

目录

1 面向对象分析概述	4
1.1 传统的软件开发方法	5
1.2 面向对象技术概述	6
2 UML 概述	7
3 面向对象的基本概念	11
4 UML 中的用例图	15
5 UML 中的静态图	17
5.1 类图	17
5.2 对象图	20
5.3 包图	20
5.4 医院病房监护系统	22
6 UML 中的动态建模技术	23
6.1 状态图	23
6.2 活动图	24
6.3 顺序图	26
6.4 合作图	28
7 单一职责原则 (SRP)	28
8 开放封闭原则 (OCP)	39
8.1 修改方式一	40
8.2 修改方式二	41
8.3 修改方式三	41
9 里氏代换原则 (LSP)	42
9.1 从代码重构的角度理解	42
10 接口隔离原则 (ISP)	45
10.1 全文搜索引擎的系统设计	45
11 依赖倒转原则 (DIP)	46
11.1 例子: 账户和账户的类型	48
12 组合/聚合复用原则 (CARP)	49

13 迪米特法则 (LoD)	50
13.1 狹义的迪米特法则	51
13.2 广义的迪米特法则	52
14 简单工厂模式	53
14.1 简单工厂模式的引进	54
14.2 简单工厂模式的结构	56
14.3 简单工厂模式的实现	57
14.4 简单工厂模式的优缺点	59
15 工厂方法模式	60
15.1 工厂方法模式的结构	61
15.2 工厂方法模式在农场系统中的实现	63
16 抽象工厂模式	65
16.1 抽象工厂模式的结构	68
16.2 抽象工厂模式的应用	71
16.3 抽象工厂模式在农场系统中的实现	73
16.4 开闭原则	77
17 单例模式 (Singleton)	77
17.1 单例模式应用场景-资源管理	78
17.2 单例类的状态	79
17.3 例子: 属性管理器	79
17.4 语言中的单例模式	83
18 原型模式 (Prototype)	84
18.1 原型模式的结构	86
19 适配器模式 (Adapter)	89
19.1 类的适配器模式的结构	90
19.2 对象的适配器模式的结构	91
19.3 Iterator 和 Enumeration	92
19.4 适配器模式在架构层次上的应用	94
20 合成模式 (Composite)	94
20.1 对象的树结构	94
20.2 合成模式的两种形式	95
20.3 应用安全式合成模式	99
20.4 应用透明式的合成模式	100

21 装饰模式 (Decoration)	103
21.1 装饰模式的结构	103
21.2 实例: 发票系统	106
22 代理模式 (proxy)	109
22.1 代理模式的结构	110
23 享元模式 (Flyweight)	111
23.1 单纯享元模式的结构	112
23.2 复合享元模式的结构	114
23.3 单例模式实现享元工厂角色	116
23.4 例子: 户外咖啡摊	117
23.5 例子: 咖啡屋	120
24 门面模式 (Facade)	124
24.1 例子: 医院	125
24.2 例子: 安全监控系统	125
24.2.1 不使用门面模式的设计	126
24.2.2 使用门面模式的设计	128
25 桥梁模式 (Bridge)	129
25.1 桥梁模式的结构	130
25.2 Java 中的 Peer 结构	131
25.3 例子	132
25.4 抽象与实现	134
26 策略模式 (Strategy)	136
26.1 介绍	136
26.2 策略模式的结构	136
26.3 例:Java 语言内部的例子	137
26.4 例: 排序策略系统	138
26.5 例: 图书折扣的计算	139
26.6 使用场景	140
27 模板方法模式 (Template Method)	141
27.1 好莱坞原则	142
27.2 例: 计算账户利息	142
27.3 模板方法模式中的方法	144
27.4 应用: 代码重构	144

28 观察者模式 (Observer)	146
28.1 观察者模式的结构	146
28.1.1 另一种方案	148
28.2 Java 提供的对观察者模式的支持	150
28.3 怎样使用 Java 对观察者模式的支持	152
28.4 Java 中的 DEM 事件机制	155
28.5 观察者模式的优缺点	155
29 命令模式 (Command)	155
29.1 命令模式的结构	155
29.2 命令模式的优劣	157
29.3 例:AudioPlayer 系统	157
30 状态模式 (State)	161
30.1 状态模式的结构	162
30.2 使用场景	163
30.3 例:TCP	163
30.4 例: 用户登录子系统	164

1 面向对象分析概述

学习目标:

- 掌握面向对象的基本思维方式、分析与设计的基本理论和方法;
- 掌握面向对象建模语言 UML 及其设计软件、软件设计模式、面向对象程序设计语言;
- 采用面向对象方法设计实现小规模的应用程序. 课程内容包括: 面向对象的基本概念, 面向对象分析与设计的基本原则、方法与过程, 面向对象设计模式.

软件产品的主要类型

从软件项目立项来源、用户群体等角度, 软件产品可划分为:

1. 互联网软件: 面向不特定人群, 由互联网企业独立投资开发并免费开放给用户, 通过服务、产品销售、广告等方式取得利润; 搜索引擎、电商平台、即时通信、信息分发 APP 等.
2. 通用软件: 面向特定人群, 由软件企业独立投资开发并采用零售拷贝、许可证、使用时间等方式发售;ERP、办公套装软件、工具软件等.
3. 定制软件: 由甲方 (业主方) 发起并提供开发经费, 仅局限于某行业、某单位或部门等少数用户;XX 集团运营管理系統、XX 城市 IOC 平台.

软件开发一般过程: 提出问题 → 可行性研究 → 需求分析 → 总体设计 → 详细设计 → 编码与测试 → 试运行与培训 → 维护 → 评价.

需求分析的主要活动

- 与用户交流, 获取拟开发系统的功能、性能、界面、安全等方面的要求; 重点解决跨领域**沟通难**的问题。如某系统的需求: 生成目标区内不同场点多概率水准的人造地震时程和天然地震时程。
- 梳理出**问题域**的业务流程、数据交换关系等信息; 一般不涉及**计算机领域**的实现细节。针对上例中的需求: 用什么数据、什么方法、怎么样的流程, 生成什么样的结果
- 形成系统需求说明书 (是软件开发所有后续工作的基础)。文字、图、表、动画、原型软件。针对上例中的分析: 把分析结果表达出来

数据交换与共享 (横向, 纵向), 应用实现层, 应用支撑层, 数据资源层, 基础设施层, 支撑硬件基础设施。

系统设计的主要活动

- 总体设计: 系统总体布局方案, 软件系统总体结构的设计, 数据存储的总体设计, 计算机和网络系统方案的选择。
- 详细设计: 代码设计, 数据库设计, 人机界面设计, 处理过程设计
- 系统实施进度与计划的制定
- 系统设计说明书的编写 (是系统设计阶段的主要成果及下一阶段工作的基础)。

1.1 传统的软件开发方法

功能分解法: 以功能为核心, 对系统逐层进行分解, 直至可直接实现的子功能, 最后设计功能间的接口, 和功能内部的数据结构、算法。

- 优点: 以功能为核心, 较为直观、自然; 逐层分解法与人类的思维模式相近, 提高了开发效率。
- 缺点: 功能和接口难以映射到问题领域中的事物; 对需求变化的适应能力差。

功能分解法示例, 成绩管理系统:

- 教师信息管理: 增删改查, 导入教师信息;
- 课程管理: 管理课程基本信息, 管理课程学生;
- 成绩维护: 录入学生成绩, 修改学生成绩.

信息建模方法: 以数据为中心的软件开发方法。

- 优点: 有关系数据库的良好数学基础, 可保证系统的数据完整性、一致性; 适合以数据处理为中心的软件系统;
- 缺点: 对功能的刻画能力较弱;

面向过程的结构化方法: 采用自顶向下, 逐步求精的思想, 把一个复杂问题的求解过程逐层、分阶段进行分解, 使得每个阶段处理的问题都控制在人们容易理解和处理的范围内。以模块为基础, 以流程图、数据流图, 数据字典, 结构化语言, 判定表, 判定树等图形表达为主要建模手段。

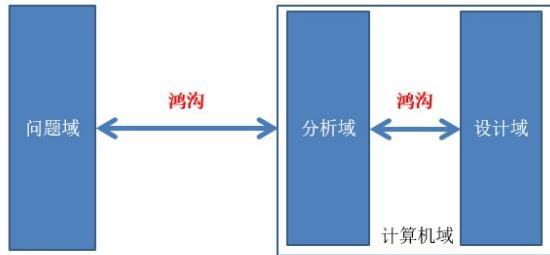
- 优点: 开发过程的整体性和逻辑性较强; 严格区分系统开发的阶段, 分离逻辑设计和物理设计; 每一阶段的工作成果是下一阶段的依据, 便于系统开发的管理和控制; 强调文档的规范化, 可为系统的开发和维护提供良好支持

- 缺点: 开发周期长, 难以适应变化; 分析与设计采用了不同的概念和表示法, 相互转换困难.

模块化: 系统划分为若干个模块, 每个模块完成一个特定的功能, 所有模块汇聚起来组成一个整体.

- 模块: 是一组能完成某个完整功能的程序语句, 包括输入输出, 逻辑处理功能, 内部信息及其运行环境.
- 模块相对独立, 模块间的接口简单

基于上述方法的软件开发过程存在两道鸿沟:



软件的分析和设计, 即对问题域的认识和描述。

1.2 面向对象技术概述

面向对象方法是一种运用对象、类、封装、继承、多态和消息等概念来构造、测试、重构软件的方法。面向对象方法是以认识论为基础, 用对象来理解和分析问题空间, 并设计和开发出由对象构成的软件系统(解空间)的方法。面向对象是一种认识客观世界的思维方法. 面向对象应用于软件开发, 软件开发是将现实世界的问题映射为计算机领域的问题的过程. 面向对象技术的核心是对象。An object is anything that has a fixed shape or form, that you can touch or see, and that is not alive. 对象可代表现实世界的任何事物, 概念。对象把数据结构和行为紧密结合在一起。面向对象技术是把一组对象有效地集成在一起形成软件的技术。

OO 的发展

- 最早 20 世纪 50 年代初, 出现对象的概念.
- 1967 年, 产生第一个面向对象的语言 Simula, 提出了数据抽象和类的概念.
- 1972 年, 产生第二个面向对象的语言 SmallTalk.
- 80 年代, 出现了 Objective-C, C++, ...
- 面向对象的分析, 设计...

面向对象的语言包含 4 个基本的分支:

1. 基于 Smalltalk 的; 包括 smalltalk 的 5 个版本, 以 Smalltalk-80 为代表。
2. 基于 C 的; 包括 objective-C, C++, Java
3. 基于 LISP 的; 包括 Flavors, XLISP, LOOPS, CLOS
4. 基于 PASCAL 的; 包括 Object Pascal, Turbo Pascal, Eiffel, Ada 95

- 面向对象开发与面向过程开发对比，其主要好处不在于减少开发时间，而在于有利于问题域的分析，促进软件可重用，减少出错，利于软件维护。
- 面向对象的开发强调从问题域的概念到软件程序和界面的直接映射；心理学的研究也表明，把客观世界看成是许多对象更接近人类的自然思维方式。
- 对象比函数更为稳定；软件需求的变动往往是功能相关的变动，而其功能的执行者-对象通常不会有大的变动。

Example: $y = kx + b$

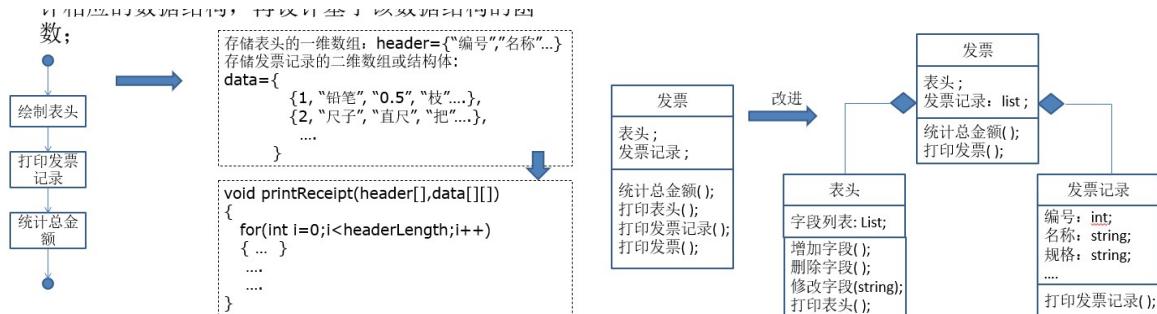
- int k; int b; float x;
- Formula(int k1, int b1, float x1){...}
- float calculateY(){return k*x+b;}

Example: Tank

- int x; int y; int width; int height; int speed; int direction;
- void move()(问题域); void fire(); void draw()(实现域).

用面向过程的结构化方法和面向对象方法来考察一个发票生成和打印系统.

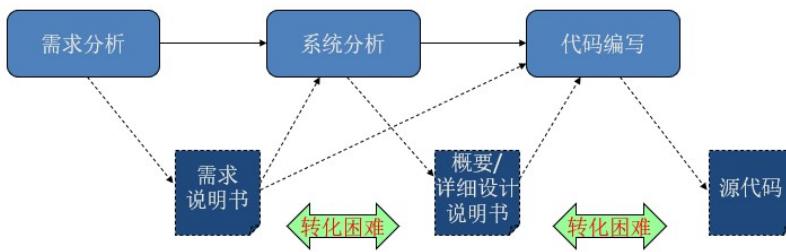
- 结构化方法：先分析发票生成和打印的过程，设计相应的数据结构，再设计基于该数据结构的函数；
- 面向对象方法：将发票看成一个对象，分析和设计对象的属性和方法；



2 UML 概述

UML (Unified Modeling Language)，全称为**统一建模语言**：是一种通用的可视化建模语言，用于说明、可视化、构造并文档化软件系统。

模型是对事物的一种抽象，人们常常在正式建造实物之前，首先建立一个简化的模型，以便更透彻地了解它的本质，抓住问题的要害；在模型中，人们总是剔除那些与问题无关的、非本质的东西，从而使模型与真实的实体相比更加简单、易于把握；**模型**：在软件开发期间所产生的中间描述或文档。



构造软件模型通常出于以下几个目地：

- 对复杂问题进行简化；
- 用于同客户或其他相关人员进行交流；
- 加强视觉效果；
- 在着手解决一个复杂问题之前，对解决方案进行检测。

1. 八十年代至九十年代，出现了数量众多的面向对象建模语言.
2. 1996 年, 由 Grady Booch、Dr.Ivar Jacobson 和 Dr.James Rumbaugh 三人正式推出 UML.
3. UML 为人们提供了从不同的角度去观察和展示系统的各种特征的一种标准表达方式。
4. UML 中，从任何一个角度对系统所作的抽象都可能需要用几种模型图来描述，而这些来自不同角度的模型图最终组成了系统的完整模型。

一般而言，可从以下几种常用视角来描述一个系统：

1. **系统的使用实例**: 从系统外部的操作者的角度描述系统的功能。
2. **系统的逻辑结构**: 描述系统内部的静态结构和动态行为，即从内部描述如何设计实现系统功能。
3. **系统的物理构成**: 描述系统由哪些程序构件所组成。
4. **系统的并发性**: 描述系统的并发性，强调并发系统中存在的各种通信和同步问题。
5. **系统的配置**: 描述系统的软件和各种硬件设备之间的配置关系。

UML 的主要构成：

1. **视图 (views)**: 从一个特殊的角度观察到的系统称为一个视图。视图由多个图 (Diagrams) 构成，它不是一个具体的图，而是在某一个抽象层上，对系统的抽象表示。
2. **图 (Diagrams)**
3. **模型元素 (Model elements)**: 代表面向对象中的类，对象，关系和消息等概念，是构成图的最基本的常用的元素。一个模型元素可以用于多个不同的图中。分为以下两类：
 - **基元素**: 是已由 UML 定义的模型元素。如：类、结点、构件、注释、关联、依赖和泛化等。
 - **构造型元素**: 在基元素的基础上增加了新的定义而构造的新的模型元素。如扩展基元素的语义（不能扩展语法结构）。构造型元素用括在双尖括号 “` `” 中的字符串表示。
4. **通用机制 (general mechanism))**

视图:

- Use case View(Use case 视图): 描述系统的外部特性、系统功能等。
- Design View(设计视图): 描述系统设计特征, 包括结构模型视图和行为模型视图, 前者描述系统的静态结构 (类图、对象图), 后者描述系统的动态行为 (交互图、状态图、活动图)。
- Process View(过程视图): 表示系统内部的控制机制。常用活动图描述过程结构, 用交互图描述过程行为。
- Implementation View(实现视图): 表示系统的实现特征, 常用构件图表示。
- Deployment View(配置视图): 配置视图描述系统的物理配置特征。用配置图表示。

图: UML 主要提供了五类十种图形:

- **用例图** (Use case diagram): 从用户角度描述系统功能, 并指出各功能的操作者。
- **静态图** (Static diagram): 表示系统的静态结构, 包括**类图**、**对象图**、**包图**。
 - 类图描述系统中类的静态结构。不仅定义系统中的类, 表示类之间的联系如关联、依赖、聚合等, 也包括类的内部结构 (类的属性和操作)
 - 对象图是类图的实例, 几乎使用与类图完全相同的标识。他们的不同点在于对象图显示类的多个对象实例, 而不是实际的类。
 - 包由包或类组成, 表示包与包之间的关系。包图用于描述系统的分层结构。UML 1.1 不再认为包图是一类独立的图形.
- **行为图** (Behavior diagram): 描述系统的动态行为。包括**状态图**、**活动图**。
 - 状态图描述类的对象所有可能的状态以及事件发生时状态的转移条件。通常, 状态图是对类图的补充。
 - 活动图描述满足用例要求所要进行的活动以及活动间的约束关系, 有利于识别并行活动。
- **交互图** (Interactive diagram): 描述对象间的交互关系。包括**顺序图**、**合作图** (**通信图**)。
 - 顺序图显示对象之间的动态合作关系, 它强调对象之间消息发送的顺序, 同时显示对象之间的交互;
 - 合作图描述对象间的协作关系, 合作图跟顺序图相似, 显示对象间的动态合作关系。
 - UML 2.0 中新增时间图、交互概述图。
- **实现图** (Implementation diagram): 用于描述系统的物理实现, 包括**构件图**、**配置图**。
 1. 构件图描述代码部件的物理结构及各部件之间的依赖关系。
 2. 配置图定义系统中软硬件的物理体系结构。它可以显示实际的计算机和设备 (用节点表示) 以及它们之间的连接关系, 也可显示连接的类型及部件之间的依赖性。

类图

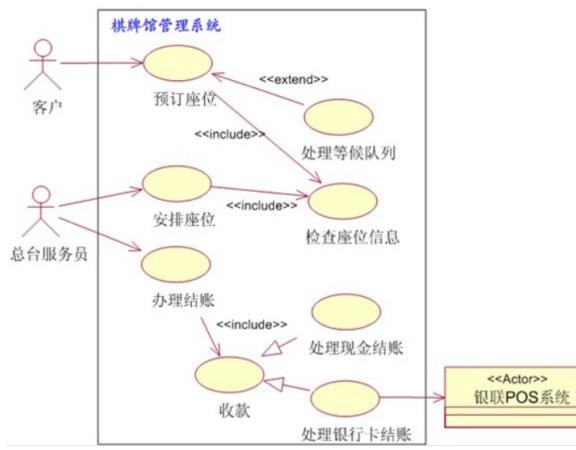


图 1: 用例图

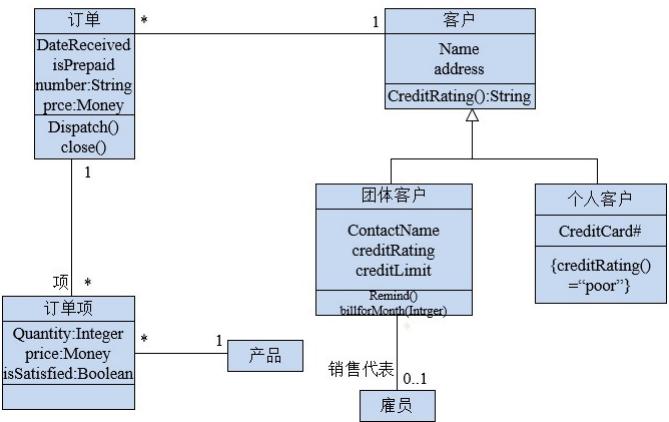


图 2: 类图

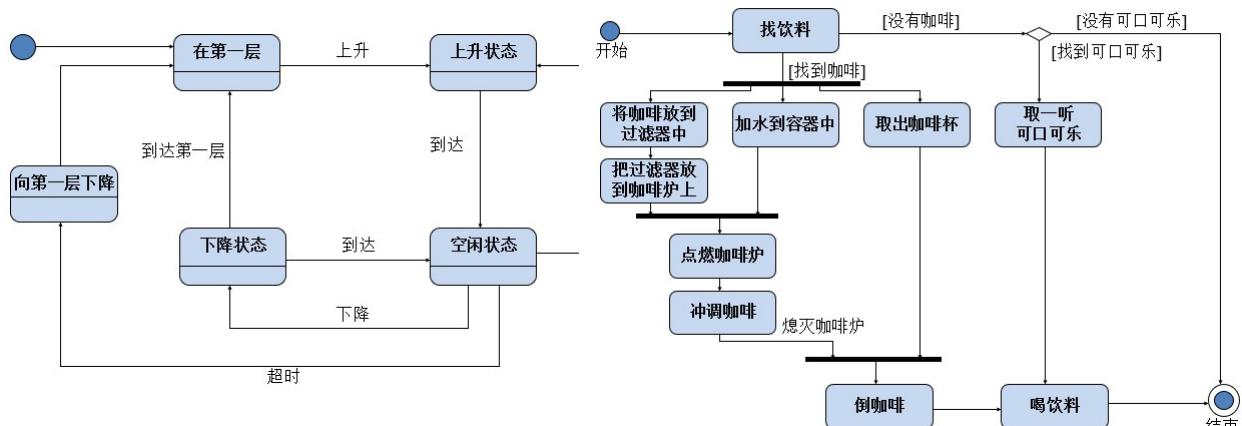


图 3: 状态图

图 4: 活动图

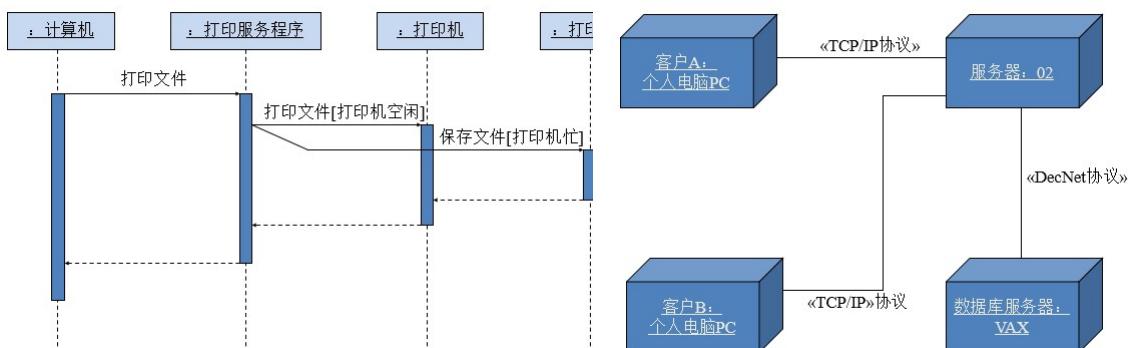


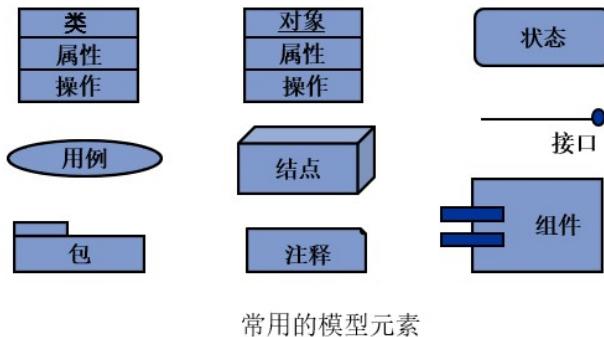
图 5: 顺序图

图 6: 配置图

UML 的应用领域:UML 的目标是以面向对象图的方式来描述任何类型的系统，具有广泛的应用领域。其中最常用的是建立软件系统的模型，但它同样可以用于描述非软件领域的系统，如机械系统、企业机构或业务过程，以及具有实时要求的工业系统或工业过程等。UML 可应用于软件开发的各个阶段。

常用的模型元素：

模型元素在图中用其相应的视图元素（符号）表示，图中给出了常用的元素符号：类、对象、结点、包和组件等。



常用的模型元素

模型元素与模型元素之间的连接关系也是模型元素，常见的关系有关联（association）、泛化（generalization）、依赖(dependency)和聚合（aggregation），其中聚合是关联的一种特殊形式。这些关系的图示符号如图所示。



常用的模型元素-连接关系

3 面向对象的基本概念

对象概念的三种观点

- 从数据结构的角度看，对象是一种复杂的数据类型。
- 从软件结构的角度看，对象是一个完备的模块，包含了能完成一定功能的函数和局部数据。
- 从分析设计的角度看，对象是一个活动的实体，可以代表世界的万事万物。

对象的内部结构: 对象是一个拥有属性、行为和标志符的实体。

- 属性:** 第一种观点：属性也就是变量；第二种观点：属性是程序要处理的数据；第三种观点：描述对象的状态、特征。

- **行为:** 第一种观点: 行为也就是函数; 第二种观点: 行为是程序所完成功能的实现; 第三种观点: 对象具有的改变自身或其他对象状态的活动.
- **标志符:** 用于区别对象.

类的概念

- **类:** 对一组相似的对象的描述, 这一组对象有共同的属性和行为.
- **对象与类的关系:** 对象是类的实例, 类是对象的模板.
- 同一类的对象, 具有不同的属性值, 但具有相同的方法.

方法的类型

- **属性过程:** 对属性的存取操作, 维护对象的状态.
- **服务函数:** 为其他函数提供服务. 比如字符串查找, 排序等.
- **接口函数:** 类和外界打交道的接口, 类通过接口函数为外界提供服务.
- **对象控制函数:** 实现对象生命周期的典型功能, 控制对象的创建和销毁.

面向对象系统的基本特征:

- 利用对象进行**抽象**: 主要是提炼相对某种目的的重要的方面, 而忽略次要的方面. 目的决定了哪些方面是重要的, 因此, 根据目的的不同, 对同一事物可以有不同的抽象. 是所有程序设计方法的基本工具. 自上而下逐步细化, 自下而上逐步抽象.
- **封装:** 信息隐藏, 阻止外界直接对类的状态信息的访问, 仅提供方法用以访问和改变它的状态, 提高类的安全性. 提高对象的独立性, 有利于灵活地局部修改, 提升了程序的可维护性. 封装是所有常用的信息系统开发方法的普遍特点. 传统方法将数据和功能分开放装. 面向对象技术则是把功能和数据封装进入对象.
- **消息通信:**
 - 是对象协作的灵活机制; 模拟现实系统中对象之间的联系; 对象之间联系的方法-利用消息进行通信.
 - 消息: 从发送方向接收方发出的执行服务的请求. 发送消息通过调用某个类的方法来实现. 接收消息通过被其他对象调用本类的方法被实现.
- **生命周期:** 设计期-类的生命周期 (设计, 实现); 运行期-对象的生命周期.
 1. 对象被创建 (调用构造函数, 实例化)
 2. 存在
 3. 消亡 (区分对象的正常和意外消亡)
- **类层次结构 (继承和组合)**
- **多态:**
 1. 与继承相关的概念, 从共同的基类派生出不同的子类, 不同子类对象呈现出多种形态. (第三种观点)
 2. 同一对象引用, 可以指向不同子类的对象. 用父类引用去操作子类对象.(第二种观点)
 3. 方法与对象相关, 由对象 (具体类型) 才能确定方法.(第一种观点)

类之间的关联: 描述类之间的关系.

- 普通关联 (互相独立的类)
- 层次结构 (互相不独立的类)
 1. 整体和部分之间的关系: 聚合 (弱关联关系, 部分可有可无, 整体和部分可相对独立) 和组合 (强关联关系, 部分不可或缺, 整体和部分不可分离)
 2. 泛化/特殊化: 继承.

概念之间的关系: 如车辆分为机动车和非机动车.

继承:

- 体现了类之间的关联关系, 该关系把类分成父类和子类.
- 代表了概念之间的扩展关系, 与人们认知事物的认知过程一致:
- 由一般到具体, 由模糊到清晰.(第三种观点)
- 能够实现某个类无需重新定义就拥有另一个类的某些属性和方法, 达到复用和灵活设计的目的, 也利于代码的统一维护。(第二种观点)
- 子类是父类类型的一种. (第一种观点)

继承的若干种情况:

- **一般继承 (单继承):** 一个父类拥有一个或多个子类, 一个子类只有一个父类.
- **多继承:** 一个子类拥有多个父类, 描述了现实系统里的概念叠加. 多继承可能带来定义冲突, 如两个父类具有同名的方法. 铁锅就是金属和容器这两个概念的叠加.
- **实现式继承:** 父类方法只有声明, 没有实现.

继承属性, 继承方法:

- 子类具有父类的所有属性.
- **重载 (overload):** 在同一个类内部, 为同名的方法指定不同的参数表并进行不同的实现.
- **覆盖 (override):** 子类为超类的属性和方法指定了新的定义.

在设计类之间的继承关系时, 应注意:

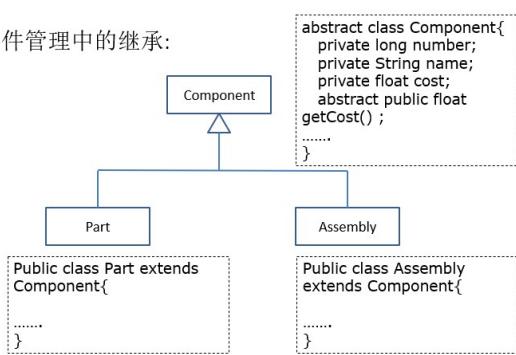
- 用 isa 进行继承关系的测试. an A(子类) is a B(父类) (A 是一个 B)
- 父类和子类之间要确实存在继承关系. 如错误的设计:(父类) 猫科动物-(子类) 狗
- 子类的对象在其生存期内必须保持独特性. 如错误的设计:(父类) 正常用户-(子类) 欠费用户
- 所有继承下来的特性在每个子类中都必须有意义. 如错误的设计: 车 (父类) 含属性-油量, 汽车 (子类), 自行车 (子类).

Example: 零件管理: 在制造业中, 某些类别的复杂产品是由零件装配而成的。常见的需求是记录所有的零件的库存以及这些零件的信息, 包括:

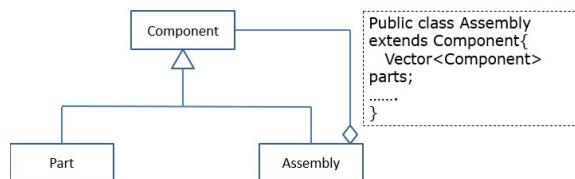
1. 零件的目录查找号 (整数);
2. 名字 (字符串);
3. 价格 (浮点数);

零件可以装配成更复杂的结构, 成为组件。一个组件可以包含多个零件, 而且可以形成层次结构, 即一个组件可以由多个子组件构成, 每个子组件又可以由零件或下级子组件构成。

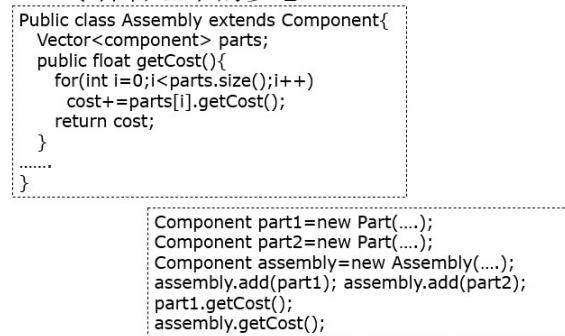
- 零件管理中的继承:



- 零件管理中的组合\聚合:



- 零件管理中的多态:



继承, 重载, 多态: 是为了提高系统的灵活性, 降低类之间的耦合性.

