

“十二五”普通高等教育本科国家级规划教材

北京高等教育精品教材

21世纪软件工程专业规划教材

软件工程导论（第6版）



第11章 面向对象设计

第11章 面向对象设计

分析是提取和整理用户需求，并建立问题域精确模型的过程。设计则是把分析阶段得到的需求转变成符合成本和质量要求的、抽象的系统实现方案的过程。

从面向对象分析到面向对象设计(OOD)，是一个逐渐扩充模型的过程。或者说，面向对象设计就是用面向对象观点建立求解域模型的过程。

本章首先讲述为获得优秀设计结果应该遵循的准则，然后具体讲述面向对象设计的任务和方法。

主要内容

- | | | | |
|-------|-----------|--------|-----------|
| 11. 1 | 面向对象设计的准则 | 11. 7 | 设计任务管理子系统 |
| 11. 2 | 启发规则 | 11. 8 | 设计数据管理子系统 |
| 11. 3 | 软件重用 | 11. 9 | 设计类中的服务 |
| 11. 4 | 系统分解 | 11. 10 | 设计关联 |
| 11. 5 | 设计问题域子系统 | 11. 11 | 设计优化 |
| 11. 6 | 设计人机交互子系统 | | |

主要内容



11.1 面向对象设计的准则

11.2 启发规则

11.3 软件重用

11.4 系统分解

11.5 设计问题域子系统

11.6 设计人机交互子系统

11.7 设计任务管理子系统

11.8 设计数据管理子系统

11.9 设计类中的服务

11.10 设计关联

11.11 设计优化

11.1 面向对象设计的准则

1. 模块化
2. 抽象
3. 信息隐藏
4. 弱耦合
5. 强内聚
6. 可重用

11.1 面向对象设计的准则

1. 模块化

- 对象就是模块
- 把数据结构和操作这些数据的方法紧密地结合在一起

2. 抽象

- 过程抽象
- 数据抽象：类
- 参数化抽象：C++的“模板”

11.1 面向对象设计的准则

3. 信息隐藏

- 通过对对象的封装性实现
- 类分离了接口与实现，支持信息隐藏

4. 弱耦合

- 耦合：一个软件结构内不同模块之间互连的紧密程度
- 弱耦合：系统中某一部分的变化对其他部分的影响降到最低程度
- 对象之间的耦合：交互耦合&继承耦合

11.1 面向对象设计的准则

5. 强内聚

- 内聚衡量一个模块内各个元素彼此结合的紧密程度
- 在设计时应该力求做到高内聚
- 面向对象设计的3种内聚：
服务内聚、类内聚、一般\特殊内聚

6. 可重用

- 尽量使用已有的类
- 如果确实需要创建新类，则在设计这些新类的协议时，
应该考虑将来的可重复使用性

主要内容

11.1 面向对象设计的准则

11.2 启发规则

11.3 软件重用

11.4 系统分解

11.5 设计问题域子系统

11.6 设计人机交互子系统

11.7 设计任务管理子系统

11.8 设计数据管理子系统

11.9 设计类中的服务

11.10 设计关联

11.11 设计优化

11.2 启发规则

1. 设计结果应该清晰易懂
2. 一般-特殊结构的深度应适当
3. 设计简单的类
4. 使用简单的协议
5. 使用简单的服务
6. 把设计变动减至最小

11.2 启发规则

1. 设计结果应该清晰易懂

- 提高软件可维护性和可重用性的重要措施
- 保证设计结果清晰易懂的主要因素：
 - (1) 用词一致
 - (2) 使用已有的协议
 - (3) 减少消息模式的数目
 - (4) 避免模糊的定义

2. 一般-特殊结构的深度应适当

- 类等级中包含的层数次适当
- 一个中等规模(大约包含100个类)的系统中，类等级层数次应保持为 7 ± 2 。

11.2 启发规则

3. 设计简单的类

- 尽量设计小而简单的类
- 注意以下几点：
 - (1) 避免包含过多的属性
 - (2) 有明确的定义
 - (3) 尽量简化对象之间的合作关系
 - (4) 不要提供太多服务

4. 使用简单的协议

- 消息中的参数不要超过3个

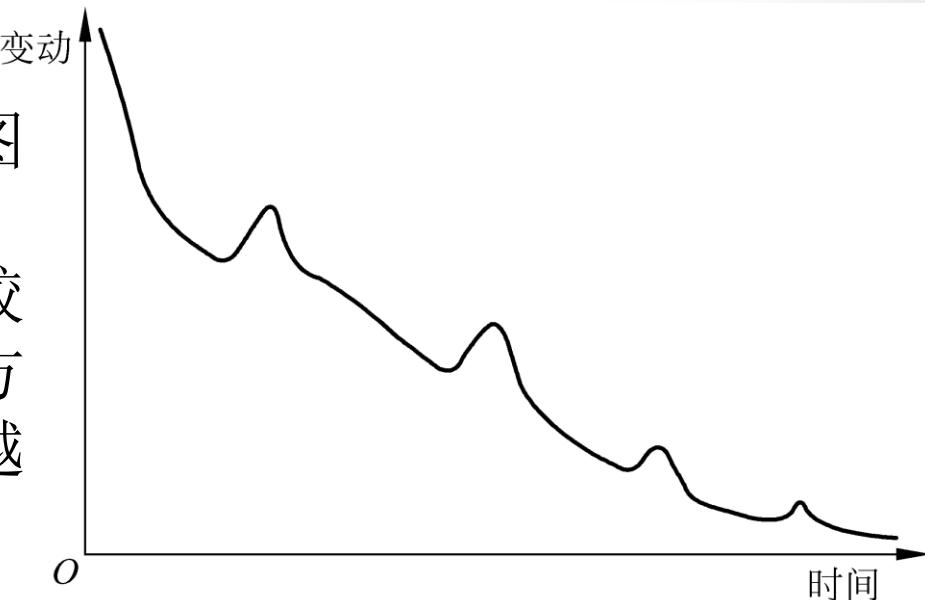
11.2 启发规则

5. 使用简单的服务

- 类中的服务通常都很小
- 尽量避免使用复杂的服务

6. 把设计变动减至最小

- 理想的设计变动曲线如右图所示
- 在设计的早期阶段，变动较大，随着时间推移，设计方案日趋成熟，改动也越来越小了



主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
-  11.3 软件重用
- 11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统

- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.3 软件重用

11.3.1 概述

11.3.2 类构建

11.3.3 设计简单的类

11.3 软件重用

11.3.1 软件重用概述

1. 重用

- 重用也叫再用或复用，是指同一事物不作修改或稍加改动就多次重复使用。
- 广义地说，软件重用可分为以下3个层次：
 - (1) 知识重用
 - (2) 方法和标准的重用
 - (3) 软件成分的重用
- 上述前两个重用层次属于知识工程研究的范畴，本节仅讨论软件成分重用问题。

11.3 软件重用

2. 软件成分的重用级别

- 代码重用
 - (1) 源代码剪贴
 - (2) 源代码包含
 - (3) 继承
- 设计结果重用
重用某个软件系统的设计模型(即求解域模型)
- 分许结果重用
更高级别的重用，即重用某个系统的分析模型

11.3 软件重用

3. 典型的可重用软件成分

主要有以下10种：

- (1) 项目计划
- (2) 成本估计
- (3) 体系结构
- (4) 需求模型和规格说明
- (5) 设计
- (6) 源代码
- (7) 用户文档和技术文档
- (8) 用户界面
- (9) 数据
- (10) 测试用例

