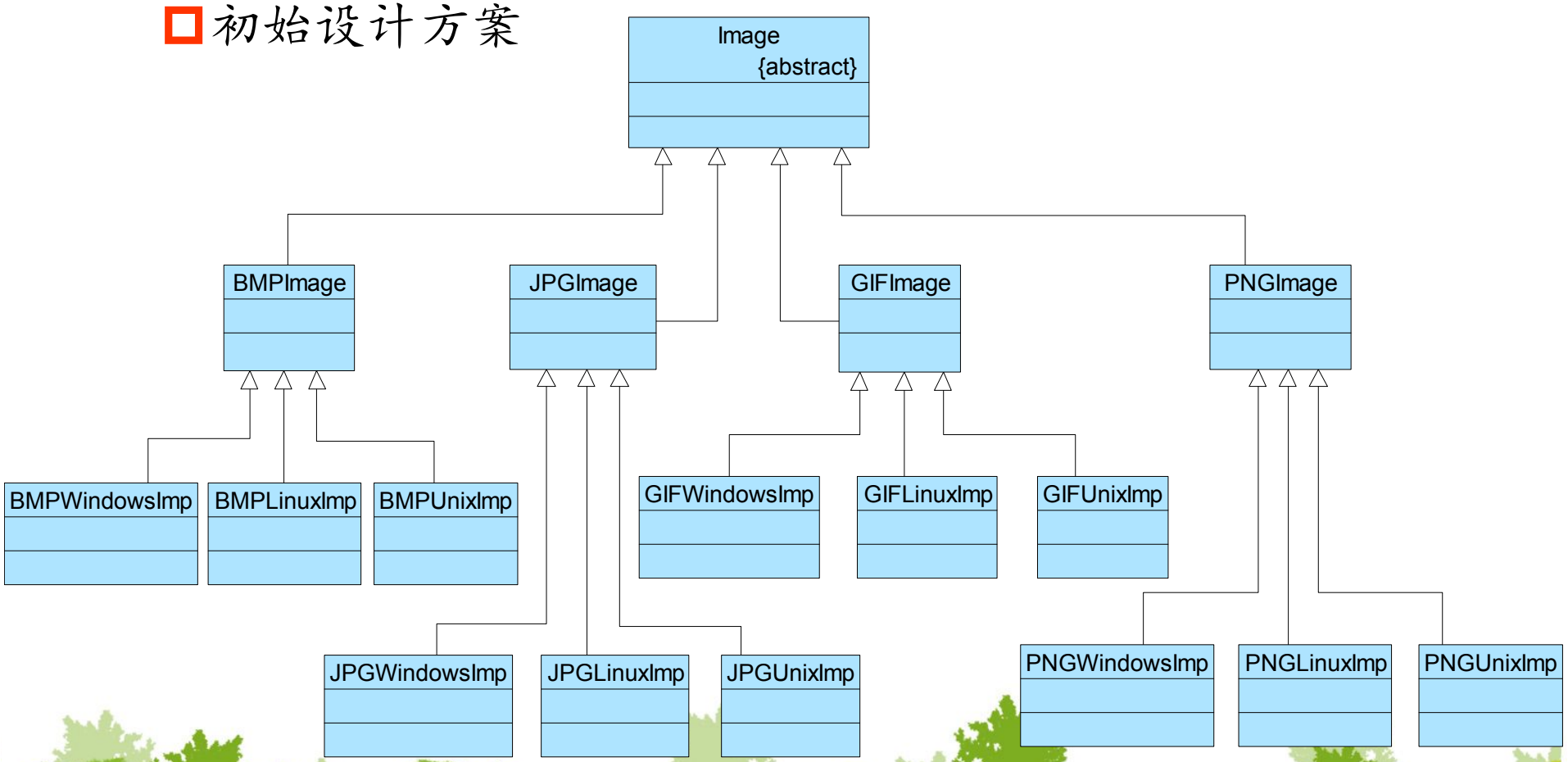
✓庞大的跨平台图像浏览系统

实例说明

某软件公司要开发一个跨平台图像浏览系统，要求该系统能够显示BMP、JPG、GIF、PNG等多种格式的文件，并且能够在Windows、Linux、Unix等多个操作系统上运行。系统首先将各种格式的文件解析为像素矩阵(Matrix)，然后将像素矩阵显示在屏幕上，在不同的操作系统中可以调用不同的绘制函数来绘制像素矩阵。

✓庞大的跨平台图像浏览系统

□初始设计方案



✓庞大的跨

- 问题

(1) 采用
加，具
所支持

(2) 系统
增加新
导致系

中类的个数急剧增
图像文件格式数×

图像文件格式还是
量的具体类，这将
和维护开销

✓不够灵活的影院售票系统

实例说明

某软件公司为某电影院开发了一套影院售票系统，在该系统中需要为不同类型的用户提供不同的电影票打折方式，具体打折方案如下：

- (1) 学生凭学生证可享受票价8折优惠；
- (2) 年龄在10周岁及以下的儿童可享受每张票减免10元的优惠（原始票价需大于等于20元）；
- (3) 影院VIP用户除享受票价半价优惠外还可进行积分，积分累计到一定额度可换取电影院赠送的奖品。

该系统在将来可能还要根据需要引入新的打折方式。

```
//电影票类
class MovieTicket
{
   private double price; //电影票价格
   private string type; //电影票类型
                  ……
   //计算打折之后的票价
   public double Calculate()
   {
      //学生票折后票价计算
      if(this.type.Equals("student"))
      {
         Console.WriteLine("学生票： ");
         return this.price * 0.8;
      }
      //儿童票折后票价计算
      else if(this.type.Equals("children") && this.price >= 20 )
      {
         Console.WriteLine("儿童票： ");
         return this.price - 10;
      }
       //VIP票折后票价计算
      else if(this.type.Equals("vip"))
      {
         Console.WriteLine("VIP票： ");
         Console.WriteLine("增加积分！ ");
         return this.price * 0.5;
      }
      else
      {
         return this.price; //如果不满足任何打折要求，则返回原始票价
      }
   }
}
```

弗谷

✓ 不够灵活

- 问题

(1) Movi 非常庞大，它包含各种打 中出现了较长的条件转移

(2) 在增 打折算法进行修改时必须 码，系统的灵活性和可扩

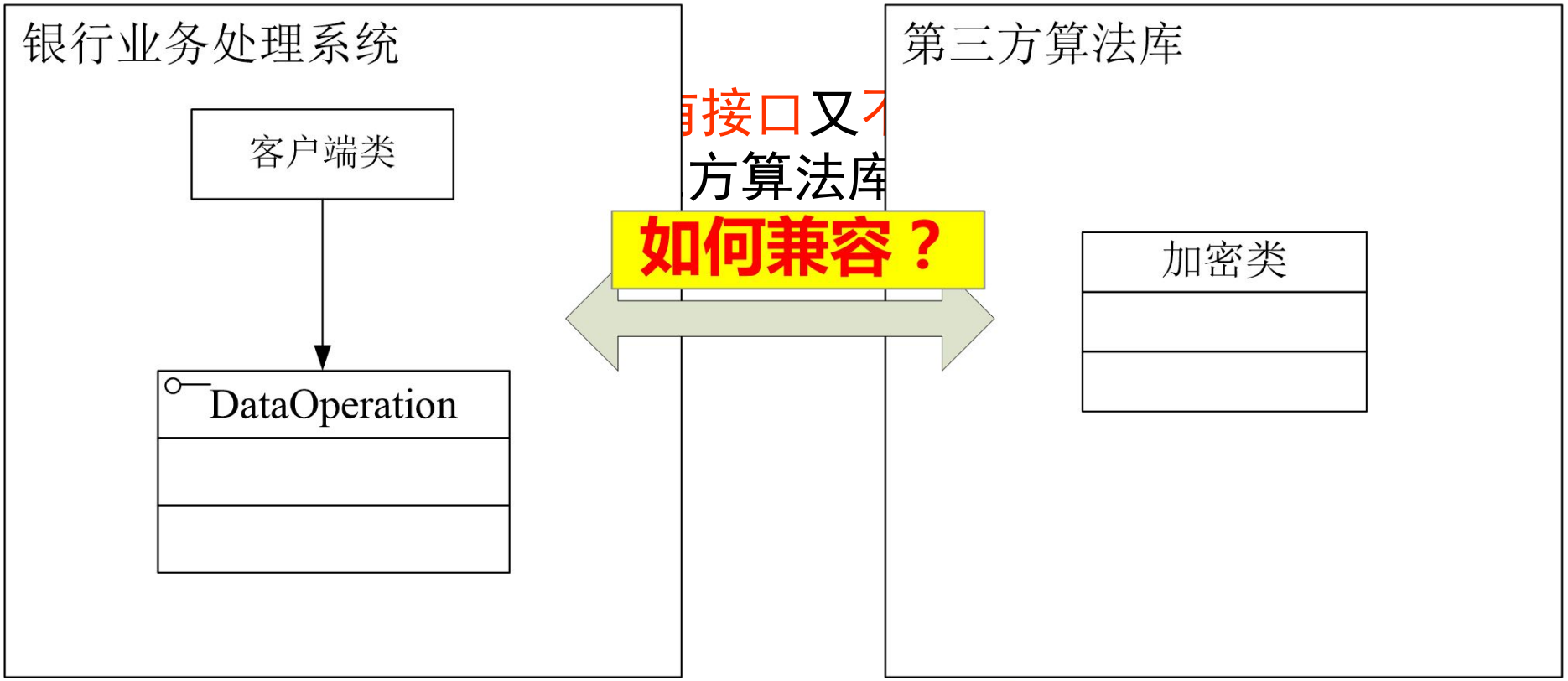
(3) 算法 统需要重用某些打折算法 制粘贴来重用，无法单独

✓重用第三方算法库时面临的问题

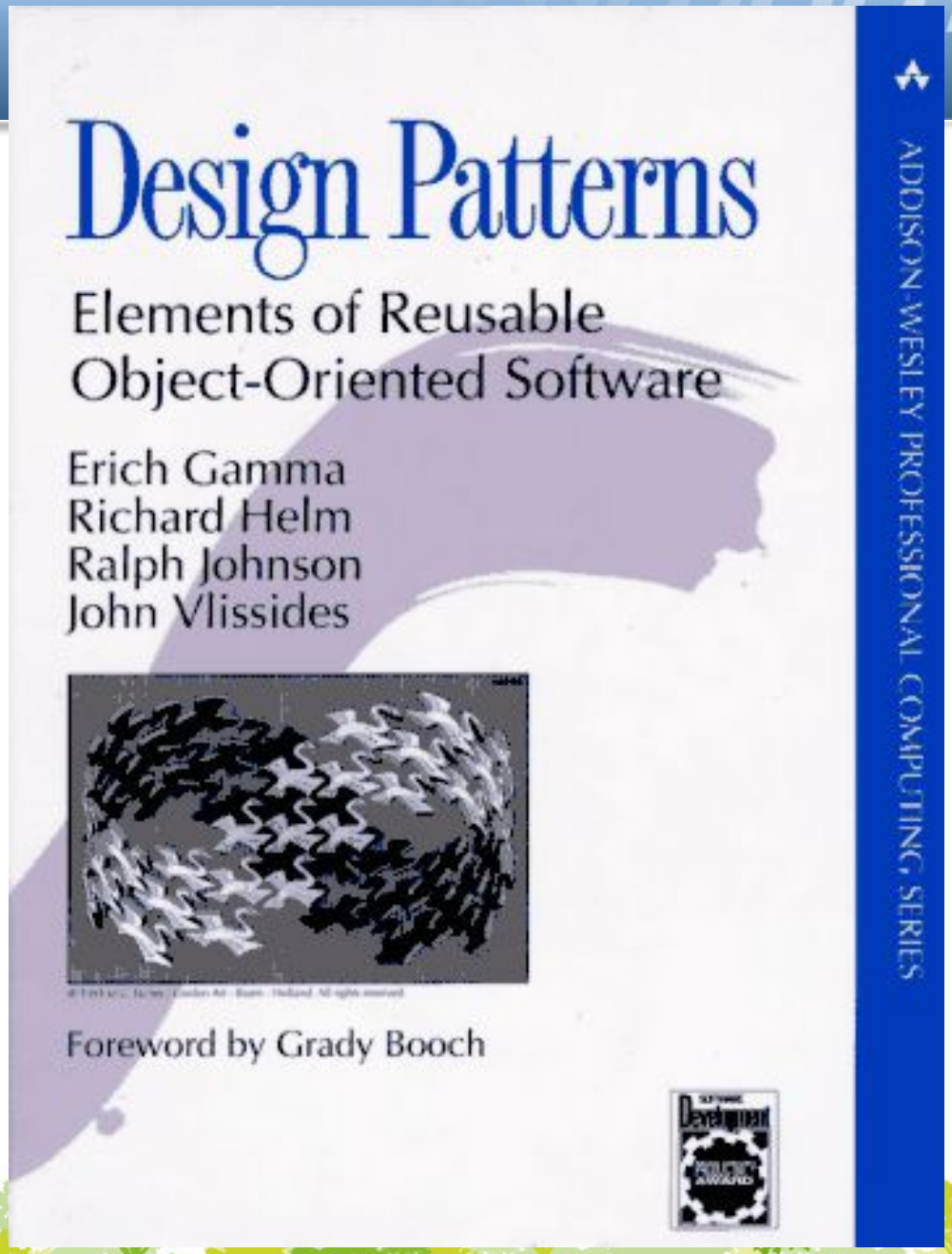
实例说明

某软件公司在开发一个银行业务处理系统时需要对其中的机密数据进行加密处理，通过分析发现，用于加密的程序已经存在于一个第三方算法库中，但是没有该算法库的源代码。在系统初始设计阶段，已定义数据操作接口DataOperation，且该接口已被很多同事使用，对该接口的修改势必导致大量代码需要产生改动。