例: 在图书馆管理系统中, 假设要实现读者借阅图书的功能. 对于不同类型的图书 (新书, 旧书, 外文书) 有不同的借阅规则 (归还日期, 费用). 请问该如何设计才能提高系统的灵活性 (将来可能会增加新的图书类型, 该类型可能具有自己特有的借阅规则)

抽象类: 描述了系统中的抽象概念. 不能实例化的类. 通常为子类定义一些公共的接口, 指定一些子类必须实现的方法.

- 抽象类中有些方法只有声明, 没有实现.
- 未实现的方法, 留给具体类去实现. 抽象类只负责定义公共接口.
- 抽象类可以具有已实现的方法, 因此也具备一般类的继承的优点.

接口 interface: 包含属性和抽象方法的特殊的类。接口定义了类之间交流的**协议**。由硬件接口得名。接口不能实例化。只能被继承和实现。

类, 抽象类, 接口三者构成完整的抽象序列机制, 描述了人们对系统的认识和理解的过程。
包: 面向对象技术提供的另一种封装机制. 包含了逻辑上功能需要相互协作的类. 可用于划分命名空间, 解决命名冲突的问题. 可用于划分子系统.

可见性: 一个类看到和使用另一个类的资源的能力。

- **公有可见性:** 公有属性和方法对整个外界都是可见的。任何其他类都可以访问类的共有属性和方法。
- **私有可见性:** 私有属性和方法只对该类的成员可见。
- **保护可见性:** 保护属性和方法只对该类和它的子类可见。
- **友类可见性:** 友类属性和方法对指定的其他的一些类 (友元类) 是可见的。

类属性: 类的所有实例共享的属性. 每个类属性只有一个拷贝, 无需创建任何类实例, 也可以访问这些类属性.

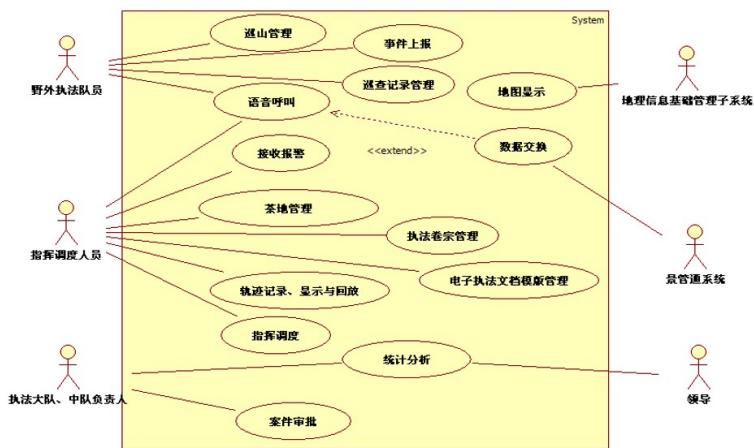
类方法: 由类定义的方法, 只能对类属性进行操作. 类方法可以在没有任何类实例的情况下被调用.

4 UML 中的用例图

用例模型: 用例模型描述的是用户所理解的系统功能。用例驱动的系统分析与设计方法, 已成为面向对象系统分析与设计的主流方法。

- 它描述了待开发系统的功能需求;
- 它将系统看作黑盒, 从外部执行者的角度来理解系统;
- 它驱动了需求分析之后各阶段的开发工作, 不仅在开发过程中保证了系统所有功能的实现, 而且被用于验证和检测所开发的系统, 从而影响到开发工作的各个阶段和 UML 的各个模型。

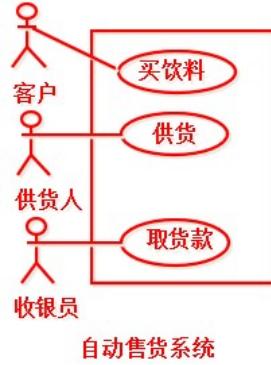
在 UML 中, 一个用例模型由若干个用例图描述, 用例图主要元素是用例和执行者。



执行者 (actor):

- 是指用户在系统中所扮演的角色。其图形化的表示是一个小人。

- 不带箭头的线段将执行者与用例连接到一起，表示两者之间交换信息，称之为通信联系。
- 单个执行者可与多个用例联系；反过来，一个用例可与多个执行者联系。
- 执行者可以不是人类用户。



用例：一个用例是用户与系统之间的一次典型交互，代表了将要开发的系统的一项功能。如：顾客通过 B2C 电子商务系统下订单。用例具有以下特点：

- 用例捕获某些用户可见的需求，实现一个具体的用户目标。
- 用例由执行者激活，并提供确切的结果给执行者。
- 用例可大可小，但它必须是对一个具体的用户目标的完整描述。

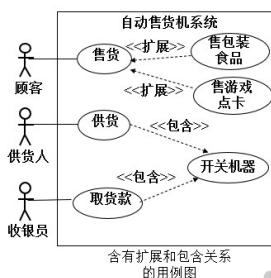
用例间的联系

- «generalization» 表示用例间的继承关系。
- «Extend» 通过向被扩展的用例添加动作来扩展用例。
- «include» 表示一个用例的行为包含了另一个用例的行为。

包括关系和扩展关系的联系和差别

联系：都是从现有的用例中抽取出公共的那部分信息，作为一个单独的用例。然后通过不同的方法来重用这个公共的用例，以降低模型维护的工作量。

差别：扩展关系中基本用例的基本流运行时，扩展用例不一定运行，即扩展用例仅仅只有在基本用例满足某种条件的时候才会运行。包括关系中基本用例的基本流运行时，包括用例一定会运行。



用例模型的建立：建立系统用例模型的过程就是对系统进行功能需求分析的过程。

1. 定义系统: 确定系统范围, 分析系统功能.
2. 确定执行者和用例: 执行者通常是使用系统功能的外部用户或系统。用例是一个子系统或系统的一个独立、完整功能。
3. 描述执行者和用例关系: 各模型元素之间有: 关联、泛化、扩展及包含等关系。
4. 确认模型: 确认用例模型与用户需求的一致性, 通常由用户与开发者共同完成。

获取执行者: 获取用例首先要找出系统的执行者。可通过用户回答一些问题的答案来识别执行者。以下问题可供参考:

- 谁使用系统的主要功能 (主要使用者)。
- 谁需要系统支持他们的日常工作。
- 谁来维护、管理使系统正常工作 (辅助使用者)。
- 系统需要操纵哪些硬件。
- 系统需要与哪些其它系统交互, 包含其它计算机系统和其它应用程序。
- 对系统产生的结果感兴趣的人或事物。

获取用例: 一旦获取了执行者, 就可以对每个执行者提出问题以获取用例。

- 执行者要求系统提供哪些功能 (执行者需要做什么)?
- 执行者需要读、产生、删除、修改或存储的信息有哪些?
- 必须提醒执行者的系统事件有哪些? 或者执行者必须提醒系统的事件有哪些?

还有一些不针对具体执行者问题 (即针对整个系统的问题):

- 系统需要何种输入输出? 输入从何处来? 输出到何处?
- 当前运行系统的主要问题?

角色描述模板: 角色名, 角色职责, 角色职责识别 (使用系统主要功能, 对系统运行结果感兴趣).

用例描述模板: 用例名, 执行者, 目标, 功能描述, 其他非功能需求 (可靠, 实时), 主要步骤, 相关用例, 相关信息 (优先级, 性能, 执行频率).

5 UML 中的静态图

静态建模: 任何建模语言都以静态建模机制为基础, 标准建模语言 UML 也不例外. 所谓静态建模是指建立系统内部结构及各部分之间的互相联系, 而这些关系不随时间而转移. **类和对象的建模**, 是 UML 静态建模的主要内容, 也是 UML 建模的基础.

5.1 类图

类图 (Class Diagram) 描述类和类之间的静态关系. 与数据模型不同, 它不仅显示了信息的结构, 同时还描述了系统的行为. 类图是定义其它图的基础.



短式和长式类图

类图的结构: 类图表示为一个划分成三个格子的长方形. 类及类型名均用英文大写字母开头, 属性及操作名为小写字母开头.

- **类名**
- **属性** (可见性, 名称, 类型, 缺省值和约束特性)
 - **属性书写的语法:** 可见性属性名: 类型 = 缺省值 {约束特性}
 - **可见性:** “+”: Public, “-”: Private, “#”: Protected.
 - **约束:** 是用户对该属性性质一个约束的说明.e.g. {只读}
- **方法书写语法:** 可见性操作名 (参数表): 返回类型 {约束特性}

使用类图的几个建议

- 不要试图使用所有的符号.
- 根据项目开发的不同阶段, 用正确的观点来画类图.
- 不要为每个事物都画一个模型, 应该把精力放在关键的领域.
- 当用一张 A4 纸不足以容纳所有的类图时, 就应该用包图.

类间的关系:

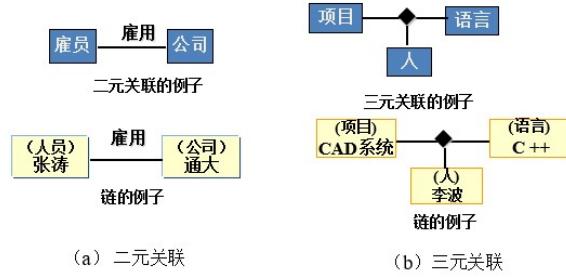


类图的层次: 类图的抽象层次和细化 (Refinement) 关系-在需求分析阶段, 类图是研究问题领域的概念; 在设计阶段, 类图描述类与类之间的接口; 而在实现阶段, 类图描述软件系统中类的实现.

- **概念层** (Conceptual) 类图描述应用领域中的概念.
- **说明层** (Specification) 类图描述软件的接口部分, 而不是软件的实现部分.
- 只有在**实现层** (Implementation) 才真正有类的概念, 并且揭示软件的实现部分.

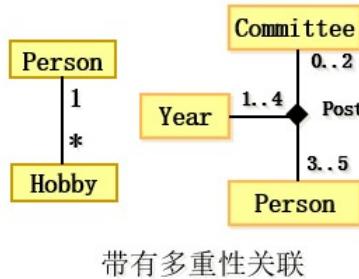
细化关系: 有两个元素 A 和 B, 若 B 元素是 A 元素的更详细描述, 则称为 B 元素细化 A 元素 ($B \rightarrow A$). 细化与类的抽象层次有密切的关系, 在构造模型时要经过逐步细化, 逐步求精的过程.

关联和链: 关联 (association) 是两个或多个独立类之间固有的关系. 链 (link) 是关联的具体体现. 关联分为二元关联 (binary), 三元关联 (ternary), 多元关联 (higher order).



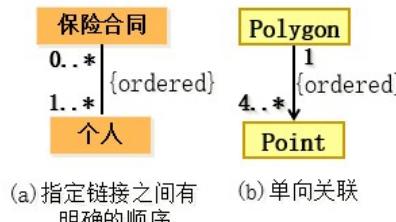
关联的重数: 重数 (multiplicity) 表示多少个对象参与到关联关系中, 重数的默认值为 1. 常用重数符号有:

- “0..1”: 表示 0 或 1
- “0..*” 或 “*”: 表示 0 或多个
- “1..*”: 表示 1 或多个
- “1,3,7”: 表示 1 或 3 或 7 (枚举型)



带有多重性关联

有序关联与导航 (导引): 在关联的多端标注 {ordered} 指明这些类之间的关系是有序的. 关联可以用箭头, 表示该关联关系的方向 (单向或双向), 称为导引或导航 (navigation).



有序关联与导航

依赖: 依赖关系描述的是两个模型元素 (类, 组合, 用例等) 之间的使用与被使用的关系, 其中一个模型元素是独立的, 另一个模型元素是非独立的 (或依赖的) . 如图表示类 A 依赖于类 B 的一个友元依赖关系.



依赖的形式可能是多样的，针对不同的依赖的形式，依赖关系有不同的变体 (varieties): 如友元、访问、调用、创建等.

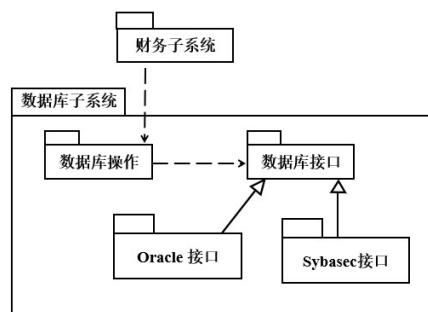
5.2 对象图

对象与类的图形表示相似，为划分成两个格子的长方形 (对象名和属性值). 对象图是类图的一种实例化. 一张对象图表示的是与其对应的类图的一个具体实例，即系统在某一时期或者某一特定时刻可能存在的具体对象实例以及它们相互之间的具体关系.

5.3 包图

为何引入包 (Package): 降低系统的复杂性.

什么是 UML 包: 是将许多类集合成为一个更高层次的单位，形成一个高内聚、低耦合的类的集合；是一种分组机制，把各种各样的模型元素通过内在的语义连接为一个整体；构成包的模型元素称为包的内容，包通常用于对模型的组织管理，因此有时又将包称为子系统 (subsystem) .

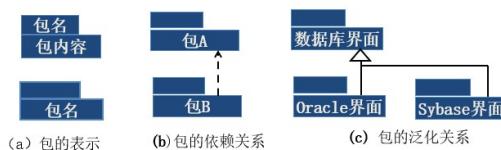


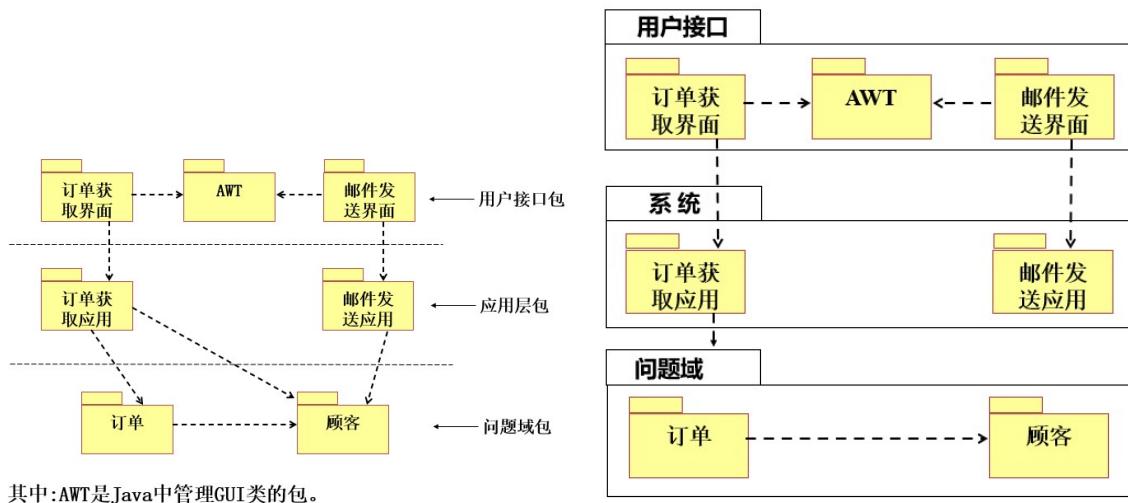
包的内容: 可以是类的列表，也可以是另一个包图，还可以是一个类图.

包之间的关系:

- **依赖关系:** 两个包中的任意两个类存在依赖关系，则包之间存在依赖关系.
- **泛化关系:** 使用继承中一般和特殊的概念来说明通用包和专用包之间的关系. 例如，专用包必须符合通用包的接口，与类继承关系类似.

包图的表示:





类的识别: 是面向对象方法的一个难点，但又是建模的关键。常用方法有：

1. **名词识别法**
2. **系统实体识别法**
3. **从用例中识别类:** 根据用例的描述来识别类
 - 用例的描述中出现哪些实体？
 - 用例执行过程中产生并存储哪些信息？
 - 与用例关联的角色向用例输入什么信息？
 - 用例又向该角色输出哪些信息？
4. **利用分解与抽象技术**
 - 分解技术: 将整体类和组合类分解. 可控制单个类的规模.
 - 抽象技术: 根据一些类的相似性建立抽象类，并建立抽象类与这些类之间的继承关系.
 - 抽象类实现了系统内部的重用，很好地控制了复杂性，并为所有子类定义了一个公共的接口，使设计局部化，提高系统的可修改性和可维护性.

关键是要定义类的属性及操作.

名词识别法: 识别问题域中的实体，实体的描述通常用名词、名词短语、名词性代词的形式出现.

- 用指定语言对系统进行描述；描述过程应与领域专家共同合作完成，并遵循问题域中的概念和命名.
- 从系统描述中标识名词、名词短语、名词性代词；识别确定（取、舍）类. 为了发现对象和类，开发人员要在系统需求和系统分析的文档中查找名词和名词短语，包括：可感知的事物、角色、事件、互相作用、人员、场所、组织、设备和地点等.
- 对类进行筛选，根据下述原则进一步确定类：
 1. 去掉冗余类：如两个类表述同一信息，应保留最具有描述能力的类.
 2. 去掉不相干的类：删除与问题无关或关系不大的类.

3. 删除模糊的类或性质独立性不强的类: 有些初始类边界定义不确切, 或范围太广, 应该删除.

4. 所描述的操作不适宜作为对象类, 所描述的只是实现过程中的暂时的对象, 应删去.

系统实体识别法: 不关心系统的运作流程及实体之间的通信状态, 而只考虑系统中的人员、设备、组织、地点、表格、报告等实体, 经过分析将他们识别为类(或对象).

被标识的实体有: 系统需要存储、分析、处理的信息实体; 系统内部需要处理的设备; 与系统交互的外部系统; 系统相关人员; 系统的组织实体.

确定关联: 使用下列标准去掉不必要的和不正确的关联

1. 若某个类已被删除, 那么与它有关的关联也必须删除或者用其他类来重新表述。
2. 不相干的关联或实现阶段的关联。删除所有问题域之外的关联或涉及实现结构中的关联,
3. 动作。关联应描述应用域的结构性质而不是瞬时事件。
4. 派生关联, 省略那些可以用其他关联来定义的关联, 因为这种关联是冗余的。

5.4 医院病房监护系统

为了对危重病人进行实时监护, 随时了解病人病情, 及时进行处理, 建立病房监护系统。

病症监视器安置在每个病床, 通过网络将病人的组合病症信号实时传送到中央监护系统进行分析处理。在中心值班室里, 值班护士使用中央监护系统对病员的情况进行监控, 监护系统实时地将病人的病症信号与标准的病诊信号进行比较分析, 当病症出现异常时, 系统会立即自动报警, 并打印病情报告和更新病历。系统根据医生的要求随时打印病人的病情报告, 系统定期自动更新病历。

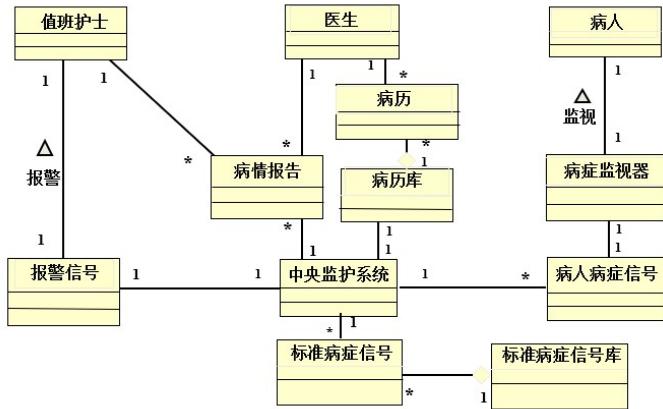
经过初步的需求分析, 得到系统功能要求:

1. 监视病员的病症 (血压、体温、脉搏等)
2. 定时更新病历
3. 病员出现异常情况时报警。
4. 随机地产生某一病员的病情报告。

通过名词识别法和系统实体识别法等方法可以确定系统的十二个类:

值班护士	医生	病人	病症监视器	病人病症信号	病情报告	病历	标准病症信号
用户名 密码 查看病情报告() 打印病情报告()	用户名 密码 查看病情报告() 要求打印病情报告() 查看病历() 要求打印病历()	姓名 性别 年龄 病症 提供病症信号()	采集频率 病症信号 提供病症信号()	脉搏 血压 体温 生成病症信号()	标题 格式 生成病情报告()	格式 病人基本情况 打印时间 生成病历() 查看病历() 打印病情报告()	脉搏 血压 体温 生成标准信号()
			格式化信号数据() 采集信号() 信号组合()				

医院病房监护系统类图



6 UML 中的动态建模技术

- 行为图 (Behavior diagram), 描述系统的行为。包括状态图和活动图。
- 交互图 (Interactive diagram), 描述对象间的交互关系。包括顺序图, 合作图。

动态建模的应用:

- 不要对系统中的每个类都画状态图。状态图描述跨越多个用例的单个对象的行为，而不适合描述多个对象间的行为合作。
- 顺序图和合作图适合描述单个用例中几个对象的行为。其中顺序图突出对象间交互的顺序，而合作图的布局方法能更清楚地表示出对象之间静态的连接关系。当行为较为简单时，顺序图和合作图是最好的选择。
- 如果想显示跨过多用例或多线程的复杂行为，可考虑使用活动图。

6.1 状态图

- 状态图 (State Diagram) 用来描述一个特定对象的所有可能状态及其引起状态转移的事件。
- 大多数面向对象技术都用状态图表示单个对象在其生命周期中的行为。
- 一个状态图包括一系列的状态以及状态之间的转移。

状态:

- 初态: 状态图的起始点
- 终态: 状态图的终点
- 中间状态: 名字域和内部转移域. 内部转移域是可选的，其中所列的动作将在对象处于该状态时执行，且该动作的执行并不改变对象的状态。
- 复合状态: 可以进一步细化的状态.
- 子状态间的关系:
 - 或关系: 在某一时刻仅可到达一个子状态。

- 与关系: 在某一时刻可同时到达多个子状态。

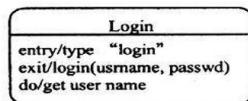


图 3 一个带有动作域的状态

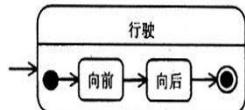


图 4 “或关系”的子状态

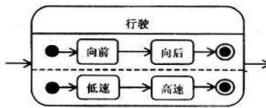
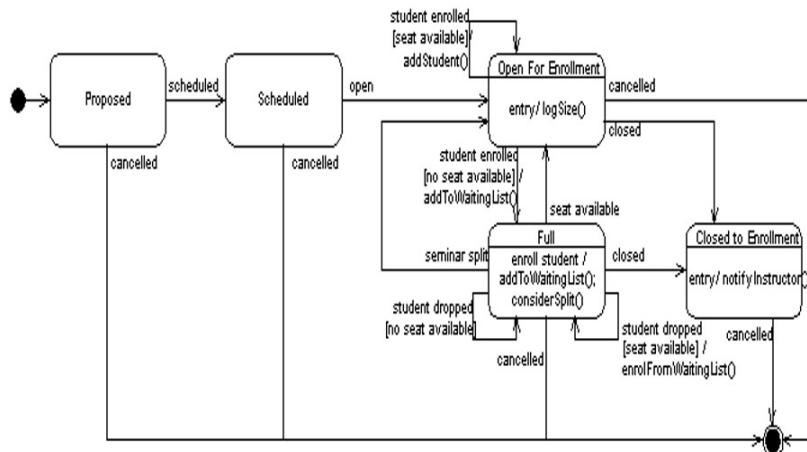


图 5 一个具有并发子状态图



6.2 活动图

活动图 (Activity Diagram) 既可用来描述类 (类的方法) 的行为, 也可以描述用例和对象内部的工作过程。

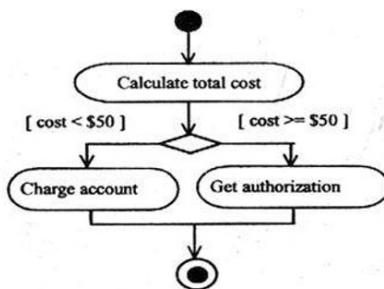
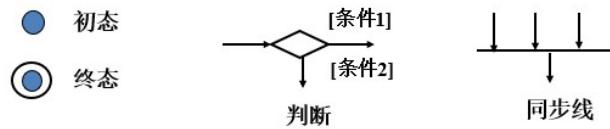


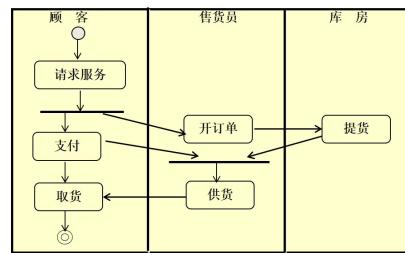
图 1 一个活动图的例子

活动图的模型元素:



转移: 转移描述活动之间的关系，描述由于隐含事件引起的活动变迁，即转移可以连接各活动。转移用带箭头的直线表示，可标注执行该转移的条件，无标注表示按顺序执行。

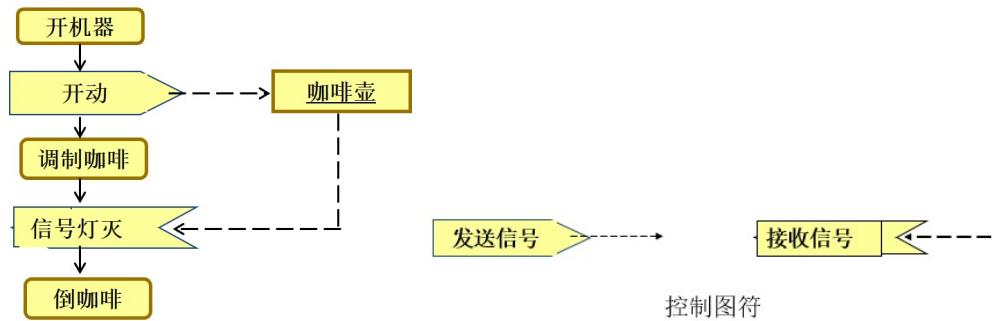
泳道: 泳道进一步描述完成活动的对象，并聚合一组活动。泳道也是一种分组机制。泳道可以直接显示动作在哪一个对象中执行，或显示的是一项组织工作的哪部分。



对象流: 活动图中可以出现对象，对象作为活动的输入 / 输出，用虚箭头表示。

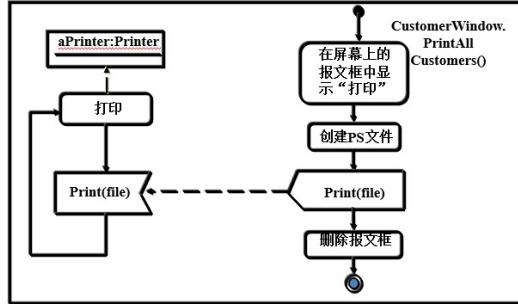


控制图符: 活动图中可发送和接收信号，发送符号对应于与转移联系在一起的发送短句。接收符号也同转移联系在一起。



发送和接收信号: 活动图中可发送和接收信号，发送符号和接收符号对应于与转移联系在一起的发送短句。转移又分两种：发送信号的转移和接收信号的转移。发送和接收信号可以和

消息的发送对象和接收对象联系在一起，如下图。发送信号动作是一种特殊的动作，它表示从输入信息创建一个信号实例，然后发送到目标对象。发送信号动作可能触发状态的转换或者活动的执行。在发送信号动作时可以包含一组带有值的参数。由于信号是一种异步消息，所以发送方立即继续执行，所有的响应都将被忽略，并未返回给发送方。

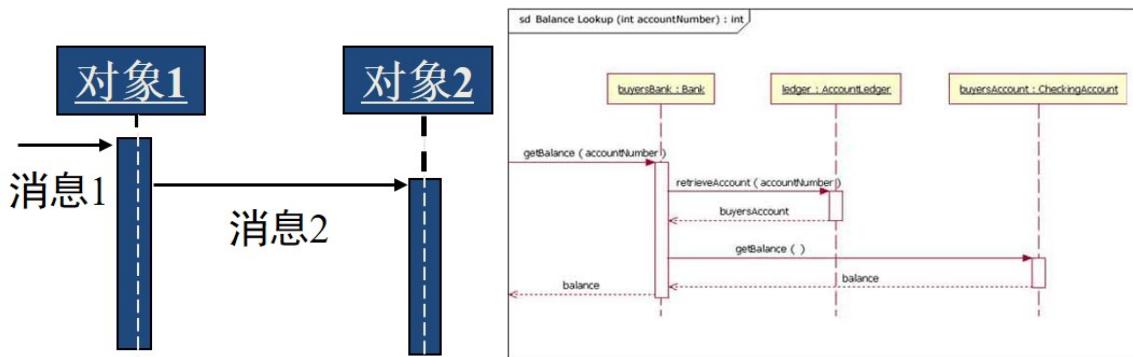


活动图中只有一个起点一个终点，表示方式和状态图一样，泳道被用来组合活动，通常根据活动的功能来组合。

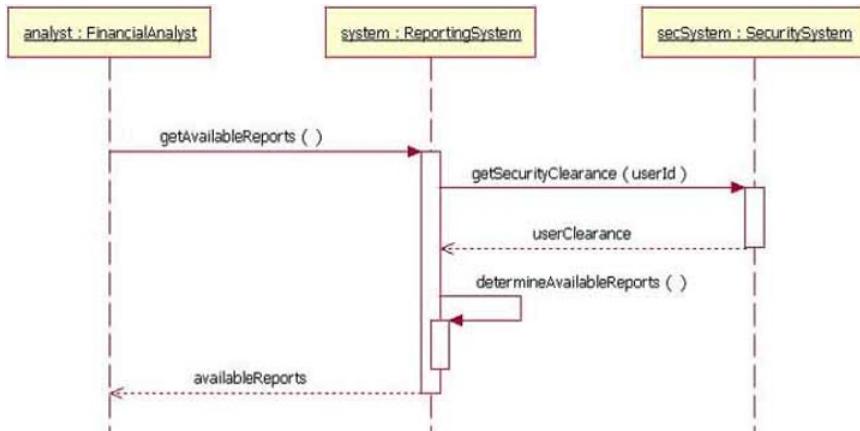
6.3 顺序图

顺序图 (Sequence Diagram) 用来描述对象之间动态的交互关系，着重体现对象间消息传递的时间顺序。顺序图的主要用途之一，是把用例表达的需求，转化为进一步、更加正式的、层次化的精细表达。

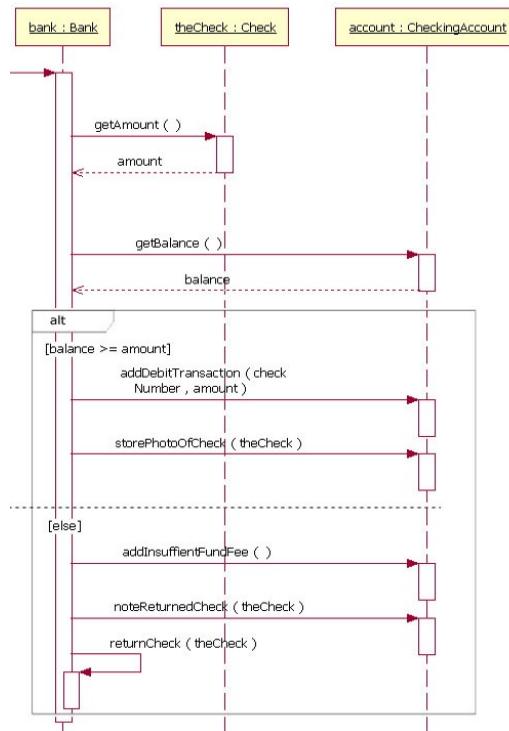
顺序图构成: 一组对象 (对象名和类名); 对象生命线 (时间轴); 对象被激活; 对象间的通信 (消息).



