第11章 面向对象设计

11.1 面向对象设计的准则

- 所谓优秀设计，就是权衡了各种因素，从而使得系统在其整个生命周期中的总开销最小的设计。
- 对大多数软件系统而言，60%以上的软件费用都用于软件维护，因此，优秀软件设计的一个主要特点就是容易维护。
- 设计准则有6条。

1. 模块化

- 对象就是模块。它是把数据结构和操作这些数据的方法紧密地结合在一起所构成的模块。

2. 抽象

- 面向对象方法不仅支持过程抽象，而且支持数据抽象。

3. 信息隐藏

- 在面向对象方法中，信息隐藏通过对象的封装性实现。

4. 弱耦合

- 耦合指不同对象之间相互关联的紧密程度。
- 一般说来，对象之间的耦合可分为两大类：
  - 交互耦合
    - 如果对象之间的耦合通过消息连接来实现，则这种耦合就是交互耦合。
    - 交互耦合应尽可能松散 。
  - 继承耦合
    - 与交互耦合相反，应该提高继承耦合程度。
    - 继承是一般化类与特殊类之间耦合的一种形式。通过继承关系结合起来的基类和派生类，构成了系统中粒度更大的模块。彼此之间应该越紧密越好。

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。
- 内聚定义为：设计中使用的一个构件内的各个元素，对完成一个定义明确的目的所做出的贡献程度。
- 在设计时应该力求做到高内聚。
- 在面向对象设计中存在下述3种内聚：
  - 服务内聚。一个服务应该完成一个且仅完成一个功能。
  - 类内聚。一个类应该只有一个用途，它的属性和服务应该是高内聚的。
  - 一般-特殊内聚。设计出的一般-特殊结构，应该符合多数人的概念。

6. 可重用

- 软件重用是提高软件开发生产率和目标系统质量的重要途径。
- 重用基本上从设计阶段开始。
- 重用有两方面的含义：
  - 一是尽量使用已有的类(包括开发环境提供的类库，及以往开发类似系统时创建的类)，
  - 二是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。

11.2 启发规则

1. 设计结果应该清晰易懂

- 用词一致。
- 使用已有的协议。
- 减少消息模式的数目。
- 避免模糊的定义。

2. 一般-特殊结构的深度应适当

- 应该使类等级中包含的层次数适当。一般说来，在一个中等规模(大约包含100个类)的系统中，类等级层次数应保持为7±2。

3. 设计简单的类

- 应该尽量设计小而简单的类，以便于开发和管理。
- 为使类保持简单，应该注意以下几点：
  - 避免包含过多的属性。
  - 有明确的定义。
  - 尽量简化对象之间的合作关系。
  - 不要提供太多服务。
  - 在开发大型软件系统时，设计出大量较小的类，需要划分“主题”。