✓重用第三方算法库时面临的问题



弗谷

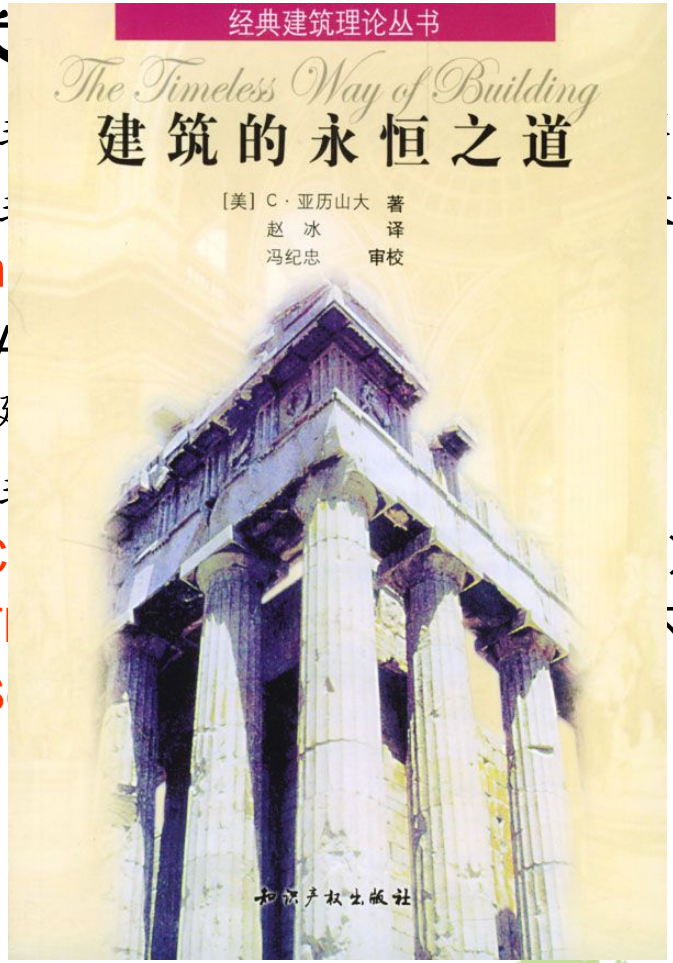

Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON WESLEY PROFESSIONAL COMPUTING SERIES

天群

1. 引言
2. 设计模式概述
3. 面向对象的设计原则
4. UML概述

贯豪毫弓碍赛眉专取属

- ✓ 模式
  - 模
  - 模
    Ch
  - 《A
    个
  - 模
    C
    T
    S



软件业

学环境结构中心研究所所长

Buildin

—253

)

要解决的

各种物理

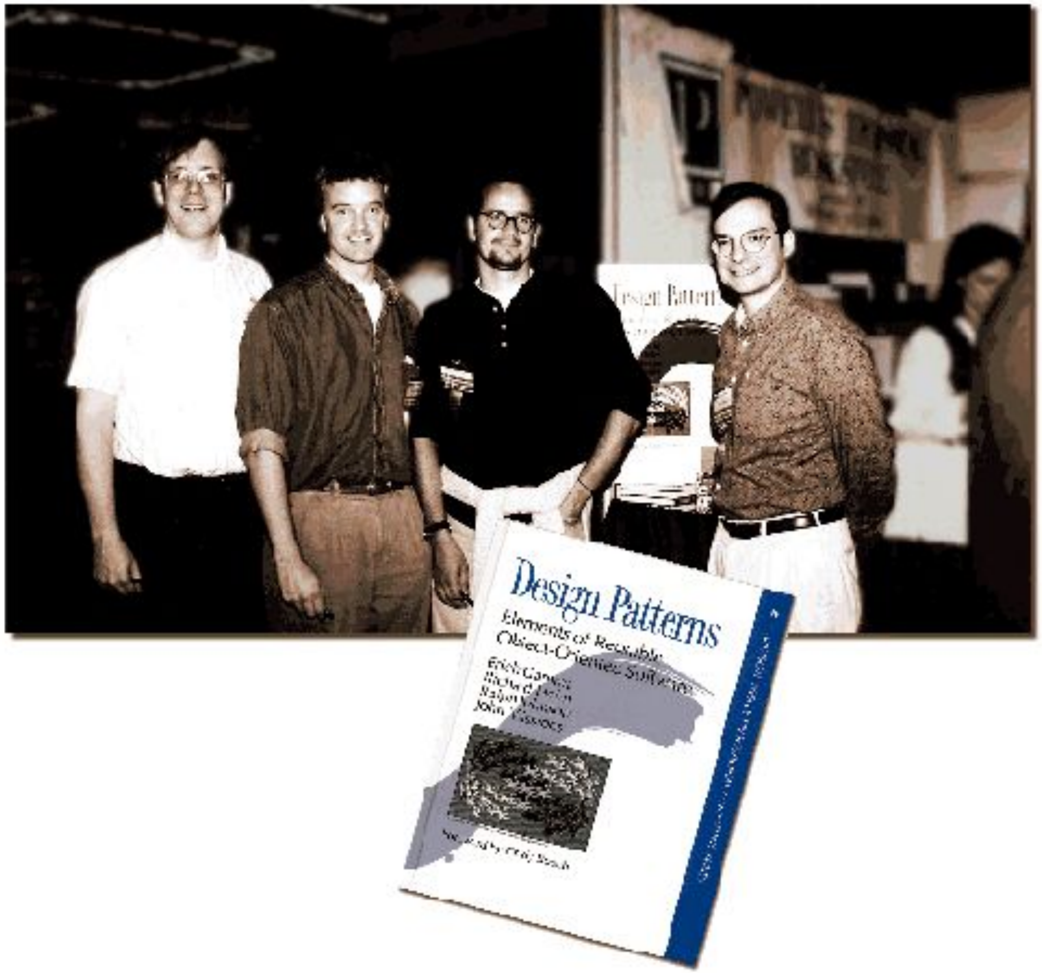
- 模式的诞生与定义
  - ✓ Alexander给出了关于模式的经典定义：
    - 每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心，通过这种方式，人们可以无数次地重用那些已有的解决方案，无须再重复相同的工作

模式是在特定环境下人们解决某类重复出现问题的一套成功或有效的解决方案。

A pattern is a successful or efficient solution to a recurring problem within a context.

贯豪毫弓碍赛眉专取属

- 软
  - ✓ hristopher
    市规划领域的重大
  - ✓ ich Gamma,
    Vlissides)"于
    用频率较高的设计
    法在分析、设计和



贯豪毫弓碍赛眉专取属

Gang of Four

Erich Gamma

苏黎世大学计算机科学博士，是Eclipse、JUnit 等项目的负责人

Richard Helm

墨尔本大学计算机科学博士，原IBM 研究员，现供职于波士顿顾问集团

Ralph Johnson

康奈尔大学计算机科学博士，伊利诺伊大学教授

John Vlissides

斯坦福大学计算机科学博士，原IBM研究员，于2005年11月24日因脑瘤去世，享年44岁

✓ 软件模式概述

- 软件模式：在一定条件下的软件开发问题及其解法

  问题描述

  前提条件（环境）

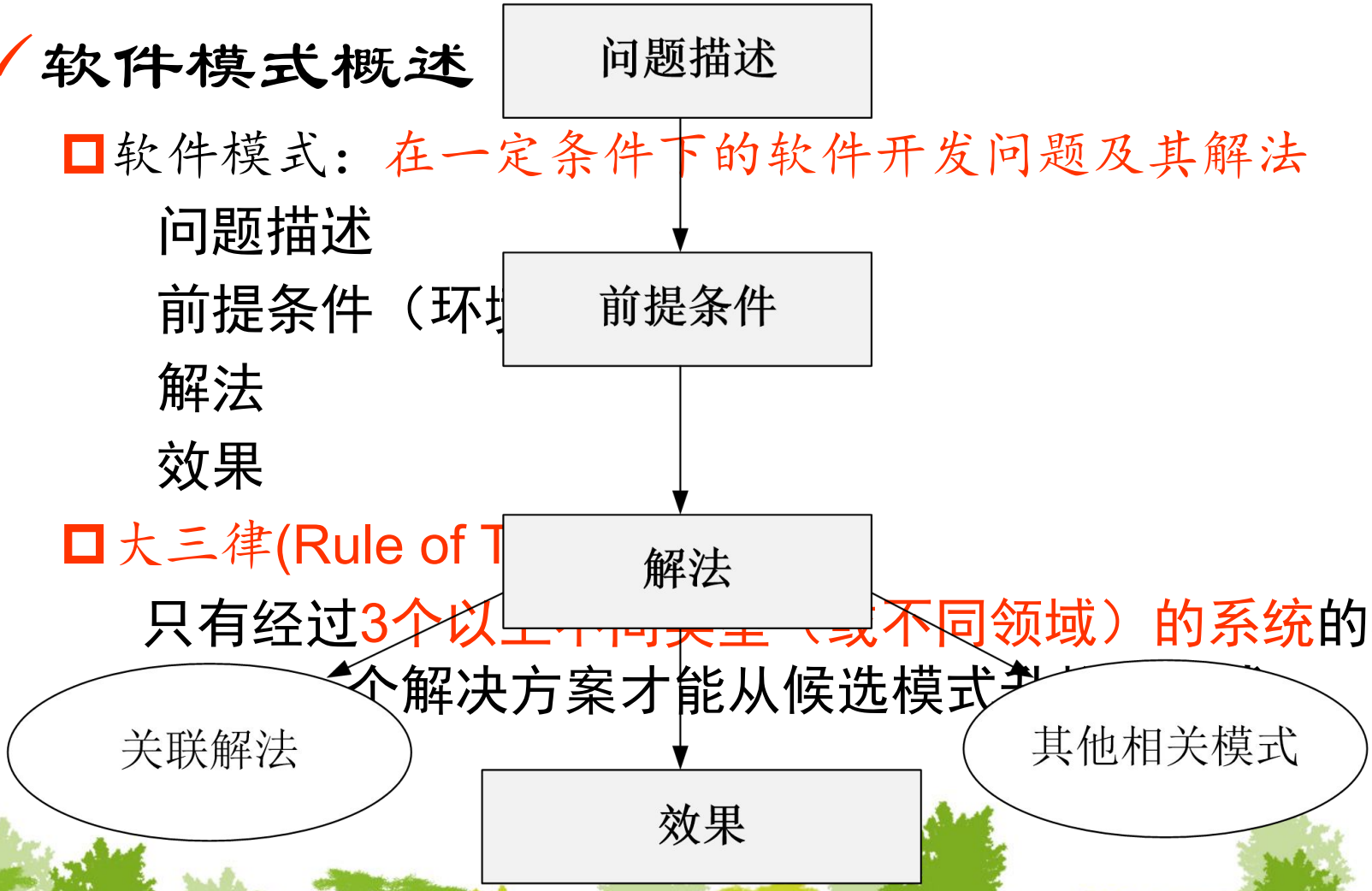
  解法

  效果

- 大三律(Rule of T

  只有经过3个以上不同类型（或不同领域）的系统的校

  个解决方案才能从候选模式



✓设计模式的基本要素

- 设计模式一般包含模式名称、问题、目的、解决方案、效果、实例代码和相关设计模式等基本要素，4个关键要素如下：
  - 模式名称 (Pattern Name)
  - 问题 (Problem)
  - 解决方案 (Solution)
  - 效果 (Consequences)

✓设计模式的分类

- 根据目的（模式是用来做什么的）可分为创建型(Creational)，结构型(Structural)和行为型(Behavioral)三类：
  - **创建型模式**主要用于创建对象
  - **结构型模式**主要用于处理类或对象的组合
  - **行为型模式**主要用于描述类或对象如何交互和怎样分配职责

✓设计模式的分类

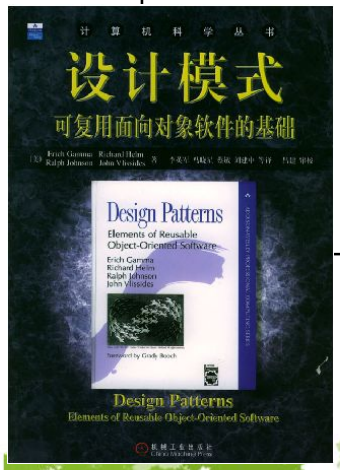
- 根据范围，即模式主要是处理类之间的关系还是处理对象之间的关系，可分为类模式和对象模式两种：