11.3 软件重用

11.3.2 类构件

面向对象技术中的“类”，是比较理想的可重用软构件，不妨称之为类构件。

1. 可重用软构件应具备的特点

- 为使软构件也像硬件集成电路那样，能在构造各种各样的软件系统时方便地重复使用，就必须使它们满足下列要求：
 - (1) 模块独立性强
 - (2) 成具有高度可塑性
 - (3) 接口清晰、简明、可靠
 - (4) 需求模型和规格说明

11.3 软件重用

2. 类构件的重用方式

- 实例重用
 - (1) 使用适当的构造函数，按照需要创建类的实例
 - (2) 用几个简单的对象作为类的成员创建出一个更复杂的类
- 继承重用

继承性提供了一种对已有的类构件进行裁剪的机制
- 多态重用
 - (1) 使对象的对外接口更加一般化，降低了消息连接的复杂程度
 - (2) 提供一种简便可靠的软构件组合机制

11.3 软件重用

11.3.3 软件重用的效益

1. 质量

理想情况下，为了重用而开发的软件构件已被证明是正确的，且没有缺陷。

事实上，由于不能定期进行形式化验证，错误可能而且也确实存在。

但是，随着每一次重用，都会有一些错误被发现并被清除，构件的质量也会随之改善。

随着时间的推移，构件将变成实质上无错误的。

11.3 软件重用

2. 生产率

当把可重用的软件成分应用于软件开发的全过程时，创建计划、模型、文档、代码和数据所需花费的时间将减少，从而将用较少的投入给客户提供相同级别的产品，因此，生产率得到了提高。

由于应用领域、问题复杂程度、项目组的结构和大小、项目期限、可应用的技术等许多因素都对项目组的生产率有影响，因此，不同开发组织对软件重用带来生产率提高的数字的报告并不相同，但基本上30%~50%的重用大约可以导致生产率提高25%~40%。

11.3 软件重用

3. 成本

软件重用带来的净成本节省可以用下式估算：

$$C = Cs - Cr - Cd$$

- **Cr**是与重用相关联的成本
 - (1) 领域分析与建模的成本
 - (2) 设计领域体系结构的成本
 - (3) 为便于重用而增加的文档的成本
 - (4) 维护和完善可重用的软件成分的成本
 - (5) 为从外部获取构件所付出的版税和许可证费用
 - (6) 创建（或购买）及运行重用库的费用
 - (7) 对设计和实现可重用构件的人员的培训费用
- **Cd**是交付给客户的软件的实际成本
- **Cs**使用本书第13章讲述的技术来估算

主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
- 11.3 软件重用
-  11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统

- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.4 系统分解

分而治之，各个击破

软件工程师在设计比较复杂的应用系统时普遍采用的策略，也是首先把系统分解成若干个比较小的部分，然后再分别设计每个部分。

系统的主要组成部分称为子系统

面向对象设计模型的4大组成部分可以想象成整个模型的4个垂直切片。



11.4 系统分解

1. 子系统之间的两种交互方式

(1) 客户-供应商关系

- 作为“客户”的子系统调用作为“供应商”的子系统，后者完成某些服务工作并返回结果。
- 前者必须了解后者的接口，然而后者却无须了解前者的接口，因为任何交互行为都是由前者驱动的。

(2) 平等伙伴关系

- 每个子系统都可能调用其他子系统，每个子系统都必须了解其他子系统的接口。
- 子系统之间的交互更复杂，这种交互方式还可能存在通信环路。

总地说来，单向交互比双向交互更容易理解，也更容易设计和修改，因此应该尽量使用客户-供应商关系

11.4 系统分解

2. 组织系统的两种方案

(1) 层次组织

- 把软件系统组织成一个层次系统，每层是一个子系统。
- 上层在下层的基础上建立，下层为实现上层功能而提供必要的服务。
- 每一层内所包含的对象，彼此间相互独立，而处于不同层次上的对象，彼此间往往有关联。
- 在上、下层之间存在客户-供应商关系。低层相当于供应商，上层相当于客户。
- 层次结构又可进一步划分成两种模式：封闭式和开放式。

11.4 系统分解

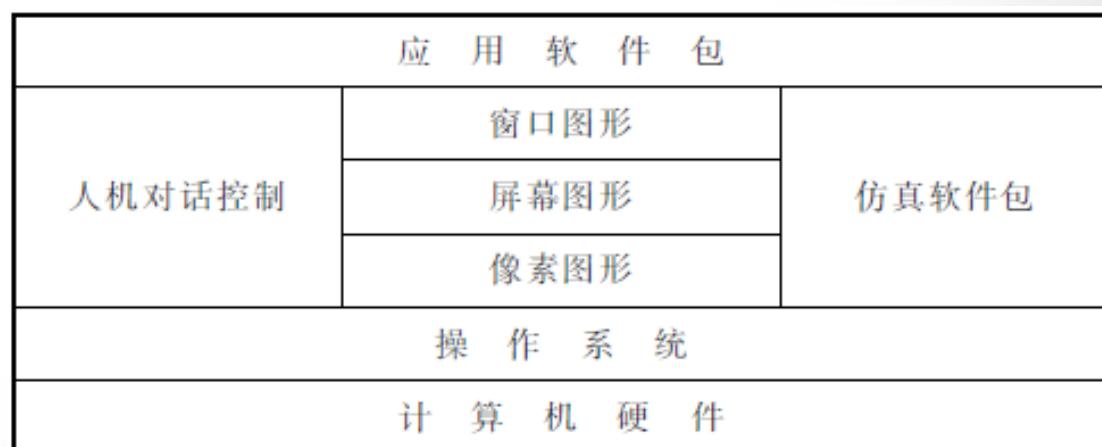
2. 组织系统的两种方案

(2) 块状组织

- 把软件系统垂直地分解成若干个相对独立的、弱耦合的子系统。
- 一个子系统相当于一块，每块提供一种类型的服务。

利用层次和块的各种可能的组合，可以成功地把多个子系统组成一个完整的软件系统。

右图表示一个混合使用层次与块状的应用系统的组织结构。



11.4 系统分解

2. 组织系统的两种方案

(3) 设计系统的拓扑结构

由子系统组成完整的系统时，典型的拓扑结构有管道形、树形、星形等。

设计者应该采用与问题结构相适应的、尽可能简单的拓扑结构，以减少子系统之间的交互数量。

主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
- 11.3 软件重用
- 11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统

- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.5 设计问题域子系统

在面向对象设计过程中，可能对面向对象分析所得出的问题域模型做补充或修改

1. 调整需求
2. 重用已有的类
3. 把问题域类组合在一起
4. 添加一般化类以建立协议
5. 调整继承类层次
6. ATM系统实例

11.5 设计问题域子系统

1. 调整需求

两种情况会导致修改通过面向对象分析所确定的系统需求

- 用户需求或外部环境发生了变化。
- 分析员对问题域理解不透彻或缺乏领域专家帮助，以致面向对象分析模型不能完整、准确地反映用户的真实需求。

11.5 设计问题域子系统

2. 重用已有的类

如果有可能重用已有的类，则重用已有类的典型过程如下

- ① 选择有可能被重用的已有类，标出这些候选类中对本问题无用的属性和服务，尽量重用那些能使无用的属性和服务降到最低程度的类。
- ② 在被重用的已有类和问题域类之间添加泛化关系(即从被重用的已有类派生出问题域类)。
- ③ 标出问题域类中从已有类继承来的属性和服务，现在已经无须在问题域类内定义它们了。
- ④ 修改与问题域类相关的关联，必要时改为与被重用的已有类相关的关联。