4. 使用简单的协议

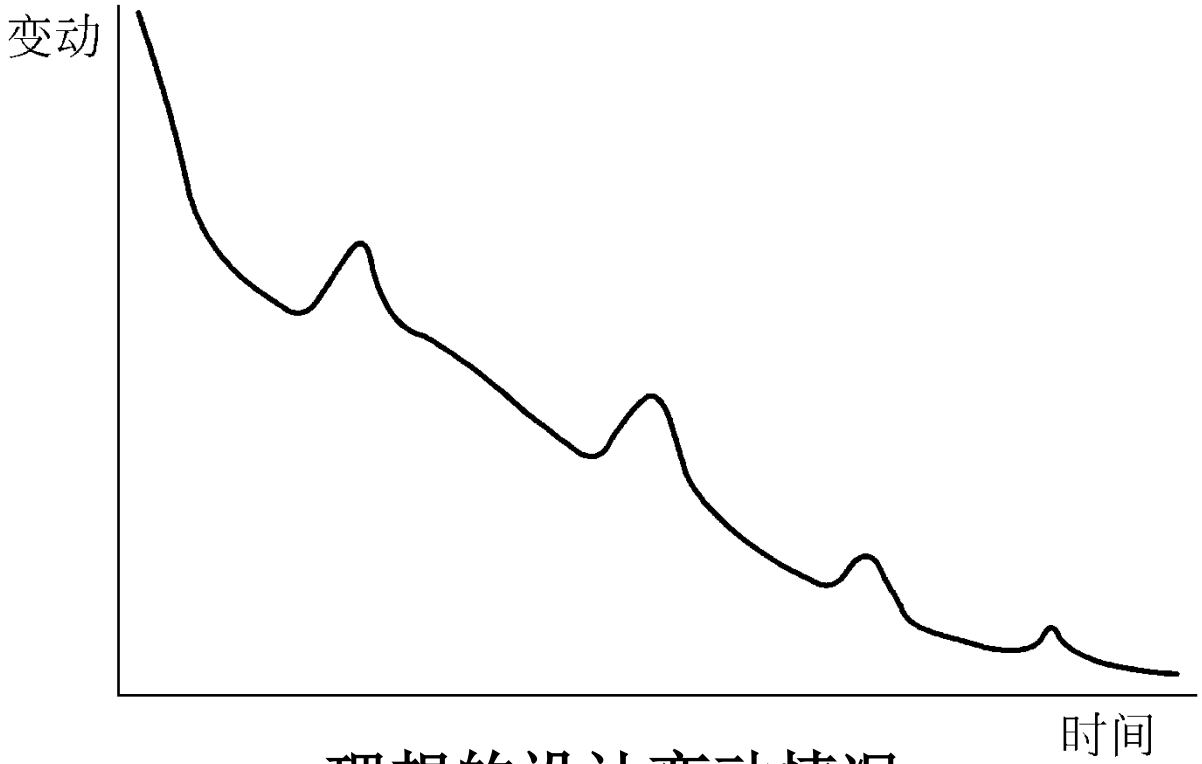
- 一般说来，消息中的参数不要超过3个。当然，不超过3个的限制也不是绝对的。

5. 使用简单的服务

- 一般说来，应该尽量避免使用复杂的服务。
- 如果一个服务中包含了过多的源程序语句，或者语句嵌套层次太多，或者使用了复杂的CASE语句，则应该仔细检查这个服务，设法分解或简化它，考虑用一般-特殊结构代替。

6. 把设计变动减至最小

- 出现必须修改设计的情况，应该使修改的范围尽可能小。



理想的设计变动情况

11.3 软件重用

11.3.1 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：
  - 知识重用(软件工程知识的重用)。
  - 方法和标准的重用(面向对象方法或国家制定的软件开发规范的重用)。
  - 软件成分的重用。

2. 软件成分的重用级别

- 代码重用：通常把它理解为调用库中的模块。代码重用的几种形式：
  - 源代码剪贴
  - 源代码包含
  - 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)。这个级别的重用有助于把一个应用系统移植到完全不同的软硬件平台上。
- 分析结果重用：重用某个系统的分析模型。这种重用特别适用于用户需求未改变，但系统体系结构发生了根本变化的场合。

3. 典型的可重用软件成分

- 项目计划
- 成本估计
- 体系结构
- 需求模型和规格说明
- 设计
- 源代码
- 用户文档和技术文档
- 用户界面
- 数据
- 测试用例

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，称之为类构件。

1. 可重用软构件应具备的特点
   - 模块独立性强
   - 具有高度可塑性
   - 接口清晰、简明、可靠
- 精心设计的“类”基本上能满足上述要求，可以认为它是可重用软构件的雏形。

2. 类构件的重用方式

- 实例重用
  - 由于类的封装性，使用者无须了解实现细节就可以使用适当的构造函数，按照需要创建类的实例。这是最基本的重用方式。
- 继承重用
  - 继承重用提供了一种安全地修改已有类构件，以便在当前系统中重用的手段。
- 多态重用
  - 利用多态性不仅可以使对象的对外接口更加一般化，从而降低了消息连接的复杂程度，而且还提供了一种简便可靠的软构件组合机制。

11.3.3 软件重用的效益

1. 质量

- 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。

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，生产率得到了提高。基本上30%~50%的重用大约可以导致生产率提高25%~40%。