  类模式处理类和子类之间的关系，这些关系通过继承建立，在编译时刻就被确定下来，是一种静态关系

  对象模式处理对象间的关系，这些关系在运行时变化，更具动态性

GoF贯豪毫弓练仍

范围\目的	创建型模式	结构型模式	行为型模式
类模式	工厂方法模式	（类）适配器模式	解释器模式 模板方法模式
对象模式	抽象工厂模式 建造者模式 原型模式 单例模式	（对象）适配器模式 桥接模式 组合模式 装饰模式 外观模式 享元模式 代理模式	职责链模式 命令模式 迭代器模式 中介者模式 备忘录模式 观察者模式 状态模式 策略模式 访问者模式



GoF贯豪毫弓练仍

- 创建型模式
  - ✓ 抽象工厂模式(Abstract Factory) ★★★★★
  - ✓ 建造者模式(Builder) ★★☆☆☆
  - ✓ 工厂方法模式(Factory Method) ★★★★★
  - ✓ 原型模式(Prototype) ★★★☆☆
  - ✓ 单例模式(Singleton) ★★★★☆

GoF贯豪毫弓练仍

- 结构型模式
  - ✓ 适配器模式(Adapter) ★★★★☆
  - ✓ 桥接模式(Bridge) ★★★☆☆
  - ✓ 组合模式(Composite) ★★★★☆
  - ✓ 装饰模式(Decorator) ★★★☆☆
  - ✓ 外观模式(Facade) ★★★★★
  - ✓ 享元模式(Flyweight) ★☆☆☆☆
  - ✓ 代理模式(Proxy) ★★★★☆

GoF贯豪毫弓练仍

- ***行为型模式***
  - ✓ *职责链模式(Chain of Responsibility)* ★★☆☆☆
  - ✓ *命令模式(Command)* ★★★★☆
  - ✓ *解释器模式(Interpreter)* ★☆☆☆☆
  - ✓ *迭代器模式(Iterator)* ★★★★★
  - ✓ *中介者模式(Mediator)* ★★☆☆☆
  - ✓ *备忘录模式(Memento)* ★★☆☆☆
  - ✓ *观察者模式(Observer)* ★★★★★
  - ✓ *状态模式(State)* ★★★☆☆
  - ✓ *策略模式(Strategy)* ★★★★☆
  - ✓ *模板方法模式(Template Method)* ★★★☆☆
  - ✓ *访问者模式(Visitor)* ★☆☆☆☆

设计模式学习步骤

- 我们将按照以下次序来学习设计模式:
  - 模式动机与定义
  - 模式结构与分析
  - 模式实例与解析
  - 模式效果与应用
  - 模式扩展

贯豪毫弓碍伙玉

- ✓ 融合了众多专家的经验，并以一种标准的形式供广大开发人员所用
- ✓ 提供了一套通用的设计词汇和一种通用的语言，以方便开发人员之间进行沟通和交流，使得设计方案更加通俗易懂
- ✓ 让人们可以更加简单方便地复用成功的设计和体系结构
- ✓ 使得设计方案更加灵活，且易于修改
- ✓ 将提高软件系统的开发效率和软件质量，且在一定程度上节约设计成本
- ✓ 有助于初学者更深入地理解面向对象思想，方便阅读和学习现有类库与其他系统中的源代码，还可以提高软件的设计水平和代码质量

少聲

- ✓ 设计模式是一套被反复使用、多数人知晓、经过分类编目的设计经验总结，使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。
- ✓ 设计模式一般有如下几个基本要素：模式名称、问题、目的、解决方案、效果、实例代码和相关设计模式，其中的关键元素包括模式名称、问题、解决方案和效果。
- ✓ 设计模式根据其目的可分为创建型，结构型和行为型三种；根据范围可分为类模式和对象模式两种。
- ✓ 设计模式是从许多优秀的软件系统中总结出的成功的、能够实现可维护性复用的设计方案，使用这些方案将避免我们做一些重复性的工作，从而可以设计出高质量的软件系统。

少聲

- ✓ 模式是在特定环境中解决问题的一种方案。
- ✓ GoF (Erich Gamma, Richard Helm, Ralph Johnson和John Vlissides)最先将模式的概念引入软件工程领域，他们归纳发表了23种在软件开发中使用频率较高的设计模式，旨在用模式来统一沟通面向对象方法在分析、设计和实现间的鸿沟。
- ✓ 软件模式是将模式的一般概念应用于软件开发领域，即软件开发的总体指导思路或参照样板。软件模式可以认为是对软件开发这一特定“问题”的“解法”的某种统一表示，即软件模式等于一定条件下的出现的问题以及解法。

天群

1. 引言
2. 设计模式概述
3. 面向对象的设计原则
4. UML概述

面向对象设计原则

- 面向对象设计原则概述
- 单一职责原则 

- 开闭原则 

- 里氏代换原则 

- 依赖倒转原则 

- 接口隔离原则 

- 合成复用原则 

- 迪米特法则 

途价碍台胁拿懂专台夏看懂

- 软件系统的可维护性
  - ✓ 导致一个软件设计的可维护性较低，也就是说会随着性能、用户需求的变化而“腐烂”的真正原因有四个：
    - 过于僵硬：很抱歉，这个需求暂时无法实现。
    - 过于脆弱 ：怎么回事，上线后又出问题了！
    - 复用率低 ：你参照那个类写就可以了。
    - 黏度过高 ：明明修改的是A功能，结果B功能出问题了...

- 一个好的系统设计应该有如下性质：
  可扩展性
  灵活性
  可插入性

途价碍台胁拿懂咖台夏看懂

✓ 软件的可维护性和可复用性

- 软件的复用(Reuse)或重用拥有众多优点，如可以提高软件的开发效率，提高软件质量，节约开发成本，恰当的复用还可以改善系统的可维护性。
- 面向对象设计复用的目标在于实现支持可维护性的复用。
- 在面向对象的设计里面，可维护性复用都是以面向对象设计原则为基础的，这些设计原则首先都是复用的原则，遵循这些设计原则可以有效地提高系统的复用性，同时提高系统的可维护性。

✓ 可维护性(Maintainability)：指软件能够被理解、改正、适应及扩展的难易程度

✓ 可复用性(Reusability)：指软件能够被重复使用的难易程度

鬼吗寻踩贯豪去刚练仍

常用的面向对象设计原则包括7个，这些原则并不是孤立存在的，它们相互依赖，相互补充

设计原则名称	定 义	使用频率
单一职责原则 (Single Responsibility Principle, SRP)	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	★★★★☆
开闭原则 (Open-Closed Principle, OCP)	软件实体应当对扩展开放，对修改关闭	★★★★★
里氏代换原则 (Liskov Substitution Principle, LSP)	所有引用基类的地方必须能透明地使用其子类的对象	★★★★★
依赖倒转原则 (Dependence Inversion Principle, DIP)	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	★★★★★
接口隔离原则 (Interface Segregation Principle, ISP)	客户端不应该依赖那些它不需要的接口	★★☆☆☆
合成复用原则 (Composite Reuse Principle, CRP)	优先使用对象组合，而不是继承来达到复用的目的	★★★★☆
迪米特法则 (Law of Demeter, LoD)	每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位	★★★☆☆

鬼吗寻踩贯豪去刚比释

- ✓ 可维护性(Maintainability)：指软件能够被理解、改正、适应及扩展的难易程度
- ✓ 可复用性(Reusability)：指软件能够被重复使用的难易程度
- ✓ 面向对象设计的目标之一在于支持可维护性复用，一方面需要实现设计方案或者源代码的复用，另一方面要确保系统能够易于扩展和修改，具有良好的可维护性

鬼吗寻踩贯豪去刚比释

- ✓ 面向对象设计原则为支持可维护性复用而诞生
- ✓ 指导性原则，非强制性原则
- ✓ 每一个设计模式都符合一个或多个面向对象设计原则，面向对象设计原则是用于评价一个设计模式的使用效果的重要指标之一

卖乙艺轧去刚

✓单一职责原则定义

- 单一职责原则是最简单的面向对象设计原则，用于控制类的粒度大小

单一职责原则：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。

就一个类而言，应该仅有一个引起它变化的原因

Single Responsibility Principle (SRP): Every object should have **a single responsibility**, and that responsibility should be entirely encapsulated by the class.

✓单一职责原则分析

- 一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小
- 当一个职责变化时，可能会影响其他职责的运作
- 将这些职责进行分离，将不同的职责封装在不同的类中
- 将不同的变化原因封装在不同的类中
- 单一职责原则是实现高内聚、低耦合的指导方针

There should never be more than one reason for a class to change.

✓ 单一职责原则实例

实例说明

某软件公司开发人员针对CRM（Customer Relationship Management，客户关系管理）系统中的客户信息图表统计模块提出了如图2-1所示的初始设计方案。

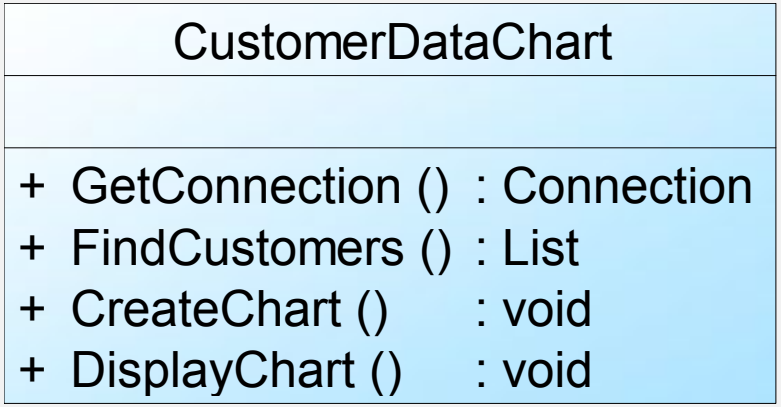


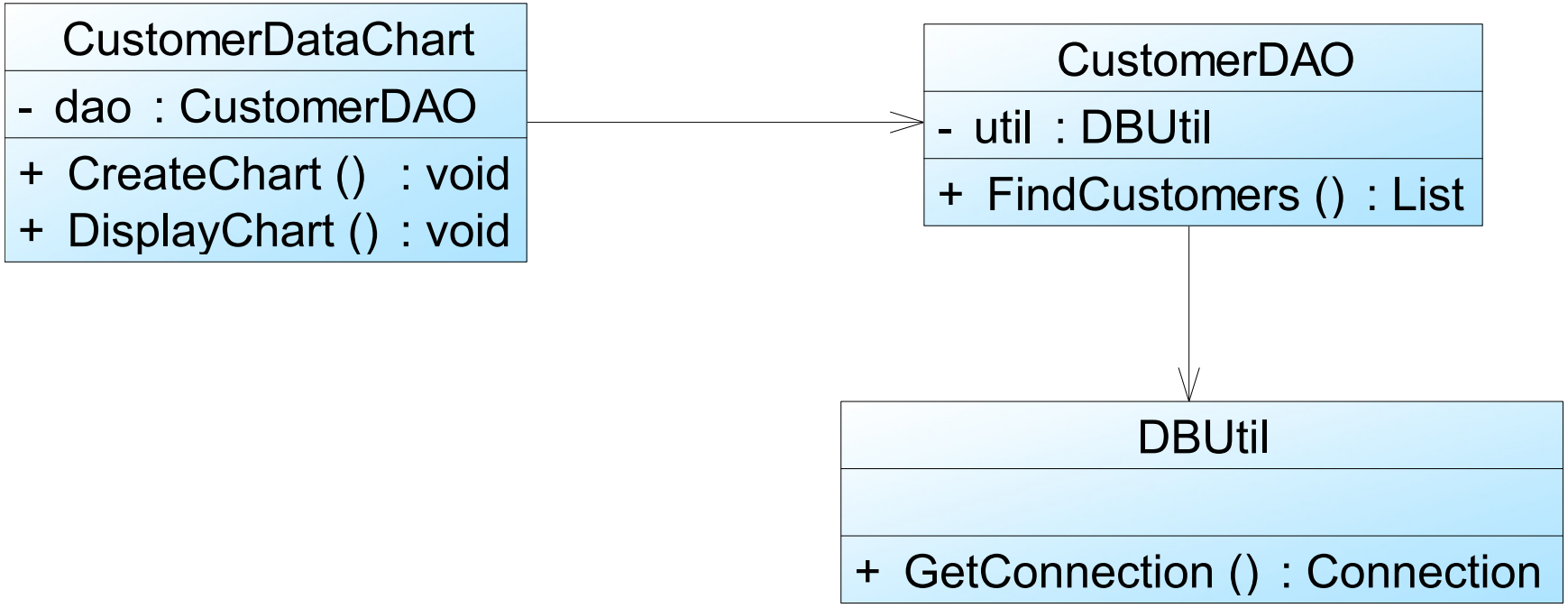
图2-1 初始设计方案结构图

在图2-1中，GetConnection()方法用于连接数据库，FindCustomers()用于查询所有的客户信息，CreateChart()用于创建图表，DisplayChart()用于显示图表。

现使用单一职责原则对其进行重构。

✓ 单一职责原则实例

□ 实例解析



异需去刚

✓开闭原则定义

- 开闭原则是面向对象的可复用设计的第一块基石，是最重要的面向对象设计原则

> **开闭原则：**软件实体应当**对扩展开放，对修改关闭**。
>
> **Open-Closed Principle (OCP):** Software entities should be **open for extension**, but **closed for modification**.