11.5 设计问题域子系统

3. 把问题域类组合在一起

在面向对象设计过程中，设计者往往通过引入一个根类而把问题域类组合在一起。

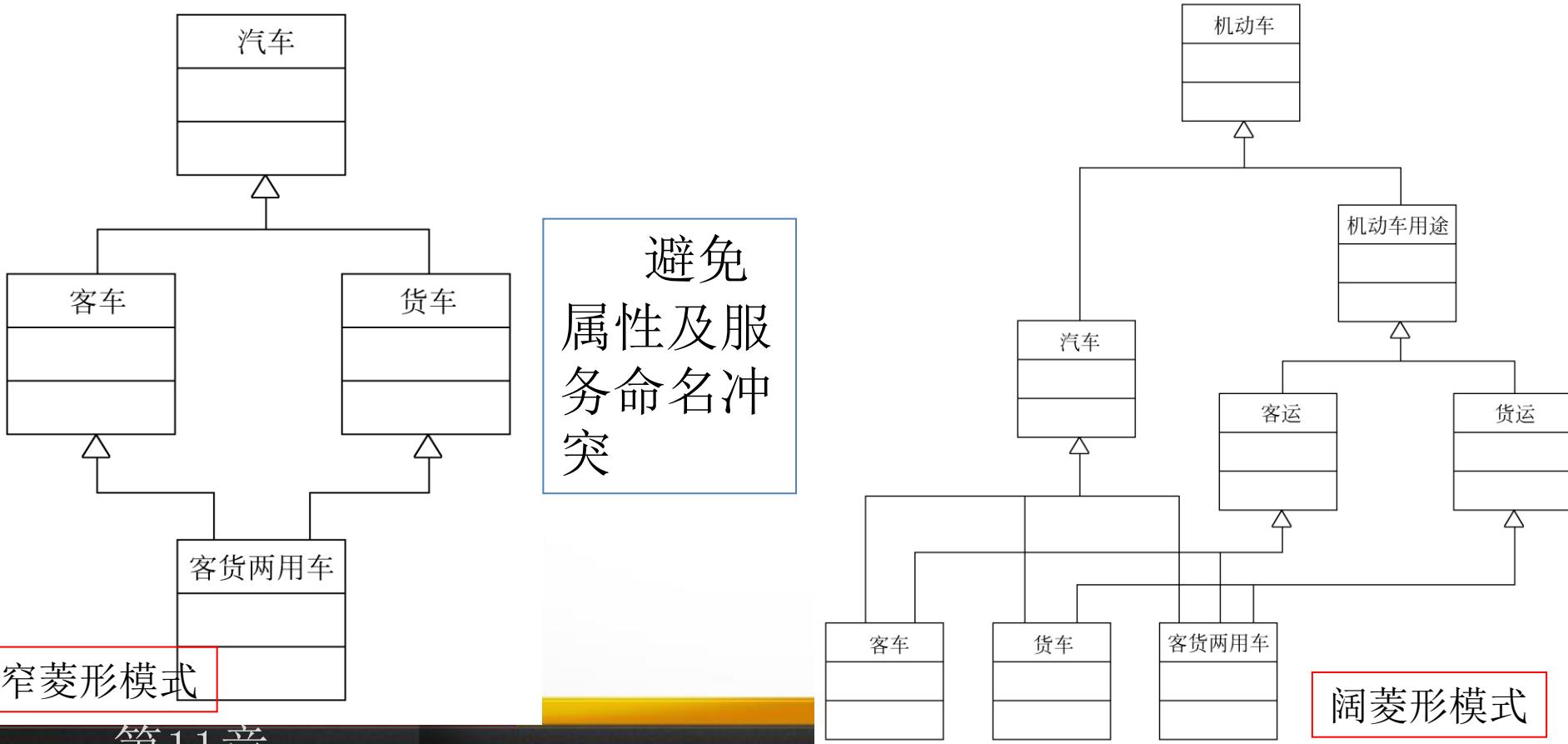
4. 添加一般化类以建立协议

一些具体类需要有一个公共的协议，也就是说，它们都需要定义一组类似的服务(很可能还需要相应的属性)。在这种情况下可以引入一个附加类(例如根类)，以便建立这个协议。

11.5 设计问题域子系统

5. 调整继承类层次

(1) 使用多重继承机制



窄菱形模式

阔菱形模式

11.5 设计问题域子系统

5. 调整继承类层次

(1) 使用多重继承机制

如果面向对象分析模型中包含了多重继承关系，然而所使用的程序设计语言却并不提供多重继承机制，则必须修改面向对象分析的结果。即使使用支持多重继承的语言，有时也会出于实现考虑而对面向对象分析结果作一些调整。

下面分情况讨论。

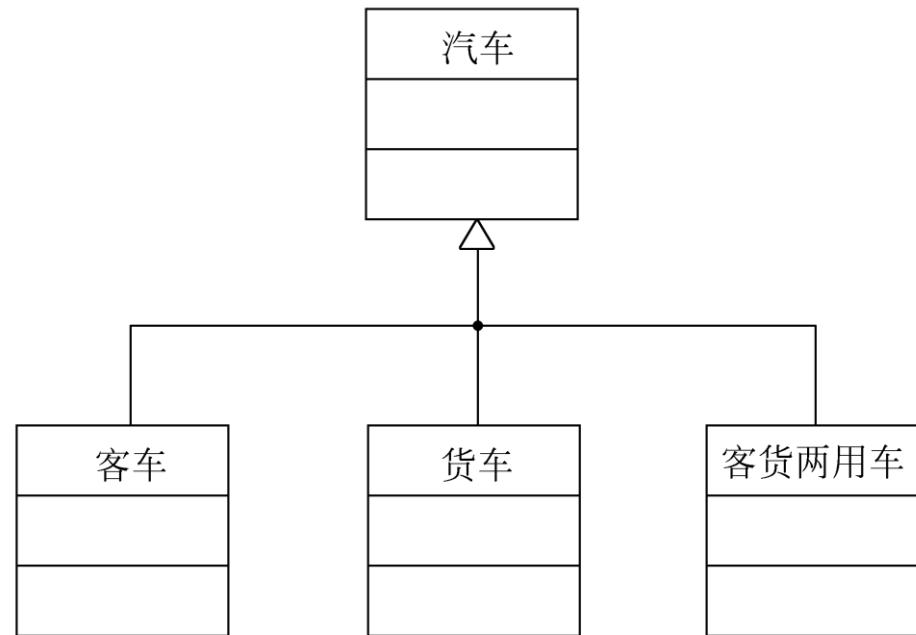
- 窄菱形模式，出现属性及服务命名冲突的可能性比较大；
- 阔菱形模式，属性及服务的名字发生冲突的可能性比较小，但是，它需要用更多的类才能表示同一个设计。