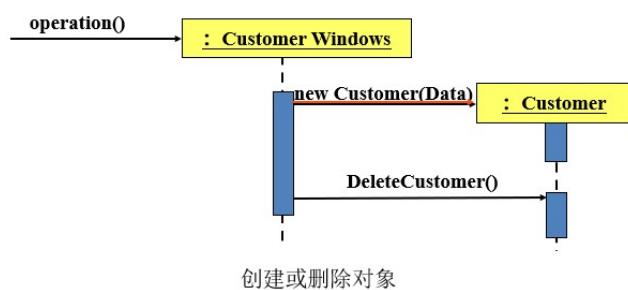
对象向自身发送消息:



顺序图中的选择:

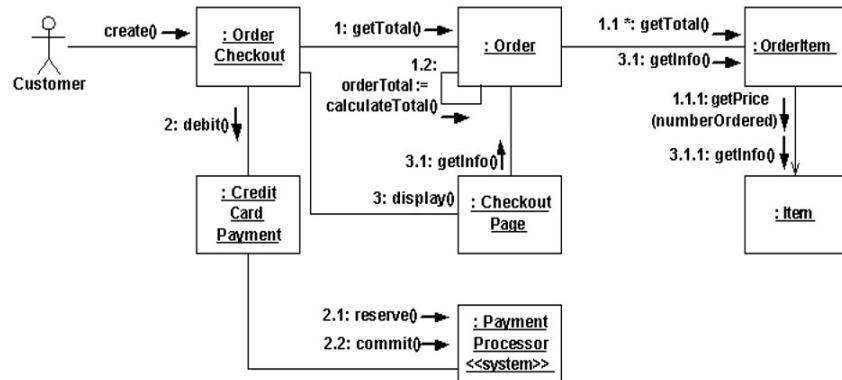


在顺序图中，还可以描述一个对象通过发送一条消息来创建另一个对象。



6.4 合作图

合作图 (Collaboration Diagram) 用于描述相互合作的对象间的交互关系和链接关系。着重体现交互对象间的静态链接关系。



合作图中对象的外观与顺序图中的一样。如果一个对象在消息的交互中被创建，则可在对象名称之后标以 {new}。类似地，如果一个对象在交互期间被删除，则可在对象名称之后标以 {destroy}。

消息: 在对象之间的静态链接关系上可标注消息。用标号表示，标号有 3 种：

- 顺序执行: 按整数大小执行. $1, 2, \dots$
 - 嵌套执行: 标号中带小数点. $1.1, 1.2, 1.3, \dots$
 - 并行执行: 标号中带小写字母. $1.1.1a, 1.1.1b, \dots$

控制信息:

- 条件控制信息, 如: $[x > y]$.
 - 重复控制信息, 如: $*[I = 1..n]$.

7 单一职责原则 (SRP)

Single Responsibility Principle

OO 设计的 SOLID 原则: 为实现易维护、可扩展的软件系统，应遵循一定的设计原则。

缩写	全称	翻译
SRP	The Single Responsibility Principle	单一职责/责任原则
OCP	The Open Closed Principle	开放封闭原则
LSP	The Liskov Substitution Principle	里氏替换原则
ISP	The Interface Segregation Principle	接口分离原则
DIP	The Dependency Inversion Principle	依赖倒转原则

应只有一类原因可以引发对类的修改 (there should never be more than one reason for a class to change)。换句话说，就是让一个类只承担一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。

目的: 为了实现模块化，即高内聚、低耦合。是任何程序设计方法都遵循的原则，如面向过程方法中的函数设计。一个函数只实现一种功能；一个函数的代码行数不宜过长 (30 行?/一屏?)。

遵循 SRP 带来的好处: 可以降低类的复杂度，提高类的可读性；控制变更影响的范围；综合以上两点，SRP 可提高系统的可维护性；SRP 是所有原则中的最简单的，同时也是最难正确运用的。对象识别与职责划分，是 OO 设计永恒的主题。

```
interface Room
{
    void service();
};

class Kitchen implements Room
{
    void service()
    {
        //实现做饭的功能;
    }
};

class Bedroom implements Room
{
    void service()
    {
        //实现休息的功能;
    }
};
```

案例：“图书管理系统”面向对象分析与设计

例如，“图书管理系统”面向对象分析与设计大致过程如下：

1. 需求调查分析

需求调查分析的结果一般用文字描述，必要时也可用业务流程图辅助描述。“图书管理系统”需求陈述如下：

在图书管理系统中，管理员要为每个读者建立借阅账户，并给读者发放不同类别的借阅卡（借阅卡可提供卡号、读者姓名），账户内存储读者的个人信息和借阅记录信息。持有借阅卡的读者可以通过管理员（作为读者的代理人与系统交互）借阅、归还图书，不同类别的读者可借阅图书的范围、数量和期限不同，可通过互联网或图书馆内查询终端查询图书信息和个人借阅情况，以及续借图书（系统审核符合续借条件）。读者类别可以按需要增加、删除或修改。

借阅图书时，先输入读者的借阅卡号，系统验证借阅卡的有效性和读者是否可继续借阅图书，无效则提示其原因，有效则显示读者的基本信息（包括照片），供管理员人工核对。然后输入要借阅的书号，系统查阅图书信息数据库，显示图书的基本信息，供管理员人工核对。最后提交借阅请求，若被系统接受则存储借阅纪录，并修改可借阅图书的数量。归还图书时，输入读者借阅卡号和图书号（或丢失标记号），系统验证是否有此借阅纪录以及是否超期借阅，无则提示，有则显示读者和图书的基本信息供管理员人工审核。如果有超期借阅或丢失情况，先转入过期罚款或图书丢失处理。然后提交还书请求，系统接受后删除借阅纪录，并登记并修改可借阅图书的数量。

图书管理员定期或不定期对图书信息进行入库、修改、删除等图书信息管理以及注销（不外借），包括图书类别和出版社管理。

2. 用例建模

（1）确定执行者

通过对系统需求陈述的分析，可以确定系统有两个执行者：管理员和读者。简要描述如下：

- 1) 管理员：管理员按系统授权维护和使用系统不同功能，可以创建、修改、删除读者信息和图书信息即读者管理和图书管理，借阅、归还图书以及罚款等即借阅管理。
- 2) 读者：通过互联网或图书馆查询终端，查询图书信息和个人借阅信息，还可以在符合续借的条件下自己办理续借图书。

（2）确定用例

在确定执行者之后，结合图书管理的领域知识，进一步分析系统的需求，可以确定系统的用例有：

- 借阅管理：包含借书、还书（可扩展过期和丢失罚款）、续借、借阅情况查询；
- 读者管理：包含读者信息和读者类别管理；

- 图书管理：包含图书信息管理、图书类别管理、出版社管理、图书注销和图书信息查询。

下面是借阅情况查询、读者信息管理、读者类别管理、图书类别管理、出版社管理和图书信息查询等用例的简要描述：

- 1) 借阅情况查询：读者通过互联网或图书查询终端登录系统后，查阅个人的所有借阅纪录。
- 2) 读者信息管理：管理员登录后，对读者详细信息进行增、删、改等维护管理。
- 3) 读者类别管理：管理员登录后，对读者类别进行增、删、改等维护管理。
- 4) 图书类别管理：管理员登录后，对图书类别进行增、删、改等维护管理。
- 5) 出版社信息管理：管理员登录后，对出版社详细信息进行增、删、改等维护管理。
- 6) 图书信息查询：读者或管理员通过互联网或图书查询终端登录后，查询所需要的图书信息。

下面是借书、还书、续借、图书信息管理、图书注销等用例的详细描述：

1) 借书

用例名称：借书

参与的执行者：管理员

前置条件：一个合法的管理员已经登录到这个系统

事件流：

```

A.输入读者编号;
提示超期未还的借阅记录;
B.输入图书编号;
If 选择“确定”then
    If 读者状态无效 或 该书“已”注销 或 已借书数>=可借书数 Then
        给出相应提示;
    Else
        添加一条借书记录;
        “图书信息表”中“现有库存量”-1;
        “读者信息表”中“已借书数量”+1;
        提示执行情况;
    Endif
    清空读者、图书编号等输入数据;
Endif
If 选择“重新输入”then
    清空读者、图书编号等输入数据;
Endif
If 选择“退出”then
    返回上一级界面;
Endif
返回 A.等待输入下一条;

```

后置条件：如果是有效借书，在系统中保存借阅纪录，并修改图书库存量和读者借书数量。

2) 还书

用例名称：还书

参与的执行者：管理员

前置条件：一个合法的管理员已经登录到这个系统

事件流：

A.输入读者编号；

提示超期未还的借阅记录；

If 有超期 then

 提示，调用“计算超期罚款金额”；

Endif

If 丢失 then

 选择该书借阅记录；

 调用“计算丢失罚款金额”+调用“计算超期罚款金额”；

Endif

If 选择“确定”还书 then //要先交罚款后才能还

B.输入图书编号；

If 读者状态无效 或 该图书标号不在借书记录中 then

 提示该读者借书证无效或该图书不是该读者借阅的；

Else

 添加一条还书记录；

 删除该借书记录；

 “图书信息表”中“现有库存量”+1；

 “读者信息表”中“已借书数量”-1；

 提示执行情况；

Endif

 清空读者、图书编号等输入数据；

Endif

If 选择“重新输入”then

 清空读者、图书编号等输入数据；

Endif

If 选择“退出”then

 返回上一级界面；

Endif

 返回 A.等待输入下一条；

后置条件：如果是有效还书，在系统中删除借阅纪录，并修改图书库存量和读者借书数量。

3) 续借

用例名称：续借

参与的执行者：管理员、读者

前置条件：一个合法的管理员或读者已经登录到这个系统

事件流：

A.输入读者编号；

提示超期未还的借阅记录;
 If 有超期 then
 提示，调用“计算超期罚款金额”;
 Endif
 选择该书借阅记录;
 Endif
 If 选择“确定”续借 then
 If 该图书已超期 或 该图书续借次数>=可续借次数 then
 提示该读者该图书已超期或该图书续借次数>可续借次数，不能续借;
 Else
 修改该书借阅记录中的“应归还日期”;
 图书续借次数+1;
 提示执行情况;
 Endif
 清空读者、图书编号等输入数据;
 Endif
 If 选择“重新输入”then
 清空读者书编号等输入数据;
 Endif
 If 选择“退出”then
 返回上一级界面;
 Endif
 返回 A.等待输入下一条;

后置条件：如果是有效续借，在系统中修改借阅纪录。

4) 图书信息管理

用例名称：图书信息管理

参与的执行者：管理员

前置条件：一个合法的管理员已经登录到这个系统

事件流：

(参见附录 D “图书管理系统软件设计规格说明书” 中“书籍信息管理”模块详细设计，这里略)

后置条件：如果是有效操纵，在系统中增加、修改、删除图书信息纪录。

5) 图书注销

用例名称：图书注销

参与的执行者：管理员

前置条件：一个合法的管理员已经登录到这个系统

事件流：

- A.查询要注销的图书信息；
 - B.选择要注销的图书信息记录；
- If 选择“确定”注销 then
- If 该书有借阅记录 then
- 提示该书有人已借阅，不能注销；

```

Else
    添加一条注销记录;
    “图书信息表”中设定该书“已”注销;
    提示执行情况;
Endif
Endif
If 选择“退出”then
    返回上一级界面;
Endif
返回 A. 等待选择下一条或重新查询;
后置条件：如果是有效注销，在系统中保存注销纪录，并对图书信息做标记。

```

(3) 确定用例之间的关系

确定执行者和用例之后，进一步确定用例之间的关系，如图 7-35 所示。

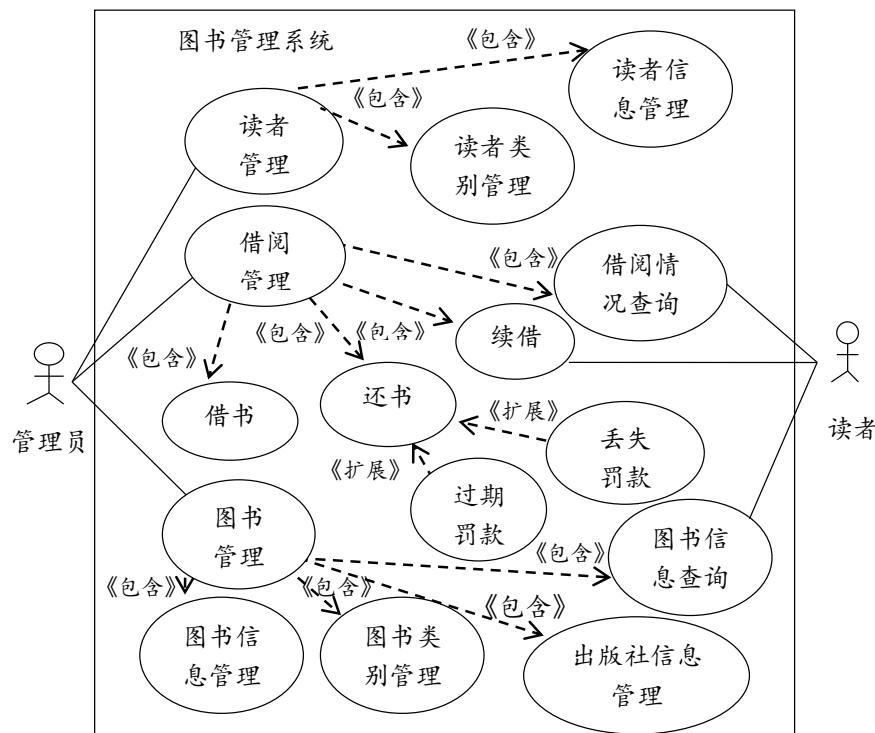


图1 “图书管理系统”用例图

3. 静态建模

首先通过寻找系统需求陈述中的名词，结合图书管理的领域知识，首先给出候选的对象类，经过筛选、审查，可确定“图书管理系统”的类有：读者、图书、借阅记录、图书注销记录、读者类别、图书类别、出版社等。然后，经过标识责任、标识协作者和复审，定义类的属性、操作和类之间的关系。

这里仅以“读者”类为例列出该类的属性和操作：

“读者”类

• 私有属性

读者编号（借书证号码和用户名与此同）：文本

读者姓名：文本

读者类别编号：文本

读者性别：文本

出生日期：时间/日期

读者状态：文本

办证日期：时间/日期

已借图书数量：数值

证件名称：文本

证件号码：文本

读者单位：文本

联系地址：文本

联系电话：文本

EMAIL：文本

用户密码：文本

办证操作员：文本

备注：文本

• 公共操作

永久写入读者信息

永久读取读者信息

新增读者

删除读者

修改读者信息

获取读者信息

查找读者信息

返回借阅数量

类之间的关系如图 2 所示。

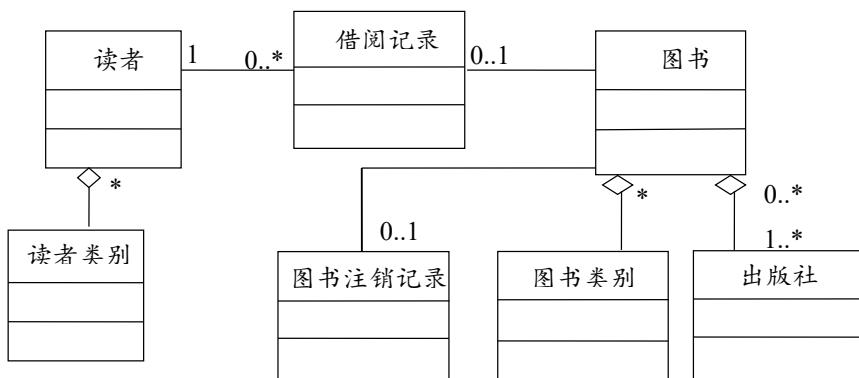


图 2 “图书管理系统”类图

4. 系统逻辑结构设计

“图书管理系统”系统逻辑结构设计用包图描述，如图 3 所示。

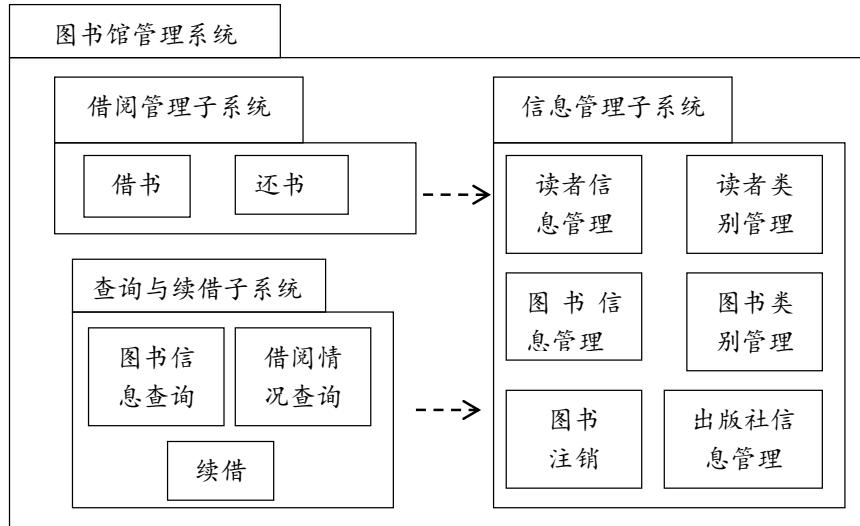


图3 “图书管理系统”包图

5. 类的细化设计

对象设计主要有两个任务：

一是对类的属性和操作的实现细节进行设计。

如上面“读者”类的属性“联系电话”有多个时，决定用一个链表或数组来存放，也可能需要增加属性和操作，如“读者”类中增加属性“相片”，操作增加“打印与发生过期通知书”，而后设计每一个操作的算法。

二是分别从人机交互、数据管理、任务管理和问题域方面考虑，以实现的角度添加一些类，或优化类的结构。

如从数据管理方面，需要添加一个“永久数据”类作为需要永久保存数据类的父类，承担读写数据库的责任；从人机交互方面，需要添加一个“对话框”类（其父类是“窗口”类）来实现人机交互的功能，则图 2 可改进为图 4。

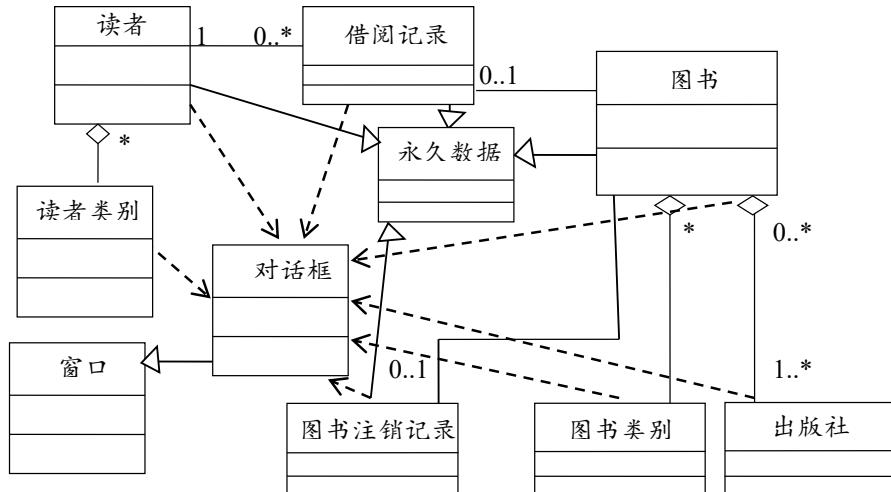


图4 “图书管理系统”细化设计后的类图

6. 动态建模

必要时，可针对系统的某一功能画出完成此功能的对象之间交互消息的顺序图，如“借书”功能的消息交互顺序如图 5 所示。

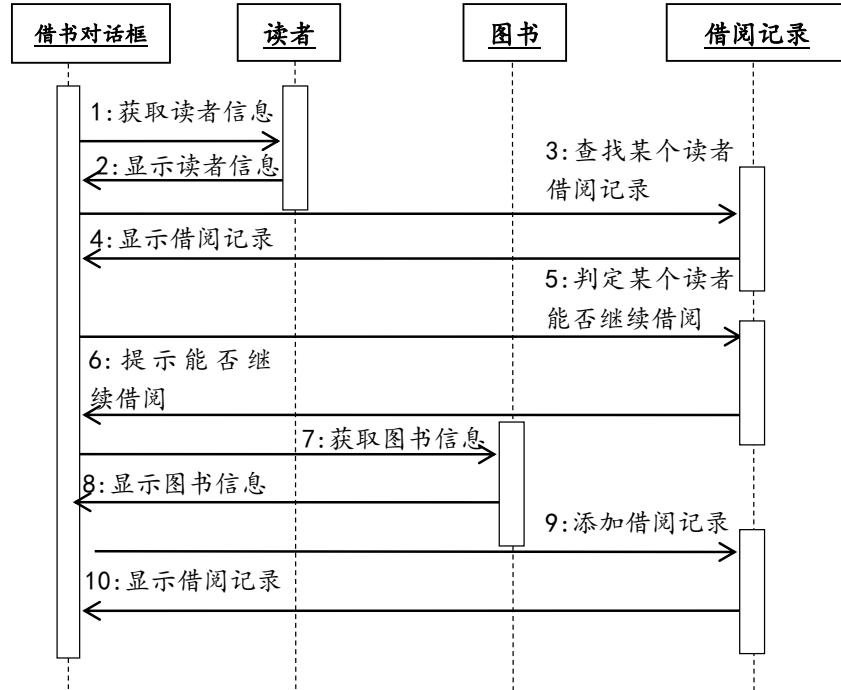


图 5 “借书”顺序图

必要时，可针对系统的某一类对象画出表示该对象在系统中的状态变化过程，如“图书”对象的状态变化如图 6 所示。

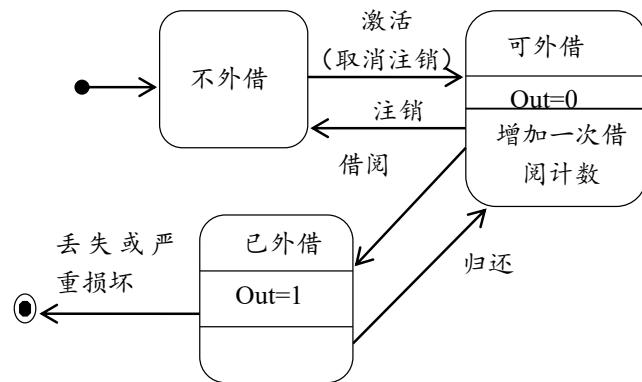


图6 “图书”对象状态图

7. 物理建模

“图书管理系统”物理结点分布如图 7 所示。

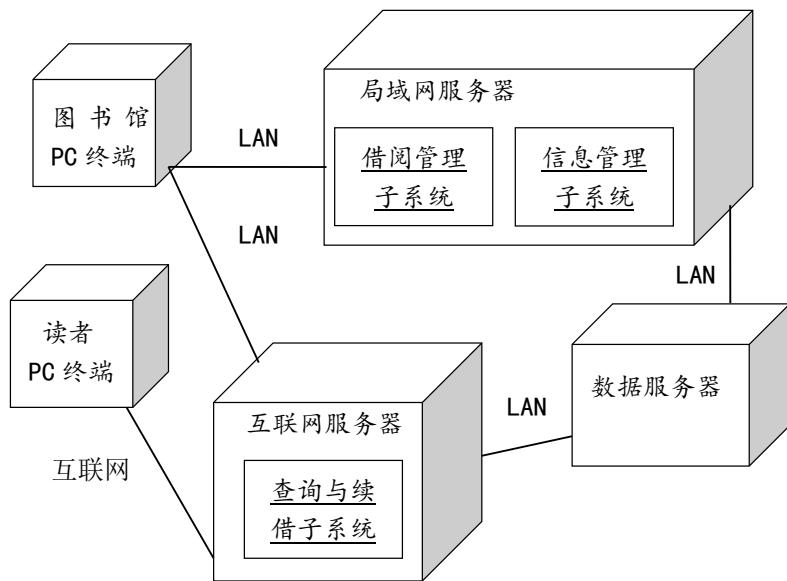


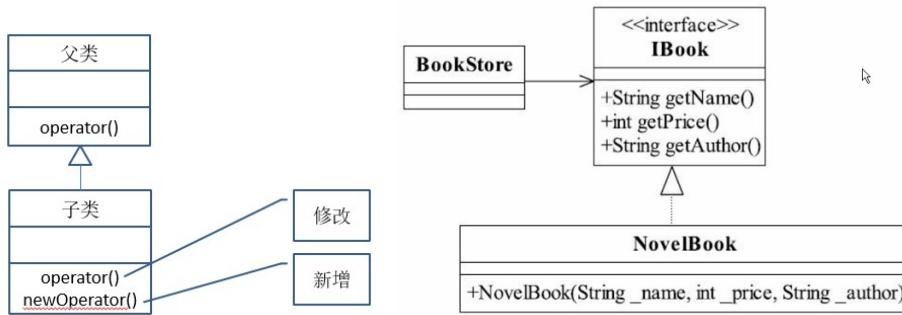
图7 “图书管理系统”部署图

8 开放封闭原则 (OCP)

Open Close Principle

软件应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。

Meyer 开闭原则:Bertrand Meyer 在 1988 年提出，一个实体（类、函数）类的实现只应该因错误而修改，新的或者改变的特性应该通过新建不同的类实现。



遵循 OCP 带来的好处:

- 提高程序的可重用性: 若应对需求的变更，都是对原有的类进行修改，则可能使原有的类累积过多的功能，则不利于功能重用。
- 提高程序的可维护性: 采用新增代码的方式应对新需求，可降低修改原有代码带来的难度；避免引入新错误；降低测试难度。

```
1 // 书籍接口
2 public interface IBook {
3     //名称
4     public String getName();
5     //售价
6     public int getPrice();
7     //作者
8     public String getAuthor();
9 }
```

```
1 // 小说类
2 public class NovelBook implements IBook {
3     private String name;
4     private int price;
5     private String author;
6     //构造函数
7     public NovelBook(String _name,int _price,String _author){
8         this.name = _name;
9         this.price = _price;
```

```

10     this.author = _author;
11 }
12 //获得作者
13 public String getAuthor() { return this.author; }
14 //获得书名
15 public String getName() { return this.name; }
16 //获得价格
17 public int getPrice() { return this.price; }
18 }
```

```

1 // 书店类
2 public class BookStore {
3     private final static ArrayList<IBook> bookList = new ArrayList<IBook>();
4     //static静态模块初始化数据
5     static{
6         bookList.add(new NovelBook("天龙八部",3200,"金庸"));
7         bookList.add(new NovelBook("巴黎圣母院",5600,"雨果"));
8         bookList.add(new NovelBook("悲惨世界",3500,"雨果"));
9         bookList.add(new NovelBook("战争和人",4300,"王火")); }
10    //模拟书店卖书
11    public static void main(String[] args) {
12        System.out.println("-----书店卖出去的书籍记录如下: -----");
13        for(IBook book:bookList){
14            System.out.println("书籍名称: " + book.getName()
15                         +"\t书籍作者: "+book.getAuthor()
16                         +"\t书籍价格: "+book.getPrice());
17        }
18    }
```

假设现在需要对书籍采取打折措施，每本小说可打 8 折。

8.1 修改方式一

修改书籍接口，增加 `getDiscountRate()` 方法。修改本该稳定的顶端抽象类接口，影响面过大；

```

1 // 书籍接口
2 public interface IBook {
3     //名称
4     public String getName();
5     //售价
```

```
6 public int getPrice();
7 //作者
8 public String getAuthor();
9 //折扣
10 public float getDiscountRate();
11 }
```

```
1 //书店类
2 public class BookStore{
3 book.getPrice()*book.getDiscountRate();
4 // ...
5 }
```

```
1 //小说类
2 public class NovelBook implements IBook {
3 //折扣
4 private float discountRate;
5 public float getDiscountRate();
6 // ...
7 }
```

8.2 修改方式二

修改小说类的 `getPrice()` 方法, 可能影响原来依赖 `NovelBook` 类的模块; 可能引入新的 Bug.

```
1 //小说类
2 public class NovelBook implements IBook {
3 //折扣
4 private float discountRate;
5 public NovelBook(String name,int price,String author,float discount);
6 public float getPrice() {
7     return price*discountRate;
8 }
9 // ...
10 }
```

8.3 修改方式三

新增支持打折的小说类:

```
1 public class NovelBookWithDiscount extends NovelBook{
2     //折扣
3     private float discountRate;
4     public NovelBookWithDiscount(String name,
5         int price, String author,
6         float discount);
7     public float getPrice() //重写父类中的方法
8     {
9         return price*discountRate;
10    }
11    // ...
12 }
```

9 里氏代换原则 (LSP)

Liskov Substitution Principle

1988 年, 由麻省理工的 Barbara Liskov 提出, 其思想表达为: 如果对每一个类型为 T1 的对象 o1, 都有类型为 T2 的对象 o2, 使得以 T1 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 T2 是类型 T1 的子类型.

里氏代换原则是继承复用的基石. 只有当派生类可以替换掉基类, 软件功能不会受到影响, 基类才能真正被复用, 而派生类也才能够在基类的基础上增加新的功能.

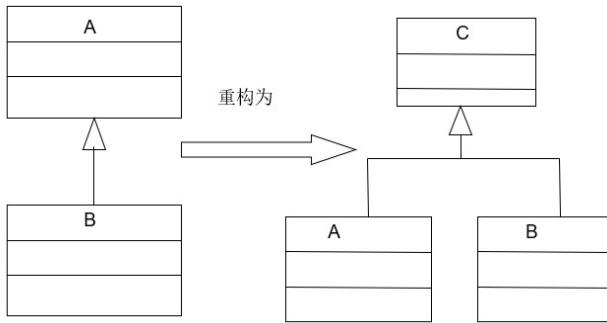
JAVA 语言对里氏代换原则的支持: 在编译时期, JAVA 语言会检查一个程序是否符合里氏代换原则. 里氏代换原则要求凡是使用基类的地方, 子类型一定适用, 因此子类必须具备基本类型的全部接口.

Java 语言对里氏代换支持的局限: Java 编译器的检查是有局限的, JAVA 编译器不能检查一个系统在实现和业务逻辑上是否满足里氏代换原则.