异需去刚

✓开闭原则分析

- 开闭原则由Bertrand Meyer于1988年提出
- 在开闭原则的定义中，软件实体可以是一个软件模块、一个由多个类组成的局部结构或一个独立的类
- 开闭原则是指软件实体应尽量在不修改原有代码的情况下进行扩展
- 抽象化是开闭原则的关键
- 相对稳定的抽象层 + 灵活的具体层
- 对可变性封装原则(Principle of Encapsulation of Variation, EVP)：找到系统的可变因素并将其封装起来

✓ 里氏代换原则分析

- 里氏代换原则由2008年图灵奖得主、美国第一位计算机科学女博士、麻省理工学院教授Barbara Liskov和卡内基.梅隆大学Jeannette Wing教授于1994年提出

芭芭拉·利斯科夫（Barbara Liskov），美国计算机科学家，2008年图灵奖得主，2004年约翰.冯诺依曼奖得主，美国工程院院士，美国艺术与科学院院士，美国计算机协会会士。现任麻省理工学院电子电气与计算机科学系教授，她是美国第一个计算机科学女博士。

✓里氏代换原则定义

里氏代换原则：所有引用基类的地方必须能透明地使用其子类的对象。

Liskov Substitution Principle (LSP): Functions that use pointers or references to **base classes** must be able to use objects of **derived classes** without knowing it.

闯泡令摘去刚

✓ 里氏代换原则分析

- 在软件中将一个基类对象替换成它的子类对象，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对象的话，那么它不一定能够使用基类对象
- 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

> ***我喜欢动物 → 我喜欢狗***
>
> ***因为狗是动物*** ☺

侦辨倘透去刚

✓依赖倒转原则定义

依赖倒转原则：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。**

Dependency Inversion Principle (DIP): High level modules should not depend upon low level modules, both should depend upon abstractions. **Abstractions should not depend upon details, details should depend upon abstractions**.

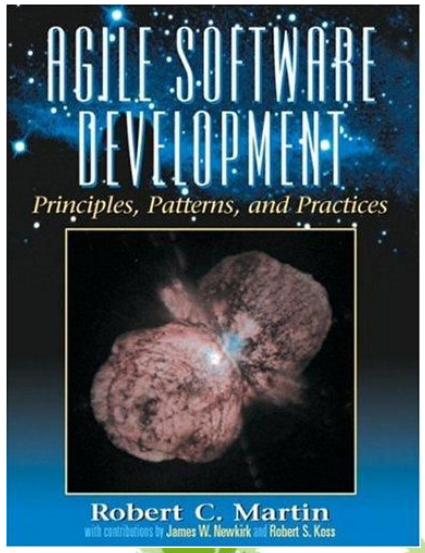
要针对接口编程，不要针对实现编程

Program to an interface, not an implementation.

侦辨倘透去刚

✓ 依赖倒转原则分析

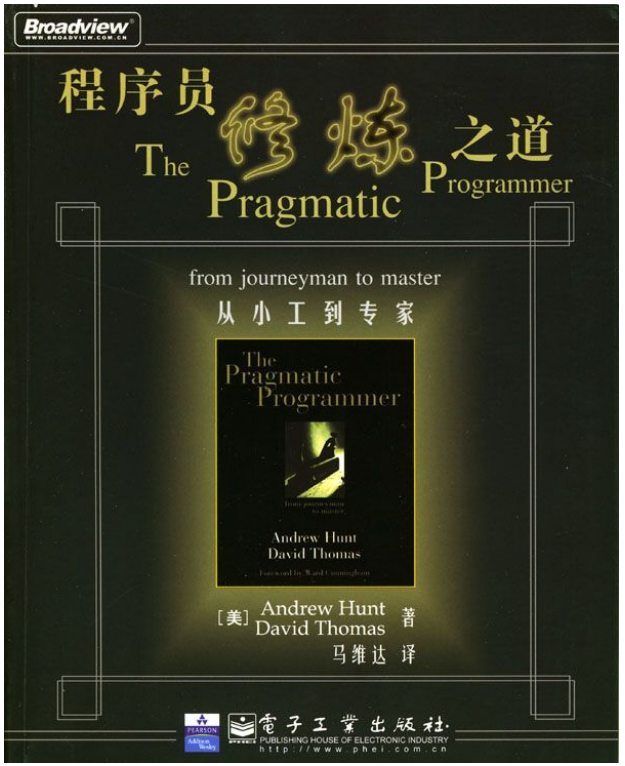
- 依赖倒转原则是Robert C. Martin在1996年为“C++ Reporter”所写的专栏Engineering Notebook的第三篇，后来加入到他在2002年出版的经典著作《Agile Software Development, Principles, Patterns, and Practices》一书中



侦辨倘透去刚

✓ 依赖倒转原则分析

- 在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等
- 在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中
- 我们常常使用的Structs，所有具体ACTION都存放于配置文件中



侦辨倘透去刚

✓ 依赖倒转原则分析

- 针对抽象层编程，将具体类的对象通过依赖注入(Dependency Injection, DI)的方式注入到其他对象
  - 构造注入
  - 设值注入（Setter注入）
  - 接口注入

某软件公司开发人员在开发CRM系统时发现：该系统经常需要将存储在TXT或Excel文件中的客户信息转存到数据库中，因此需要进行数据格式转换。在客户数据操作类CustomerDAO中将调用数据格式转换类的方法来实现格式转换，初始设计方案结构如图2-3所示：

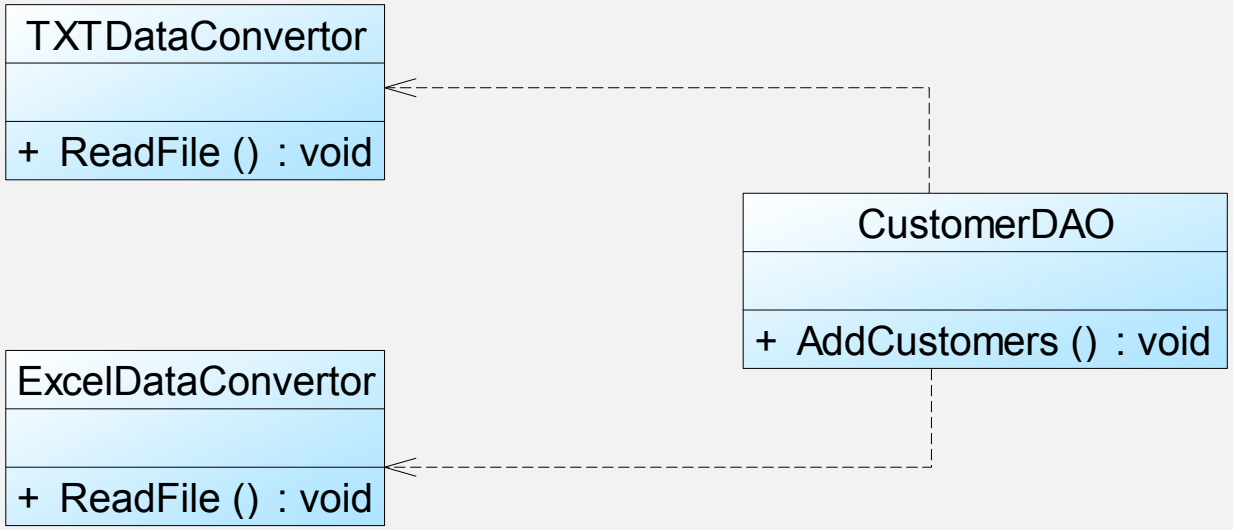
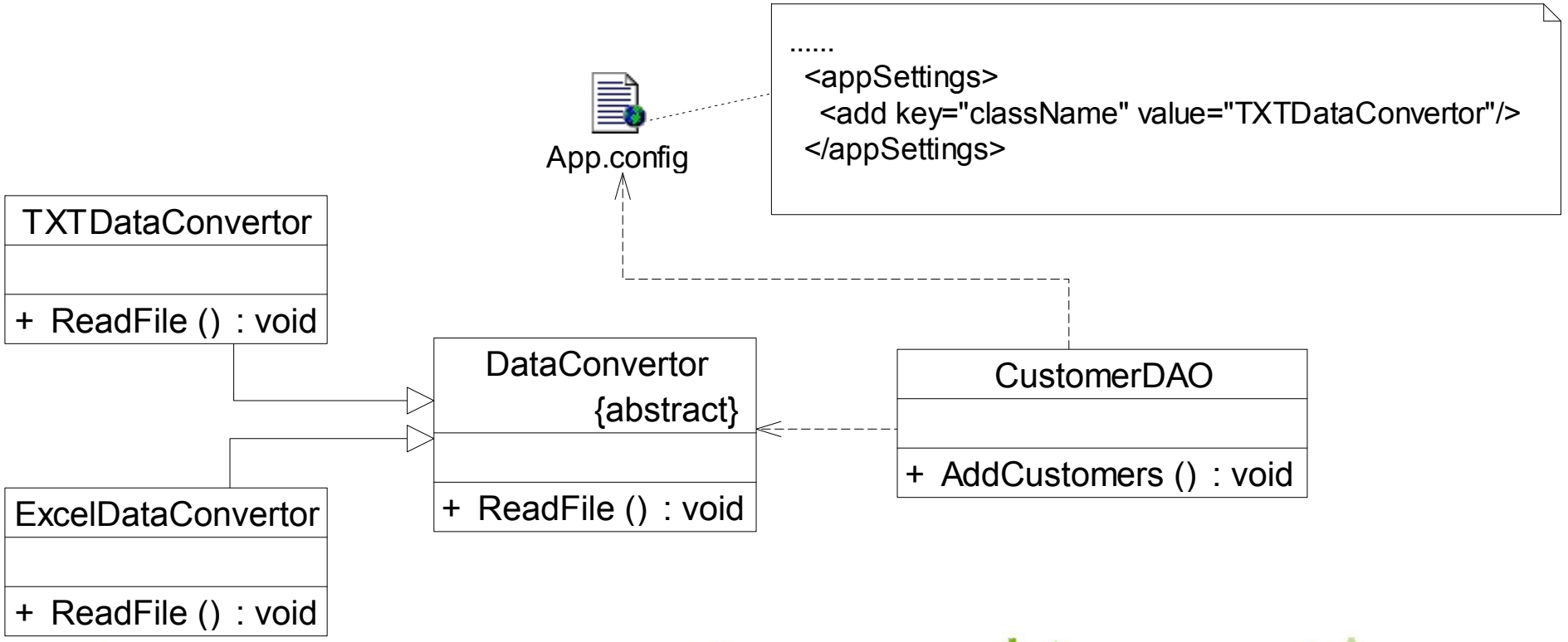


图2-3 初始设计方案结构图

在编码实现图2-3所示结构时，该软件公司开发人员发现该设计方案存在一个非常严重的问题，由于每次转换数据时数据来源不一定相同，因此需要经常更换数据转换类，例如有时候需要将TXTDataConvertor改为ExcelDataConvertor，此时，需要修改CustomerDAO的源代码，而且在引入并使用新的数据转换类时也不得不修改CustomerDAO的源代码，系统扩展性较差，违反了开闭原则，需要对该方案进行重构。

✓OCP/LSP/DIP综合实例

实例解析



撤古颠筒去刚

✓ 接口隔离原则定义

接口隔离原则：客户端不应该依赖那些它不需要的接口。

Interface Segregation Principle (ISP): Clients **should not be forced to depend upon interfaces that they do not use**.