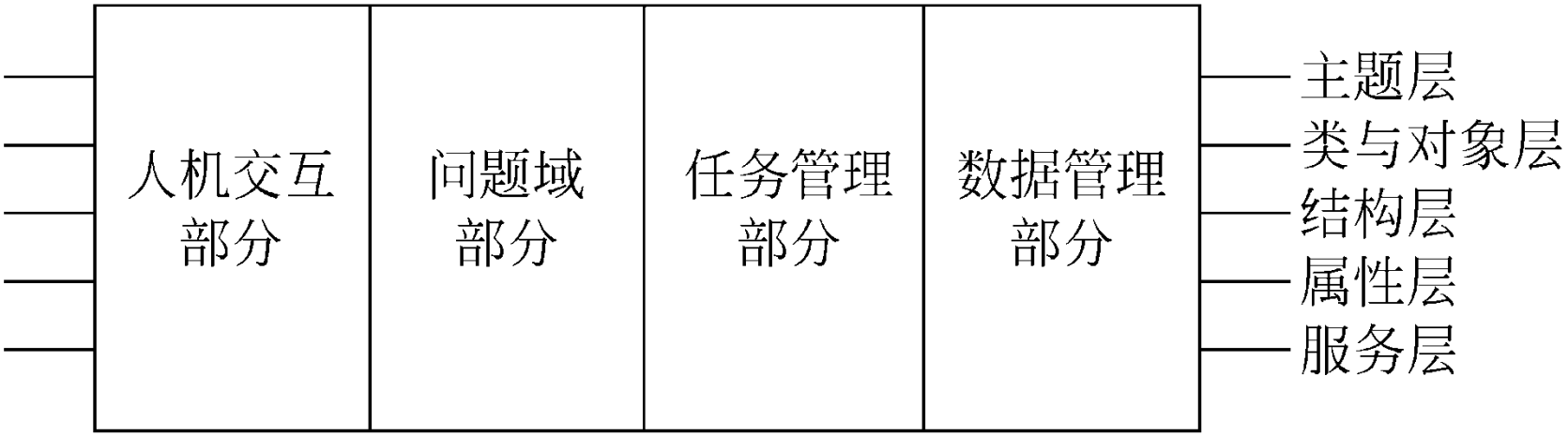
3. 成本

- 净成本节省可以用下式估算：$C=C_s-C_r-C_d$
- 其中，C_s是项目从头开发所需要的成本；C_r是与重用相关联的成本；C_d是交付给客户的软件的实际成本。

11.4 系统分解

- 系统的主要组成部分称为子系统。通常根据所提供的功能来划分子系统。一般说来，子系统的数目应该与系统规模基本匹配。
- 各个子系统之间应该具有尽可能简单、明确的接口。因此，可以相对独立地设计各个子系统。
- 在划分和设计子系统时，应该尽量减少子系统彼此间的依赖性。

- 面向对象设计模型，与面向对象分析模型一样，也由主题、类与对象、结构、属性、服务等5个层次组成。
- 大多数系统的面向对象设计模型，在逻辑上都由4大部分组成。
  - 问题域子系统
  - 人机交互子系统
  - 任务管理子系统
  - 数据管理子系统
- 可以把4大组成部分想象成整个模型的4个垂直切片。



典型的面向对象设计模型

1. 子系统之间的两种交互方式

- 客户-供应商关系(Client-supplier)
  - 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。
  - 任何交互行为都是由前者驱动的。
- 平等伙伴关系(peer-to-peer)
  - 每个子系统都可能调用其他子系统，因此，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。
- 总的说来，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。

2. 组织系统的两种方案

- 层次组织(水平)
  - 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。在上、下层之间存在客户-供应商关系。
    - 封闭式
    - 开放式
- 块状组织(垂直)
  - 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。
- 混合使用层次结构和块状结构

典型应用系统的组织结构

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。

11.5 设计问题域子系统

- 面向对象设计仅需从实现角度对问题域模型做一些补充或修改，主要是增添、合并或分解类与对象、属性及服务，调整继承关系等等。
- 当问题域子系统过分复杂庞大时，应该把它进一步分解成若干个更小的子系统。

1. 调整需求

- 有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：
  - 一是用户需求或外部环境发生了变化；
  - 二是分析员对问题域理解不透彻或缺乏领域专家帮助。
- 无论出现上述哪种情况，通常都只需简单地修改面向对象分析结果，然后再把这些修改反映到问题域子系统中。

2. 重用已有的类

- 代码重用从设计阶段开始，在研究面向对象分析结果时就应该寻找使用已有类的方法。
- 重用已有类的典型过程如下：
  - 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。
  - 添加泛化关系。
  - 标出从已有类继承来的属性和服务。
  - 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。