9.1 从代码重构的角度理解

里氏代换原则讲的是基类与子类的关系. 只有当这种关系存在时, 里氏代换关系才能存在, 反之不存在. 如果有两个具体类 A 和类 B 之间的关系违反了里氏代换原则的设计, 根据具体情况可以在下面的两种重构方案中选择一个:

创建新的抽象类 C, 作为两个类的基类:



下面介绍一个判断正方形是否是长方形的问题。一个长方形(Rectangle)类和正方形(Square)类的代码如下：

```

1 public class Rectangle{
2 ...
3     void setWidth(int width){
4         this.width=width;
5     }
6     void setHeight(int height){
7         this.height=height
8     }
9 }
```

```

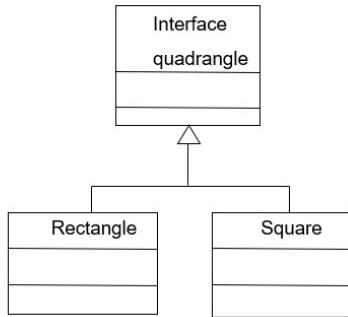
1 public class Square{
2 ...
3     void setWidth(int width){
4         this.width=width;
5         this.height=width;
6     }
7     void setHeight(int height){
8         this.setWidth(height);
9     }
10 }
```

假设有如下函数，需要对长方形对象的边长进行改动：

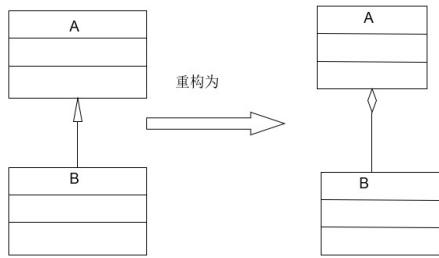
```

1 public void resize(Rectangle r){
2     while(r.getHeight()<=r.getWidth()){
3         r.setHeight(r.getWidth()+1);
4     }
5 }
```

如果将正方形作为长方形的子类, 这样, 只要 width 或 height 被赋值, 那么 width 和 height 会同时赋值, 从而长方形的长和宽总是一样的. 换言之, 里氏代换原则就被破坏了. Rectangle 和 Square 都应该同属于四边形 (Quadrangle) 类的子类, 如下图所示:



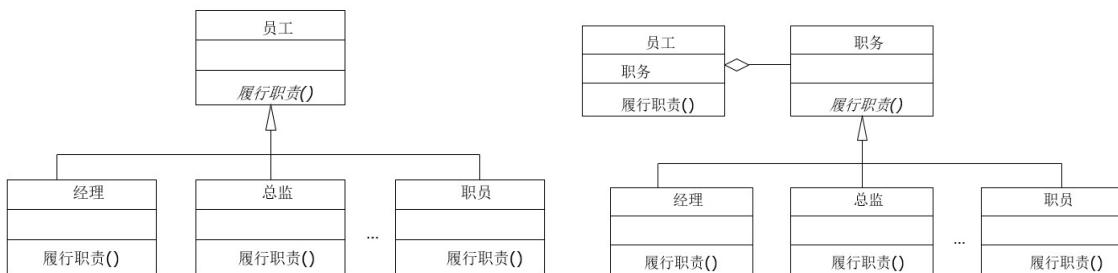
从 B 到 A 的继承关系改为委派 delegate 关系, 如下图:



```

1 class A {
2     void method1();
3     void method2() {
4         b.method2();
5     }
6     B b;
7 }
8 class B {
9     void method2();
10}
  
```

从“员工”类派生出“经理”、“总监”、“组长”、“职员”等:



将不恰当的继承关系改为组合/聚合关系。将“职务”作为“员工”的一个成员：

10 接口隔离原则 (ISP)

Interface Segregation Principle

接口隔离原则指的是：使用多个专门的细粒度接口比总使用单一的粗粒度接口要好。换言之，从一个客户端类的角度来讲：一个类对另外一个类的依赖应当是建立在最小的接口上。

角色的合理划分：可以将接口理解为一个类所具有的公有方法的集合。接口的划分就直接带来类型的划分。一个接口相当于剧本中的一种角色，而此角色在一个舞台上由哪一个演员来演则相当于接口的实现。因此，一个接口应当简单地代表一个角色，而不是多个。我们将这种角色划分的原则叫做角色隔离原则。

接口污染：过于臃肿的接口是对接口的污染。准确而恰当地划分角色以及角色所对应的接口，是面向对象设计的一个重要组成部分。将没有关系的接口合并在一起，形成一个臃肿的大接口，是对角色和接口的污染。

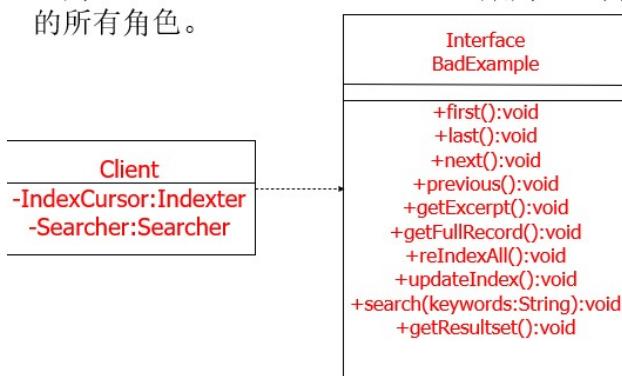
定制服务也是一个重要的设计原则。含义：如果客户端仅仅需要某一些方法的话，就应当向客户端提供这些需要的方法，而不要提供其不需要的方法。这样做的效果：

- 很整洁（设计师需要花较多时间在分析和划分这些接口）
- 系统的可维护性。（尽量不提供 public 接口）

10.1 全文搜索引擎的系统设计

一个动态的资料网站将大量的文件资料存储在文件中或关系数据库里面，用户可以通过输入一个和数个关键词进行全网站的全文搜索。这个搜索引擎需要维护一个索引库，索引库以文本文件方式存于文件系统中。在源数据被修改，删除或者增加时，搜索引擎要做相应的动作，以保证引擎的索引文件也被相应的更新。

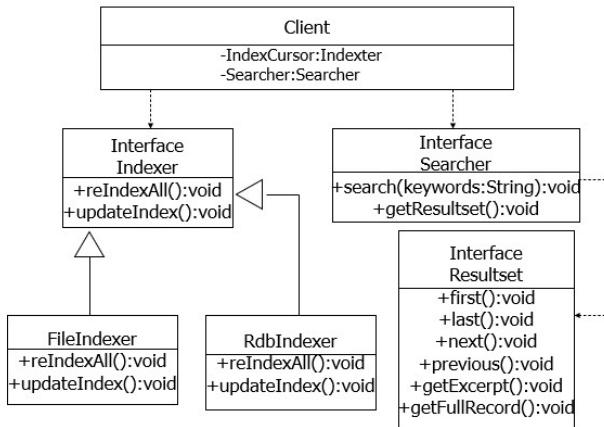
首先，下面所示为一个不好的解决方案。一个叫做 BadExample 的接口负责所有的操作，从提供搜索功能到建立索引功能，甚至包括搜索结果集合的功能均在一个接口内提供。如下图所示：



这个解决方案违反了角色风格原则，把不同功能的接口放在一起，由一个接口给出包括搜

索器角色，索引生成器以及搜索结果集角色在内的所有角色。

角色的分割：



由图中可看出，搜索引擎的功能被分割成三个角色：搜索器角色，索引生成器角色，搜索结果集角色。

以“2”为例，由于索引生成因数据的格式不同而不同，故分为 RdbIndexer 和 FileIndexer 两种实现。FileIndexer 类代表对诸如 *.txt, *.html, *.doc 以及 *.pdf 等文件类型的数据生成全文索引，而 RdbIndexer 则针对关系数据库的数据进行全文索引生成。这两个实现扮演的同为索引生成器角色，就好像扮演同样角色的两个不同演员一样。

搜索器角色则是与索引生成器角色完全不同的角色，它提供用户全文搜索功能。用户传进一些关键字，搜索器角色则返回一个 ResultSet。搜索结果集角色就是 ResultSet。它给用户提供对集合进行迭代走访的功能，如 first() 将光标移到集合的第一个元素； last() 将光标移到集合的最后一个元素； next() 将光标移到集合的下一个元素； previous() 将光标移到集合的前一个元素；而 getExcerpt() 则返回当前记录的摘要；而 getFullRecord() 则将记录的全文返回。

11 依赖倒转原则 (DIP)

Dependence Inversion Principle

倒转的意义：传统的面向过程的设计办法倾向于使高层次的模块依赖于低层次的模块；抽象层次依赖于具体层次。依赖倒转原则就是要把这个错误的依赖关系倒转过来，这就是依赖倒转原则的来由。抽象层次含有宏观的和重要业务逻辑，是必然性的体现，而具体层次是含有一些次要的与实现有关的算法和逻辑，带有相当大的偶然性选择。

复用与可维护性的倒转：从复用的角度来看，抽象层次的模块是设计者应当复用的。但是传统的过程性的设计中，复用却侧重于具体层次模块的复用，比如算法复用，数据结构复用，函数库复用等，都不可避免是具体层次模块的复用。较高层次的结构依赖于较低层次的结果，接下去不断的循环直到依赖于每一行的代码。较低层次的修改就会影响到较高层次的修改，直到高层次逻辑的修改。

三种耦合关系：

- 零耦合：如果两个类没有耦合关系，就称为零耦合。

- 具体耦合：具体耦合发生在两个具体的（可实例化的）类之间，经由一个类对另一个具体类的直接引用造成的。
- 抽象耦合：抽象耦合关系发生在一个具体类和一个抽象类（或者 Java 接口）之间，使两个必须发生联系的类之间存有最大的灵活性。

简单的说，DIP 要求依赖于抽象耦合。依赖倒转的表述是：抽象不应当依赖于细节，细节应当依赖于抽象。另一种表述是：要针对接口编程，不要针对实现编程。

针对接口编程：使用 Java 接口和抽象 Java 类进行变量的类型声明，参数的类型声明，方法的返回类型声明，以及数据类型的转换等。不要针对实现编程：不应当使用具体 Java 类进行变量的类型声明。设计师希望遵守开闭原则，那么倒转依赖原则就是达到要求的途径。

如何做到依赖倒转原则：以抽象方式耦合是依赖倒转原则的关键。由于一个抽象耦合关系总要涉及到具体类从抽象类继承，并且保证在任何引用到基类的地方都可以转换成其子类，因此，里氏代换原则是依赖倒转原则的基础。

依赖倒转原则的优缺点：

- 依赖倒转原则要求模块之间依赖于抽象耦合。
- 抽象不应当依赖于细节；细节应当依赖于抽象。
- 实施重点：从问题的具体细节中分离出抽象，以抽象方式对类进行耦合；
- 不足：导致生成大量的类；

抽象方式耦合的局限：在某些情况下，如果一个具体类发生变化的可能性很小，那么抽象耦合能发挥的好处便十分有限，这时使用具体耦合反而会更好。

变量被声明时的类型叫做变量的**静态类型**，变量所引用的对象的真实类型叫做变量的**实际类型**。

引用对象的抽象类型：很多情况下，当一个 Java 程序需要引用一个对象，如果这个对象有一个抽象类型的话，应当使用这个抽象类型作为变量的静态类型。这就是面向接口编程的具体表现。

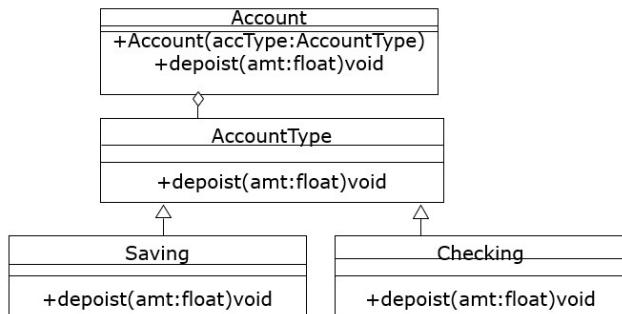
若“家禽”代表抽象，“鸡”代表具体，则：`家禽 x = new 鸡();`

尽量不要使用：`Vector employees = new Vector();`，应当使用：`List employees = new Vector();`。

区别：前者使用具体类作为变量的类型，而后者使用一个抽象类（List 是 Java 接口）作为类型。

好处：在决定将 Vector 类型转换成 ArrayList 时，需要改动的很少。`List employees = new ArrayList();`。程序具有更好的灵活性，因为除去调用构造函数的那一行语句之外，程序的其余部分根本察觉不到变化。因此，只要一个被引用的对象存在抽象类型，就应当在任何引用此对象的地方使用抽象类型，包括变量的类型声明，方法返还类型的声明，属性变量类型的声明等。

11.1 例子：账户和账户的类型



```
1 // Account
2 Public class Account {
3     private AccountType accountType;
4     public Account(AccountType accType)
5     {
6         accountType=accType;
7     }
8     public void deposit(float amt)
9     {
10        accountType.deposit(amt);
11    }
12 }
13 Account acc=new Account(new Checking());
```

```
1 // AccountType
2 abstract public class AccountType {
3     public abstract void deposit(float amt);
4 }
```

```
1 // Saving
2 public class Saving extends AccountType {
3     public void deposit(float amt) {
4         //write your code here
5     }
6 }
```

```
1 // Checking
2 public class Checking extends AccountType {
3     public void deposit(float amt) {
```

```
4 //write your code here  
5 }  
6 }
```

在这个例子里，Account 类依赖于 AccountType 这个抽象类型，而不是它的子类型。AccountType 有两个子类型：

- 储蓄账号：以 Saving 具体类代表
- 支票账号：以 Checking 具体类代表

Account 类并不依赖于具体类，因此当有新的具体类型添加到系统中时，Account 类不必改变。例如，如果系统引进了一个新型的账号：MoneyMarket 类型，Account 类以及系统里面所有其他的依赖于 AccountType 抽象类的客户端均不必改变。其源代码如下：

```
1 public class MoneyMarket extends AccountType {  
2     public void deposit(float amt) {  
3         //write your code here  
4     }  
5 }
```

12 组合/聚合复用原则 (CARP)

Composition/Aggregation Reuse Principle

组合、聚合复用原则就是在一个新的对象里面应尽量使用一些已有的对象，使之成为新对象的一部份，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。这个原则有一个简短的描述：要尽量使用组合、聚合，尽量不要使用继承。

组合与聚合的区别：组合和聚合均是关联的特殊情形。聚合：拥有关系或整体与部分的关系；组合：一种强得多的拥有关系；

复用的基本种类：在面向对象的设计里，有两种基本的方法可以在不同的环境中复用已有的设计和实现，即通过组合/聚合或继承。

组合/聚合复用的优点：

- 组合/整体对象存取成分/部分对象的唯一方法是通过成分/部分对象的接口。
- 这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的，容易实现封装。
- 这种复用所需的依赖较少。
- 这种复用可以在运行时间内动态进行，新对象可以动态的引用与成分对象类型相同的其它对象。

组合/聚合复用的缺点：采用这种复用关系的系统会有较多的对象需要管理。

继承复用的优点：实现新的子类较为容易，因为超类的大部分功能可以通过继承关系自动进入子类。修改和扩展继承而来的实现较为容易。

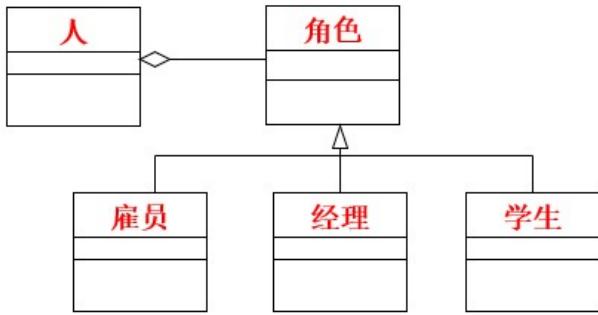
继承复用的缺点：

- 继承复用破坏封装，因为继承将超类的实现细节暴露给子类。由于超类的内部细节常常是对于子类透明的，所以这种复用是透明的复用，又称“白箱”复用。
- 如果超类发生改变，那么子类的实现也不得不发生改变。
- 从超类继承而来的实现是静态的，不可能在运行时间内发生改变，没有足够的灵活性。

区分 “Is-A” 和 “Has-A”：

- “Is-A” 代表一个类是另外一个类的一种
- “Has-A” 代表一个类是另一个类的一个部分，而不是另一个类的特殊种类。
- 对于 “Is-A” 应该考虑使用继承，而 “Has-A” 使用组合/聚合。

应当采用组合/聚合的例子：



与里氏代换原则联合使用：

- 里氏代换原则是继承复用的基石。只有当派生类可以替换掉基类，软件模块的功能不会受到影晌时，基类才真正被复用，而派生类也才能够在基类的基础上增加新的功能。
- 如果两个类的关系是“Has-A”而不是“Is-A”关系，这两个类一定违反里氏代换原则。
- 只有两个类满足里氏代换原则，才有可能是“Is-A”关系。

13 迪米特法则 (LoD)

迪米特法则 (Law of Demeter) 又称为最少知识原则 (一个对象应当对其他对象尽可能少的了解) .

迪米特法则的各种表述: 没有任何一个其他的 OO 设计法则有如此之多的表述方式，下面给出的是众多的表述中较有代表性的几种.

- 只与你直接的朋友们通信;
- 不要跟陌生人说话;
- 每一个软件模块对其他的模块都只有最少的知识，而且局限于那些与本模块密切相关的软件模块.

在上面的表述里面，什么是“直接”，“陌生”和“密切”则被有意识的模糊化了，以便在不同的环境下可以有不同的解释.

13.1 狹义的迪米特法则

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, 如果其中一个类需要调用另一个类的某一个方法的话, 可以通过第三者转发这个调用.

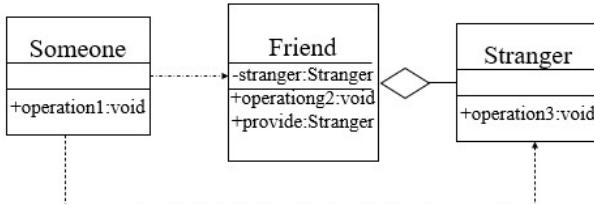
以下条件是成为“朋友”条件:

- 当前对象本身 (this)
- 以参数形式传入到当前对象方法中的对象
- 当前对象的属性直接引用的对象
- 当前对象的属性如果是一个聚集, 那么聚集中的元素也都是朋友
- 当前对象所创建的对象

狭义的迪米特法则的缺点:

- 会在系统内造出大量的小方法, 散落在系统的各个角落. 这些方法仅仅是传递间接的调用, 并且与系统中的业务逻辑无关.
- 为了克服狭义迪米特法则的缺点, 可以使用依赖倒转原则, 引入一个抽象的类型引用“抽象陌生人”对象, 使“A”依赖于“抽象陌生人 C”, 换言之, 就是将“抽象陌生人 C”变成朋友.

不满足迪米特法则的设计: 有三个类, 分别是 Someone, Friend 和 Stranger. 其中 Someone 与 Friend 是朋友, 而 Friend 与 Stranger 是朋友. 相关类图如下所示:



```
1 // Someone
2 public class Someone {
3     public void operation1(Friend friend) {
4         Stranger stranger=friend.provide();
5         stranger.operation3();
6     }
7 }
```

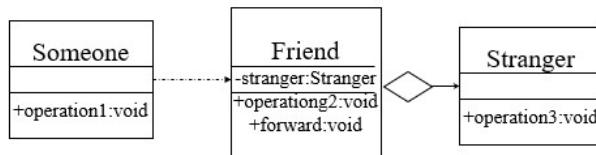
```
1 // Friend
2 public class Friend {
3     private Stranger stranger=new Stranger();
4     public void operation2() {}
5     public Stranger provide() {
```

```

6     return stranger;
7 }
8 }
```

显然,Someone 的方法 operation1() 不满足迪米特法则. 因为这个方法引用了 Stranger 对象, 而 Stranger 对象不是 Someone 的朋友.

可以使用迪米特法则对上面的例子改造, 改造的方法就是调用转发. 改造后的情况如下图所示:



```

1 // Someone
2 public class Someone {
3     public void operation1(Friend friend) { friend.forward(); }
4 }
```

```

1 // Friend
2 public class Friend {
3     private Stranger stranger=new Stranger();
4     public void operation2() {
5         System.out.println( "In Friend.operation2()" );
6     }
7     public void forward() {
8         stranger.operation3();
9     }
10 }
```

Friend 类的 forward() 方法所做的就是以前 Someone 要做的事情, 使用了 Stranger 的 operation3() 方法, 而这种 forward() 方法叫做转发方法. 由于使用了调用转发, 使得调用的具体细节被隐藏在 Friend 内部, 从而使 Someone 与 Stranger 之间的直接联系被省略掉了. 这样一来, 使得系统内部的耦合度降低. 在系统的某一个类需要修改时, 仅仅会直接影响到这个类的朋友们, 而不会直接影响到其余部分.

13.2 广义的迪米特法则

一个模块设计得好坏的一个重要的标志就是该模块在多大的程度上将自己的内部数据与实现有关的细节隐藏起来. 信息的隐藏非常重要的原因在于, 它可以使各个子系统之间脱耦, 从而允许它们独立地被开发, 优化以及修改. 迪米特法则的主要用意是控制信息的过载. 在运用迪米特法则到系统的设计中时, 要注意以下几点:

- 在类的划分上，应当创建弱耦合的类。类之间的耦合越弱，就越有利于复用。
- 在类的结构设计上，每一个类都应当尽量降低成员的访问权限。
- 在类的设计上，只要可能，一个类应当设计成不变类。
- 在对其他类的引用上，一个对象对其他对象的引用应降到最低。
- 尽量限制局部变量的有效范围。

广义迪米特法则在类的设计上的体现：

- 优先考虑将一个类设置成不变类：Java 语言的 API 中提供了很多的不变类，不变类易于设计，实现和使用。一个对象与外界的通信大体可以分为：改变这个对象的状态的和不改变这个对象的状态的。如果一个对象的内部状态根本就是不可能改变的，那么它与外界的通信自然也就大大减少了。当设计任何一个类的时候，都优先考虑这个类的状态是否需要改变。即便一个类必须是可变类，在给它的属性设置赋值方法的时候，也要保持谨慎的态度。除非真的需要，否则不要为一个属性设置赋值方法。
- 尽量降低一个类的访问权限：在满足一个系统对这个类的需求的同时，应当尽量降低这个类的访问权限。例如：
 1. package-private：这是默认访问权限，如果一个类是 package-private 的，那么它就只能从当前库（包）访问。
 2. public：如果一个类是 public 的，那么这个类从当前库和其他库都可以访问。

14 简单工厂模式

设计模式是一套解决软件开发过程中某些常见问题的通用解决方案，是已被反复使用且证明其有效性的设计经验的总结。目的是建立具有可复用、可维护、可扩展的软件系统。

- **创建型模式**，共 5 种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
- **结构型模式**，共 7 种：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- **行为型模式**，共 11 种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

创建型的设计模式：创建模式是对类的实例化过程的抽象。一些系统在创建对象时，需要动态地决定怎样创建对象，创建哪些对象，以及如何组合和表示这些对象，实现对象的“创建”与“使用”相分离。

简单工厂模式：

- 简单工厂模式又叫做静态工厂方法 (Static Factory Method) 模式。
- 简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。
- 工厂模式专门负责将大量有共同接口的类实例化。工厂模式可以动态决定将哪一个类实例化，不必事先知道每次要实例化哪一个类。

```

1 class 消费者 {
2     public void 消费() {

```

```

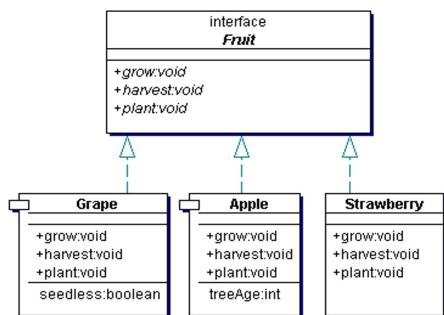
3 鼠标 obj1 = 工厂.生产( “鼠标” );
4 铅笔 obj2 = 工厂.生产( “铅笔” );
5 // ...
6 }
7 }

```

14.1 简单工厂模式的引进

比如说有一个生产水果的农场，专门向市场销售各类水果。在这个系统里需要描述下列的水果：

- 葡萄 Grape
- 草莓 Strawberry
- 苹果 Apple



```

1 public interface Fruit {
2     void grow(); // 生长
3     void harvest(); // 收获
4     void plant(); // 种植
5 }

```

```

1 public class Apple implements Fruit {
2     private int treeAge;
3     // 生长
4     public void grow() { System.out.println("Apple is growing"); }
5     // 收获
6     public void harvest() { System.out.println("Apple has been harvested"); }
7     // 种植
8     public void plant() { System.out.println("Apple has been planted"); }
9     // 树龄的取值方法
10    public int getTreeAge() { return treeAge; }
11    // 树龄的赋值方法
12    public void setTreeAge(int treeAge) { this.treeAge = treeAge; }

```

```
13 }
```

```
1 public class Grape implements Fruit {  
2     private boolean seedless;  
3     //生长  
4     public void grow() { System.out.println ("Grape is growing"); }  
5     //收获  
6     public void harvest() { System.out.println("Grape has been harvested"); }  
7     //种植  
8     public void plant() { System.out.println("Grape has been planted"); }  
9     //有无籽的取值方法  
10    public boolean getSeedless() { return seedless; }  
11    //有无籽的赋值方法  
12    public void setSeedless(boolean seedless) { this.seedless = seedless; }  
13 }
```

```
1 public class Strawberry implements Fruit {  
2     public void grow() {  
3         System.out.println("Strawberry is growing");  
4     }  
5     public void harvest() {  
6         System.out.println("Strawberry has been harvested");  
7     }  
8     public void plant() {  
9         System.out.println("Strawberry has been planted");  
10    }  
11 }
```

农场的园丁也是系统的一部分，由 FruitGardener 类代表。其结构由下面的类图描述。FruitGardener 类会根据客户端的要求，创建出不同的水果对象，比如苹果 (Apple)，葡萄 (Grape) 或草莓 (Strawberry) 的实例。

```
1 public class FruitGardener {  
2     //静态工厂方法  
3     public static Fruit factory(String which)  
4     throws BadFruitException {  
5         if (which.equalsIgnoreCase("apple")) {  
6             return new Apple();  
7         } else if (which.equalsIgnoreCase("strawberry")) {  
8             return new Strawberry();  
9         } else if (which.equalsIgnoreCase("grape")) {
```

```

10     return new Grape();
11 } else {
12     throw new BadFruitException("Bad fruit request");
13 }
14 }
15 }
16
17 class BadFruitException extends Exception {
18     public BadFruitException(String msg) {
19         super(msg);
20     }
21 }

```

可以看出，园丁类提供了一个静态工厂方法。在客户端的调用下，这个方法创建客户端所需要的水果对象。如果客户端的请求是系统所不支持的，工厂方法就会抛出一个 BadFruitException 异常。在使用时，客户端只需调用 FruitGardener 的静态方法 factory() 即可。

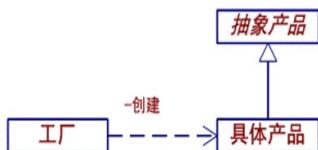
```

1 try {
2     FruitGardener.factory("grape");
3     FruitGardener.factory("apple");
4     FruitGardener.factory("strawberry");
5     // ...
6 } catch(BadFruitException e) {
7     // ...
8 }

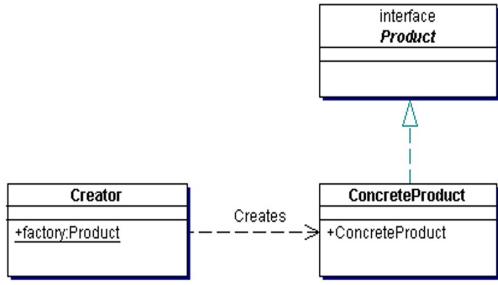
```

14.2 简单工厂模式的结构

简单工厂模式是类的创建模式，这个模式的一般性结构如下图所示。



角色与结构: 简单工厂模式就是由一个工厂类可以根据传入的参数决定创建出哪一种产品类的实例。下图所示为以一个示意性的实现为例说明简单工厂模式的结构。



从上图可看出，简单工厂模式涉及到工厂角色、抽象产品角色及具体产品角色三个角色：

- 工厂类 (Creator) 角色：担任这个角色的是工厂方法模式的核心，含有与应用紧密相关的业务逻辑。工厂类在客户端的直接调用下创建产品对象，它往往由一个具体 Java 类实现。
- 抽象产品 (Product) 角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。抽象产品角色可以用一个 Java 接口或者 Java 抽象类实现。
- 具体产品 (Concrete Product) 角色：工厂方法模式所创建的任何对象都是这个角色的实例，具体产品角色由一个具体 Java 类实现。

抽象产品角色的主要目的是给所有的具体产品类提供一个共同的类型，在最简单的情况下，可以简化为一个标识接口。

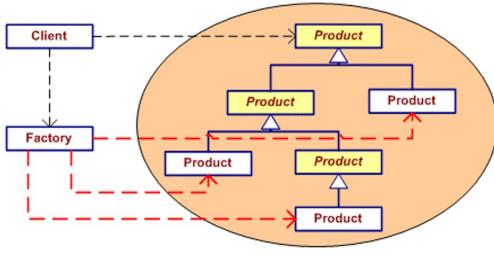
```

1 public class Creator {
2     // 静态工厂方法
3     public static Product factory() {
4         return new ConcreteProduct();
5     }
6 }
7 public interface Product { }
8 public class ConcreteProduct implements Product {
9     public ConcreteProduct() { }
10}
  
```

如果系统仅有一个具体产品角色的话，那么就可以省略掉抽象产品角色。

14.3 简单工厂模式的实现

多层次的产品结构: 在实际应用中，产品可以形成复杂的等级结构。这个时候，简单工厂模式采取的是以不变应万变的策略，一律使用同一个工厂类。如下图所示。图中从 Factory 类到各个 Product 类的虚线代表创建(依赖)关系；从 Client 到其他类的联线是一般依赖关系。



这样做的好处是设计简单，产品类的等级结构不会反映到工厂类中来。但是这样做的缺点是，增加新的产品必将导致工厂类的修改。

使用 Java 接口或者 Java 抽象类：

- 如果具体产品类彼此之间没有共同的业务逻辑，那么抽象产品角色可以由一个 Java 接口扮演；
- 相反，如果这些具体产品类彼此之间确有共同的业务逻辑，那么这些公有的逻辑就应当移到抽象角色里面，这就意味着抽象角色应当由一个抽象类扮演。在一个类型的等级结构里面，共同的代码应当尽量向上移动，以达到共享的目的，如下图所示。

工厂角色与抽象产品角色合并: 在有些情况下，工厂角色可以由抽象产品角色扮演。典型的应用就是 `java.text.DateFormat` 类，一个抽象产品类同时是子类的工厂。

```

1 Date date = new Date();
2 //日期格式，精确到日 2017-4-16
3 DateFormat df1 = DateFormat.getDateInstance();
4 System.out.println(df1.format(date));
5
6 //可以精确到秒 2017-4-16 12:43:37
7 DateFormat df2 = DateFormat.getTimeInstance();
8 System.out.println(df2.format(date));
9
10 //只显示出时时分秒 12:43:37
11 DateFormat df3 = DateFormat.getInstance();
12 System.out.println(df3.format(date));

```

三个角色全部合并: 如果抽象产品角色已经被省略，而工厂角色就可以与具体产品角色合并。换言之，一个产品类为创建自身的工厂。显然，三个原本独立的角色：工厂角色、抽象产品以及具体产品角色都已经合并成为一个类，这个类自行创建自己的实例。

```

1 public class ConcreteProduct {
2     public ConcreteProduct() { }
3     // 静态工厂方法
4     public static ConcreteProduct factory() {
5         return new ConcreteProduct();
6     }