11.5 设计问题域子系统

5. 调整继承类层次

(2) 使用单重继承机制

如果打算使用仅提供单继承机制的语言实现系统，则必须把面向对象分析模型中的多重继承结构转换成单继承结构。



显然，在多重继承结构中的某些继承关系，经简化后将不再存在，这表明需要在各个具体类中重复定义某些属性和服务。

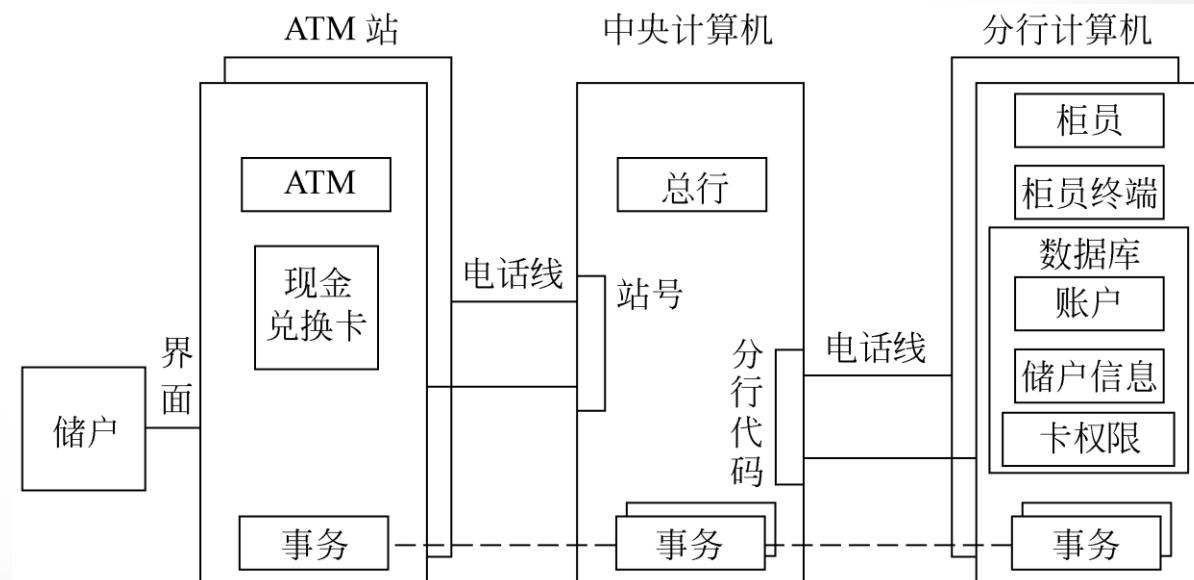
11.5 设计问题域子系统

6. ATM系统实例

ATM三个子系统为星形拓扑结构；物理联结用专用电话线实现；根据ATM站号和分行代码，区分由每个ATM站和每台分行计算机联向中央计算机的电话线。

在面向对象分析过程中已经对ATM系统做了相当仔细的分析，而且假设所使用的实现环境能完全支持模型的实现。

因此，在设计阶段无须对已有的问题域模型作实质性的修改或扩充。



主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
- 11.3 软件重用
- 11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统

- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.6 设计人机交互子系统

1. 分类用户
2. 描述用户
3. 设计命令层次
4. 设计人机交互类

11.6 设计人机交互子系统

1. 分类用户

为了更好地了解用户的需要与爱好，以便设计出符合用户需要的界面，设计者首先应该把将来可能与系统交互的用户分类。

- 按技能水平分类(新手、初级、中级、高级)。
- 按职务分类(总经理、经理、职员)。
- 按所属集团分类(职员、顾客)。

11.6 设计人机交互子系统

2. 描述用户

了解将来使用系统的每类用户的情况

- 用户类型。
- 使用系统欲达到的目的。
- 特征(年龄、性别、受教育程度、限制因素等)。
- 关键的成功因素(需求、爱好、习惯等)。
- 技能水平。
- 完成本职工作的脚本。

11.6 设计人机交互子系统

3. 设计命令层次

(1) 研究现有的人机交互含义和准则

Windows已经成了微机上图形用户界面事实上的工业标准

- 基本外观及给用户的感受都是相同的
 - (1) 每个程序至少有一个窗口，它由标题栏标识；
 - (2) 程序中大多数功能可通过菜单选用；
 - (3) 选中某些菜单项会弹出对话框，用户可通过它输入附加信息；
 - (4)
- 广大用户习以为常的许多约定
 - (1) File菜单的最后一个菜单项是Exit；
 - (2) 在文件列表框中用鼠标单击某个表项，则相应的文件名变亮，若用鼠标双击则会打开该文件；
 - (3)

11.6 设计人机交互子系统

3. 设计命令层次

(2) 确定初始的命令层次

- 所谓命令层次，实质上是用过程抽象机制组织起来的、可供选用的服务的表示形式。
- 设计命令层次时，通常先从对服务的过程抽象着手，然后再进一步修改它们，以适合具体应用环境的需要。

(3) 精化命令层次

- 次序
- 整体-部分关系
- 宽度和深度
- 操作步骤

11.6 设计人机交互子系统

4. 设计人机交互类

人机交互类与所使用的操作系统及编程语言密切相关

例如，在Windows环境下运行的Visual C++语言提供了MFC类库，设计人机交互类时，往往仅需从MFC类库中选出一些适用的类，然后从这些类派生出符合自己需要的类就可以了。

主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
- 11.3 软件重用
- 11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统



- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.7 设计任务管理子系统

设计工作的一项重要内容就是，确定哪些是必须同时动作的对象，哪些是相互排斥的对象。然后进一步设计任务管理子系统。

1. 分析并发性

- 如果两个对象彼此间不存在交互，或者它们同时接受事件，则这两个对象在本质上是并发的。
- 通过检查各个对象的状态图及它们之间交换的事件，能够把若干个非并发的对象归并到一条控制线中。
- 在计算机系统中用任务(task)实现控制线，一般认为任务是进程(process)的别名。通常把多个任务的并发执行称为多任务。

划分任务，可以简化系统的设计及编码工作。这种并发行为既可以在不同的处理器上实现，也可以在单个处理器上利用多任务操作系统仿真实现。

11.7 设计任务管理子系统

2. 设计任务管理子系统

- (1) 确定事件驱动型任务
- (2) 确定时钟驱动型任务
- (3) 确定优先任务
- (4) 确定关键任务
- (5) 确定协调任务
- (6) 尽量减少任务数
- (7) 确定资源需求

11.7 设计任务管理子系统

2. 设计任务管理子系统

(1) 确定事件驱动型任务

事件驱动任务可能主要完成通信工作

工作过程如下：

- 任务处于睡眠状态(不消耗处理器时间), 等待来自数据线或其他数据源的中断;
- 接收到中断唤醒该任务, 接收数据并放入内存缓冲区或其他目的地, 通知需要知道这件事的对象;
- 该任务又回到睡眠状态。

11.7 设计任务管理子系统

2. 设计任务管理子系统

(2) 确定时钟驱动型任务

任务每隔一定时间间隔就被触发以执行某些处理

工作过程如下：

- 任务设置了唤醒时间后进入睡眠状态；
- 任务睡眠(不消耗处理器时间)，等待来自系统的中断；
- 一旦接收到这种中断，任务就被唤醒并做它的工
作，通知有关的对象，然后该任务又回到睡眠状
态。

11.7 设计任务管理子系统

2. 设计任务管理子系统

(3) 确定优先任务

优先任务可以满足高优先级或低优先级的处理需求

- 高优先级：某些服务具有很高的优先级，为了在严格限定的时间内完成这种服务，可能需要把这类服务分离成独立的任务。
- 低优先级：与高优先级相反，有些服务是低优先级的，属于低优先级处理(通常指那些背景处理)。设计时可能用额外的任务把这样的处理分离出来。