3. 把问题域类组合在一起

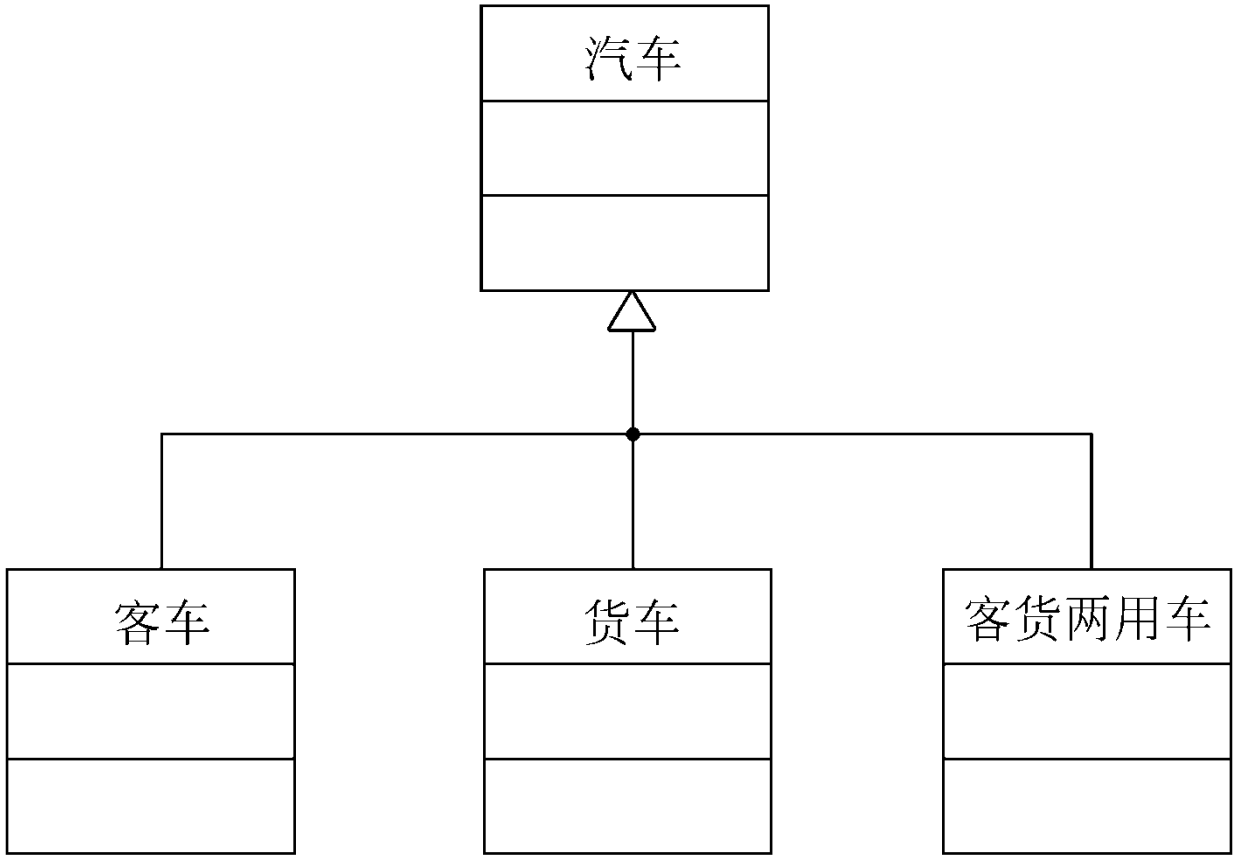
- 通过引入一个根类而把问题域类组合在一起。此外，这样的根类还可以用来建立协议。

4. 增添一般化类以建立协议

- 在设计过程中常常发现，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。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。