✓ 接口隔离原则分析

- 当一个接口太大时，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
- 每一个接口应该承担一种相对独立的角色，不干不该干的事，该干的事都要干

✓接口隔离原则分析

- “接口”定义(1)：一个类型所提供的所有方法特征的集合。一个接口代表一个角色，每个角色都有它特定的一个接口，“角色隔离原则”
- “接口”定义(2)：狭义的特定语言的接口。接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口，每个接口中只包含一个客户端所需的方法，“定制服务”

撤古颠筒去刚

某软件公司开发人员针对CRM系统的客户数据显示模块设计了如图2-5所示接口，其中方法DataRead()用于从文件中读取数据，方法TransformToXML()用于将数据转换成XML格式，方法CreateChart()用于创建图表，方法DisplayChart()用于显示图表，方法CreateReport()用于创建文字报表，方法DisplayReport()用于显示文字报表。

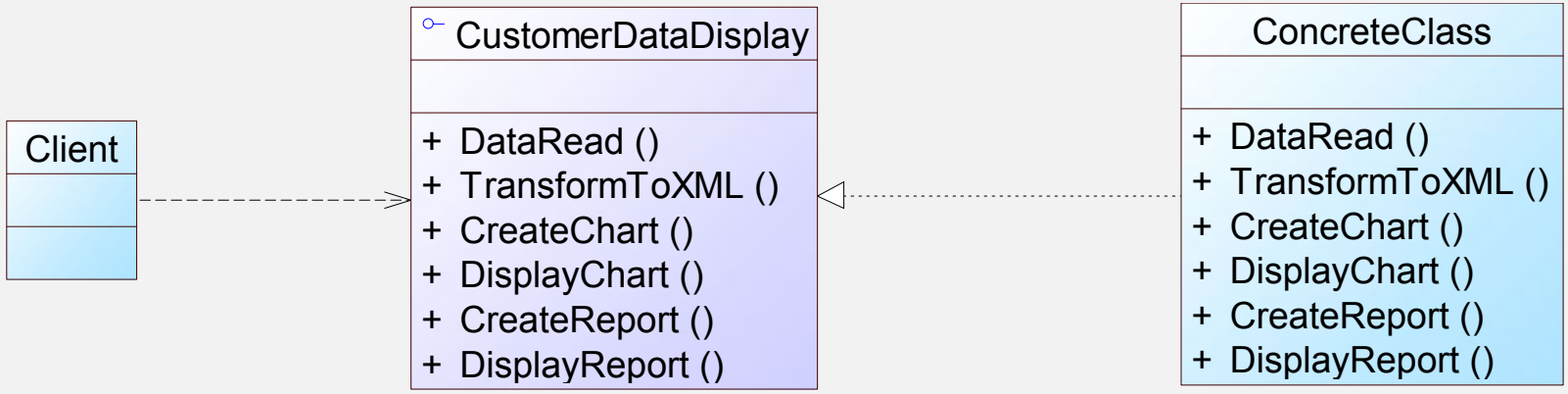


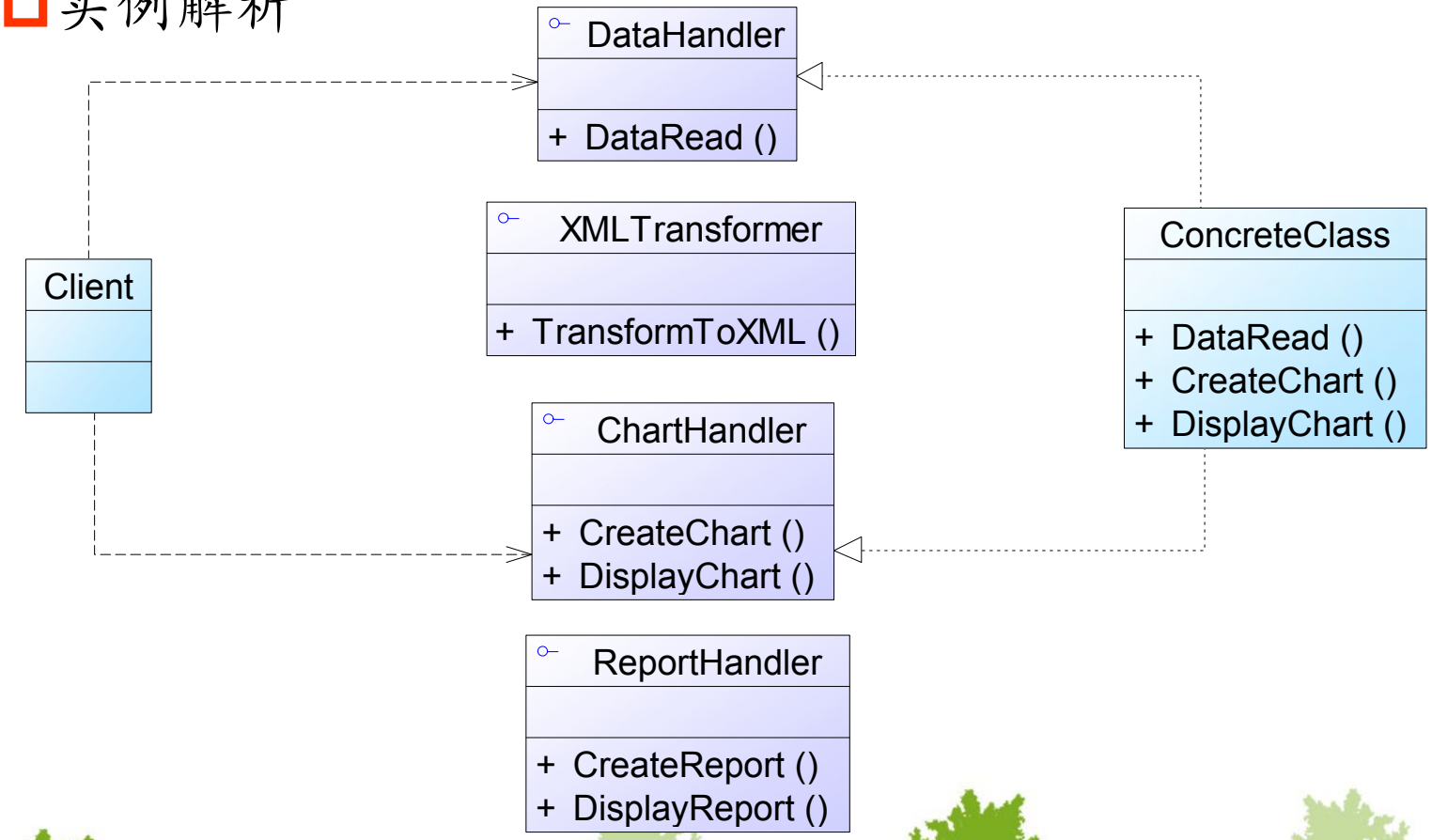
图2-5 初始设计方案结构图

在实际使用过程中开发人员发现该接口很不灵活，例如：如果一个具体的数据显示类无须进行数据转换（源文件本身就是XML格式），但由于实现了该接口，不得不实现其中声明的TransformToXML()方法（至少需要提供一个空实现）；如果需要创建和显示图表，除了需要实现与图表相关的方法外，还需要实现创建和显示文字报表的方法，否则程序在编译时将报错。

现使用接口隔离原则对其进行重构。

✓接口隔离原则实例

实例解析



✓合成复用原则定义

- 合成复用原则又称为组合/聚合复用原则(Composition/ Aggregate Reuse Principle, CARP)

合成复用原则：优先**使用对象组合，而不是继承**来达到**复用**的目的。

Composite Reuse Principle (CRP)： **Favor composition** of objects **over inheritance** as a **reuse** mechanism.

✓合成复用原则分析

- 合成复用原则就是在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，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功能的目的
- 复用时要尽量使用组合/聚合关系（关联关系），少用继承

✓合成复用原则分析

- 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）
- 组合/聚合复用：耦合度相对较低，有选择性地调用成员对象的操作；可以在运行时动态进行，新对象可以动态地引用与成员对象类型相同的其他对象。（“黑箱”复用）

某软件公司开发人员在初期的CRM系统设计中，考虑到客户数量不多，系统采用Access作为数据库，与数据库操作有关的类，例如CustomerDAO类等都需要连接数据库，连接数据库的方法GetConnection()封装在DBUtil类中，由于需要重用DBUtil类的GetConnection()方法，设计人员将CustomerDAO作为DBUtil类的子类，初始设计方案结构如图2-7所示。

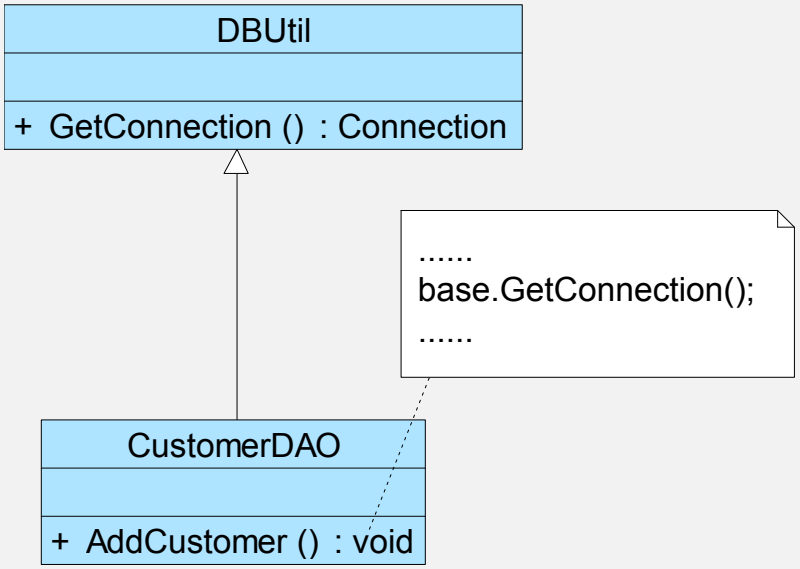


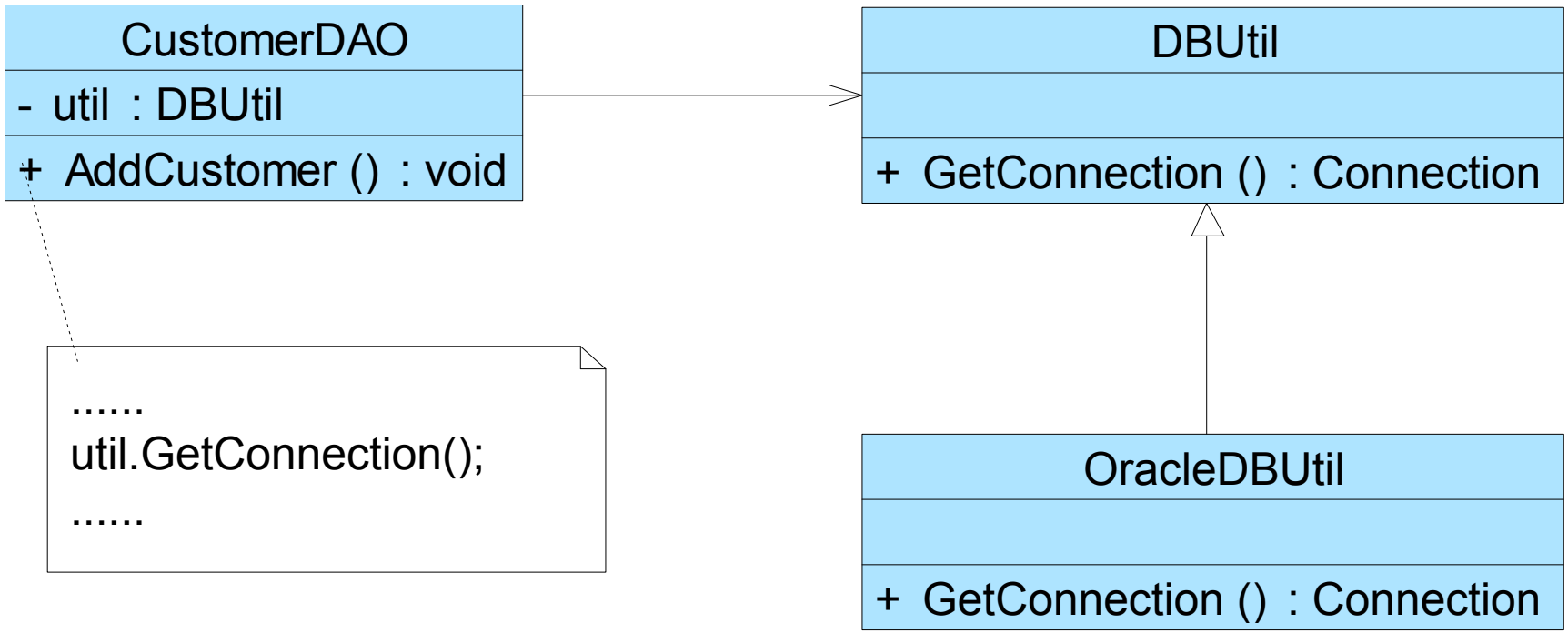
图2-7 初始设计方案结构图

随着客户数量的增加，系统决定升级为Oracle数据库，因此需要增加一个新的OracleDBUtil类来连接Oracle数据库，由于在初始设计方案中CustomerDAO和DBUtil之间是继承关系，因此在更换数据库连接方式时需要修改CustomerDAO类的源代码，将CustomerDAO作为OracleDBUtil的子类，这将违背开闭原则。当然也可以直接修改DBUtil类的源代码，这同样也违背了开闭原则。

现使用合成复用原则对其进行重构。

✓合成复用原则实例

- 实例解析



酱绝电涂刚

✓迪米特法则定义

- 迪米特法则又称为最少知识原则(Least Knowledge Principle, LKP)

迪米特法则：每一个软件单位对其他的单位都只有**最少的知识**，而且局限于那些与本单位密切相关的软件单位。

Law of Demeter (LoD): Each unit should have **only limited knowledge** about other units: only units "closely" related to the current unit.