11.7 设计任务管理子系统

2. 设计任务管理子系统

(4) 确定关键任务

- 关键任务是有关系统成功或失败的关键处理，这类处理通常都有严格的可靠性要求。
- 在设计过程中可能用额外的任务把这样的关键处理分离出来，以满足高可靠性处理的要求。
- 对高可靠性处理应该精心设计和编码，并且应该严格测试。

11.7 设计任务管理子系统

2. 设计任务管理子系统

(5) 确定协调任务

- 当系统中存在3个以上任务时，就应该增加一个任务，用它作为协调任务。
- 引入协调任务会增加系统的总开销(增加从一个任务到另一个任务的转换时间)，但是引入协调任务有助于把不同任务之间的协调控制封装起来。
- 使用状态转换矩阵可以比较方便地描述该任务的行为。
- 这类任务应该仅做协调工作，不要让它再承担其他服务工作。

11.7 设计任务管理子系统

2. 设计任务管理子系统

(6) 尽量减少任务数

必须仔细分析和选择每个确实需要的任务，使系统中包含的任务数尽量少。

(7) 确定资源需求

使用多处理器或固件，主要是为了满足高性能的需求。设计者必须通过计算系统载荷(即每秒处理的业务数及处理一个业务所花费的时间)，来估算所需要的CPU(或其他固件)的处理能力。

11.7 设计任务管理子系统

2. 设计任务管理子系统

设计者应该综合考虑各种因素，以决定哪些子系统用硬件实现，哪些子系统用软件实现。

使用硬件实现某些子系统的主要原因可能是：

- 现有的硬件完全能满足某些方面的需求，例如，买一块浮点运算卡比用软件实现浮点运算要容易得多。
- 专用硬件比通用的CPU性能更高。例如，目前在信号处理系统中广泛使用固件实现快速傅里叶变换。

设计者在决定到底采用软件还是硬件的时候，必须综合权衡一致性、成本、性能等多种因素，还要考虑未来的可扩充性和可修改性。

主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
- 11.3 软件重用
- 11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统

- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.8 设计数据管理子系统

数据管理子系统是系统存储或检索对象的基本设施，它建立在某种数据存储管理系统之上，并且隔离了数据存储管理模式(文件、关系数据库或面向对象数据库)的影响。

11.8.1 选择数据存储管理模式

1. 文件管理系统

- 文件管理系统是操作系统的一个组成部分，使用它长期保存数据具有成本低和简单等特点，
- 但是，文件操作的级别低，为提供适当的抽象级别还必须编写额外的代码。
- 此外，不同操作系统的文件管理系统往往有明显差异。

11.8 设计数据管理子系统

2. 关系数据库管理系统

- 关系数据库管理系统的理论基础是关系代数，它不仅理论基础坚实而且有下列一些主要优点
 - (1) 提供了各种最基本的数据管理功能
 - (2) 为多种应用提供了一致的接口
 - (3) 标准化的语言(SQL语言)
- 为了做到通用与一致，关系数据库管理系统通常都相当复杂，且有下述一些具体缺点
 - (1) 运行开销大
 - (2) 不能满足高级应用的需求
 - (3) 与程序设计语言的连接不自然

11.8 设计数据管理子系统

3. 面向对象数据库管理系统

面向对象数据库管理系统主要有两种设计途径

- 扩展的关系数据库管理系统

- (1)在关系数据库的基础上，增加了抽象数据类型和继承机制。
(2)增加了创建及管理类和对象的通用服务。

- 扩展的面向对象程序设计语言

- (1)扩充了面向对象程序设计语言的语法和功能，增加了在数据库中存储和管理对象的机制。

- (2)开发人员可以用统一的面向对象观点进行设计，不再需要区分存储数据结构和程序数据结构(即生命周期短暂的数据)。

主要内容

- 11.1 面向对象设计的准则
- 11.2 启发规则
- 11.3 软件重用
- 11.4 系统分解
- 11.5 设计问题域子系统
- 11.6 设计人机交互子系统

- 11.7 设计任务管理子系统
- 11.8 设计数据管理子系统
- 11.9 设计类中的服务
- 11.10 设计关联
- 11.11 设计优化

11.9 设计类中的服务

面向对象分析得出的对象模型，通常并不详细描述类中的服务。面向对象设计则是扩充、完善和细化面向对象分析模型的过程，设计类中的服务是它的一项重要工作内容。

11.9.1 确定类中应有的服务

确定类中应有的服务需要综合考虑对象模型、动态模型和功能模型，才能正确确定类中应有的服务。对象模型是进行对象设计的基本框架。

11.9 设计类中的服务

- 一张状态图描绘了一类对象的生命周期，图中的状态转换是执行对象服务的结果。
- 功能模型指明了系统必须提供的服务。
- 状态图中状态转换所触发的动作，在功能模型中有时可能扩展成一张数据流图。
- 数据流图中的某些处理可能与对象提供的服务相对应，有一些规则有助于确定操作的目标对象(即应该在该对象所属的类中定义这个服务)。
- 当一个处理涉及多个对象时，通常在起主要作用的对象类中定义这个服务。

11.9 设计类中的服务

11.9.2 设计实现服务的方法

1. 设计实现服务的算法

应该考虑下列几个因素：

(1) 算法复杂度。

通常选用复杂度较低(即效率较高)的算法，但也不要过分追求高效率，应以能满足用户需求为准。

(2) 容易理解与容易实现。

容易理解与容易实现的要求往往与高效率有矛盾，设计者应该对这两个因素适当折衷。

(3) 易修改。

应该尽可能预测将来可能做的修改，并在设计时预先做些准备。

11.9 设计类中的服务

2. 选择数据结构

在分析阶段，仅需考虑系统中需要的信息的逻辑结构，在面向对象设计过程中，则需要选择能够方便、有效地实现算法的物理数据结构。

3. 算法与数据结构的关系

设计阶段是解决“怎么做”的时候了，因此，确定实现服务方法中所需要的算法与数据结构是非常关键的。主要考虑下列因素：

- (1) 分析问题寻找数据特点，提炼出所有可行有效的算法；
- (2) 定义与所提炼算法相关联的数据结构；
- (3) 依据此数据结构进行算法的详细设计；
- (4) 进行一定规模的实验与评测；
- (5) 确定最佳设计。

11.9 设计类中的服务

4. 定义内部类和内部操作

在面向对象设计过程中，可能需要增添一些在需求陈述中没有提到的类，这些新增加的类，主要用来存放在执行算法过程中所得出的某些中间结果。

此外，复杂操作往往可以用简单对象上的更低层操作来定义。因此，在分解高层操作时常常引入新的低层操作。在面向对象设计过程中应该定义这些新增加的低层操作。

主要内容

- | | |
|----------------|----------------|
| 11.1 面向对象设计的准则 | 11.7 设计任务管理子系统 |
| 11.2 启发规则 | 11.8 设计数据管理子系统 |
| 11.3 软件重用 | 11.9 设计类中的服务 |
| 11.4 系统分解 | 11.10 设计关联 |
| 11.5 设计问题域子系统 | 11.11 设计优化 |
| 11.6 设计人机交互子系统 | |
- 