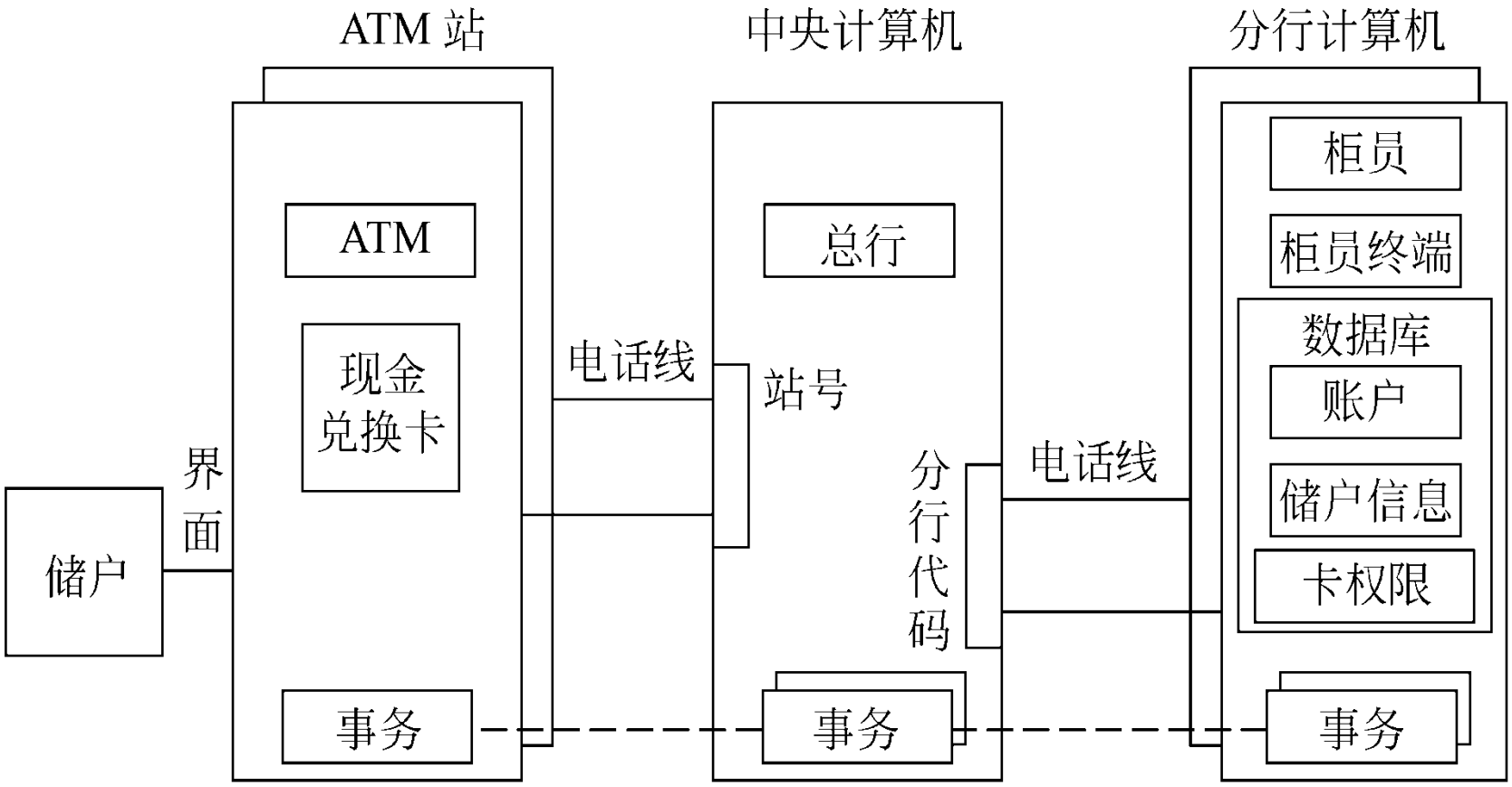
5. 调整继承层次

- 使用多重继承机制
  - 避免出现属性及服务的命名冲突。
  - 窄菱形模式。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。
- 使用单继承机制
  - 把多重继承结构简化成单一的单继承层次结构。
  - 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。



把多重继承简化为单一层次的单继承

6. ATM系统实例



ATM系统问题域子系统的结构

11.6 设计人机交互子系统

- 在面向对象设计过程中，则应该对系统的人机交互子系统进行详细设计，以确定人机交互的细节。
- 使用由原型支持的系统化的设计策略，是成功地设计人机交互子系统的关键。

1. 分类用户

- 应该深入到用户的工作现场，仔细观察用户是怎样做他们的工作的。
- 设计者首先应该把将来可能与系统交互的用户分类。几个不同角度：
  - 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)。

2. 描述用户

- 应该仔细了解将来使用系统的每类用户的情况，记录各项信息：
  - 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。

3. 设计命令层次

- 研究现有的人机交互含义和准则
- 确定初始的命令层次
- 精化命令层次
  - 次序
  - 整体-部分关系
  - 宽度和深度
  - 操作步骤

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。

11.7 设计任务管理子系统

1. 分析并发性

- 通过面向对象分析建立起来的动态模型，是分析并发性的主要依据。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。

2. 设计任务管理子系统

- 确定事件驱动型任务
  - 这类任务可能主要完成通信工作。
  - 工作过程：任务处于睡眠状态，接收到外部中断就被唤醒，接收数据并放入内存缓冲区或其他目的地，通知需要知道的对象，然后又回到睡眠状态。
- 确定时钟驱动型任务
  - 每隔一定时间间隔就被触发以执行某些处理。
  - 工作过程：任务设置了唤醒时间后进入睡眠状态；一旦接收到这种系统中断，就被唤醒并工作，通知有关的对象，然后该任务又回到睡眠状态。

- 确定优先任务
  - 高优先级：为了在严格限定的时间内完成这种服务，可能需要分离成独立的任务。
  - 低优先级：设计时可能用额外的任务把它分离出来。
- 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。