```

这种退化的简单工厂模式与单例模式以及多例模式有相似之处，但是并不等于单例或者多例模式。

产品对象的循环使用和登记式的工厂方法: 在很多情况下，产品对象可以循环使用。换言之，工厂方法可以循环使用已经创建出来的对象，而不是每一次都创建新的产品对象。如果工厂方法总是循环使用同一个产品对象，那么这个工厂对象可以使用一个属性来存储这个产品对象。每一次客户端调用工厂方法时，工厂方法总是提供这同一个对象。

```

1 class Factory {
2     static Product obj;
3     public static void factory(int type) {
4         // ...
5         if(obj==null)
6             obj=new ConcreteProduct();
7         return obj;
8         // ...
9     }
10 }
```

14.4 简单工厂模式的优缺点

优点: 客户端则可以免除直接创建产品对象的责任，而仅仅负责消费产品。对于消费者角色来说，任何时候需要某种产品，只需要向工厂角色(下订单)请求即可，而无需知道产品创建细节。实现了客户端类与产品类的解耦。

缺点: 当产品种类增加时，工厂类的工厂方法也必须随之修改。

这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类(God Class)。如果这个全能类代表的是农场的一个具体园丁的话，那么这个园丁就需要对所有的产品负责，成了农场的关键人物，他什么时候不能正常工作了，整个农场都要受到影响。

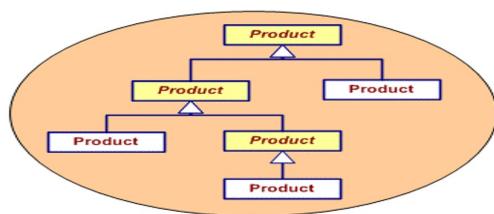
对开闭原则的支持: 开闭原则要求一个系统的设计能够允许系统在无需修改的情况下，扩展其功能。那么简单工厂模式是否满足这个条件？要回答这个问题，首先需要将系统划分成不同的子系统，再考虑功能扩展对于这些子系统的要求。一般而言，一个系统总可以划分成为产品的消费者角色(Client)、产品的工厂角色(Factory)以及产品角(Product)三个子系统。在这个系统中，功能的扩展体现在引进新的产品上。开闭原则要求系统允许当新的产品加入系统中，而无需对现有代码进行修改。这一点对于产品的消费角色是成立的，而对于工厂角色是不成立的。简单工厂角色只在有限的程度上支持开闭原则。

15 工厂方法模式

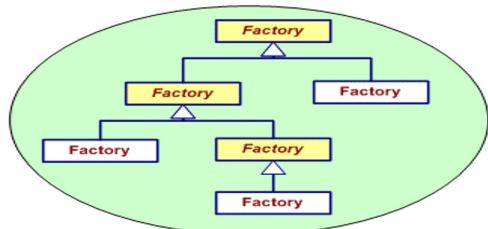
工厂方法模式 (Factory Method) 是类形式的创建模式，又叫做虚拟构造器 (Virtual Constructor) 模式或者多态性工厂 (Polymorphic Factory) 模式。工厂方法模式将定义一个创建产品对象的工厂接口，将实际创建功能放到具体工厂子类中。

简单工厂模式的优缺点: 在简单工厂模式中，一个工厂类处于对产品类实例化的中心位置上，它知道每一个产品，它决定哪一个产品类应当被实例化。这个模式的优点是允许客户端相对独立于产品创建的过程，并且在系统引入新产品的时候无需修改客户端，也就是说，它在某种程度上支持开闭原则。这个模式的缺点是对开闭原则的支持不够，因为如果有新的产品加入到系统中去，就需要修改工厂类，将必要的逻辑加入到工厂类中。

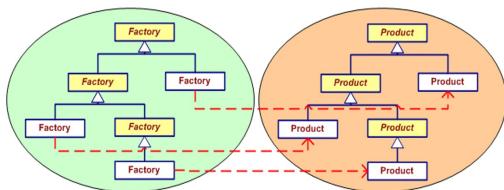
平行的等级结构: 在一个系统设计中，常常是首先有产品角色，然后有工厂角色。在可以应用工厂方法模式的情形下，一般都会有一个产品的等级结构，其由一个（甚至多个）抽象产品和多个具体产品组成。产品的等级结构如下图所示，树图中有阴影的是树枝型节点。



在上面的产品等级结构中，出现了多于一个的抽象产品类，以及多于两个的层次。这其实是真实的系统中常常出现的情况。当将工厂方法模式应用到这个系统中去的时候，常常采用的一个做法是按照产品的等级结构设计一个同结构的工厂等级结构。工厂的等级结构如下图所示，树图中有阴影的是树枝型节点。

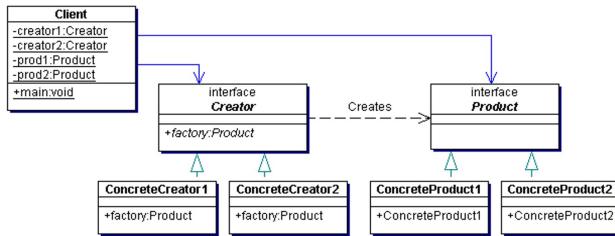


然后由相应的工厂角色创建相应的产品角色，工厂方法模式的应用如下图所示，图中的虚线代表创建（依赖）关系。工厂方法模式并没有限制产品等级结构的层数。



15.1 工厂方法模式的结构

为说明工厂方法模式的结构，下面以最简单的情形为例。此示意性系统类图如下所示。



从上图可以看出，这个使用了工厂方法模式的系统涉及到以下的角色：

- 抽象工厂 (Creator) 角色：担任这个角色的是工厂方法模式的核心，它是应用无关的。任何在模式中创建对象的工厂类必须实现这个接口。在上面的系统中这个角色由 Java 接口 Creator 扮演；在实际的系统中，这个角色也常常使用抽象 Java 类实现。
- 具体工厂 (Concrete Creator) 角色：担任这个角色的是实现了抽象工厂接口的具体 Java 类。具体工厂角色含有与应用密切相关的逻辑，并且受到应用程序的调用以创建产品对象。在本系统中给出了两个这样的角色，也就是具体 Java 类 ConcreteCreator1 和 ConcreteCreator2。
- 抽象产品 (Product) 角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。在本系统中，这个角色由 Java 接口 Product 扮演；在实际的系统中，这个角色也常常使用抽象 Java 类实现。
- 具体产品 (Concrete Product) 角色：这个角色实现了抽象产品角色所声明的接口。工厂方法模式所创建的每一个对象都是某个具体产品角色的实例。
- 客户端 (Client) 角色：为了说明这个系统的使用办法，特地引进了一个客户端角色 Client。这个角色创建工作对象，然后调用工厂对象的工厂方法创建相应的产品对象。

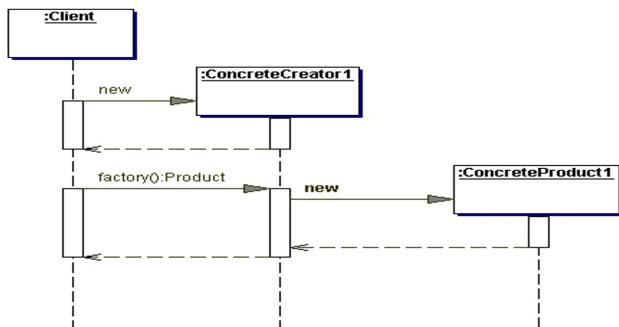
```
1 public interface Creator {  
2     // 工厂方法  
3     public Product factory();  
4 }  
5  
6 public interface Product {}  
7  
8 public class ConcreteCreator1 implements Creator {  
9     // 工厂方法  
10    public Product factory() { return new ConcreteProduct1(); }  
11 }  
12  
13 public class ConcreteProduct1 implements Product {  
14     public ConcreteProduct1() {
```

```

15     // do something
16 }
17 }
18
19 public class Client {
20     private static Creator creator1, creator2;
21     private static Product prod1, prod2;
22     public static void main(String[] args) {
23         creator1 = new ConcreteCreator1();
24         prod1 = creator1.factory();
25         creator2 = new ConcreteCreator2();
26         prod2 = creator2.factory();
27     }
28 }

```

工厂方法模式的活动序列图:



Client 对象的活动可以分成两部分。

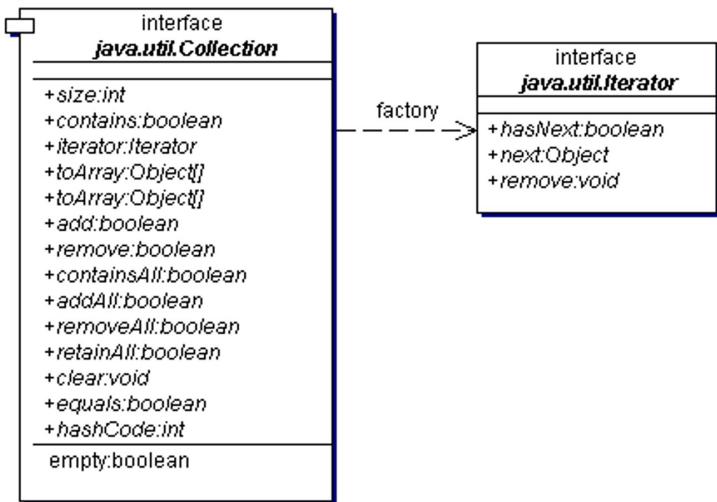
1. 客户端创建 ConcreteCreator1 对象。这时客户端所持有变量的静态类型是 Creator, 而实际类型是 ConcreteCreator1。然后, 客户端调用 ConcreteCreator1 对象的工厂方法 factory(), 接着后者调用 ConcreteProduct1 的构造子创建出产品对象。
2. 客户端创建一个 ConcreteCreator2 对象, 然后调用 ConcreteCreator2 对象的工厂方法 factory(), 而后者调用 ConcreteProduct2 的构造子创建出产品对象.

工厂方法模式和简单工厂模式: 工厂方法模式和简单工厂模式在结构上的不同是很明显的。工厂方法模式的核心是一个抽象工厂类, 而简单工厂模式把核心放在一个具体类上。工厂方法模式可以允许很多具体工厂类从抽象工厂类中将创建行为继承下来, 从而可以成为多个简单工厂模式的综合, 进而推广了简单工厂模式。工厂方法模式退化后可以变得很像简单工厂模式。设想如果非常确定一个系统只需要一个具体工厂类, 那么就不妨把抽象工厂类合并到具体的工厂类中去。由于反正只有一个具体工厂类, 所以不妨将工厂方法改成为静态方法, 这时候就得到了简单工厂模式。

工厂方法模式之所以有一个别名叫**多态性工厂模式**, 显然是因为具体工厂类都有共同的接

口，或者都有共同的抽象父类。如果系统需要加入一个新的产品，那么所需要的就是向系统中加入一个这个产品类以及它所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的产品类而言，这个系统完全支持开闭原则。

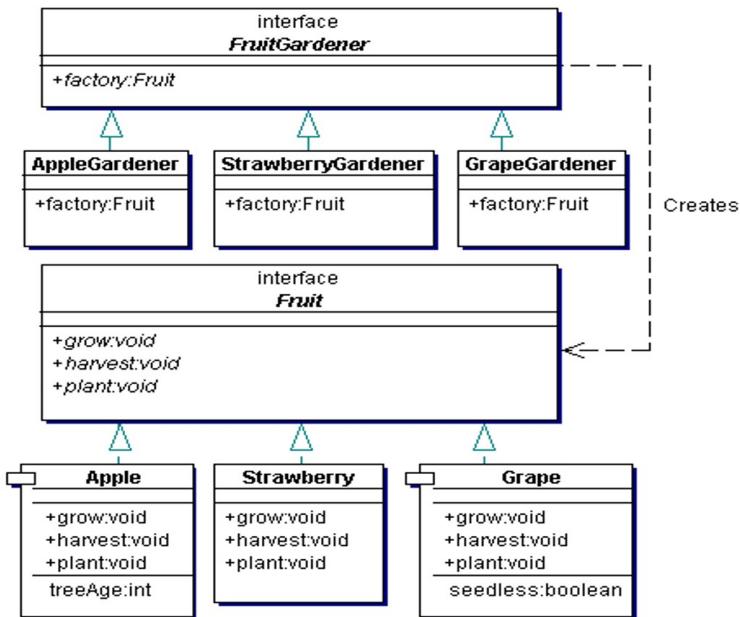
Java 语言中工厂方法模式的例子：在 Java 聚集中的应用，Java 聚集是 Java1.2 版提出来的。多个对象聚在一起形成的总体称之为聚集 (Aggregate)，聚集对象是能够包容一组对象的容器对象。所有的 Java 聚集都实现 java.util.Collection 接口，这个接口规定所有的 Java 聚集必须提供一个 iterator() 方法，返还一个 Iterator (迭代器：first();next();) 类型的对象，如下图所示。一个具体的 Java 聚集对象会通过这个 iterator() 方法接口返还一个具体的 Iterator 类。可以看出，这个 iterator() 方法就是一个工厂方法。



15.2 工厂方法模式在农场系统中的实现

取代了过去的全能角色的是一个抽象的园丁角色，这个角色规定出具体园丁角色需要实现的具体职能，而真正负责作物管理的则是负责各种作物的具体园丁角色。此处仍然考虑前面所讨论过的植物，包括葡萄 (Grape)、草莓 (Strawberry) 以及苹果 (Apple) 等。专业化的管理要求将有专门的园丁负责专门的水果，比如苹果由苹果园丁负责，草莓有草莓园丁负责，而葡萄由葡萄园丁负责。这些苹果园丁、草莓园丁以及葡萄园丁都是实现了抽象的水果园丁接口的具体工厂类，而水果园丁则扮演抽象工厂角色。

多态农场系统的设计图就如下图所示。抽象工厂类 FruitGardener 是工厂方法模式的核心，但是它并不负责具体水果或蔬菜的种植。相反地，这项权力被交给子类，即 AppleGardener StrawberryGardener 以及 GrapeGardener。



```

1 // 抽象工厂角色
2 public interface FruitGardener {
3     // 工厂方法
4     public Fruit factory();
5 }
6
7 public class AppleGardener implements FruitGardener {
8     // 工厂方法
9     public Fruit factory() {
10         return new Apple();
11     }
12 }
13
14 public class StrawberryGardener implements FruitGardener {
15     // 工厂方法
16     public Fruit factory() {
17         return new Strawberry ();
18     }
19 }
20
21 public class GrapeGardener implements FruitGardener {
22     // 工厂方法
23     public Fruit factory() {
24         return new Grape ();
25     }
26 }
  
```

```

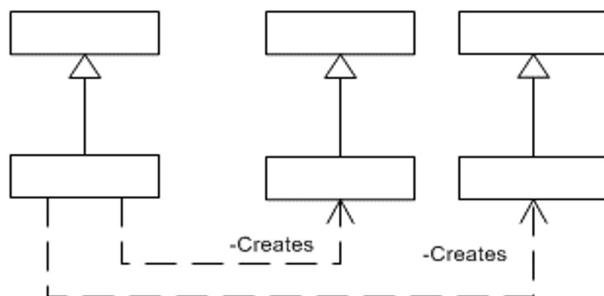
25 }
26 }
27
28 public interface Fruit {
29     abstract void grow();
30     abstract void harvest();
31     abstract void plant();
32 }
33
34 public class Apple implements Fruit {
35     private int treeAge;
36     public void grow() {
37         System.out.println("Apple is growing...");
38     }
39     public void harvest() {
40         System.out.println("Apple has been harvested.");
41     }
42     public void plant() {
43         System.out.println("Apple has been planted.");
44     }
45     public int getTreeAge() { return treeAge; }
46     public void setTreeAge(int treeAge) { this.treeAge = treeAge; }
47 }

```

16 抽象工厂模式

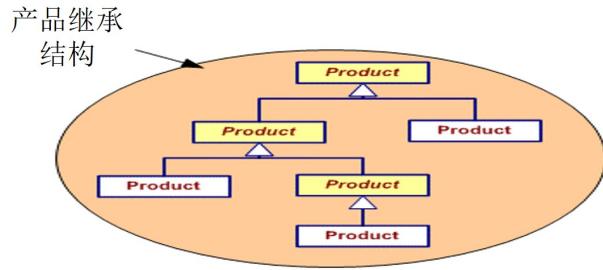
Abstract Factory

抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式的简略类图如下所示。

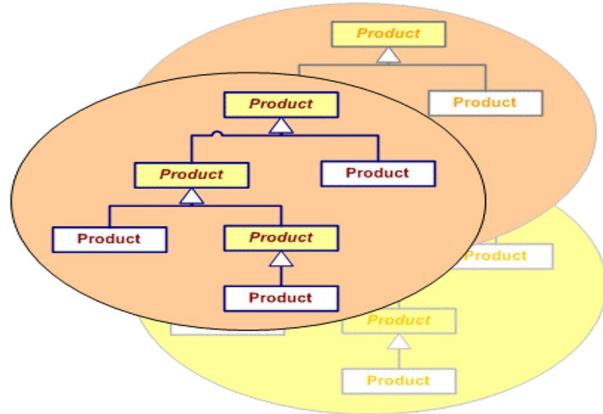


抽象工厂模式面对的问题是多个产品继承结构的系统设计。抽象工厂模式与工厂方法模式

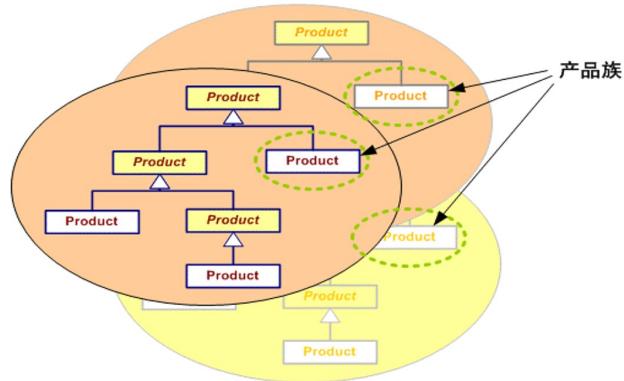
的最大区别就在于，工厂方法模式针对的是一个产品继承结构；而抽象工厂模式则需要面对多个产品继承结构。下图给出了一个产品继承结构。



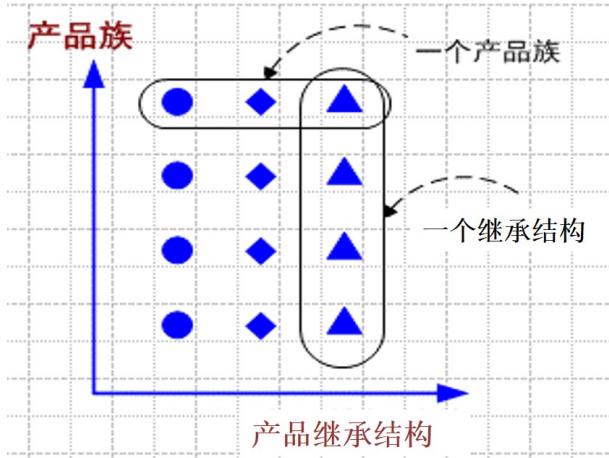
下图则给出了多个相平行的产品继承结构的例子。



为了方便引进抽象工厂模式，特地引进一个新的概念：**产品族**（Product Family）。所谓产品族，是指位于不同产品继承结构中，功能相关联的产品组成的家族。比如在下图中，箭头所指就是三个功能相关联的产品，它们位于三个不同的继承结构中的相同位置上，组成一个产品族。

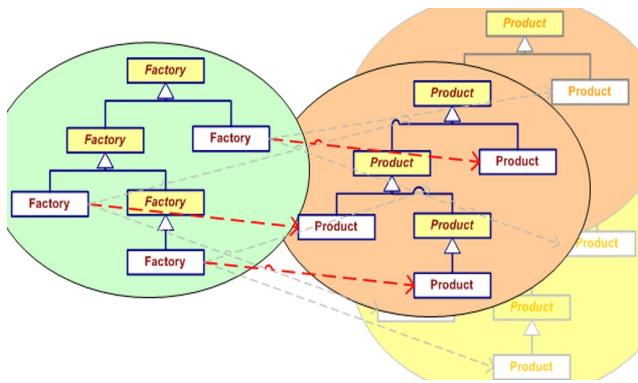


显然，每一个产品族中含有产品的数目，与产品继承结构的数目是相等的。产品的继承结构和产品族将产品按照不同方向划分，形成一个二维的坐标系，如下图所示。

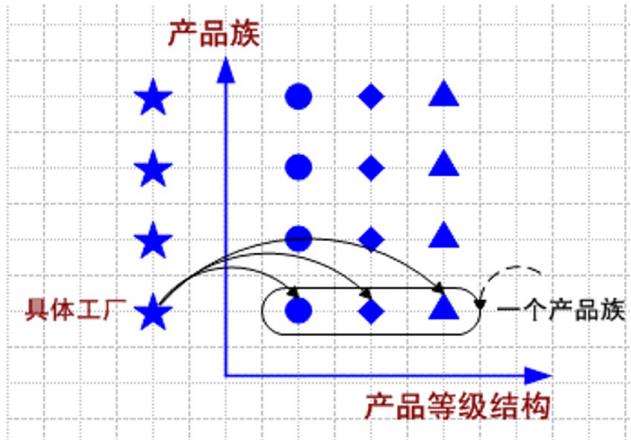


在坐标图中有四个产品族，分布于三个产品继承结构中。在上面的坐标图中，横轴表示产品继承结构，纵轴表示产品族。可以看出，图中一共有四个产品族，分布于三个不同的产品继承结构中。只要指明一个产品所处的产品族以及它所属的继承结构，就可以惟一地确定这个产品。

上面所给出的三个不同的继承结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂继承结构来对付这三个产品继承结构。由于这三个产品继承结构的相似性，会导致三个平行的工厂继承结构。随着产品继承结构的数目增加，工厂方法模式所给出的工厂继承结构的数目也会随之增加。

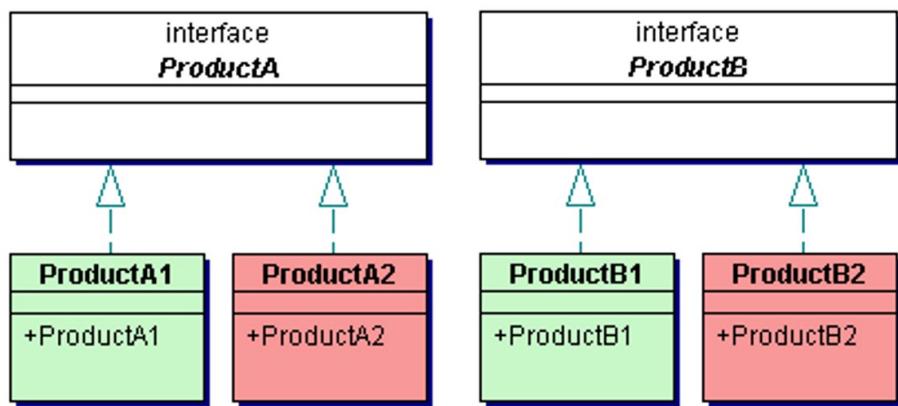


可以看出，一个工厂继承结构可以创建出分属于不同产品继承结构的一个产品族中的所有对象；显然，这时候抽象工厂模式比工厂方法模式更有效率。可以看出，对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族、但是分属于不同继承结构的产品。

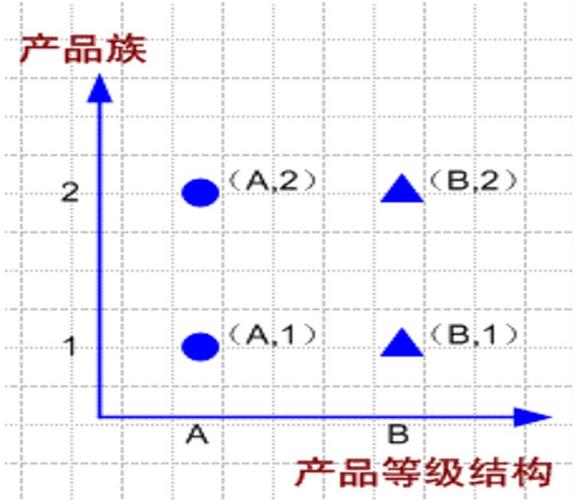


16.1 抽象工厂模式的结构

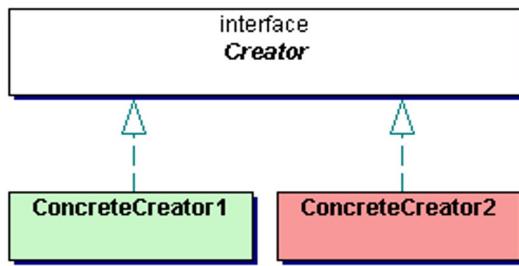
通过使用抽象工厂模式，可以处理具有相同（或者相似）继承结构的多个产品族中的产品对象创建问题。比如下面就是两个具有相同继承结构的产品继承结构 A 和产品继承结构 B 的结构图。



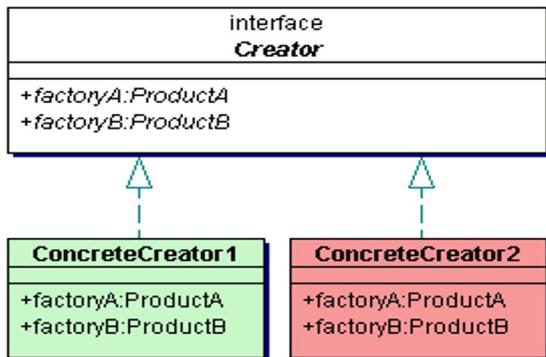
如果使用坐标图描述的话，会看到在图上出现两个继承结构 A 和 B，以及两个产品族 1 和 2。如下图所示。在下面的图中，每一个坐标点都代表一个具体产品角色。



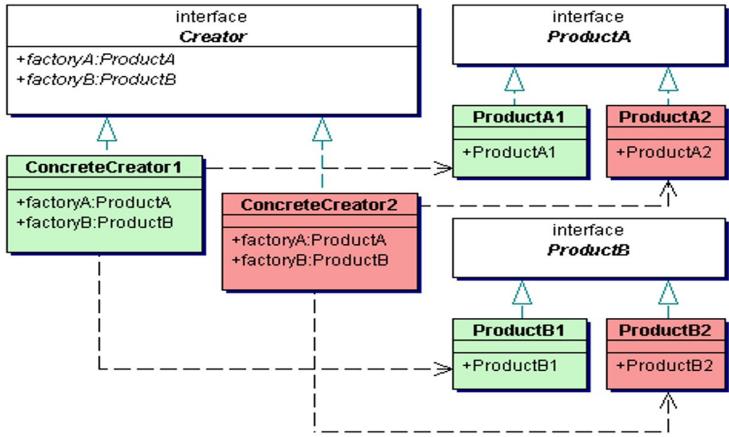
如果使用工厂方法模式处理的话，就必须要有两个独立的工厂族。由于这两个产品族的继承结构相同，因此，使用同一个工厂族也可以处理这两个产品族的创建问题。后者就是抽象工厂模式，这样根据产品角色的结构图，就不难给出工厂角色的结构设计图，如下图所示。



由于每个具体工厂角色都需要负责两个不同继承结构的产品对象的创建，因此每个工厂角色都需要提供两个工厂方法，分别用于创建两个继承结构的产品。既然每个具体工厂角色都需要实现这两个工厂方法，所以这种情况就具有一般性，不妨抽象出来，移动到抽象工厂角色 Creator 中加以声明。产品继承结构 A 和产品继承结构 B 的结构图如下所示。可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品继承结构的产品对象。



采用抽象工厂模式设计出的系统类图如下所示。



从上图可以看出，抽象工厂模式涉及到以下的角色。

- 抽象工厂 (Abstract Factory) 角色：担任这个角色的是抽象工厂模式的核心，它是与应用系统的业务逻辑无关的。通常使用 Java 接口或者抽象 Java 类实现，而所有的具体工厂类必须实现这个 Java 接口或继承这个抽象 Java 类。
- 具体工厂类 (Concrete Factory) 角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的业务逻辑紧密相关的。通常使用具体 Java 类实现这个角色。
- 抽象产品 (Abstract Product) 角色：担任这个角色的类是抽象工厂模式所创建的对象的父类，或它们共同拥有的接口。通常使用 Java 接口或者抽象 Java 类实现这一角色。
- 具体产品 (Concrete Product) 角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定实现了应用系统的业务逻辑。通常使用具体 Java 类实现这个角色。

首先给出工厂角色的源代码，可以看出，抽象工厂角色规定出两个工厂方法，分别提供两个不同继承结构的产品对象。

```

1 public interface Creator {
2     // 产品继承结构 A 的工厂方法
3     public ProductA factoryA();
4     // 产品继承结构 B 的工厂方法
5     public ProductB factoryB();
6 }
```

下面给出具体工厂角色 **ConcreteCreator1** 的源代码。这个具体工厂类实现了抽象工厂角色所要求的两个工厂方法，分别提供两个产品继承结构中的某一个产品对象。

```

1 public class ConcreteCreator1 implements Creator {
2     // 产品继承结构 A 的工厂方法
3     public ProductA factoryA() { return new ProductA1(); }
4     // 产品继承结构 B 的工厂方法
5     public ProductB factoryB() { return new ProductB1(); }
```

一般而言，有多少个产品继承结构，在工厂角色中就有对应个数的个工厂方法。每一个产品继承结构中有多少具体产品，就有多少个产品族，也就会在工厂继承结构中发现多少个具体工厂。

下面给出具体工厂角色 ConcreteCreator2 的源代码。这个具体工厂类实现了抽象工厂角色所要求的两个工厂方法，分别提供两个产品继承结构中的另一个产品对象。

```

1 public class ConcreteCreator2 implements Creator {
2     // 产品继承结构A 的工厂方法
3     public ProductA factoryA() { return new ProductA2(); }
4     // 产品继承结构B 的工厂方法*/
5     public ProductB factoryB() { return new ProductB2(); }
6 }
7
8 public interface ProductA { }
9 public class ProductA1 implements ProductA { }
10 public class ProductA2 implements ProductA { }
11
12 public interface ProductB { }
13 public class ProductB1 implements ProductB { }
14 public class ProductB2 implements ProductB { }

```

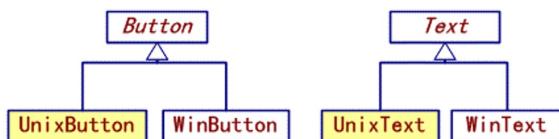
在以下情况下应当考虑使用抽象工厂模式：

1. 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节。这对于所有形态的工厂模式都是重要的；
2. 这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品；
3. 同属于同一个产品族的产品是在一起使用的，这一约束必须要在系统的设计中体现出来；

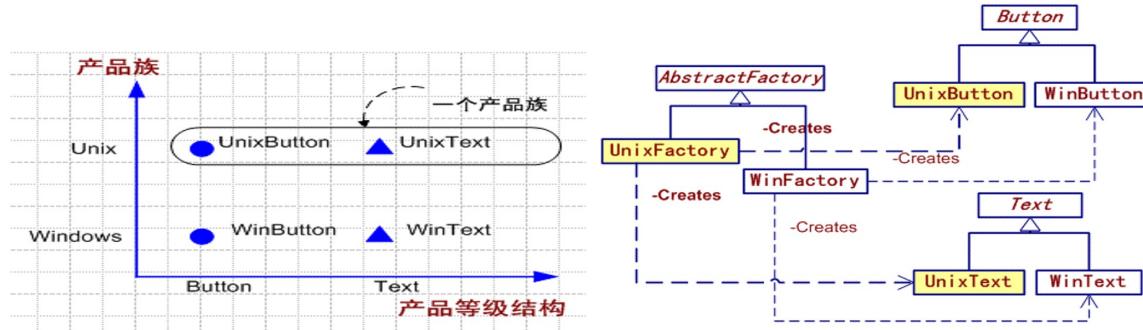
16.2 抽象工厂模式的应用

抽象工厂模式的起源或者说最早的应用，是用于创建分属于不同操作系统的视窗构件。比如，命令按键（Button）与文字框（Text）都是视窗构件，在 UNIX 操作系统的视窗环境和 Windows 操作系统的视窗环境中，这两个构件有不同的本地实现，它们的细节也有所不同。