11.10 设计关联

1. 关联的遍历
2. 实现单向关联
3. 实现双向关联
4. 关联对象的实现

在对象模型中，关联是联结不同对象的纽带，它指定了对象相互间的访问路径。在面向对象设计过程中，设计人员必须确定实现关联的具体策略。既可以选定一个全局性的策略统一实现所有关联，也可以分别为每个关联选择具体的实现策略，以与它在应用系统中的使用方式相适应。

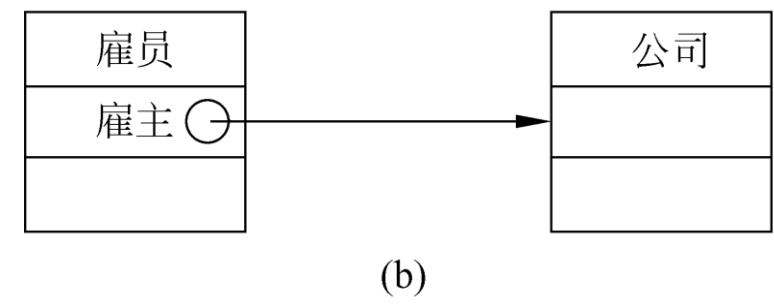
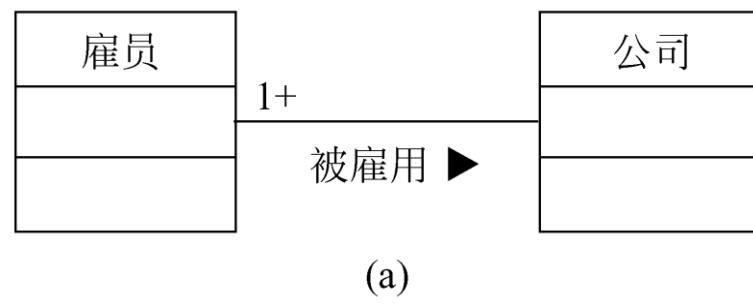
11.10 设计关联

1. 关联的遍历

- 在应用系统中，使用关联有两种可能的方式：单向遍历和双向遍历。
- 在使用原型法开发软件的时候，原型中所有关联都应该是双向的，以便于增加新的行为，快速地扩充和修改原型。

2. 实现单向关联

用指针可以方便地实现单向关联。如果关联的重数是一元的(如图a所示)，则实现关联的指针是一个简单指针；如果重数是多元的，则需要用一个指针集合实现关联(参见图b)。



11.10 设计关联

3. 实现双向关联

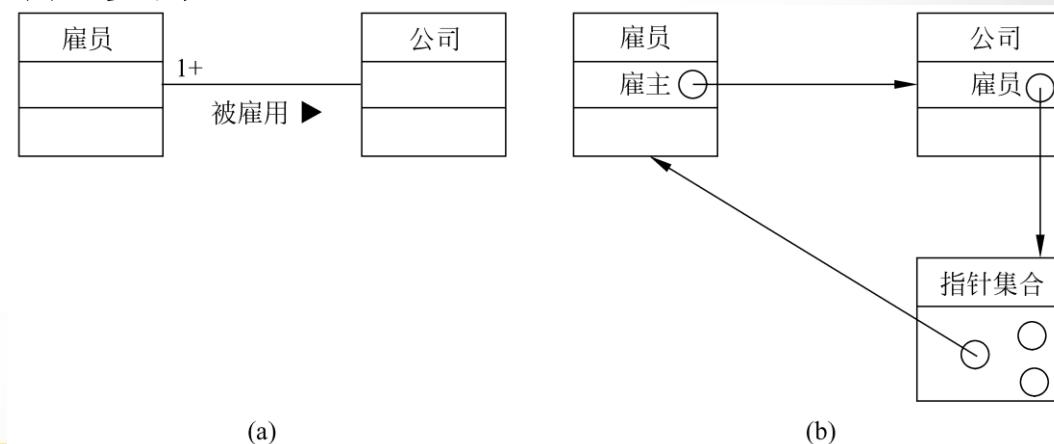
实现双向关联有下列3种方法：

(1) 只用属性实现一个方向的关联

当需要反向遍历时就执行一次正向查找。如果两个方向遍历的频度相差很大，而且需要尽量减少存储开销和修改时的开销，则这是一种很有效的实现双向关联的方法。

(2) 两个方向的关联都用属性实现

这种方法能实现快速访问，但是，如果修改了一个属性，则相关的属性也必须随之修改，才能保持该关联链的一致性。当访问次数远远多于修改次数时，这种实现方法很有效。



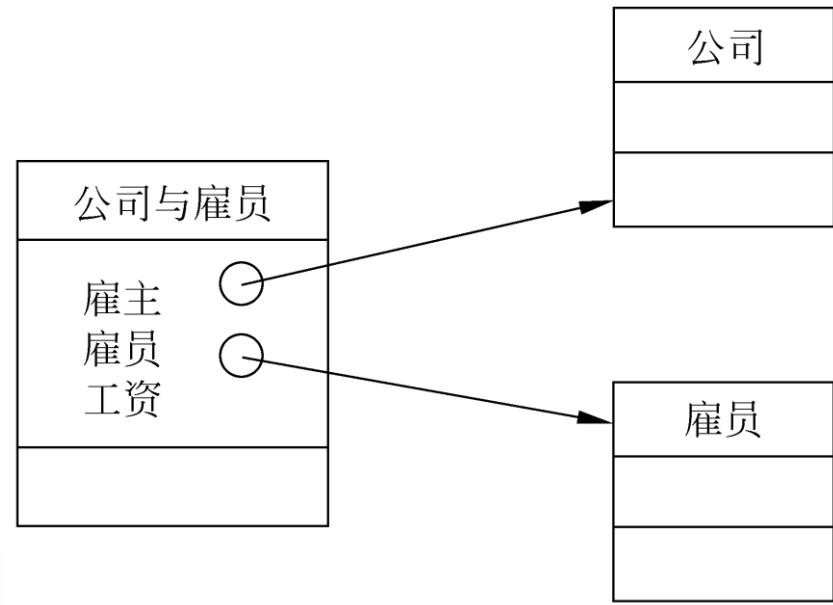
11.10 设计关联

3. 实现双向关联

实现双向关联有下列3种方法：

(3) 用独立的关联对象实现双向关联

关联对象不属于相互关联的任何一个类，它是独立的关联类的实例。如右图所示。



11.10 设计关联

4. 关联对象的实现

9.4.2节曾经讲过，可以引入一个关联类来保存描述关联性质的信息，关联中的每个连接对应着关联类的一个对象。

实现关联对象的方法取决于关联的重数。

- 对于一对一关联来说，关联对象可以与参与关联的任一个对象合并。
- 对于一对多关联来说，关联对象可以与“多”端对象合并
- 如果是多对多关联，则关联链的性质不可能只与一个参与关联的对象有关，通常用一个独立的关联类来保存描述关联性质的信息，这个类的每个实例表示一条具体的关联链及该链的属性（参见上页图）。

主要内容

- | | |
|----------------|----------------|
| 11.1 面向对象设计的准则 | 11.7 设计任务管理子系统 |
| 11.2 启发规则 | 11.8 设计数据管理子系统 |
| 11.3 软件重用 | 11.9 设计类中的服务 |
| 11.4 系统分解 | 11.10 设计关联 |
| 11.5 设计问题域子系统 | 11.11 设计优化 |
| 11.6 设计人机交互子系统 | |
- 