- 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。
- 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。应该使系统中包含的任务数尽量少。
- 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。

11.8 设计数据管理子系统

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，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。

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：
  - 提供了各种最基本的数据管理功能。
  - 为多种应用提供了一致的接口。
  - 标准化的语言(SQL语言)。
- 关系数据库管理系统通常都相当复杂，具体缺点：
  - 运行开销大。
  - 不能满足高级应用的需求。
  - 与程序设计语言的连接不自然。

3. 面向对象数据库管理系统

- 面向对象数据库管理系统是一种新技术，主要有两种设计途径：
  - 扩展的关系数据库管理系统，是在关系数据库的基础上，增加了抽象数据类型和继承机制，此外还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，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。

11.8.2 设计数据管理子系统

1. 设计数据格式

- 文件系统（基于第一范式）
- 关系数据库管理系统（基于第三范式）
- 面向对象数据库管理系统
  - 扩展的关系数据库途径：使用与关系数据库管理系统相同的方法。
  - 扩展的面向对象程序设计语言途径：不需要规范化属性的步骤，因为数据库管理系统本身具有把对象值映射成存储值的功能。

2. 设计相应的服务

- 文件系统
  - 需要知道打开哪个(些)文件，怎样把文件定位到正确的记录上，怎样检索旧值，以及怎样用现有值更新它们。
- 关系数据库管理系统
  - 应该知道访问哪些数据库表，怎样访问所需要的行，怎样检索旧值，以及怎样用现有值更新它们。
- 面向对象数据库管理系统
  - 扩展的关系数据库途径：与关系数据库管理系统方法相同。
  - 扩展的面向对象程序设计语言途径：无须增加服务，已经给每个对象提供了“存储自己”的行为。

11.9 设计类中的服务

11.9.1 设计类中应有的服务

- 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，通常只在每个类中列出很少几个最核心的服务。设计者必须把动态模型中对象的行为以及功能模型中的数据处理，转换成由适当的类所提供的服务。
- 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。数据流图中的某些处理可能与对象提供的服务对应。