✓ 迪米特法则分析

- 迪米特法则来自于1987年美国东北大学(Northeastern University)一个名为“Demeter”的研究项目
- 迪米特法则要求一个软件实体应当尽可能少地与其他实体发生相互作用
- 应用迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系

✓迪米特法则分析

- 不要和“陌生人”说话 (Don't talk to strangers.)
- 只与你的直接朋友通信 (Talk only to your immediate friends.)
  - (1) 当前对象本身(this)
  - (2) 以参数形式传入到当前对象方法中的对象
  - (3) 当前对象的成员对象
  - (4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友
  - (5) 当前对象所创建的对象
- 任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”
- 在应用迪米特法则时，一个对象只能与直接朋友发生交互，不要和“陌生人”发生直接交互，这样做可以降低系统的耦合度，一个对象的改变不会给太多其他对象带来影响

酱绝电涂刚

✓ 迪米特法则分析

- 迪米特法则要求在设计系统时，应该尽量减少对象之间的交互
- 如果两个对象之间不必彼此直接通信，那么这两个对象就不应该发生任何直接的相互作用
- 如果其中一个对象需要调用另一个对象的方法，可以通过“第三者”转发这个调用
- 通过引入一个合理的“第三者”来降低现有对象之间的耦合度

✓ 迪米特法则分析

- 应用迪米特法则注意点：
  - 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，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影响
  - 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
  - 在类的设计上，只要有可能，一个类型应当设计成不变类
  - 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

某软件公司所开发CRM系统包含很多业务操作窗口，在这些窗口中，某些界面控件之间存在复杂的交互关系，一个控件事件的触发将导致多个其他界面控件产生响应。例如，当一个按钮(Button)被单击时，对应的列表框(List)、组合框(ComboBox)、文本框(TextBox)、文本标签(Label)等都将发生改

2-9所示

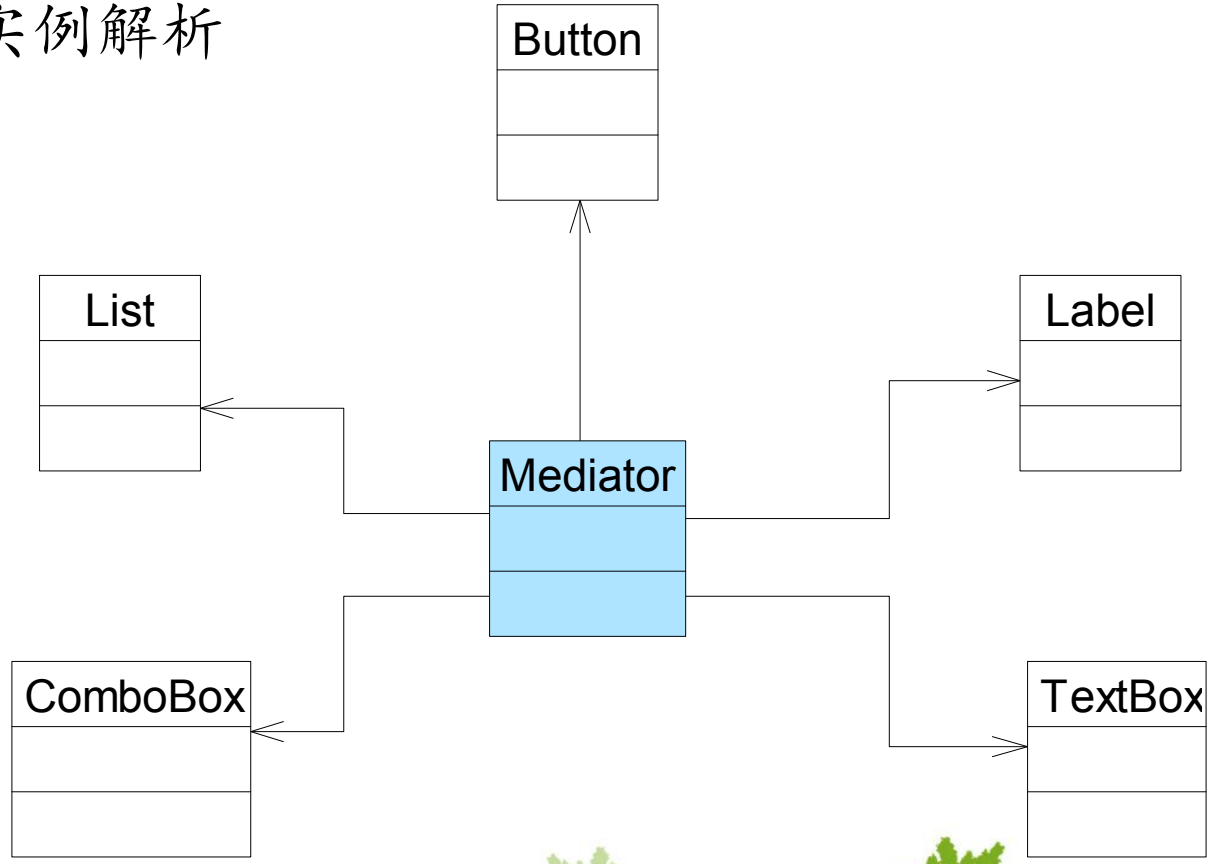
加新的

，也不便于增加和删除控件。

现使用迪米特法则对其进行重构。

✓迪米特法则实例

- 实例解析



柏繁少聋

- ✓ 单一职责原则要求一个类只负责一个功能领域中的相应职责。
- ✓ 开闭原则要求一个软件实体应当对扩展开放，对修改关闭。
- ✓ 里氏代换原则是指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。
- ✓ 依赖倒转原则是指抽象不应当依赖于细节，细节应当依赖于抽象，即要针对接口编程，不要针对实现编程。
- ✓ 接口隔离原则是指使用多个专门的接口比使用单一的总接口要好。
- ✓ 合成复用原则就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用已有功能的目的。
- ✓ 狭义的迪米特法则要求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。

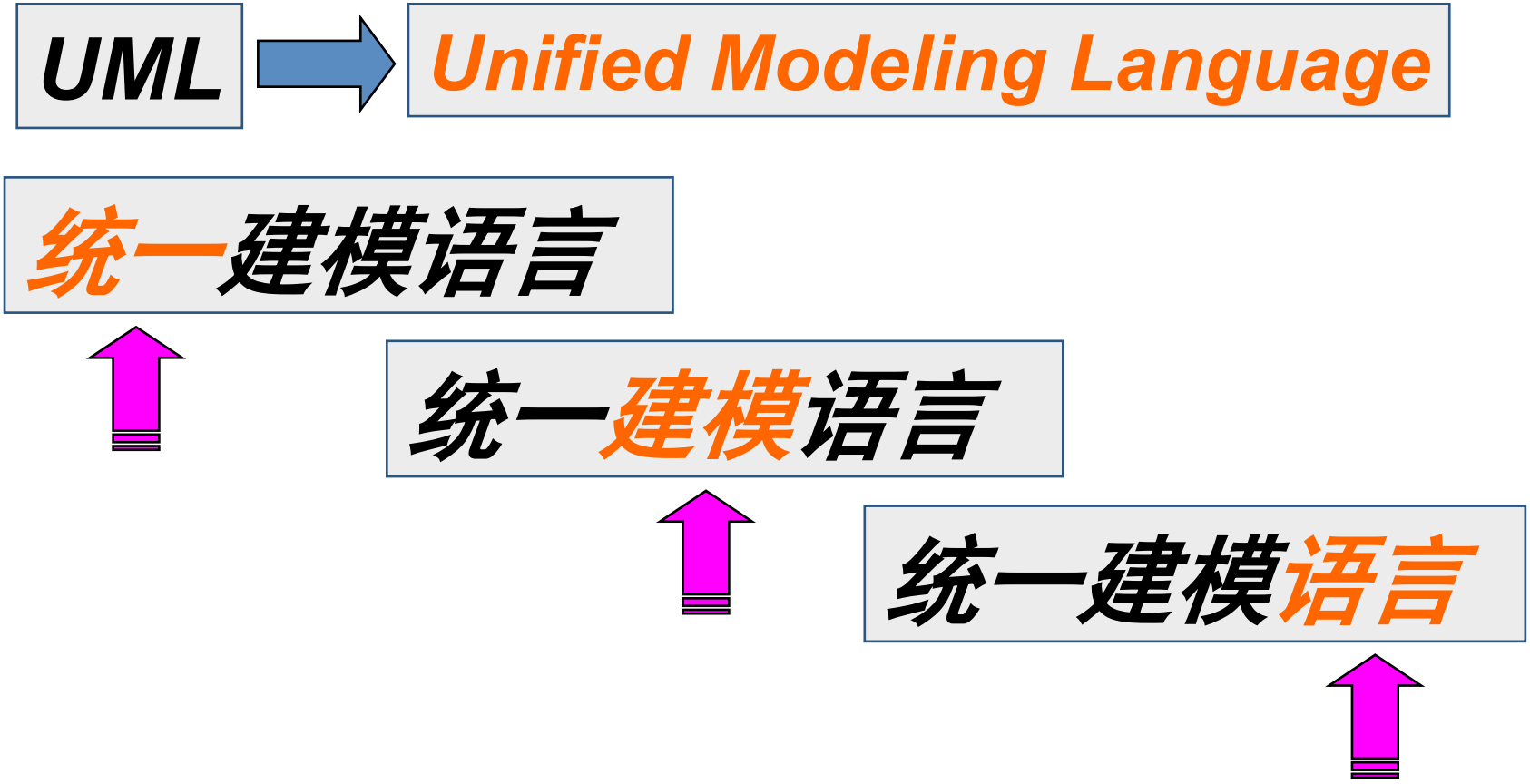
1 引言

2 设计模式概述

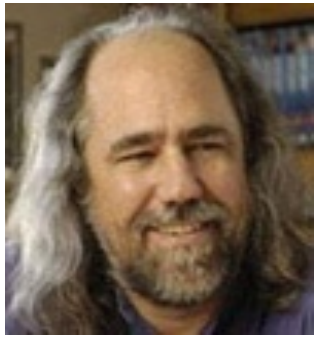
3 面向对象的设计原则

4 UML概述

UML比释



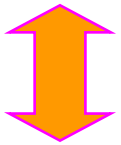
UML比释



Ivar Jacoboson

Grady Booch

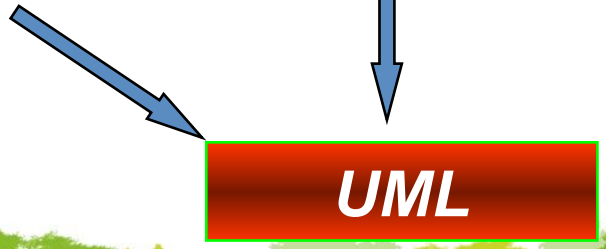
James Rumbaugh



Object-Oriented Software Engineering(OOSE)

Booch开发方法

Object Modeling Technique(OMT)



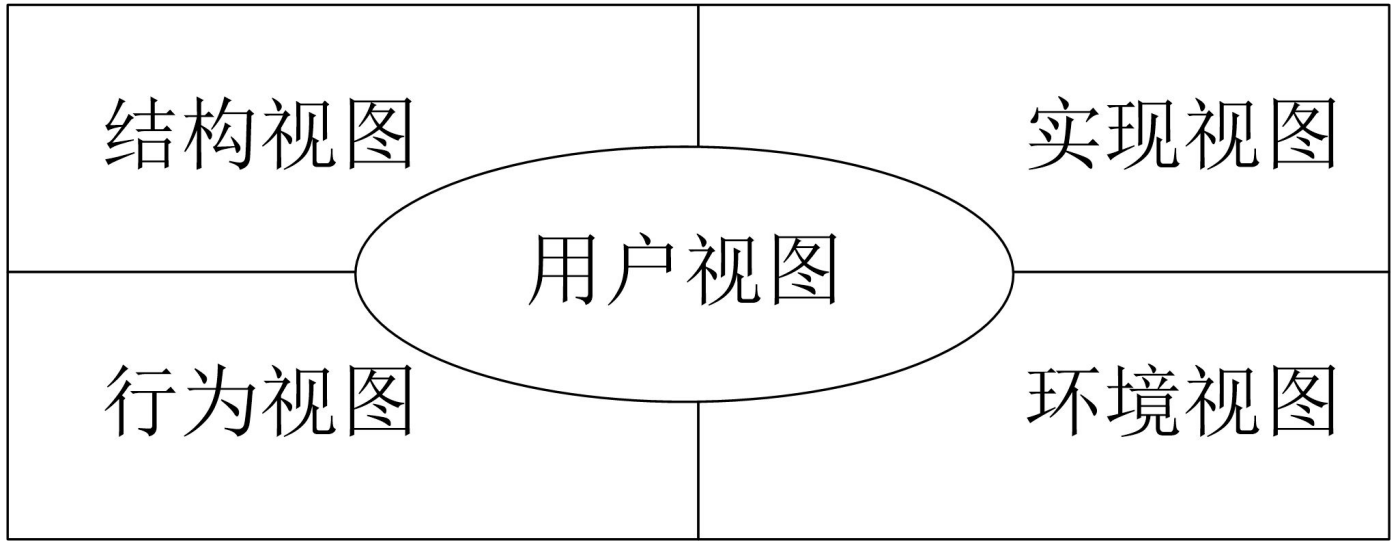
UML比释

- ✓UML是一个通用的可视化建模语言，不同于编程语言，它通过一些标准的图形符号和文字来对系统进行建模
- ✓用于对软件进行描述、可视化处理、构造和建立软件系统制品的文档
- ✓是一套总结了以往建模技术的经验并吸收了当今最优秀成果的标准建模方法

UML比释

✓UML的结构

- 视图(View)



UML比释

✓UML的结构

- 图(Diagram)：13种(UML 2.X)

  用例图(Use Case Diagram)，类图(Class Diagram)，对象图(Object Diagram)，包图(Package Diagram)，组合结构图(Composite Structure Diagram)，状态图(State Diagram)，活动图(Activity Diagram)，顺序图(Sequence Diagram)，通信图(Communication Diagram)，定时图(Timing Diagram)，交互概览图(Interaction Overview Diagram)，组件图(Component Diagram)，部署图(Deployment Diagram)

UML比释

- ✓ UML的结构
  - 模型元素(Model Element)