11.11 设计优化

11.11.1 确定优先级

11.11.2 提高效率的几项技术

11.11.3 调整继承关系

11.11 设计优化

11.11.1 确定优先级

系统的各项质量指标并不是同等重要的，设计人员必须确定各项质量指标的相对重要性(即确定优先级)，以便在优化设计时制定折衷方案。

- 系统的整体质量与设计人员所制定的折衷方案密切相关。最终产品成功与否，在很大程度上取决于是否选择好了系统目标。
- 在折衷方案中设置的优先级应该是模糊的。事实上，不可能指定精确的优先级数值(例如速度48%，内存25%，费用8%，可修改性19%)。
- 最常见的情况，是在效率和清晰性之间寻求适当的折衷方案。

下面两小节分别讲述在优化设计时提高效率的技术，以及建立良好的继承结构的方法。

11.11 设计优化

11.11.2 提高效率的几项技术

1. 增加冗余关联以提高访问效率

- 在面向对象分析过程中，应该避免在对象模型中存在冗余的关联，因为冗余关联不仅没有增添任何信息，反而会降低模型的清晰程度。
- 但是，在面向对象设计过程中，当考虑用户的访问模式，及不同类型的访问彼此间的依赖关系时，就会发现，分析阶段确定的关联可能并没有构成效率最高的访问路径。
- 下面用设计公司雇员技能数据库的例子，说明分析访问路径及提高访问效率的方法。

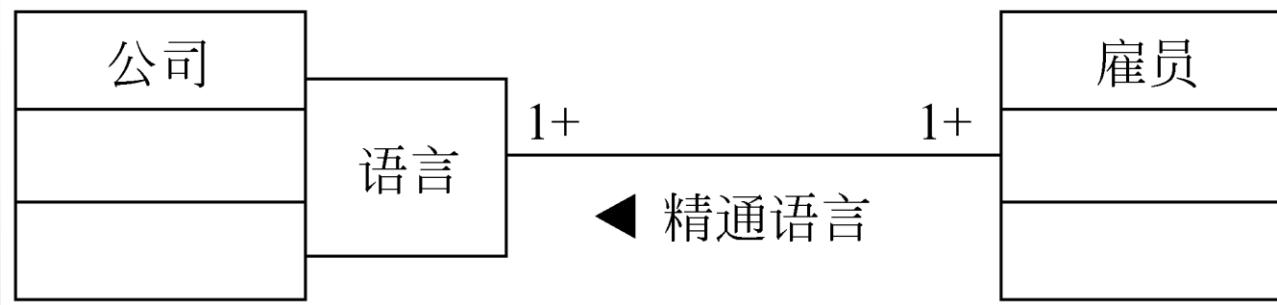
下图是从面向对象分析模型中摘取的一部分。公司类中的服务 `find_skill` 返回具有指定技能的雇员集合。例如，用户可能询问公司中会讲日语的雇员有哪些人。



- 假设某公司共有2000名雇员，平均每名雇员会10种技能，则简单的嵌套查询将遍历雇员对象2000次，针对每名雇员平均再遍历技能对象10次。如果全公司仅有5名雇员精通日语，则查询命中率仅有1/4000。
- 提高访问效率的一种方法是使用哈希(Hash)表：“具有技能”这个关联不再利用无序表实现，而是改用哈希表实现。只要“会讲日语”是用唯一一个技能对象表示，这样改进后就会使查询次数由20000次减少到2000次。

但是，当仅有极少数对象满足查询条件时，查询命中率仍然很低。在这种情况下，更有效的提高查询效率的改进方法是，给那些需要经常查询的对象建立索引。

例如，针对上述例子，可以增加一个额外的限定关联“精通语言”，用来联系公司与雇员这两类对象，如下图所示。利用适当的冗余关联，可以立即查到精通某种具体语言的雇员，而无须多余的访问。当然，索引也必然带来开销：占用内存空间，而且每当修改基关联时也必须相应地修改索引。因此，应该只给那些经常执行并且开销大、命中率低的查询建立索引。



11.11 设计优化

2. 调整查询次序

改进了对象模型的结构，从而优化了常用的遍历之后，接下来就应该优化算法了。

优化算法的一个途径是尽量缩小查找范围。

例如，假设用户在使用上述的雇员技能数据库的过程中，希望找出既会讲日语又会讲法语的所有雇员。如果某公司只有5位雇员会讲日语，会讲法语的雇员却有200人，则应该先查找会讲日语的雇员，然后再从这些会讲日语的雇员中查找同时又会讲法语的人。

11.11 设计优化

3. 保留派生属性

- 通过某种运算而从其他数据派生出来的数据，是一种冗余数据。通常把这类数据“存储”（或称为“隐藏”）在计算它的表达式中。如果希望避免重复计算复杂表达式所带来的开销，可以把这类冗余数据作为派生属性保存起来。
- 派生属性既可以在原有类中定义，也可以定义新类，并用新类的对象保存它们。每当修改了基本对象之后，所有依赖于它的、保存派生属性的对象也必须相应地修改。

11.11 设计优化

11.11.3 调整继承关系

- 在面向对象设计过程中，建立良好的继承关系是优化设计的一项重要内容。继承关系能够为一个类族定义一个协议，并能在类之间实现代码共享以减少冗余。
- 一个基类和它的子孙类在一起称为一个类继承。在面向对象设计中，建立良好的类继承是非常重要的。利用类继承能够把若干个类组织成一个逻辑结构。

下面讨论与建立类继承有关的问题。

1. 抽象与具体
2. 为提高继承程度而修改类定义
3. 利用委托实现行为共享

11.11 设计优化

1. 抽象与具体

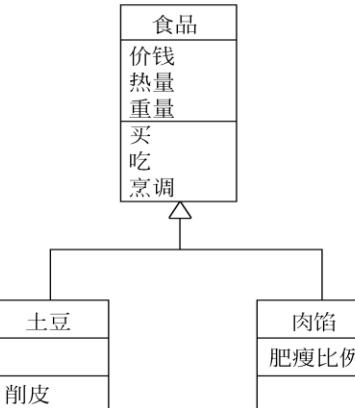
- 在设计类继承时，很少使用纯粹自顶向下的方法。
- 通常的作法是，首先创建一些满足具体用途的类，然后对它们进行归纳，一旦归纳出一些通用的类以后，往往可以根据需要再派生出具体类。
- 在进行了一些具体化(即专门化)的工作之后，也许就应该再次归纳了。对于某些类继承来说，这是一个持续不断的演化过程。