11.9.2 设计实现服务的方法

1. 设计实现服务的算法

- 设计实现服务的算法时，应该考虑下列几个因素：
  - 算法复杂度。
  - 容易理解与容易实现。
  - 易修改。

2. 选择数据结构

- 在面向对象设计过程中，需要选择能够方便、有效地实现算法的物理数据结构。

3. 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。

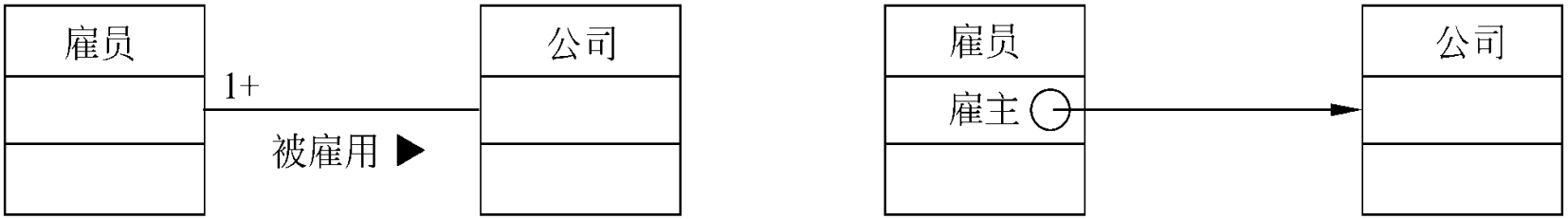
11.10 设计关联

1. 关联的遍历

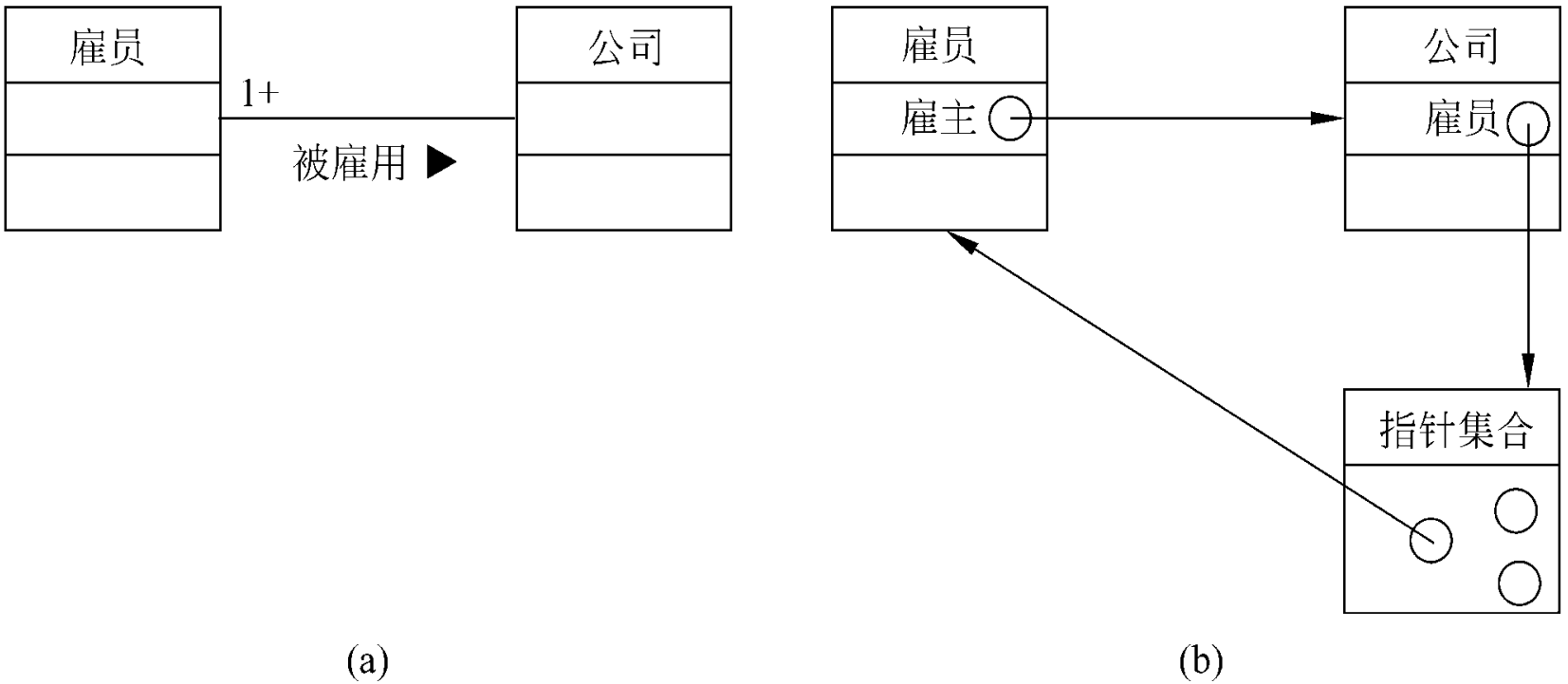
- 使用关联有两种可能的方式：单向遍历和双向遍历。
- 单向遍历实现起来比较简单，双向遍历实现起来稍微麻烦一些。

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，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。