    UML图中所使用的一些概念，对应于普通的面向对象概念

    同一个模型元素可以在多个不同的UML图中使用，但是，无论在哪个图中，同一个模型元素都必须保持相同的意义并具有相同的符号

  - 通用机制(General Mechanism)

    UML提供的通用机制为模型元素提供额外的注释、语义和其他信息，包括扩展机制，允许用户对UML进行扩展

绞专绞碍UML词竹

✓ 类

- 类(Class)封装了数据和行为，是面向对象的重要组成部分，它是具有相同属性、操作、关系的对象集合的总称
- 在系统中，每个类都具有一定的职责，职责指的是类要完成什么样的功能，要承担什么样的义务。一个类可以有多种职责，设计得好的类通常有且仅有一种职责。在定义类的时候，将类的职责分解成为类的属性和操作（即方法）
- 类的属性即类的数据职责，类的操作即类的行为职责

绞专绞碍UML词竹

✓ 类

- 类实例化成对象(Object)，对象对应于某个具体的事物，是类的实例(Instance)
- 类图(Class Diagram)使用出现在系统中的不同类来描述系统的静态结构，它用来描述不同的类以及它们之间的关系

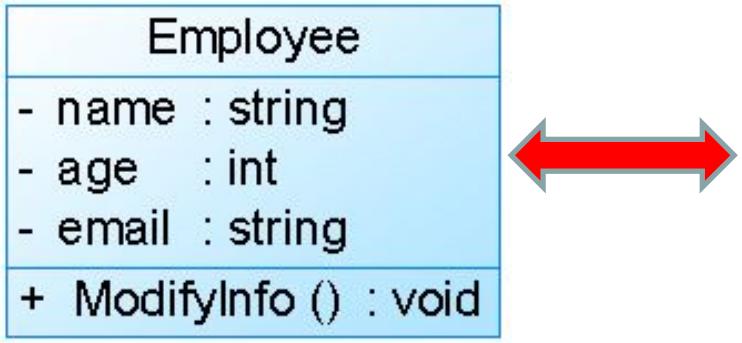
绞专绞碍UML词竹

✓类的UML图示

- 在UML类图中，类一般由三部分组成：

  第一部分是类名：每个类都必须有一个名字，类名是一个字符串。

  按照C#语言的命名规范，类名中每一个单词的首字母均大写。



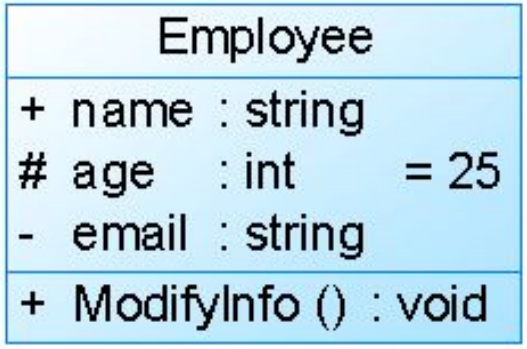

```
public class Employee
{
        private string name;
        private int age;
        private string email;

        public void ModifyInfo()
        {
                ......
        }
}
```

✓类的UML图示

第二部分是类的属性(Attributes)：属性是指类的性质，即类的成员变量。一个类可以有任意多个属性，也可以没有属性。

按照C#语言的命名规范，属性名中的第一个单词全小写，之后每个单词首字母大写。（**驼峰命名法**）

可见性 名称:类型 [= 默认值]

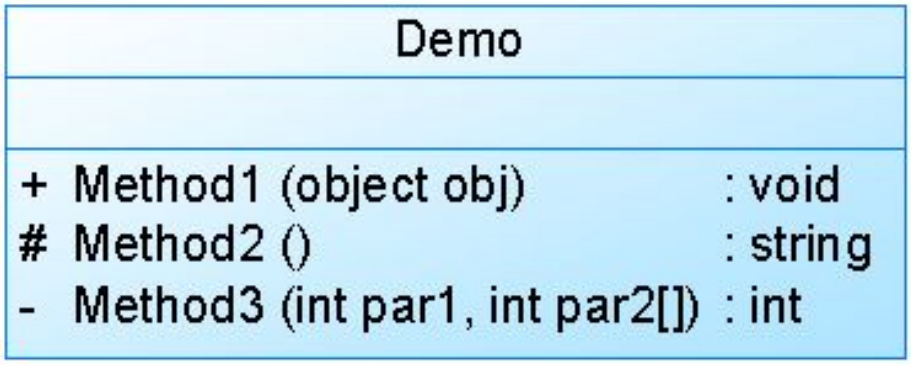


✓类的UML图示

第三部分是类的操作(Operations)：操作是类的任意一个实例对象都拥有的行为，是类的成员方法。

按照C#语言的命名规范，方法名中的每个单词首字母都大写。（**帕斯卡命名法**）

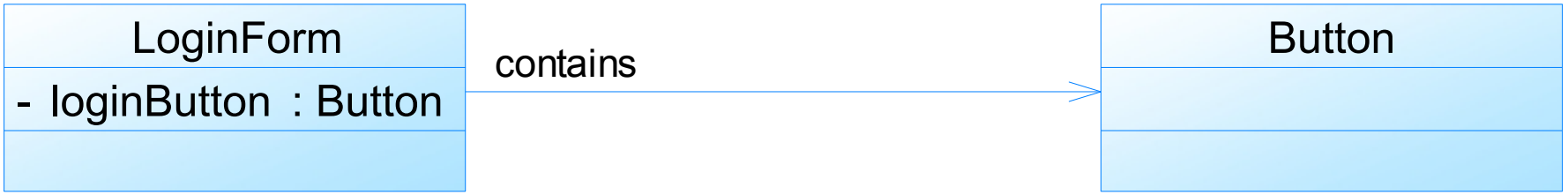
可见性 名称(参数列表) [:返回类型]



✓关联关系

- 关联(Association)关系是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系。
- 在UML类图中，用实线连接有关联关系的对象所对应的类，在使用C#、C++和Java等编程语言实现关联关系时，通常将一个类的对象作为另一个类的成员变量。
- 在使用类图表示关联关系时可以在关联线上标注角色名。

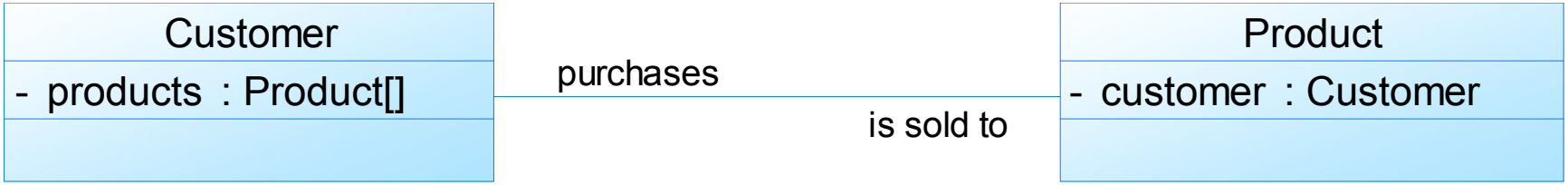
✓关联关系




```
public class LoginForm
{
    private Button loginButton;
    ……
}
public class Button
{
    ……
}
```

✓关联关系

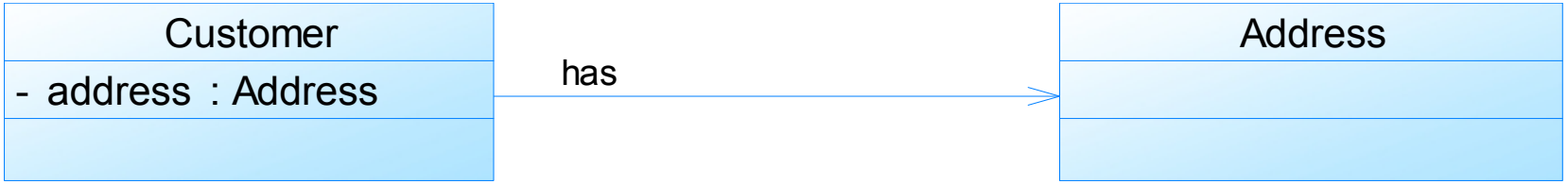
- 双向关联：默认情况下，关联是双向的




```
public class Customer
{
    private Product[] products;
    ……
}
public class Product
{
    private Customer customer;
    ……
}
```

✓关联关系

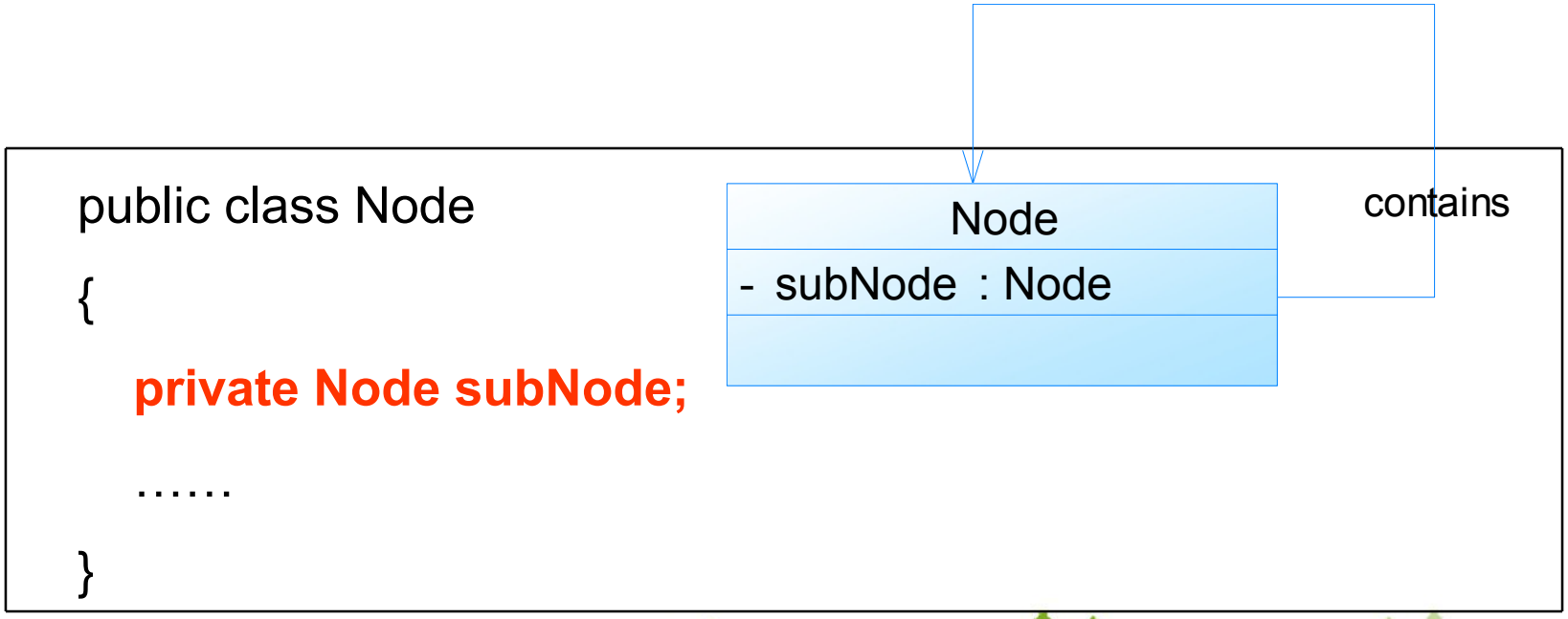
- 单向关联：类的关联关系也可以是单向的，单向关联用带箭头的实线表示




```
public class Customer
{
    private Address address;
    ……
}
public class Address
{
    ……
}
```

✓关联关系

- 自关联：在系统中可能会存在一些类的属性对象类型为该类本身，这种特殊的关联关系称为自关联



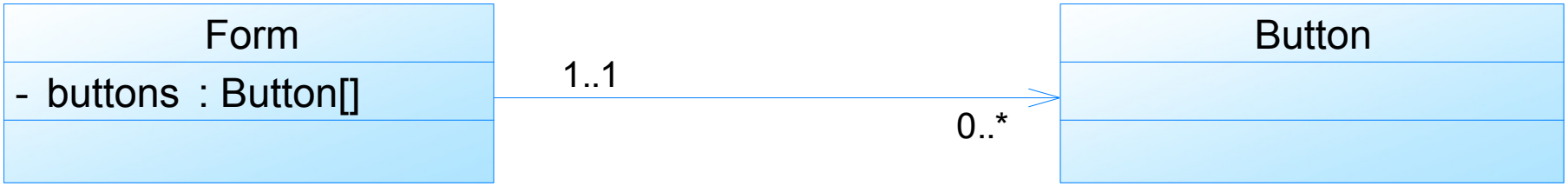
✓关联关系

- 多重性关联：多重性关联关系又称为重数性(Multiplicity)关联关系，表示两个关联对象在数量上的对应关系。在UML中，对象之间的多重性可以直接在关联直线上用一个数字或一个数字范围表示

表示方式	多重性说明
1..1	表示另一个类的一个对象只与该类的一个对象有关系
0..*	表示另一个类的一个对象与该类的零个或多个对象有关系
1..*	表示另一个类的一个对象与该类的一个或多个对象有关系
0..1	表示另一个类的一个对象没有或只与该类的一个对象有关系
m..n	表示另一个类的一个对象与该类最少m，最多n个对象有关系 (m≤n)