在每一个操作系统中，都有一个视窗构件组成的构件家族。在这里就是 Button 和 Text 组成的产品族。而每一个视窗构件都构成自己的继承结构，由一个抽象角色给出抽象的功能描述，而由具体子类给出不同操作系统下的具体实现，如下图所示。



可以发现在上面的产品类图中，有两个产品的继承结构，分别是 Button 继承结构和 Text 继承结构。同时有两个产品族，也就是 UNIX 产品族和 Windows 产品族。UNIX 产品族由 UnixButton 和 UnixText 产品构成；而 Windows 产品族由 WinButton 和 WinText 产品构成。



系统对产品对象的创建需求由一个工厂的继承结构满足；其中有两个具体工厂角色，即 UnixFactory 和 WinFactory。其中 UnixFactory 对象负责创建 Unix 产品族中的产品，而 WinFactory 对象负责创建 Windows 产品族中的产品。这就是抽象工厂模式的应用，抽象工厂模式的解决方案如图所示。

```

1 abstract class Button {
2     public abstract void paint();
3 }
4
5 class WinButton extends Button {
6     public void paint() {
7         System.out.println("I'm a WinButton");
8     }
9 }
10
11 class OSXButton extends Button {
12     public void paint() {
13         System.out.println("I'm an OSXButton");
14     }
15 }
16
17 abstract class GUIFactory {
18     public static GUIFactory getFactory() {
19         //从配置文件中读取操作系统类型
20         int sys = readFromConfigFile("OS_TYPE");
21         if (sys == 0) {
22             return new WinFactory();
23         } else {

```

```

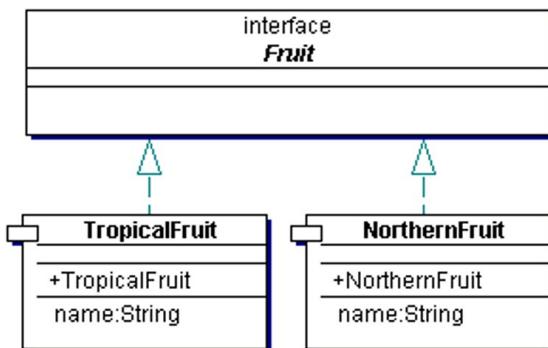
24     return new OSXFactory();
25   }
26 }
27 public abstract Button createButton();
28 }

29
30 //生成 Windows 界面的工厂
31 class WinFactory extends GUIFactory {
32   public Button createButton() { return new WinButton(); }
33 }
34
35 //生成 OS X 界面的工厂
36 class OSXFactory extends GUIFactory {
37   public Button createButton() { return new OSXButton(); }
38 }
39
40 public class Application {
41   public static void main(String[] args) {
42     GUIFactory factory = GUIFactory.getFactory();
43     Button button = factory.createButton();
44     button.paint();
45   }
46 }

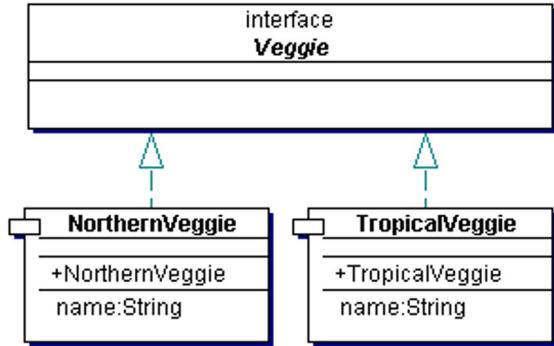
```

16.3 抽象工厂模式在农场系统中的实现

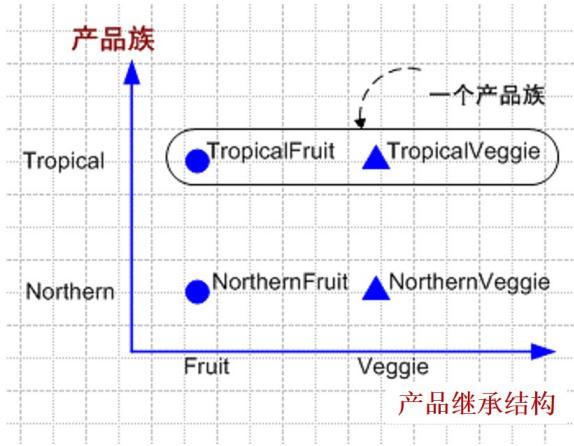
农场公司再次面临新的大发展，一项重要的工作，就是引进塑料大棚技术，在大棚里种植热带（Tropical）和北方的水果和蔬菜。因此，在这个系统里面，产品分成两个继承结构：水果（Fruit）和蔬菜（Veggie）。下面就是水果（Fruit）的类图。



下面则是蔬菜（Veggie）的类图。

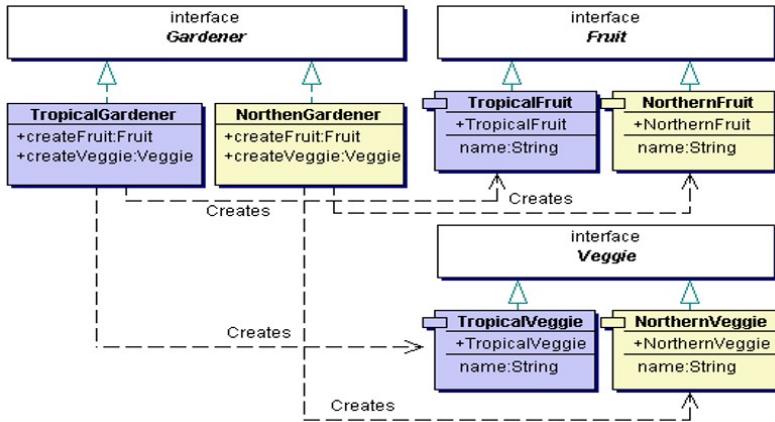


下面是描述这个系统的产品角色的相图。



可以看出,这个系统的产品可以分成两个继承结构:Fruit 和 Veggie,以及两个产品族:Tropical 和 Northern。坐标图上出现了四个坐标点,分别代表 TropicalFruit (热带水果)、TropicalVeggie (热带蔬菜)、NorthernFruit (北方水果) 以及 NorthernVeggie (北方蔬菜) 等四个产品。显然可以使用一个工厂族来封装它们的创建过程。这个工厂族的继承结构应当与产品族的继承结构完全平行,园丁继承结构的类图如下图所示。系统所需要的是产品的实例,而工厂则是对产品创建过程的封装。

系统设计: 与抽象工厂模式的各个角色相对照,不难发现,所谓各个园丁其实就是各个工厂角色,而蔬菜和水果角色则是产品角色。将抽象工厂模式应用于农场系统中,系统的设计图如下所示。



种在田间的北方作物与种在大棚的热带作物都是系统的产品，它们分属于两个产品族。显然，北方作物是要种植在一起的，而大棚作物是要另外种植在一起的。这些分别体现在系统的设计上，就正好满足了使用抽象工厂模式的第三个条件。

```

1 // 抽象工厂类
2 public interface Gardener {
3     public Fruit creatFruit();
4     public Veggie createVeggie();
5 }
6
7 // 具体工厂类
8 public class NorthernGardener implements Gardener {
9     // 水果的工厂方法
10    public Fruit createFruit(String name) {
11        return new NorthernFruit(name);
12    }
13    // 蔬菜的工厂方法
14    public Veggie createVeggie(String name) {
15        return new NorthernVeggie(name);
16    }
17 }
18
19 // 具体工厂类
20 public class TropicalGardener implements Gardener {
21     // 水果的工厂方法
22     public Fruit createFruit(String name) {
23         return new TropicalFruit(name);
24     }
25     // 蔬菜的工厂方法
  
```

```
26     public Veggie createVeggie(String name) {
27         return new TropicalVeggie(name);
28     }
29 }
30
31 public interface Veggie {}
32
33 // 具体产品类
34 public class NorthernVeggie implements Veggie{
35     private String name;
36     public NorthernVeggie(String name) { }
37     public String getName() { return name; }
38     public void setName(String name) { this.name = name; }
39 }
40
41 // 具体产品类
42 public class TropicalVeggie implements Veggie {
43     private String name;
44     public TropicalVeggie(String name) { this.name = name; }
45     public String getName() { return name; }
46     public void setName(String name) { this.name = name; }
47 }
48
49 public interface Fruit { }
50
51 public class NorthernFruit implements Fruit {
52     private String name;
53     public NorthernFruit(String name) { }
54     public String getName() { return name; }
55     public void setName(String name) { this.name = name; }
56 }
57
58 public class TropicalFruit implements Fruit {
59     private String name;
60     public TropicalFruit(String name) { }
61     public String getName() { return name; }
62     public void setName(String name) { this.name = name; }
63 }
64 // 在使用时，客户端只需要创建具体工厂的实例，
```

65 // 然后调用工厂对象的工厂方法，就可以得到所需要的产品对象。

16.4 开闭原则

开闭原则要求一个软件系统可以在不修改原有代码的情况下，通过扩展达到增强其功能的目的。对于一个涉及到多个产品继承结构和多个产品族的系统，其功能的增强不外乎两个方面：

- 增加新的产品族；
- 增加新的产品继承结构。

那么抽象工厂模式是怎样支持这两方面功能增强的呢？

- 增加新的产品族：只需要向系统中加入新的具体工厂类就可以了，没有必要修改已有的工厂角色或者产品角色。因此，在系统中的产品族增加时，抽象工厂模式是支持开闭原则的。
- 增加新的产品继承结构：需要修改所有的工厂角色，给每一个工厂类都增加一个新的工厂方法，而这显然是违背开闭原则的。换言之，对于产品继承结构的增加，抽象工厂模式是不支持开闭原则的。

综合起来，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，而不能为新的产品继承结构的增加提供这样的方便。

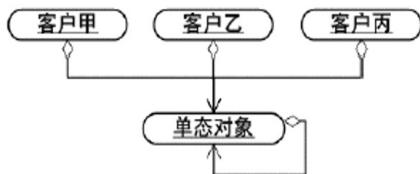
17 单例模式 (Singleton)

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为**单例类**。

单例模式的要点：

- 某个类只能有一个实例；
- 它必须自行创建这个实例；
- 必须自行向外提供这个实例。

在下面的对象图中，有一个“单例对象”，而“客户甲”、“客户乙”和“客户丙”是单例对象的三个客户对象。可以看到，所有的客户对象共享一个单例对象。而且从单例对象到自身的连接线可以看出，单例对象持有对自己的引用。

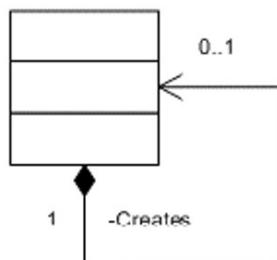


17.1 单例模式应用场景-资源管理

一些资源管理器常常设计成单例模式。在计算机系统中，需要管理的资源包括软件外部资源，譬如每台计算机可以有若干个打印机，但只能有一个 Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

需要管理的资源包括软件内部资源，譬如，大多数的软件都有一个（甚至多个）属性（properties）文件存放系统配置。这样的系统应当由一个对象来管理一个属性文件。需要管理的软件内部资源，比如负责记录网站来访人数的部件，记录软件系统内部事件、出错信息的部件等。这些部件都必须集中管理，不可政出多头。

单例模式的结构: 虽然单例模式中的单例类被限定只能有一个实例，但是单例模式和单例类可以很容易被推广到任意且有限多个实例的情况，这时候称它为多例模式（Multiton Pattern）和多例类（Multiton Class）。单例类的简略类图如下所示。



饿汉式单例类:

```
1 public class EagerSingleton {
2     private static final EagerSingleton m_instance = new EagerSingleton();
3     // 私有的默认构造函数
4     private EagerSingleton() { }
5     // 静态工厂方法
6     public static EagerSingleton getInstance() { return m_instance; }
7 }
```

懒汉式单例类: 与饿汉式单例类相同之处是，类的构造函数是私有的。与饿汉式单例类不同的是，懒汉式单例类在第一次被引用时将自己实例化。

```
1 public class LazySingleton {
2     private static LazySingleton m_instance = null;
3     // 私有的默认构造函数，保证外界无法直接实例化
4     private LazySingleton() { }
5     // 静态工厂方法，返还此类的唯一实例
6     synchronized public static LazySingleton getInstance() {
7         if (m_instance == null) {
```

```
8     m_instance = new LazySingleton();
9 }
10 return m_instance;
11 }
12 }
```

使用单例模式有一个很重要的必要条件：要求一个类只有一个实例时，才应当使用单例模式。反过来说，如果一个类可以有几个实例共存，那么就没有必要使用单例类。

17.2 单例类的状态

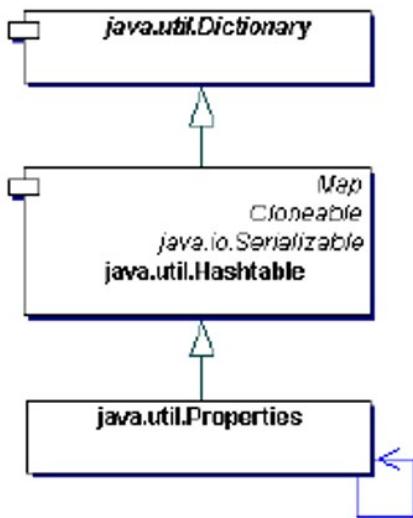
- 有状态的单例类：一个单例类可以是有状态的（stateful），一个有状态的单例对象一般也是可变（mutable）单例对象。有状态的可变的单例对象常常当做状态库（repository）使用。比如一个单例对象可以持有一个 int 类型的属性，用来给一个系统提供一个数值唯一的序列号码，作为某个销售系统的账单号码。当然，一个单例类可以持有一个聚集，从而允许存储多个状态。
- 没有状态的单例类：另一方面，单例类也可以是没有状态的（stateless），仅用做提供工具性函数的对象。既然是为了提供工具性函数，也就没有必要创建多个实例，因此使用单例模式很合适。一个没有状态的单例类也就是不变（Immutable）单例类。

17.3 例子：属性管理器

这里给出一个读取属性（properties）文件的单例类，作为单例模式的一个实用的例子。属性文件如同老式的 windows 编程时的.ini 文件，用于存放系统的配置信息。配置信息在属性文件中以属性的方式存放，一个属性就是两个字符串组成的对子，其中一个字符串是键（key），另一个字符串是这个键的值（value）。大多数的系统都有一些配置常量，这些常量如果是存储在程序内部的，那么每一次修改这些常量都需要重新编译程序。将这些常量放在配置文件中，系统通过访问这个配置文件取得配置常量，就可以通过修改配置文件而无需修改程序而达到更改系统配置的目的。系统也可以在配置文件中存储一些工作环境信息，这样在系统重启时，这些工作信息可以延续到下一个运行周期中。假定需要读取的属性文件就在当前目录中，且文件名为 singleton.properties。这个文件中有如下的一些属性项。

```
1 // 属性文件内容
2 node1.item1=How
3 node1.item2=are
4 node2.item1=you
5 node2.item2=doing
6 node3.item1=?
```

Java 提供了一个工具类，称做属性类，可以用来完成属性文件的操作。这个属性类的继承关系可以从下面的类图中看清楚。



属性类提供了读取属性和设置属性的各种方法。其中读取属性的方法有：

- `contains(Object value)`、`containsKey(Object key)`: 如果给定的参数或属性关键字在属性表中有定义，该方法返回 True，否则返回 False。
- `getProperty(String key)`、`getProperty(String key, String default)`: 根据给定的属性关键字获取关键字值。
- `list(PrintStream s)`、`list(PrintWriter w)`: 在输出流中输出属性表内容。
- `size()`: 返回当前属性表中定义的属性关键字个数。

设置属性的方法有：

- `put(Object key, Object value)`: 向属性表中追加属性关键字和关键字的值
- `remove(Object key)`: 从属性表中删除关键字。

从属性文件加载属性数据的方法为 `load(InputStream inStream)`，可以从一个输入流中读入一个属性列，如果这个流是来自一个文件的话，这个方法就从文件中读入属性。将属性存入属性文件的方法有几个，重要的一个时 `store(OutputStream out, String header)`，将当前的属性列写入一个输出流，如果这个输出流是导向一个文件的，那么这个方法就将属性流存入文件。

为什么使用单例模式: 属性是系统的一种“资源”，应当避免有多于一个的对象读写属性。此外，属性的读取可能会在很多地方发生。换言之，属性管理器应当自己创建自己的实例，并且自己向系统全程提供这一实例。因此，属性文件管理器应当是一个单例模式负责。

系统设计: 系统的核心是一个属性管理器，也就是一个叫做 `ConfigManager` 的类，这个类应当是一个单例类。因此，这个类应当有一个静态工厂方法，不妨叫 `getInstance()`，用于提供自己的实例。为简单起见，在这里采取“饿汉”方式实现 `ConfigManager`。

```

1 import java.util.Properties;
2 import java.io.FileInputStream;
3 import java.io.File;
  
```

```
4 public class ConfigManager {  
5     // 属性文件全名  
6     private static final String PFILE = System.getProperty("user.dir")  
7         + File.separator + "Singleton.properties";  
8     // 对应于属性文件的文件对象变量  
9     private File m_file = null;  
10    // 属性文件的最后修改日期  
11    private long m_lastModifiedTime = 0;  
12    // 属性文件所对应的属性对象变量  
13    private Properties m_props = null;  
14    // 本类惟一的一个实例  
15    private static ConfigManager m_instance = new ConfigManager();  
16    // 私有的构造函数，用以保证外界无法直接实例化  
17    private ConfigManager() {  
18        m_file = new File(PFILE);  
19        m_lastModifiedTime = m_file.lastModified();  
20        if(m_lastModifiedTime == 0) {  
21            System.err.println(PFILE + " file does not exist!");  
22            System.exit(1);  
23        }  
24        m_props = new Properties();  
25        try {  
26            m_props.load(new FileInputStream(PFILE));  
27        } catch(Exception e) {  
28            e.printStackTrace();  
29        }  
30    }  
31    /**  
32     * 静态工厂方法  
33     * @return 返回 ConfigManager 类的单一实例  
34     */  
35    synchronized public static ConfigManager getInstance() {  
36        return m_instance;  
37    }  
38    /**  
39     * 读取一特定的属性项  
40     *  
41     * @param name 属性项的项名  
42     * @param defaultVal 属性项的默认值
```

```

43 * @return 属性项的值 (如此项存在) , 默认值 (如此项不存在)
44 */
45 final public Object getConfigItem(String name, Object defaultValue) {
46     long newTime = m_file.lastModified();
47     // 检查属性文件是否被其他程序
48     // (多数情况是程序员手动) 修改过
49     // 如果是, 重新读取此文件
50     if(newTime == 0) {
51         // 属性文件不存在
52         if(m_lastModifiedTime == 0) {
53             System.err.println(PFILE+ " file does not exist!");
54         } else {
55             System.err.println(PFILE+ " file was deleted!!!");
56         }
57         return defaultValue;
58     } else if(newTime > m_lastModifiedTime) {
59         // Get rid of the old properties
60         m_props.clear();
61         try {
62             m_props.load(new FileInputStream(PFILE));
63         } catch(Exception e) {
64             e.printStackTrace();
65         }
66     }
67     m_lastModifiedTime = newTime;
68     Object val = m_props.getProperty(name);
69     if (val == null) {
70         return defaultValue;
71     } else {
72         return val;
73     }
74 }
75 }
```

下面的源代码演示了怎样调用 ConfigManager 来读取属性文件。

```

1 BufferedReader reader = new BufferedReader(
2     new InputStreamReader(System.in));
3 System.out.println("Type quit to quit");
4 do {
5     System.out.print("Property item to read:");
6 }
```

```

6 String line = reader.readLine();
7 if(line.equals("quit")) { break; }
8 System.out.println(ConfigManager.getInstance()
9     .getConfigItem(line, "Not found."));
10 } while(true);

```

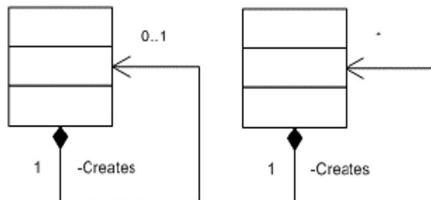
相关模式:有一些模式可以使用单例模式，如抽象工厂模式可以使用单例模式，将具体工厂类设计成单例类。

```

1 DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();
2 DocumentBuilder builder=factory.newDocumentBuilder();
3 Document document=builder.parse(new File("Dom_3.xml"));

```

单例模式的精神可以推广到多于一个实例的情况。这时候这种类叫做多例类，这种模式叫做多例模式。单例类（左）和多例类（右）的类图如下所示。



17.4 语言中的单例模式

Java 的 Runtime 对象:在 Java 内部,java.lang.Runtime 对象就是一个使用单例模式的例子。在每一个 Java 应用程序里面，都有惟一的一个 Runtime 对象。通过这个 Runtime 对象，应用程序可以与其运行环境发生相互作用。Runtime 类提供一个静态工厂方法 getRuntime():
`public static Runtime getRuntime();` 通过调用此方法，可以获得 Runtime 类惟一的一个实例：
`Runtime rt = Runtime.getRuntime();`

Runtime 对象通常的用途包括:执行外部命令；返回现有内存即全部内存；运行垃圾收集器；加载动态库等。

下面的例子演示了怎样使用 Runtime 对象运行一个外部程序。

```

1 import java.io.*;
2 public class CmdTest {
3     public static void main(String[] args)
4         throws IOException {
5         Process proc = Runtime.getRuntime().exec("notepad.exe");
6     }
7 }

```

18 原型模式 (Prototype)

原型模式的目的: 通过给出一个原型对象来指明所要创建的对象类型，然后用复制这个原型对象的办法创建出更多的同类型对象。

浅复制: 被复制对象的所有变量都含有与原对象相同的值，而所有的对其它对象的引用都仍然指向原来的对象。浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

深复制: 被复制对象的所有变量都含有与原对象相同的值，除去那些引用其它对象的变量。那些引用其它对象的变量将指向被复制过的新对象。深复制把要复制的对象所引用的对象都复制了一遍，而这种对被引用到的对象的复制叫做间接复制。

原型模式的优点:

- 优点: 简化了对象的创建过程。在有大对象或多个对象的复制时，可提高系统性能。
- 缺点: 每一个类必须额外增加一个克隆方法。深复制的实现较为复杂。

当把一个类实例化时，此类的数据成员都被复制到属于此数据类型的一个新的实例中去。如：

Panda myPanda=new Panda(); 上面的语句做了如下的事情：

1. 创建了一个 Panda 类型的变量，名称为 myPanda
2. 建立了一个 Panda 类型的对象
3. 使变量 myPanda 指到这个新的对象

对象的创建与它们的引用是独立的。

最后一行把 myPanda 的引用赋值给 thatPanda，使得 myPanda 和 thatPanda 同时指向同一个 Panda 对象。

```
1 Panda myPanda, thatPanda;  
2 myPanda = new Panda();  
3 thatPanda = myPanda;
```

创建了两个 Panda 类的对象。第一个对象被创建出来时，立即被引用，第二个对象被创建出来时，也立即被引用，而同时对第一个对象的引用就不存在了。在以后的代码中，第一个对象也不可能再被引用了。Java 的垃圾收集器会在某个时候把它收集走。

```
1 Panda myPanda, thatPanda;  
2 myPanda = new Panda();  
3 myPanda = new Panda();
```

Java 对象的复制:

- Java 的所有类都是从 java.lang.Object 类继承而来的，而 Object 类提供下面的方法对对象进行复制:`protected Object clone()`
- 子类可以对这个方法重新实现，提供满足自己需要的复制方法。对象通常都有对其它的对象的引用。当使用 `clone()` 方法复制一个对象时，此对象对其它对象的引用也同时会被复制一份。
- Java 提供的 Cloneable 接口的作用是在运行时通知 java 虚拟机可以安全地在这个类上使用 `clone()` 方法。通过调用 `clone()` 方法可以得到一个对象的复制。

- 由于 Object 类本身并不实现 Cloneable 接口，因此如果类没有实现 Cloneable 接口而调用 clone() 方法会抛出 CloneNotSupportedException 异常。

PandaToClone 类的 clone() 方法提供复制自己实例的任务，源代码如下：

```

1 class PandaToClone implements Cloneable {
2     private int height,weight,age;
3     public PandaToClone(int height,weight) {
4         this.age=0;
5         this.weight=weight;
6         this.height=height;
7     }
8     public void setAge(int age) { this.age = age; }
9     public int getAge() { return age; }
10    public int getHeight() { return height; }
11    public int getWeight() { return weight; }
12    public Object clone() {
13        PandaToClone temp=new PandaToClone(height, weight);
14        temp.setAge(age);
15        //注意返还的值的类型必需是Object
16        return (Object) temp;
17    }
18 }
```

客户端的代码如下：

```

1 public class Client {
2     private PandaToClone thisPanda, thatPanda;
3     public static void main(String[] args) {
4         thisPanda=new PandaToClone(15, 25);
5         thisPanda.setAge(3);
6         // 通过第一个对象的clone()方法创建第二个对象
7         thatPanda = (PandaToClone) thisPanda.clone();
8         System.out.println("Age of this panda:" + thisPanda.getAge());
9         System.out.println(" height:" + thisPanda.getHeight());
10        System.out.println(" weight:" + thisPanda.getWeight());
11        System.out.println("Age of that panda:" + thatPanda.getAge());
12        System.out.println("    height:" + thatPanda.getHeight());
13        System.out.println("    weight:" + thatPanda.getWeight());
14    }
15 }
```

从系统的运行结果看，克隆对象与原对象的状态是完全一样的。

克隆满足的条件:

- 对任何的对象 `x`, 都有 `x.clone() != x`。克隆对象与原对象不是同一个对象。
- 对任何的对象 `x`, 都有 `x.clone().getClass() == x.getClass()` 克隆对象与原对象的类型一样。
- 如果对象 `x` 的 `equals()` 方法定义恰当的话, `x.clone().equals(x)` 是成立的。

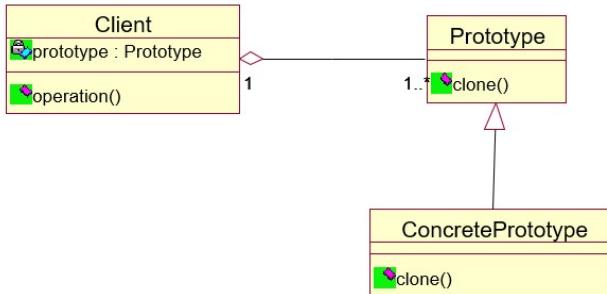
Java 的 API 中, 凡是提供了 `clone()` 的类, 都满足以上三个条件。一般来说, 前两个条件是必需的, 第三个是可选的。

equals() 方法的讨论: 通过继承 `java.lang.Object` 对象的 `equals()` 方法是不够的。例如: 以下是 `java.lang.Object` 对象的 `equals()` 方法的源代码

```
1 public boolean equals(Object obj) { return (this == obj); }
```

也就是说, 当两个变量指向同一个对象时, `equals()` 方法才会返还 `true`。显然, 克隆的对象不相等。假设被克隆的对象按照它们的内部状态是否可变, 划分成可变对象和不变对象, 可变对象和不变对象所提供的 `equals()` 工作方式应当是不同的。不变对象只有当它们是同一个对象时, `equals()` 才会返回 `true`, 可以从 `java.lang.Object` 继承这个方法。可变对象必需含有相同的状态才能返回 `true`, 因此可变对象必需自行实现 `equals()` 方法。

18.1 原型模式的结构



以上是第一种形式的原始原型模式, 这种形式涉及到三个角色:

- 客户端 (Client) 角色: 客户类提出创建对象的请求。
- 抽象原型 (Prototype) 角色: 这是一个抽象角色, 通常由一个 java 接口或抽象类实现。此角色给出所有的具体原型类所需的接口。
- 具体原型 (Concrete Prototype) 角色: 被复制的对象。此角色需要实现抽象原型角色所要求的接口。

下面的程序给出了一个示意性的实现, 下面是源代码:

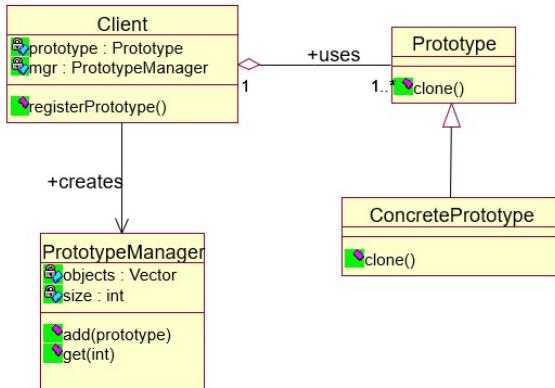
```
1 public class Client {
2     private Prototype prototype;
3     public void operation(Prototype example) {
```

```

4     Prototype p = (Prototype) example.clone();
5 }
6 }
7 // 抽象原型角色声明了一个clone()方法
8 public interface Prototype extends Cloneable {
9     Object clone();
10 }
11 // 具体原型角色实现clone()方法
12 public class ConcretePrototype implements Prototype {
13     /* 克隆方法 */
14     public Object clone() {
15         try {
16             return new ConcretePrototype();
17         } catch(CloneNotSupportedException e) {
18             return null;
19         }
20     }
21 }

```

登记式的原型模式:



该模式有如下的角色：

- 客户端（Client）角色：客户端类向原型管理器提出存储或提取对象的请求。
- 抽象原型（Prototype）角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体原型类所需的接口。
- 具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象的原型角色所要求的接口。
- 原型管理器（Prototype Manager）角色：管理记录每一个被创建的对象。

下面给出一个示意性代码。首先抽象类型角色声明了一个方法，即 clone() 方法

```

1 public interface Prototype extends Cloneable {
2     public Object clone();
3 }
4
5 public class ConcretePrototype implements Prototype {
6     string gender;
7     public synchronized Object clone() {
8         Prototype temp=null;
9         temp=new ConcretePrototype ();
10        return temp;
11    }
12 }

```

原型管理器角色保持一个聚集，作为对所有原型对象的登记，这个角色提供必要的方法，供外界增加新的原型对象和取得已经登记过的对象。其源代码如下：

```

1 import java.util.Map;
2 public class PrototypeManager {
3     /* 用来记录原型的编号和原型实例的对应关系 */
4     private static Map<String,Prototype> map = new
5         HashMap<String, Prototype>();
6     /* 私有化构造方法，避免外部创建实例 */
7     private PrototypeManager() { }
8     /**
9      * 向原型管理器里面添加或是修改某个原型注册
10     * @param prototypeId 原型编号
11     * @param prototype 原型实例
12     */
13     public synchronized static void setPrototype(String prototypeId,
14         Prototype prototype){
15         map.put(prototypeId, prototype);
16     }
17     /**
18      * 获取某个原型编号对应的原型实例
19      * @param prototypeId 原型编号
20      * @return 原型编号对应的原型实例
21      * @throws Exception 如果原型编号对应的实例不存在，
22      * 则抛出异常
23      */
24     public synchronized static Prototype getPrototype(String prototypeId)