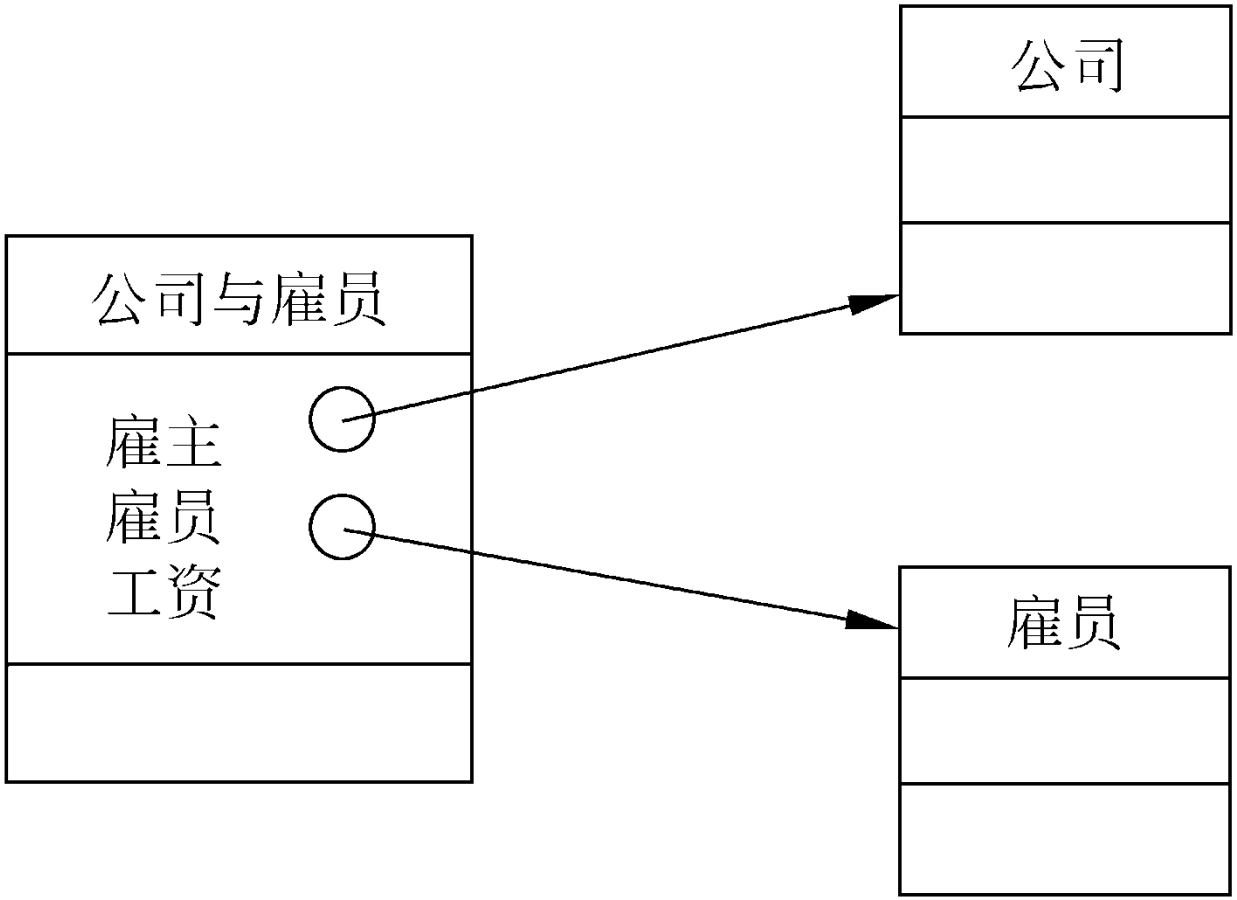
用指针实现单向关联



用指针集合实现双向关联

3. 实现双向关联

- 许多关联都需要双向遍历，当然，两个方向遍历的频度往往并不相同。实现双向关联有下列3种方法：
  - 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。
  - 两个方向的关联都用属性实现。当访问次数远远多于修改次数时，这种实现方法很有效。
  - 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。



用对象实现关联

4. 关联对象的实现

- 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。

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。
- 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。

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

- 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。

2. 调整查询次序

- 优化算法的一个途径是尽量缩小查找范围。

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。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。

11.11.3 调整继承关系

1. 抽象与具体

- 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。

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，则可以把它们抽取出来放在一个共同的祖先类中，供其子类继承。

3. 利用委托实现行为共享

- 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。

利用委托(commitment)

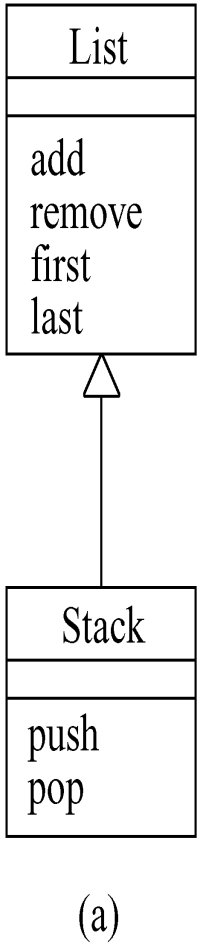
例：实现 Stack

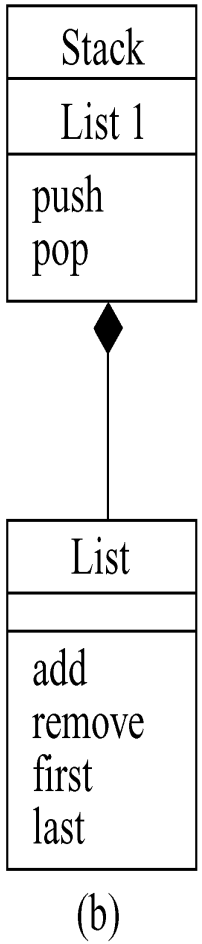
方法1：从 List 派生

push = last + add

pop = last + remove

问题：Stack.first 也是合法的





方法2：把 List 作为 Stack 的一个 attribute，称为 commitment。这种方法比较安全(Stack.first 为非法)。

小结

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