右图为一个人们在日常生活中熟悉的设计类继承的例子，说明上述从具体到抽象，再到具体的过程。

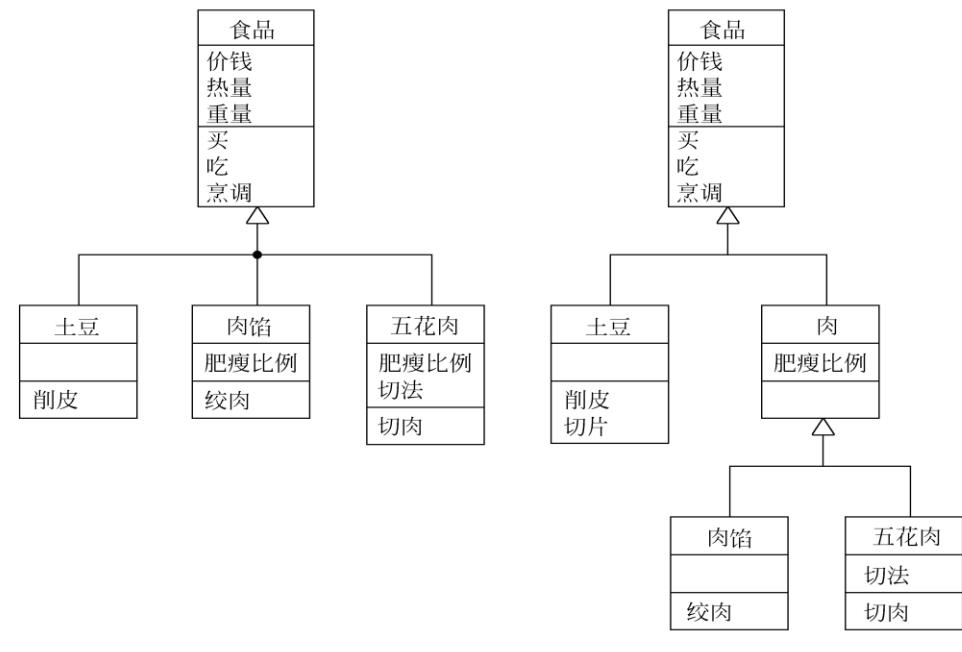
- (a) 先创建一些具体类；
- (b) 归纳出抽象类；
- (c) 进一步具体化；
- (d) 再次归纳

肉馅	土豆
价钱	价钱
热量	热量
重量	重量
肥瘦比例	
买	买
吃	吃
烹调	烹调

(a)



(b)



11.11 设计优化

2. 为提高继承程度而修改类定义

- 如果在一组相似的类中存在公共的属性和公共的行为，则可以把这些公共的属性和行为抽取出来放在一个共同的祖先类中，供其子类继承，如上图(a)和(b)所示。
- 在对现有类进行归纳的时候，要注意下述两点：
 - (1)不能违背领域知识和常识；
 - (2)应该确保现有类的协议(即同外部世界的接口)不变。
- 更常见的情况是，各个现有类中的属性和行为(操作)，虽然相似却并不完全相同，在这种情况下需要对类的定义稍加修改，才能定义一个基类供其子类从中继承需要的属性或行为。

有时抽象出一个基类之后，在系统中暂时只有一个子类能从它继承属性和行为，显然，在当前情况下抽象出这个基类并没有获得共享的好处。但是，这样做通常仍然是值得的，因为将来可能重用这个基类。

11.11 设计优化

3. 利用委托实现行为共享

仅当存在真实的一般-特殊关系(即子类确实是父类的一种特殊形式)时，利用继承机制实现行为共享才是合理的。

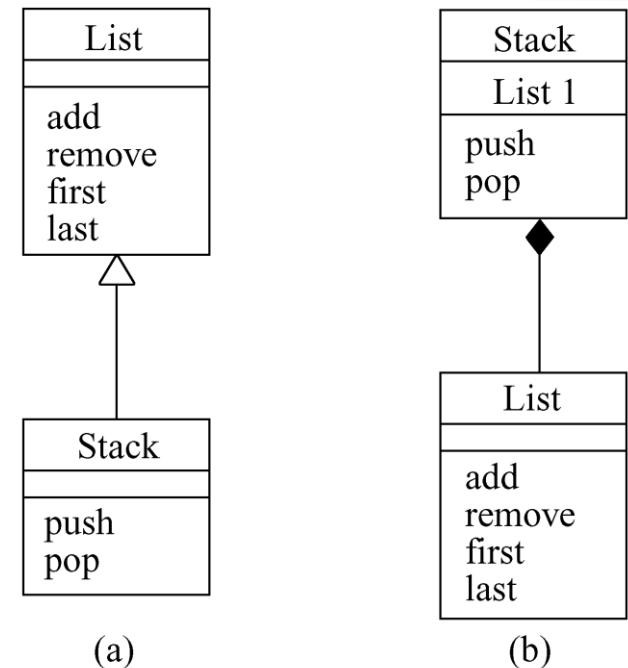
有时程序员只想用继承作为实现操作共享的一种手段，并不打算确保基类和派生类具有相同的行为。在这种情况下，如果从基类继承的操作中包含了子类不应有的行为，则可能引起麻烦。

11.11 设计优化

3. 利用委托实现行为共享

例如，假设程序员正在实现一个 Stack(后进先出栈)类，类库中已经有一个 List(表)类。

- 如果程序员从 List 类派生出 Stack 类，则如右图 (a) 所示：把一个元素压入栈，等价于在表尾加入一个元素；把一个元素弹出栈，相当于从表尾移走一个元素。
- 但是，与此同时，也继承了一些不需要的表操作。例如，从表头移走一个元素或在表头增加一个元素。万一用户错误地使用了这类操作，Stack 类将不能正常工作。



11.11 设计优化

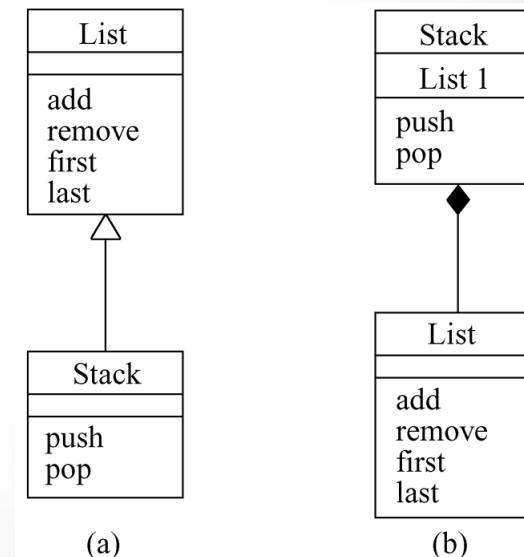
3. 利用委托实现行为共享

如果只想把继承作为实现操作共享的一种手段，则利用**委托**(即把一类对象作为另一类对象的属性，从而在两类对象间建立组合关系)也可以达到同样目的，而且这种方法更安全。

使用委托机制时，只有有意义的操作才委托另一类对象实现，因此，不会发生不慎继承了无意义(甚至有害)操作的问题。

右图(b)描绘了委托List类实现Stack类操作的方法。

Stack类的每个实例都包含一个私有的List类实例(或指向List类实例的指针)。Stack对象的操作push(压栈)，委托List类对象通过调用last(定位到表尾)和add(加入一个元素)操作实现，而pop(出栈)操作则通过List的last和remove(移走一个元素)操作实现。



本章小结

1. 结合面向对象方法学固有的特点讲述了面向对象设计准则，并介绍了一些有助于提高设计质量的启发式规则。
2. 结合面向对象方法学的特点，对软件重用做了较全面的介绍，其中着重讲述了类构件重用技术。
3. 大多数求解空间模型，在逻辑上由4大部分组成。分别讲述了问题域子系统、人机交互子系统、任务管理子系统和数据管理子系统的设计方法。此外还讲述了设计类中服务的方法及实现关联的策略。
4. 通常应该在设计工作开始之前，对系统的各项质量指标的相对重要性做认真分析和仔细权衡，制定出恰当的系统目标。在设计过程中根据既定的系统目标，做必要的优化工作。

本章结束