```

```

25     throws Exception{
26     Prototype prototype = map.get(prototypeId);
27     if(prototype == null){
28         throw new Exception("您希望获取的原型还没有注册或已被销毁");
29     }
30     return prototype;
31 }
32 }
```

客户端角色 Client 的源代码如下:

```

1 public class Client {
2     try{
3         Prototype p1 = new ConcretePrototype();
4         // 获取原型来创建对象
5         PrototypeManager.setPrototype("p1", p1);
6         Prototype p2 = PrototypeManager.getPrototype("p1").clone();
7     } catch (Exception e) { }
8 }
```

19 适配器模式 (Adapter)

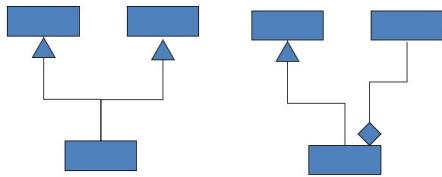
结构模式描述如何将类或者对象组织在一起形成特定的结构，以实现更为复杂、强大的功能。结构模式可以分为：

- 类的结构模式：使用继承来把类、接口等组合起来，以形成更大的结构。类的结构模式是静态的。典型例子是类形式的适配器模式。
- 对象的结构模式：描述怎样把各种不同类型的对象组合在一起，以实现新的功能的方法。对象的结构模式是动态的。典型的结构模式有代理模式，其它包括合成模式、享元模式、装饰模式、对象形式的适配器模式。

适配器模式 (Adapter Pattern) 把一个类的接口变成客户端所需要的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

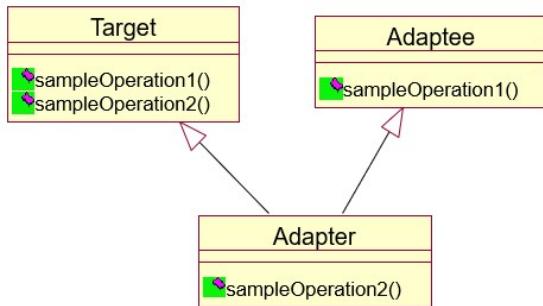
名称的由来：这很像变压器，变压器把一种电压转换成另一种电压。把美国的电器拿回中国大陆来用的时候，用户就面临电压不同的问题。美国的生活用电电压是 110v，而中国的电压是 220v。如果要在中国使用美国的电器，就必须有一个能把 220v 电压转换成 110v 电压的变压器。而这就像是本模式所做的事，因此此模式也常常被称为变压器模式。

适配器模式的两种形式：适配器模式有类的适配器模式和对象的适配器模式两种不同的形式。如下图所示，左边是类的适配器模式，右边是对象的适配器模式。



19.1 类的适配器模式的结构

类的适配器模式把被适配的类的 API 转换成为目标类的 API，其静态结构图如下图所示。



Adaptee 类没有提供 sampleOperation2() 方法，而客户端则需要使用这个方法。为使客户端能够使用 Adaptee 类，提供一个中间环节，即类 Adapter，把 Adaptee 的 API 与 Target 类的 API 衔接起来。

模式所涉及的角色有：

- 目标 (Target) 角色：这就是所期待得到的接口。
- 源 (Adaptee) 角色：现有需要适配的接口。
- 适配器 (Adapter) 角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

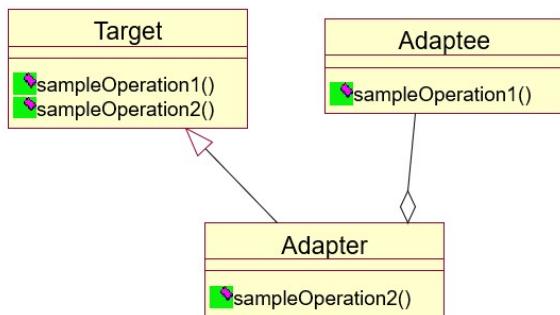
```

1 public interface Target {
2     void sampleOperation1(); //源类的方法
3     void sampleOperation2(); //源类没有的方法
4 }
5
6 public class Adaptee {
7     public void sampleOperation1() { }
8 }
9
10 public class Adapter extends Adaptee implements Target {
11     /* 由于源类没有方法sampleOperation2,因此适配器类补上这个方法 */
12     public void sampleOperation2() {
13         // Write your code here
14     }
15 }
```

```
14 }
15 }
```

19.2 对象的适配器模式的结构

与类的适配器模式一样，对象的适配器模式把被适配的类的 API 转换成为目标类的 API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到 adaptee 类，而是使用委派关系连接到 Adaptee 类。对象的适配器模式的静态结构如下图所示。



```
1 public interface Target {
2     void sampleOperation1(); // 源类有的方法
3     void sampleOperation2(); // 源类没有的方法
4 }
5
6 public class Adaptee {
7     public void sampleOperation1() { }
8 }
9
10 public class Adapter implements Target {
11     private Adaptee adaptee;
12     public Adapter(Adaptee adaptee) {
13         super();
14         this.adaptee = adaptee;
15     }
16     public void sampleOperation1() {
17         adaptee.sampleOperation1();
18     }
19     public void sampleOperation2() {
20         // Write your code here
21     }
22 }
```

适配器模式的目的是将接口不同而功能相同或相近的接口加以转换，这里面包括适配器角色补充了一个源角色没有的方法。

对象型的适配器模式的效果：

- 一个适配器可以把多种不同的源适配到同一个目标。即同一个适配器可以把源类和它的子类都适配到目标接口。
- 与类的适配器模式相比，要想替换源类的方法就不容易。如果一定要置换源类的一个或多个方法，就要先做一个源类的子类，将源类的方法替换，然后把源类的子类当作真正的源来适配。
- 虽然要想替换源类的方法不容易，但要想增加新的方法则很方便，而新增加的方法可适用于所有的源。

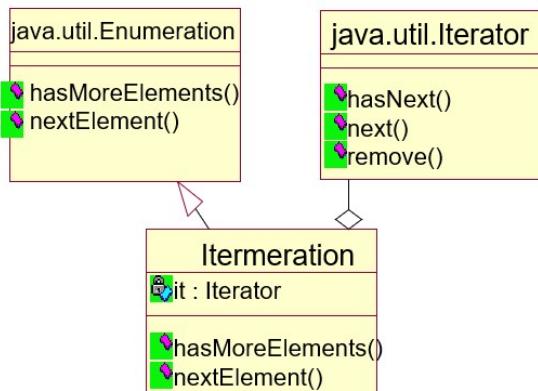
什么情况下使用适配器模式：

- 系统需要使用现有的类，而此类的接口不符合系统的需要。
- 想要建立一个可以重复使用的类。用于一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有很复杂的接口。

19.3 Iterator 和 Enumeration

在 JDK 1.0 和 1.1 版本里没有 java 聚集 (collection) 的框架，这一框架是在 JDK 1.2 版本中给出的。与此相对应，JDK 1.0 和 1.1 版本提供了 enumeration 接口，而 JDK 1.2 版本给出了 iterator 接口。如果有许多的 Java 代码是为老版本 Java 编译器写的，使用的是 Enumeration，现在想使用新版本编译器和新的 Java 聚集库包的话，需要将已有代码的 Iterator 接口换成 Enumeration 接口。因为 Java 聚集要求 Iterator 接口，这样才能使已有的代码可以使用新版本的聚集对象。

从 Iterator 到 Enumeration 的适配：适配器 Itermeration 使用了对象的适配器模式，将一个 Iterator 对象封装在一个 Enumeration 类的具体类里。如下图所示：



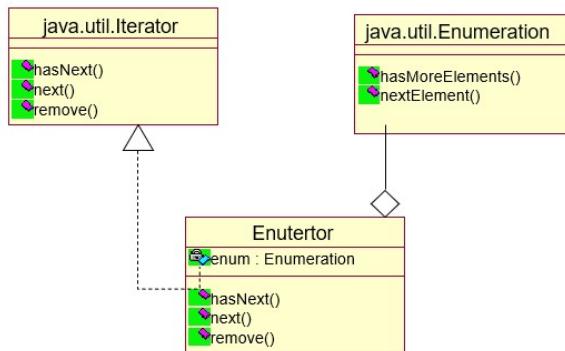
```
1 import java.util.Enumeration;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
```

```

4 public class Itermeration implements Enumeration {
5     private Iterator it;
6     public Itermeration(Iterator it) {
7         this.it = it;
8     }
9     public boolean hasMoreElements() {
10        return it.hasNext();
11    }
12    public Object nextElement() throws NoSuchElementException {
13        return it.next();
14    }
15 }

```

从 Enumeration 到 Iterator 的适配: Enuterator 使用了对象的适配器模式将 Enumeration 接口适配到 Iterator 接口。如下图所示:



```

1 import java.util.Enumeration;
2 import java.util.Iterator;
3 import java.util.NoSuchElementException;
4 public class Enuterator implements Iterator {
5     private Enumeration enum;
6     public Enuterator(Enumeration enum) { this.enum = enum; }
7     public boolean hasNext() { return enum.hasMoreElements(); }
8     public Object next() throws NoSuchElementException {
9         return enum.nextElement();
10    }
11    public void remove() {
12        throw new UnsupportedOperationException();
13    }
14 }

```

19.4 适配器模式在架构层次上的应用

- WINE 是一个开源的免费软件，允许在 linux 环境里运行 windows 程序。WINE 提供了从 Linux 到 Window 图形界面的适配器。
- MKS Toolkit 软件是一个为 Windows 系统设计的商业软件，它提供了 C shell、Korn shell、Perl 以及多种 UNIX 命令的解释器。在安装了此软件之后，Windows 的用户可以在自己的系统上使用 Unix 的 C Shell 和 Korn shell 指令集合，以及如 ls、vi、grep 等的 unix 命令。也就是说 MKS Toolkit 提供了 UNIX 命令集从 UNIX 到 windows 的适配，是一种命令集层次上的适配器模式。
- JDBC 驱动软件与适配器模式：JDBC 给出一个客户端的通用界面。每一个数据库引擎的 JDBC 驱动软件都是一个介于 JDBC 接口和数据库引擎之间的适配器软件。
- 抽象的 JDBC 接口和各个数据库引擎的 API 之间都需要相应的适配器软件，即为各个数据库引擎准备的驱动软件。

20 合成模式 (Composite)

合成 (Composite) 模型模式属于对象的结构模式，有时又叫做部分-整体 (Part-Whole) 模式。合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将整体对象与部分对象同等看待。

在下面的情况下应当考虑使用合成模式：

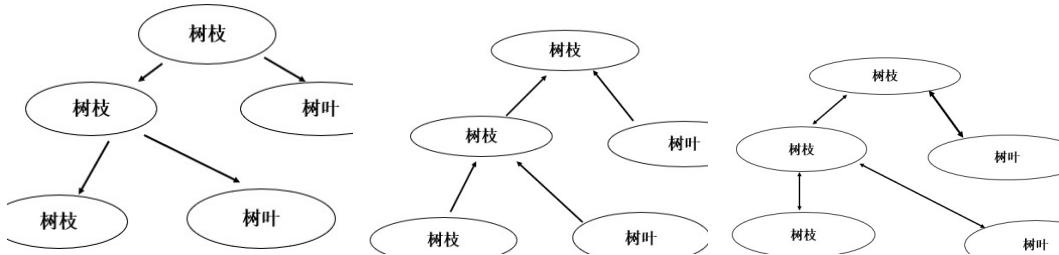
- 需要描述对象的部分和整体的等级结构；
- 需要客户端忽略掉整体对象和部分对象的区别。

合成模式的典型应用-文件系统：一个文件系统就是一个典型的合成模式系统。文件系统是一个树结构。树有节点，节点有两种，一种是树枝节点，即目录，有子树结构；另一种是文件，即树叶节点，没有子树结构。

20.1 对象的树结构

有向树结构的分类：根据信息传递的方向，有向树结构又可以分为 3 种，从上向下，从下向上和双向的。在三种有向树图中，树的节点和他们的相互关系都是一样的，但是连接他们的关系的方向却不同。

由上向下的树图：在由上向下的树图中，每一个树枝节点都有箭头指向它的所有的子节点，从而一个客户端可以要求一个树枝节点给出所有的子节点，而一个节点却不知道它父节点。在这样的树结构上，信息可以按照箭头所指的方向自上向下传播。



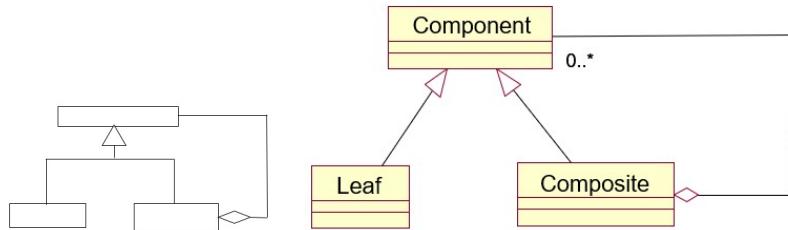
由下向上的树图: 在一个由下向上的树图中，每一个节点都有箭头指向它的父节点，但是父节点却不知道其子节点。信息可以按照箭头所指的方向自下向上传播。

双向的树图: 在一个双向的树图中，每一个节点都同时知道它的父节点和所有的子节点。在这样的树结构上，信息可以按照箭头所指的方向向两个方向传播。

树图中的两种节点: 一个树结构由 2 种节点组成：树枝节点和树叶节点。树枝节点可以有子节点，树叶节点不可以有子节点。注意一个树枝子节点可以不带任何叶子，但是它因为有带有叶子的能力，因此仍然是树枝节点而不会成为叶子节点。一个树叶节点则永远不可能带有子节点。

根节点: 一个树结构中总有至少一个节点是特殊的节点，称为根节点。一个根节点没有父节点，因为它是树结构的根。一个树的根节点一般是树枝节点，如果根节点是树叶节点的话，这个树就变成了只有这一个节点的树。

树结构的类图: 可以使用类图描述一个树结构的静态结构。下图所示的是合成模式的简略类图，同时也是一个典型的树结构的类图。



可看出上面的类图结构涉及到三个角色：

- 抽象构件 (Component) 角色：这是一个抽象角色，它给参加组合的对象规定一个接口。这个角色给出共有的接口及其默认行为。
- 树叶构件 (Leaf) 角色：代表参加组合的树叶对象。一个树叶没有下级的子对象。定义出参加组合的原始对象的行为。
- 树枝构件 (Composite) 角色：代表参加组合的所有包含子对象的对象，并给出树枝构件对象的行为。

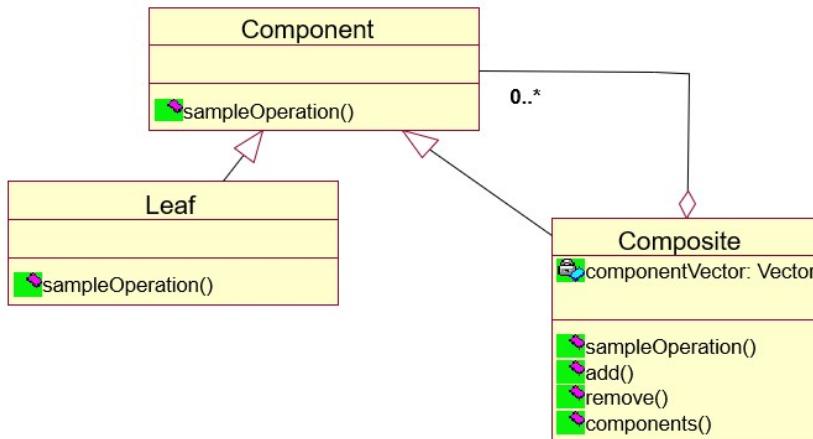
20.2 合成模式的两种形式

透明方式: 作为第一种选择，在 Component 里面声明所有的用来管理子类对象的方法，包括 add()、remove() 以及 getChild () 方法。好处是所有的构件类都有相同的接口。缺点是不够安全，因为树叶类对象和合成类对象在本质上是有区别的。树叶类对象不可能有下一个层次的

对象，因此 add()、remove() 以及 getChild () 是没有意义的，但编译时期不会出错，只在运行时期才可能会出错。

安全方式: 第二种选择是在 Composite 类里面声明所有的用来管理子类对象的方法。这样的做法是安全的做法。这个选择的缺点是不够透明，因为树叶类和合成类将具有不同的接口。

安全式的合成模式的结构: 安全式的合成模式要求管理聚集的方法只出现在树枝构件类中，而不出现在树叶构件类中。其类图如下：



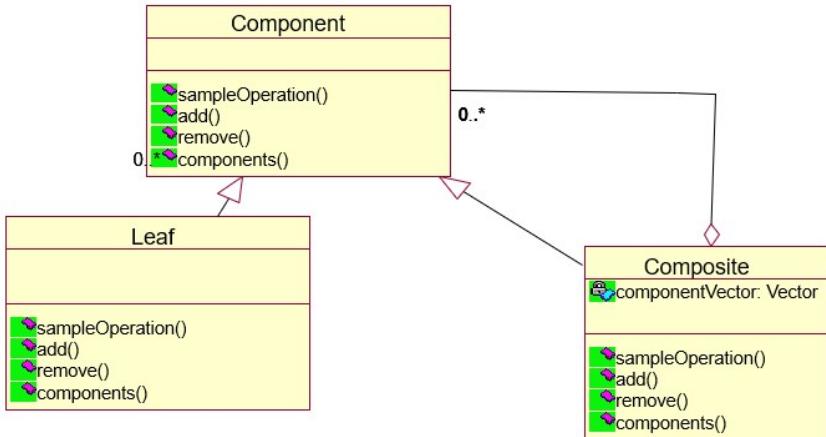
```
1 import java.util.Vector;
2 import java.util.Enumeration;
3 public interface Component {
4     Composite getComposite(); // 返回自己的实例
5     void sampleOperation(); // 某个业务逻辑方法
6 }
7 public class Composite implements Component {
8     private Vector componentVector=new java.util.Vector();
9     public Composite getComposite() {
10         return this; /* 返回自己的实例 */
11     }
12     public void sampleOperation { /* 某业务逻辑方法 */
13         Enumeration enumeration=components();
14         while (enumeration.hasMoreElements()) {
15             ((Component)enumeration.nextElement()).sampleOperation();
16         }
17     }
18     /* 聚集管理方法，增加一个子构件 */
19     public void add(Component component) {
20         componentVector.addElement(component);
21     }
}
```

```

22  /* 聚集管理方法，删除一个子构件 */
23  public void remove(Component component) {
24      componentVector.removeElement(component);
25  }
26 }
27 public class Leaf implements Component {
28     /* 某业务逻辑方法 */
29     public void sampleOperation { }
30     public Composite getComposite() { /* 返回自己的实例 */
31         // write your code here
32         return null;
33     }
34 }

```

透明式的合成模式的结构: 透明式的合成模式要求所有的具体构件类, 不论是树枝还是树叶, 均符合一个统一的接口。其示意类图如下:



```

1 import java.util.Vector;
2 import java.util.Enumeration;
3 public interface Component {
4     Composite getComposite(); // 返回自己的实例
5     void sampleoperation(); // 某个业务逻辑方法
6     /* 聚集管理方法，增加一个子构件 */
7     void add(Component component);
8     /* 聚集管理方法，删除一个子构件 */
9     void remove(Component component);
10    /* 聚集管理方法，返还聚集的Enumeration对象 */
11    Enumeration components( );
12 }

```

```

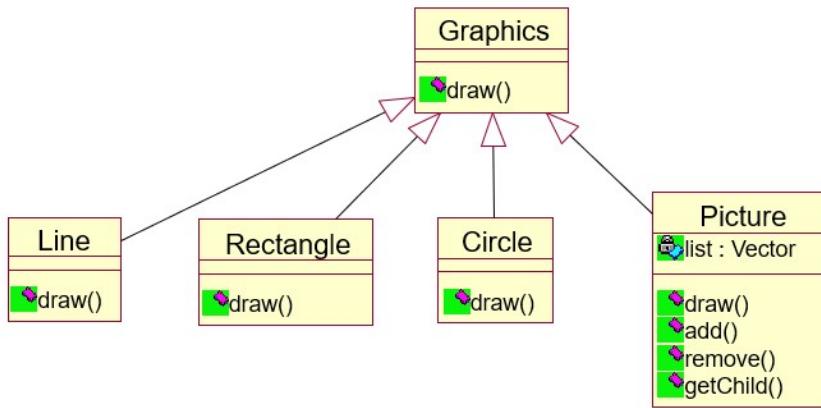
13 public class Composite implements Component {
14     private Vector componentVector=new java.util.Vector();
15     public Composite getComposite() { /*返还自己的实例 */
16         return this;
17     }
18     public void sampleOperation { /*某业务逻辑方法 */
19         Enumeration enumeration=components();
20         while (enumeration.hasMoreElements()) {
21             ((Component)enumeration.nextElement()).sampleOperation();
22         }
23     }
24     /*聚集管理方法，增加一个子构件 */
25     public void add(Component component) {
26         componentVector.addElement(component);
27     }
28     /*聚集管理方法，删除一个子构件 */
29     public void remove(Component component) {
30         componentVector.removeElement(component);
31     }
32     /*聚集管理方法，返回聚集的Enumeration对象
33     public Enumeration components() {
34         return componentVector.elements();
35     }
36 }
37 public class Leaf implements Component {
38     public void sampleOperation { /*某个业务逻辑方法 */
39         //write your code here
40     }
41     public Composite getComposite() { /*返还自己的实例 */
42         return null;
43     }
44     /*聚集管理方法，增加一个子构件 */
45     public void add(Component component) {}
46     /*聚集管理方法，删除一个子构件 */
47     public void remove(Component component){}
48     /*聚集管理方法，返回聚集的Enumeration对象 */
49     public Enumeration components() { return null; }
50 }
```

一个绘图的例子：以一个绘图软件说明合成模式的应用。一个绘图系统给出各种工具用来描绘

由线、长方形和圆形等基本图形组成的图形。一个复合的图形是由这些基本图形组成，可以运用合成模式。合成图形应当有一个列表，存储对所有基本图形的引用。复合图形的 draw() 方法在调用时，应当逐一调用所有列表上的基本图形的 draw() 方法。可以使用两种合成模式来实现：安全式和透明式。

20.3 应用安全式合成模式

安全式的合成模式意味着只有树枝构件才配有管理聚集的方法，而树叶则没有这些方法。下图所示是使用安全式的设计结构图



客户端可以调用 add() 方法加入一个基本图形,remove() 方法取消一个基本图形,getChild(index x) 得到组成复杂图形的第 x 个基本图形。

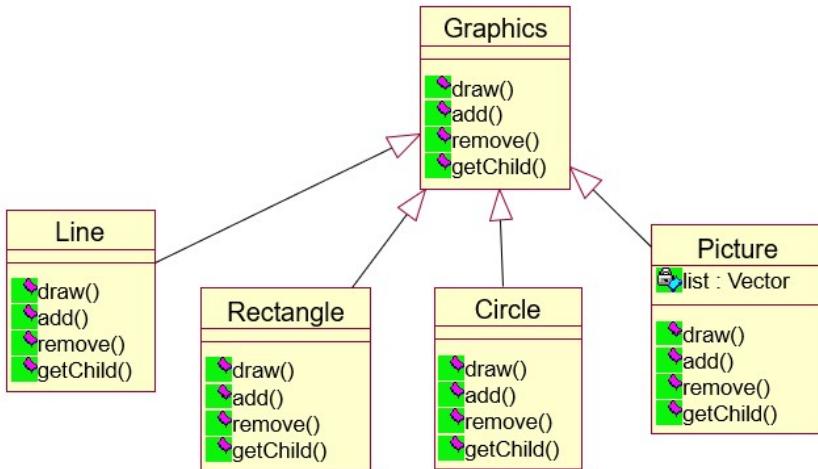
```

1 // 构件角色 Graphics
2 package com.javapatterns.composite.drawingsafe;
3 public abstract class Graphics {
4     public abstract void draw();
5 }
6 // Picture类是树枝构件角色，它实现了抽象构件角色所要求的方法，
7 // 还额外提供了用于管理子对象聚集的一系列方法
8 package com.javapatterns.composite.drawingsafe;
9 import java.util.Vector;
10 public class Picture extends Graphics {
11     private Vector list=new Vector(10);
12     public void draw() {
13         for (int i=0; i< list.size(); i++) {
14             Graphics g = (Graphics)list.get(i);
15             g.draw();
16         }
17     }
  
```

```
18 public void add(Graphics g) { /* 聚集管理方法,增加一个子构件 */
19     list.add(g);
20 }
21 public void remove(Graphics g) { /* 聚集管理方法,删除一个子构件 */
22     list.remove(g);
23 }
24 public void getChild(int i) {
25     return (Graphics)list.get(i); /* 返回一个子构件 */
26 }
27 }
28 //Line是树叶构件, 它没有任何的子对象,
29 //因此不必提供管理子对象聚集的方法
30 package com.javapatterns.composite.drawingsafe;
31 public class Line extends Graphics {
32     public void draw() {
33         //write your code
34     }
35 }
36 //树叶Rectangle
37 package com.javapatterns.composite.drawingsafe;
38 public class Rectangle extends Graphics {
39     public void draw() {
40         //write your code
41     }
42 }
43 //树叶Circle
44 package com.javapatterns.composite.drawingsafe;
45 public class Circle extends Graphics {
46     public void draw() {
47         //write your code
48     }
49 }
```

20.4 应用透明式的合成模式

透明式合成模式无论树枝和树叶都有管理聚集的方法。其设计图如下：



```

1 // 构件角色 Graphics
2 package com.javapatterns.composite.drawingtransparent;
3 abstract public class Graphics {
4     public abstract void draw();
5     public void add(Graphics g); /* 聚集管理方法,增加一个子构件 */
6     public void remove(Graphics g); /* 聚集管理方法,删除一个子构件 */
7     public Graphics getChild(int i); /* 返回一个子构件 */
8 }
9 // Picture类是树枝构件角色, 它实现了抽象构件角色所要求的方法,
10 // 还额外提供了用于管理子对象聚集的一系列方法
11 package com.javapatterns.composite.drawingtransparent;
12 import java.util.Vector;
13 public class Picture extends Graphics {
14     private Vector list=new Vector(10);
15     public void draw() {
16         for (int i=0;i< list.size();i++) {
17             Graphics g=(Graphics)list.get(i);
18             g.draw();
19         }
20     }
21     /* 聚集管理方法,增加一个子构件 */
22     public void add(Graphics g) { list.add(g); }
23     /* 聚集管理方法,删除一个子构件 */
24     public void remove(Graphics g) { list.remove(g); }
25     /* 返回一个子构件 */
26     public Graphics getChild(int i) { return (Graphics)list.get(i); }

```

```
27 }
28 // Line是树叶构件，它必需实现抽象构件角色所要求的方法
29 package com.javapatterns.composite.drawingtransparent;
30 public class Line extends Graphics {
31     public void draw() {
32         //write your code
33     }
34     /* 聚集管理方法,增加一个子构件 */
35     public void add(Graphics g) {
36         //do nothing
37     }
38     /* 聚集管理方法,删除一个子构件 */
39     public void remove(Graphics g) {
40         //do nothing
41     }
42     /* 返回一个子构件 */
43     public Graphics getChild(int i) { return null; }
44 }
45 // Rectangle是树叶构件，它必需实现抽象构件角色所要求的方法
46 package com.javapatterns.composite.drawingtransparent;
47 public class Rectangle extends Graphics {
48     public void draw() {
49         //write your code
50     }
51     public void add(Graphics g) { // 聚集管理方法,增加一个子构件
52         //do nothing
53     }
54     /* 聚集管理方法,删除一个子构件 */
55     public void remove(Graphics g) {
56         //do nothing
57     }
58     /* 返回一个子构件 */
59     public Graphics getChild(int i) { return null; }
60 }
61 // Circle是树叶构件，它必需实现抽象构件角色所要求的方法
62 package com.javapatterns.composite.drawingtransparent;
63 public class Circle extends Graphics {
64     public void draw() {
65         //write your code
```

```

66 }
67 public void add(Graphics g) { // 聚集管理方法,增加一个子构件
68     //do nothing
69 }
70 public void remove(Graphics g) { // 聚集管理方法,删除一个子构件
71     //do nothing
72 }
73 public void getChild(int i) { // 返回一个子构件
74     return null;
75 }
76 }

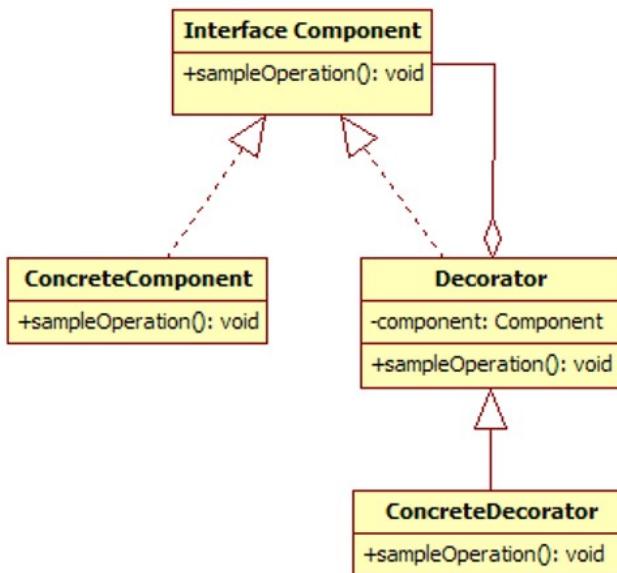
```

21 装饰模式 (Decoration)

装饰模式又称包装 (Wrapper) 模式。装饰模式以对客户端透明的方式扩展对象的功能。是继承关系的一种替代方案。

21.1 装饰模式的结构

装饰模式使用装饰类的一个子类的实例，把客户端的调用委派到其它装饰类或具体构件类。装饰模式的关键在于这种扩展是完全透明的。



模式中的角色:

- 抽象构件 (Component) 角色：给出一个接口，以规范准备接收附加责任（扩展功能）的对象。

- 具体构件 (Concrete Component) 角色：定义一个准备接收附加责任的类。
- 装饰 (Decorator) 角色：持有一个构件的实例，并定义与抽象构件接口一致的接口。
- 具体装饰 (Concrete Decorator) 角色：负责给构件对象“贴上”附加的责任。

下面给出装饰模式的示意性源代码。首先是抽象构件角色的源代码：

```

1 public interface Component {
2     void sampleOperation(); // 某业务逻辑方法
3 }
```

下面是装饰角色的源代码：

```

1 public class Decorator implements Component {
2     private Component component;
3     public Decorator(Component component) {
4         this.component = component;
5     }
6     /* 实现业务方法，委派给构件 */
7     public void sampleOperation() { component.sampleOperation(); }
8 }
```

应当指出的有以下几点：

- 在上面的装饰类里，有一个私有的属性 component，其数据类型是构件 (Component)。
- 此装饰类实现了构件 (Component) 接口。
- 接口的实现方法也值得注意，每一个实现的方法都是委派给私有的属性 component 对象，但并不仅是不单纯的委派，而是将有功能的增强。

虽然 Decorator 类不是一个抽象类，在实际应用中也不一定是抽象类，但是由于他的功能是一个抽象角色，因此也常常称它为抽象装饰。

定义中的具体构件类的示意性源代码：

```

1 public class ConcreteComponent implements Component {
2     public ConcreteComponent() { /* Write your code here */ }
3     public void sampleOperation() { /* Write your code here */ }
4 }
```

具体装饰类实现了抽象装饰类所声明的 sampleOperation() 方法：

```

1 public class ConcreteDecorator extends Decorator {
2     public void sampleOperation() {
3         // 此处可写功能增强的代码
4         super.sampleOperation();
5         // 或在此处写功能增强的代码
6     }
7 }
```

使用装饰模式的优点和缺点:

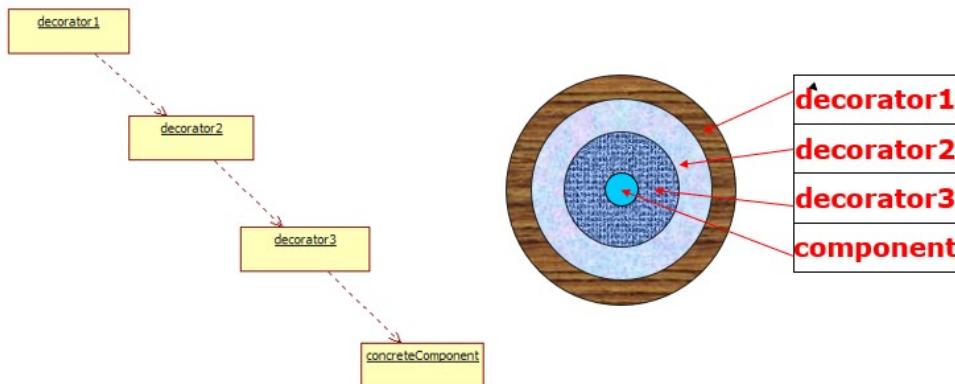
- 优点:
 1. 装饰模式与继承关系的目的都是要扩展对象的功能,但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态地决定“贴上”一个需要的“装饰”,或者除掉一个不需要的“装饰”。继承关系则不同,继承关系是静态的,它在系统运行前就决定了。
 2. 通过使用不同的具体装饰类以及这些装饰类的排列组合,可以创造出很多不同的行为的组合。
- 缺点:产生出较多的对象;比继承更易出错.