✓关联关系

- 多重性关联




```
public class Form
{
    private Button[] buttons;  //定义一个集合对象
    ……
}
public class Button
{
    ……
}
```

✓关联关系

□聚合关系

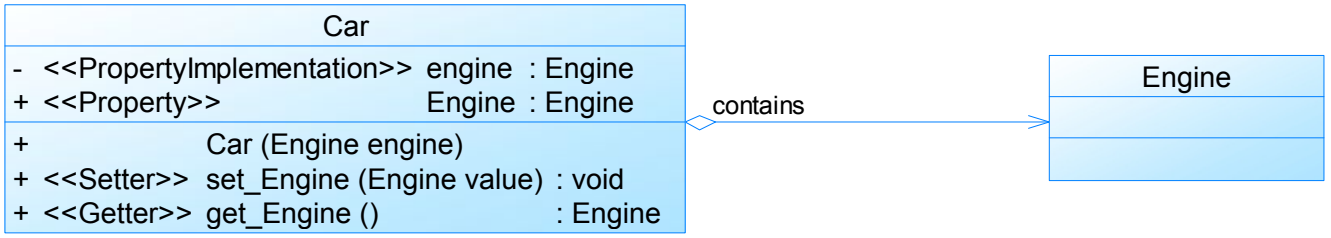
聚合(Aggregation)关系表示整体与部分的关系

在聚合关系中，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在

在UML中，聚合关系用带空心菱形的直线表示

✓关联关系

- 聚合关系




```
public class Car
{
    private Engine engine;
    public Car(Engine engine) //构造注入
    {
        this.engine = engine;
    }

    public Engine Engine
    {
        get { return engine; }
        set { engine = value; } //设值注入
    }
    ……
}
public class Engine
{
        ……
}
```

✓关联关系

□组合关系

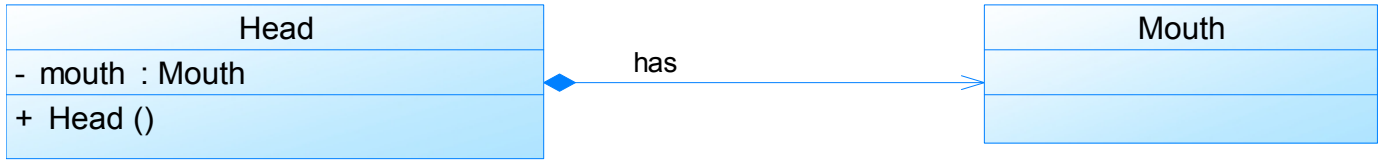
组合(Composition)关系也表示类之间整体和部分的关系，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在

成员对象与整体对象之间具有同生共死的关系

在UML中，组合关系用带实心菱形的直线表示

✓关联关系

- 组合关系



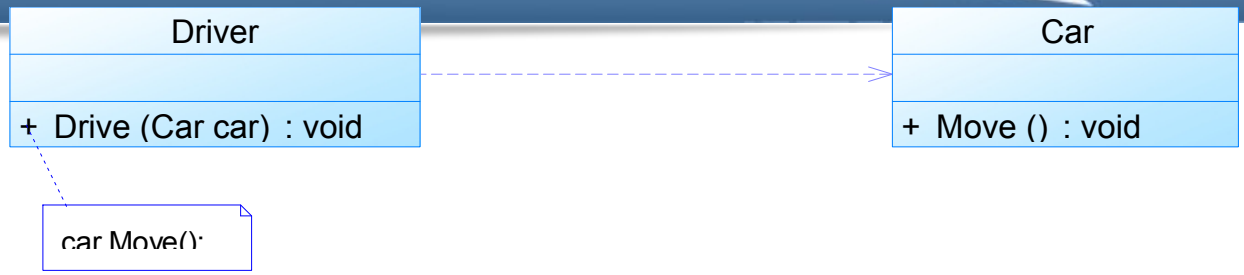

```
public class Head
{
    private Mouth mouth;
    public Head()
    {
        mouth = new Mouth();  //实例化成员类
    }
    ……
}
public class Mouth
{
        ……
}
```

✓依赖关系

- 依赖(Dependency)关系是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。
- 大多数情况下，依赖关系体现在某个类的方法使用另一个类的对象作为参数。
- 在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。

绞乌露碍兴编

✓ 依赖关系

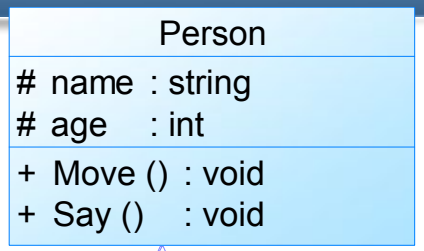


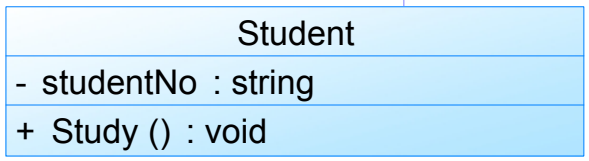

```
public class Driver
{
    public void Drive(Car car)
    {
        car.Move();
    }
    ……
}
public class Car
{
    public void Move()
    {
        ……
    }
    ……
}
```

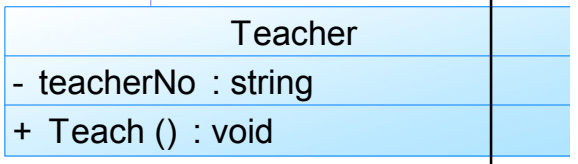
✓泛化关系

- 泛化(Generalization)关系也就是继承关系，用于描述父类与子类之间的关系，父类又称为基类或超类，子类又称为派生类。
- 在UML中，泛化关系用带空心三角形的直线来表示。
- 在用代码实现时，使用面向对象的继承机制来实现泛化关系，在C#中使用冒号“：”来实现。

✓泛化关系



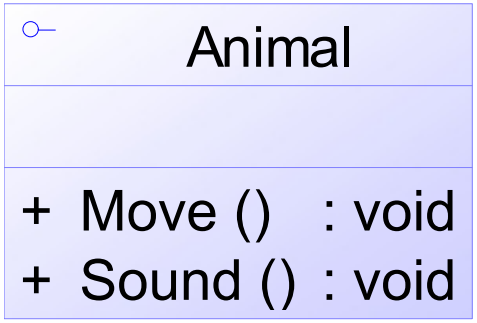





```
public class Person //父类
{
    protected string name;
    protected int age;
    public void Move()
    {
        ……
    }
    public void Say()
    {
        ……
    }
}
public class Student : Person  //子类
{
    private string studentNo;
    public void Study()
    {
        ……
    }
}
```

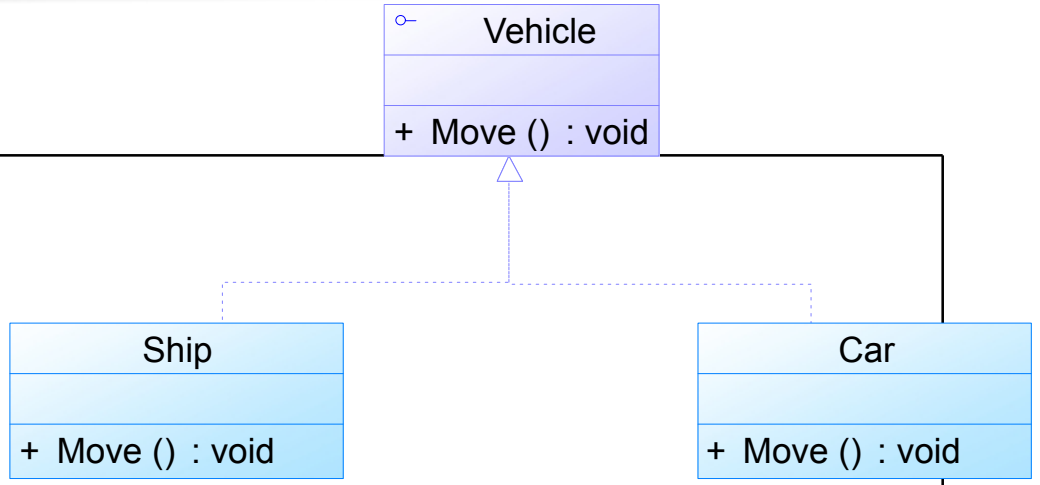
✓接口与实现关系

- 接口之间也可以有与类之间关系类似的继承关系和依赖关系，但是接口和类之间还存在一种实现(Realization)关系，在这种关系中，类实现了接口，类中的操作实现了接口中所声明的操作。
- 在UML中，类与接口之间的实现关系用带空心三角形的虚线来表示。



✓接口与实现关系

```
public interface Vehicle
{
    void Move();
}
public class Ship : Vehicle
{
    public void Move()
    {
        ……
    }
}
public class Car : Vehicle
{
    public void Move()
    {
        ……
    }
}
```

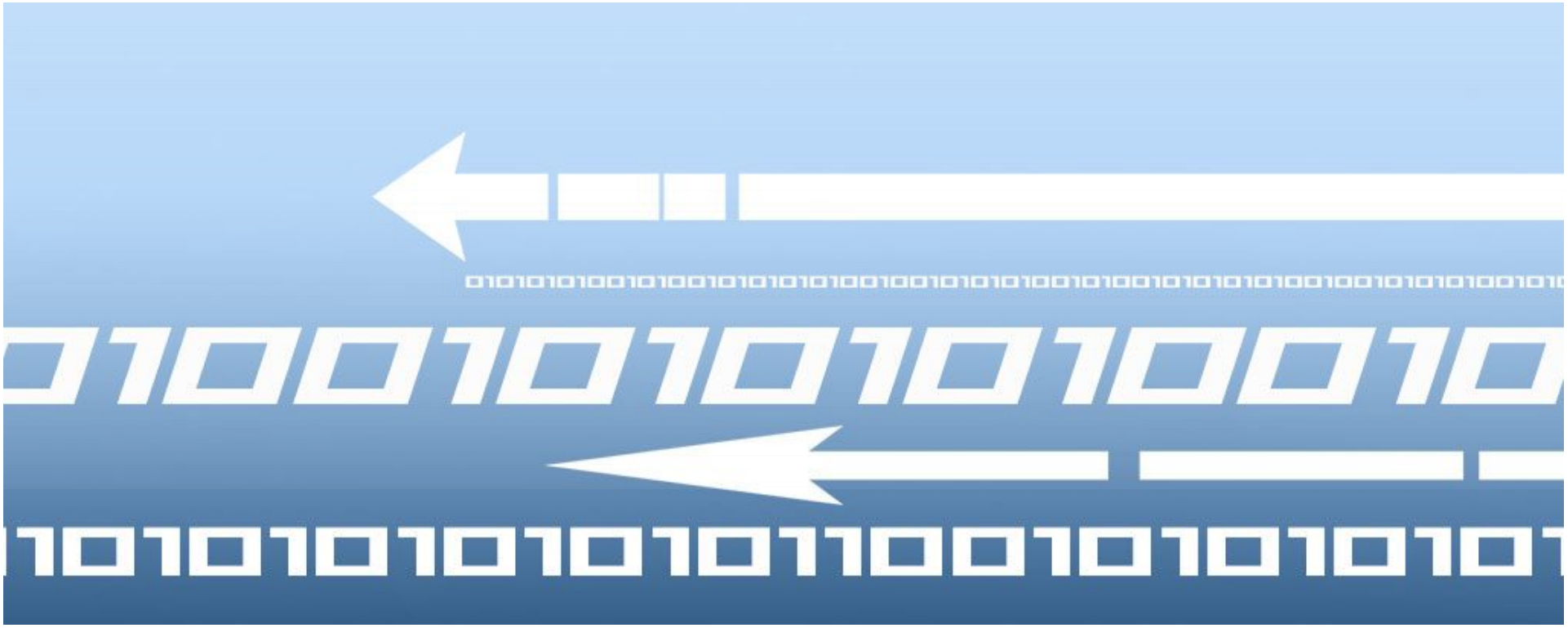


柏繁少聋

- ✓ 类包装了信息和行为，是面向对象的重要组成部分，它是具有相同属性、操作的对象集合。
- ✓ 类图使用需要出现在系统内的不同的类来描述系统的静态结构，类图包含类和它们之间的关系，它描述系统内所声明的类，但它没有描述系统运行时类的行为。
- ✓ 关联关系是一种结构化的关系，指一种对象和另一种对象有联系。
- ✓ 多重性关联关系又称为重数性关联关系，表示一个类的对象与另一个类的对象连接的个数。
- ✓ 聚合关系指的是整体与部分的关系，在聚合关系中，类A是类B的一部分，但是类A可以独立存在。

柏繁少聋

- ✓ 组合关系也表示类之间整体和部分的关系，但是组合关系中部分和整体具有相同的生存期。在组合关系中，类A包含类B，而且可以控制类B的生命周期。
- ✓ 依赖关系是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物。
- ✓ 泛化关系也就是继承关系，描述了超类与子类之间的关系。
- ✓ 类和接口之间存在实现关系。



感谢观赏

QQ：253692170

13755115139@139.com