对象图:装饰模式的对象图呈链状结构,假设共有三个具体装饰类,分别称为 Decorator1, Decorator2 和 Decorator3, 具体构件类是 ConcreteComponent。

一个典型的创建过程:

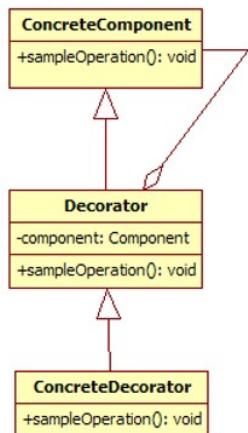
```
new Decorator1(new Decorator2(new Decorator3(new ConcreteComponent()))));
```

这就意味着 Decorator1 的对象持有一个对 Decorator2 对象的引用,后者则持有一个对 Decorator3 对象的引用,再后者持有一个对具体构件 ConcreteComponent 对象的引用,这种链式的引用关系使装饰模式看上去像是一个 LinkedList, 如图所示。

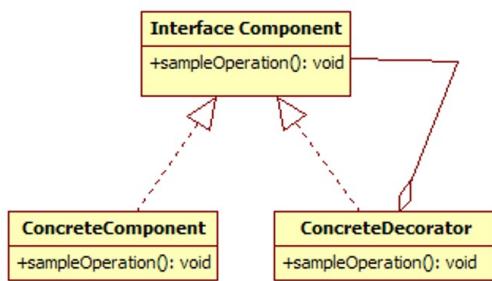


在具体实现时,要注意以下几点:

- 一个装饰类的接口必须与被装饰类的接口相容;
- 尽量保持 Component 作为一个“轻”类;
- 如果有一个 ConcreteComponent 类,而没有抽象的 Component 类,则可以把 Decorator 作为 ConcreteComponent 的子类;如图:



- 若只有一个 ConcreteDecorator 类，则没必要建立一个单独的 Decorator 类。如图：



透明性的要求：不能向客户端“暴露”装饰对象的具体类型，如下列代码所示：

```

1 Component c=new ConcreteComponent();
2 Component c1=new ConcreteDecorator1(c);
3 Component c2=new ConcreteDecorator2(c1);
  
```

而下面的做法是不对的：

```
ConcreteDecorator1 c1=new ConcreteDecorator1();
```

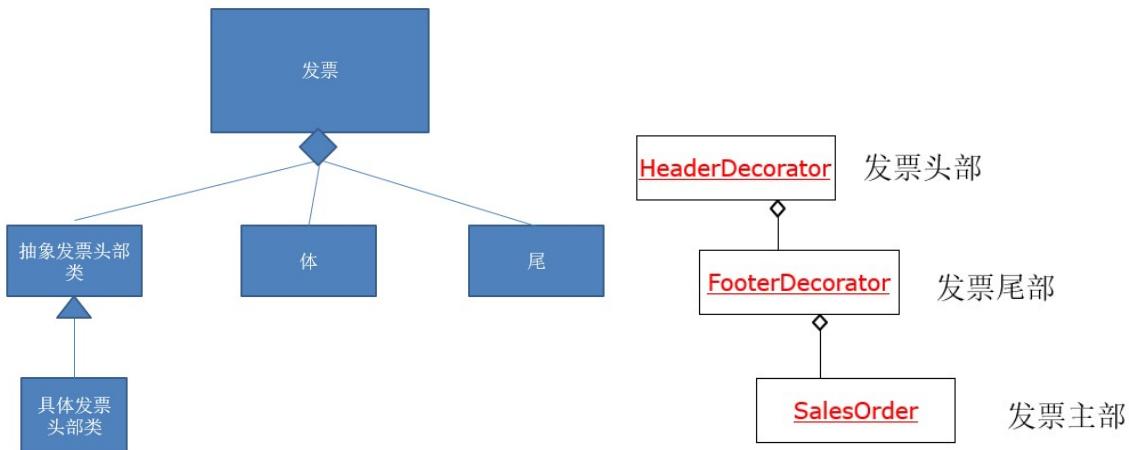
这就是前面所说的，装饰模式对客户端是完全透明的含义。因此，ConcreteDecorator 里不能有 Component 类中没有的公有方法。

21.2 实例：发票系统

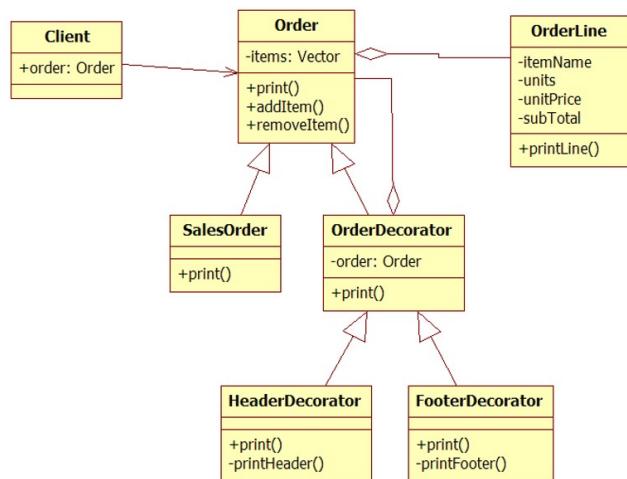
要求：有一个电子销售系统需要打印出顾客的购物发票。一张发票可以分为三个部分：

- 发票头部 (Header)：上面有顾客的名字，销售的日期；
- 发票主部：销售的货物清单，包括商品名字、购买的数量、单价、小计；
- 发票尾部：总金额

发票的头部、尾部可能有多种形式；要求系统能以灵活的方式更换头、尾，可以灵活地组合头部尾部，形成新的发票。



发票系统的类图:



```

1 // 抽象发票类
2 abstract public class Order {
3     private Vector<OrderLine> items = new Vector();
4     public void print() {
5         for (int i = 0; i < items.size(); i++) {
6             OrderLine item = (OrderLine)items.get(i);
7             item.printLine();
8         }
9     }
10    // ...
11 }
```

```

1 // 具体发票主部类
2 public class SalesOrder extends Order {
```

```
3 public void print() { super.print(); }
4 // ...
5 }
```

```
1 // 抽象装饰类
2 abstract public class OrderDecorator extends Order {
3     protected Order order;
4     public OrderDecorator(Order order) { this.order = order; }
5     // ...
6 }
```

```
1 // 具体装饰类—发票头部
2 abstract public class HeaderDecorator extends OrderDecorator {
3     private String cusName;
4     private Date date;
5     public HeaderDecorator(Order order) { super(order); }
6     public void print() {
7         printHeader();
8         order.print();
9     }
10    private void printHeader() {
11        out.println("发票头.....");
12        out.println("顾客名: " + cusName + "; 购物日期: " + date);
13        // ...
14    }
15    // ...
16 }
```

```
1 // 具体装饰类—发票尾部
2 public class FooterDecorator extends OrderDecorator {
3     public FooterDecorator(Order order) { super(order); }
4     public void print() {
5         order.print();
6         printFooter();
7     }
8     private void printFooter() {
9         out.println("发票尾.....");
10        // ...
11    }
12    // ...
```

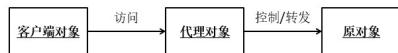
13 }

```
1 // 客户端类
2 public class Clint {
3     private Order order;
4     public static void main(String args[]) {
5         order=new SalesOrder();
6         OrderLine line1=new OrderLine();
7         line1.setName("毛巾");
8         line1.setUnits(1);
9         line1.setPrice(10);
10        order.addItem(line1);
11        order=new HeaderDecorator(new FooterDecorator(order));
12        order.print();
13    }
14 }
```

22 代理模式 (proxy)

代理模式给某一个对象提供一个代理，并由代理对象控制对原对象的引用。

代理模式的英文为 Proxy，中文可译成“代理”。所谓代理，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。在一些情况下，一个客户端对象不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。



若按使用目的划分，代理有以下几种：

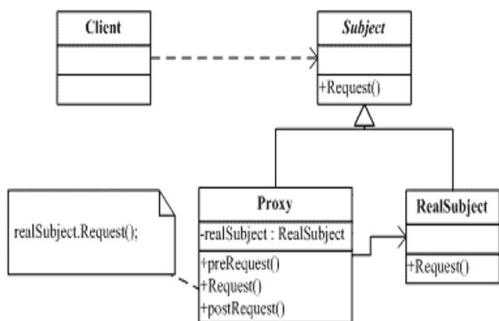
- 远程 (Remote) 代理：为一个位于不同的地址空间的对象提供一个局域代理对象。这个不同的地址空间可以是在本机器中，也可是在另一台机器中。远程代理又叫做大使 (Ambassador)。
- 虚拟 (Virtual) 代理：根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。
- Copy-on-Write 代理：虚拟代理的一种。把复制 (克隆) 拖延到只有在客户端需要时，才真正采取行动。
- 保护 (Protect or Access) 代理：控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。
- Cache 代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。

- 防火墙 (Firewall) 代理: 保护目标, 不让恶意用户接近.
- 同步化 (Synchronization) 代理: 使几个用户能够同时使用一个对象而没有冲突.

代理的例子: Windows 系统提供快捷方式 (Shortcut), 可以使任何对象同时出现在多个地方而不必修改原对象. 对快捷方式的调用完全与对原对象的调用一样, 换言之, 快捷方式对客户端是完全透明的.

22.1 代理模式的结构

代理模式的类图如图所示:



代理模式所涉及的角色:

- 抽象主体角色 (Subject): 声明了真实主体和代理主体的共同接口, 这样一来在任何使用真实主体的地方都可以使用代理主体.
- 代理主体 (Proxy) 角色: 代理主体角色内部含有对真实主体的引用, 从而可以在任何时候操作真实主体对象; 代理主体角色提供一个与真实主体角色相同的接口, 以便可以在任何时候都可以替代真实主体; 控制真实主体的引用, 负责在需要的时候创建真实主体对象 (和删除真实主体对象); 代理角色通常在将客户端调用传递给真实的主体之前或之后, 都要执行某个操作, 而不是单纯的将调用传递给真实主体对象.
- 真实主体角色 (RealSubject) 角色: 定义了代理角色所代表的真实对象.

```

1 package com.javapatterns.proxy;
2 abstract public class Subject {
3     // 声明一个抽象的请求方法
4     abstract public void request();
5 }
```

```

1 public class RealSubject extends Subject {
2     public RealSubject() { }
3     // 实现请求方法
4     public void request() {
5         System.out.println("From real subject.");
6     }
7 }
```

```
6    }
7 }
```

```
1 public class ProxySubject extends Subject {
2     private RealSubject realSubject;
3     public ProxySubject() { }
4     public void request() { // 实现请求方法
5         preRequest();
6         if (realSubject == null) {
7             realSubject = new RealSubject();
8         }
9         realSubject.request();
10        postRequest();
11    }
12    // 请求前的操作
13    private void preRequest() { }
14    // 请求后的操作
15    private void postRequest() { }
16 }
```

调用代理主体:

```
Subject subject=new ProxySubject();subject.request();
```

23 享元模式 (Flyweight)

享元模式的用意: 享元模式是对象的结构模式. 享元模式以共享的方式高效地支持大量的细粒度对象的维护与管理. 享元对象能实现共享的关键是区分**内部状态** (Internal state) 和**外部状态** (External State).

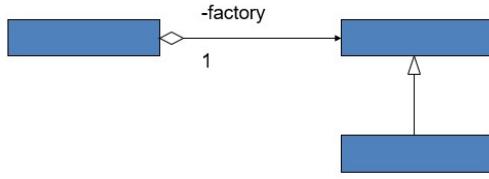
当以下所有条件都满足时, 可以考虑使用享元模式:

1. 一个系统有大量的对象.
2. 这些对象消耗大量的内存.
3. 这些对象的状态中的大部分都可以外部化.
4. 软件系统不依赖于这些对象的身份, 即这些对象可以是不可分辨的.

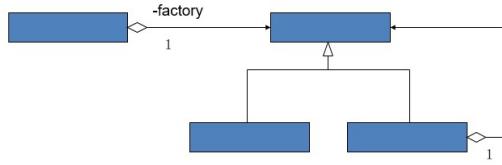
内部状态是存储在享元对象内部的并且是不会随环境改变而有所不同的. 因此, 一个享元可以具有内部状态并可以共享.

外部状态是随环境改变而改变的、不可以共享的状态. 享元对象的外部状态必须由客户端保存, 并在享元对象被创建之后, 在需要使用的时候再传入到享元对象内部. 外部状态不可以影响享元对象的内部状态. 内外部状态是互相独立的.

享元模式的种类: 根据所涉及的享元对象的内部结构, 享元模式可以分成**单纯享元模式**和**复合享元模式**两种形式. 下图是单纯享元模式的结构示意图:



下图所示是复合享元模式的结构示意图:



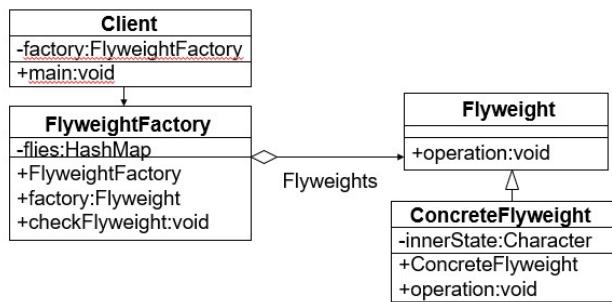
可见, 在复合享元模式中, 享元对象构成合成模式. 因此, **复合享元模式**实际上是**单纯享元模式与合成模式的组合**.

享元模式的应用:

- 享元模式在编辑器系统中大量使用. 一个文本编辑器往往会提供很多种字体, 而通常的做法就是将每一个字母作成一个享元对象. 享元对象的内部状态就是这个字母的字符数据, 而字母在文本中的位置和字体风格等其它信息则是外部状态.
- 在 Java 语言中, String 类型就使用了享元模式. String 对象是不变对象, 一旦创建出来就不能改变.

23.1 单纯享元模式的结构

单纯享元模式中, 所有享元对象都是可以共享的. 下图是一个单纯享元模式的简单实现:



单纯享元模式涉及的角色:

- 抽象享元角色: 该角色是所有的具体享元类的超类, 为这些类规定出需要实现的公共接口.
- 具体享元 (ConcreteFlyweight) 角色: 实现抽象享元角色所规定的接口.
- 享元工厂 (FlyweightFactory) 角色: 本角色负责创建和管理享元角色

- 客户端 (Client) 角色: 本角色需要维护一个对所有享元对象的引用.

```

1 // 抽象享元角色
2 abstract public class Flyweight {
3     //一个示意性的方法，参数state表示外部状态
4     abstract public void operation(String state);
5 }
```

```

1 public class ConcreteFlyweight extends Flyweight {
2     private Character innerState = null;
3     // 构造函数，内部状态作为参数传入
4     public ConcreteFlyweight(Character state) {
5         this.innerState = state;
6     }
7     //外部状态作为参数传入方法中，改变方法的行为
8     //但并不改变对象的内部状态
9     public void operation(String state) {
10        System.out.print( "\nIntrinsic State = " + innerState +
11        ", Extrinsic State = " + state);
12    }
13 }
```

客户端不可以直接将具体享元类实例化，而必须通过一个工厂对象，利用一个 factory() 方法得到享元对象。一般而言，享元对象在整个系统中只有一个，因此可以使用单例模式。当客户端需要单纯享元对象的时候，需要调用享元工厂的 factory() 方法，并传入所需的单纯享元对象的内部状态。

```

1 import java.util.Map;
2 import java.util.HashMap;
3 import java.util.Iterator;
4 public class FlyweightFactory {
5     private HashMap flies = new HashMap();
6     private Flyweight lnkFlyweight;
7     public FlyweightFactory() { } //默认构造函数
8     //构造函数,内部状态作为参数传入
9     public synchronized Flyweight factory (Character state) {
10        if (flies.containsKey(state)) {
11            return (Flyweight) flies.get(state);
12        } else {
13            Flyweight fly = new ConcreteFlyweight(state);
14            flies.put(state, fly);
```

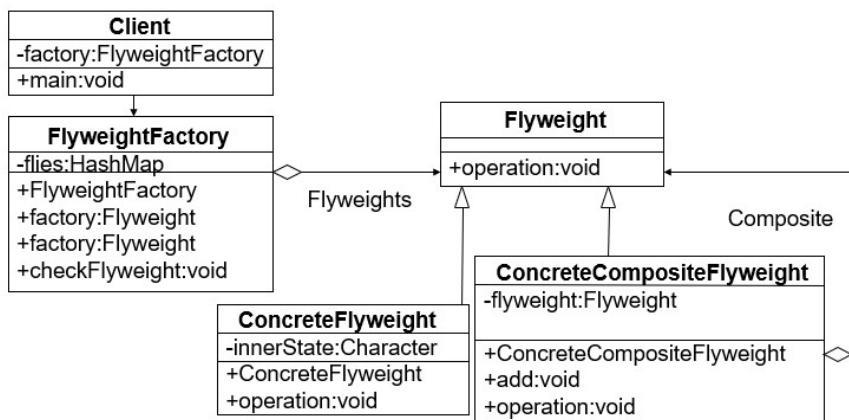
```

15     return fly;
16 }
17 }
18 public void checkFlyweight() { //辅助方法
19     Flyweight fly ;
20     int i = 0;
21     System.out.println("\n=====checkFlyweight()=====");
22     for (Iterator it = flies.entrySet().iterator(); it.hasNext();) {
23         Map.Entry e = (Map.Entry) it.next();
24         System.out.println("Item " + (++i) + " : " + e.getKey());
25     }
26     System.out.println("=====checkFlyweight()=====");
27 }
28 }

```

23.2 复合享元模式的结构

在上面的单纯享元模式中，所有的享元对象都是单纯享元对象，即都是可以直接共享的。下面考虑一个较复杂的情况，将一些单纯享元使用合成模式加以组合，形成复合享元对象。这样的复合本身不能共享，但是它们可以分解成单纯享元对象，而后者可以共享。复合享元模式的类图如下：



复合享元模式涉及的角色：

- 抽象享元角色：此角色是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。
- 具体享元 (ConcreteFlyweight) 角色：实现抽象享元角色所规定的接口。
- 复合享元 (ConcreteCompositeFlyweight) 角色：复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。
- 享元工厂 (FlyweightFactory) 角色：负责创建和管理享元角色。必须保证享元对象可以被系统适当的共享。

- 客户端 (Client) 角色: 使用享元对象, 但需要自行存储所有享元对象的外部状态.

```

1 abstract public class Flyweight {
2     //外部状态作为参数传入到方法中
3     abstract public void operation(String state);
4 }
```

具体享元角色的主要责任:

- 实现了抽象享元角色所声明的接口, 也就是 operation() 方法.Operation() 方法接收外部状态作为参数.
- 为内部状态提供存储空间的, 在本实现中就是 innerState 属性. 享元模式本身对内部状态的存储类型并无要求, 这里的内部状态就是 Character 类型, 是为了给复合享元的内部状态选做 String 类提供方便.

```

1 public class ConcreteFlyweight extends Flyweight {
2     private Character innerState = null;
3     //构造函数,内部状态作为参数传入
4     public ConcreteFlyweight(Character state) {
5         this.innerState = state;
6     }
7     //外部状态作为参数传入到方法中
8     public void operation(String state) {
9         System.out.print("\nInternal State = " + innerState
10            + " Extrinsic State = " + state);
11    }
12 }
```

```

1 import java.util.Map;
2 import java.util.HashMap;
3 import java.util.Iterator;
4 public class ConcreteCompositeFlyweight extends Flyweight {
5     private HashMap flies = new HashMap(10);
6     public ConcreteCompositeFlyweight() { }
7     //增加一个新的单纯享元对象到聚集中
8     public void add(Character key, Flyweight fly) { flies.put(key, fly); }
9     //外部状态作为参数传入到方法中
10    public void operation(String extrinsicState) {
11        Flyweight fly = null;
12        for (Iterator it = flies.entrySet().iterator(); it.hasNext(); ) {
13            Map.Entry e = (Map.Entry) it.next();
14            fly = (Flyweight) e.getValue();
```

```
15     fly.operation(extrinsicState);
16 }
17 }
18 }
```

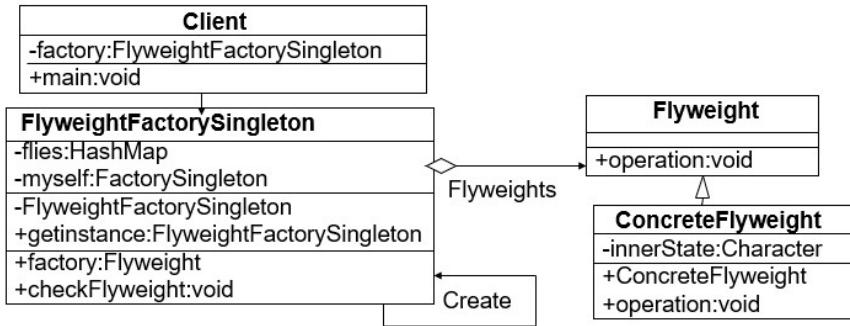
当客户端需要复合享元对象的时候，需要调用享元工厂的 factory() 方法，并传入所需的复合享元对象的所有复合元素的内部状态。

```
1 public class FlyweightFactory {
2     private HashMap flies = new HashMap();
3     public FlyweightFactory(){}
4     //复合享元工厂方法，所需状态以参数形式传入
5     //这个参数恰好可以使用 String 类型。
6     public Flyweight factory(String compositeState) {
7         ConcreteCompositeFlyweight compositeFly =
8             new ConcreteCompositeFlyweight();
9         int length = compositeState.length();
10        Character state = null;
11        for(int i = 0; i < length; i++) {
12            state = new Character (compositeState.charAt(i));
13            compositeFly.add(state, this.factory(state));
14        }
15        return compositeFly;
16    }
17    //单纯享元工厂方法，所需状态以参数形式传入
18    public Flyweight factory(Character state) {
19        //检查具有此状态的享元是否已经存在
20        if (flies.containsKey(state)) { //已存在，直接返回
21            return (Flyweight) flies.get(state);
22        } else { //不存在，创建新实例
23            Flyweight fly = new ConcreteFlyweight(state);
24            flies.put(state, fly); //将实例存储到聚集中
25            return fly; //将实例返回
26        }
27    }
28 }
```

23.3 单例模式实现享元工厂角色

系统往往只需要一个享元工厂的实例，所以享元工厂可以设计成为单例模式。

单纯享元模式中的享元工厂角色：



这是一个示意性客户端类的源代码，显示出调用单例工厂对象的 getInstance() 方法，以得到具体享元类。

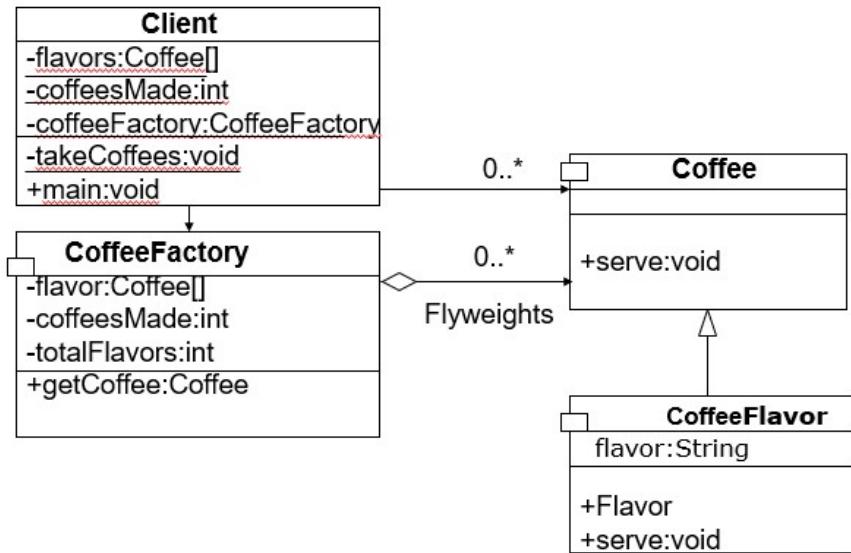
```

1 public class ClientSingleton {
2     private static FlyweightFactorySingleton factory;
3     public static void main(String[] args) {
4         //创建享元工厂对象
5         factory = FlyweightFactorySingleton.getInstance();
6         //向享元工厂对象请求一个内部状态为' a' 的对象
7         Flyweight fly = factory.factory(new Character('a'));
8         //以参数方式传入外部状态
9         fly.operation("First Call");
10        //向工厂对象请求一个内部状态为' b' 的对象
11        fly = factory.factory(new Character('b'));
12        //以参数方式传入外部状态
13        fly.operation("Second Call");
14        //向工厂对象请求一个内部为' a' 的享元对象
15        fly = factory.factory(new Character('a'));
16        //以参数方式传入外部状态
17        fly.operation("Third Call");
18
19    }
20}

```

23.4 例子：户外咖啡摊

系统的要求: 在这个咖啡摊 (Coffee Stall) 所使用的系统里，有一系列具有不同“口味 (Flavor)”的咖啡。客人到摊位上购买咖啡，所有的咖啡均放在台子上，客人自己买到咖啡后就离开摊位。咖啡有内部状态，也就是咖啡的口味；咖啡没有外部状态。如果系统为每一杯咖啡都创建一个独立的对象那么就需要创建很多小对象，所以只要把咖啡按照种类 (即口味) 划分，每一种口味的咖啡只创建一个对象，并实行共享。



在这里享元模式所涉及的角色如下:

- 抽象享元 (抽象咖啡) 角色: 此角色由 Coffee 类扮演, 它是所有的具体享元类的超类, 为这些类规定出需要实现的公共接口.
- 具体享元 (不同 “口味”的咖啡) 角色: 这个角色有 CoffeeFlavor 类扮演, 它的实例是可以共享的. 每一个不同的咖啡都对应于独立而且唯一的享元对象.
- 享元工厂 (咖啡工厂) 角色: 这个角色由 CoffeeFactory 类扮演, 它负责创建所有 “口味”的咖啡对象.

客户端并不直接创建任何 “口味” 的咖啡对象, 而是向咖啡工厂提出请求, 由咖啡工厂提高一个相应的对象. 下面是抽象享元角色的源代码:

```

1 public abstract class Coffee {
2     public abstract void serve(); //将咖啡卖给客人
3     public abstract String getFlavor(); //返回咖啡的名字
4 }
```

“口味” 角色的源代码实现了抽象 Order 角色所声明的接口:

```

1 public class CoffeeFlavor extends Coffee {
2     private String flavor;
3     public CoffeeFlavor(String flavor) { //内部状态以参数方式传入
4         this.flavor = flavor;
5     }
6     public String getFlavor() { //返回咖啡名字
7         return this.flavor;
8     }
9     public void serve() { //将咖啡卖给客人
}
```

```
10     System.out.println("Serving flavor " + flavor );
11 }
12 }
```

所有不同“口味”的咖啡对象都应当由咖啡工厂提供，而不应当由客户端直接创建。咖啡工厂的源代码如下：

```
1 public class CoffeeFactory {
2     private Coffee[] coffees = new Coffee[20];
3     private int coffeesMade = 0;
4     private int totalFlavors = 0 ;
5     //工厂方法，根据所需的口味提供咖啡
6     public Coffee getCoffee(String flavorToGet) {
7         if (totalFlavors > 0) {
8             for (int i = 0; i < totalFlavors ; i++) {
9                 if (flavorToGet.equals((coffees[i]).getFlavor())) {
10                     coffeesMade++;
11                     return coffees[i];
12                 }
13             }
14         }
15         coffees[totalFlavors] = new CoffeeFlavor(flavorToGet);
16         coffeesMade++;
17         return coffees[totalFlavors++];
18     }
19     //辅助方法，返回创建过的口味对象的个数
20     public int getTotalFlavorsMade() { return totalFlavors; }
21 }
```

下面是系统的客户端角色的源代码，该角色可代表咖啡摊的工作人员。

```
1 //客户端角色，代表咖啡摊的侍者
2 public class Client {
3     //记录卖出咖啡的总数目
4     private static Coffee[] coffees = new CoffeeFlavor[20];
5     private static int ordersMade = 0;
6     private static CoffeeFactory coffeeFactory;
7     //静态方法，提供一杯咖啡
8     private static void takeCoffees(String aFlavor) {
9         coffees[ordersMade++]=coffeeFactory.
10         getOrder(aFlavor);
11     }
```

```

12 public static void main(String[] args) {
13     //创建口味工厂
14     coffeeFactory = new CoffeeFactory();
15     //创建一个个咖啡对象
16     takeCoffees("Black Coffee");
17     takeCoffees("Capucino");
18     takeCoffees("Espresso");
19     takeCoffees("Espresso");
20     takeCoffees("Capucino");
21     takeCoffees("Capucino");
22     takeCoffees("Black Coffee");
23     takeCoffees("Espresso");
24     takeCoffees("Capucino");
25     takeCoffees("Black Coffee");
26     takeCoffees("Espresso");
27     //将所创建的咖啡对象卖给客人
28     for (int i = 0; i < coffeesMade; i++) {
29         coffees[i].serve();
30     }
31     //打印出卖出的咖啡总数
32     System.out.println("\nTotal teaFlavor objects
33     made: "+coffeeFactory.getTotalFlavorsMade());
34 }
35 }
```

可以看出，这个咖啡摊可以为客人准备最多 20 种不同口味的咖啡。虽然上面的客户端对象叫了 11 杯咖啡，但是所有咖啡的口味却只有三种，即 Capucino、Espresso 和 Black Coffee。

23.5 例子: 咖啡屋

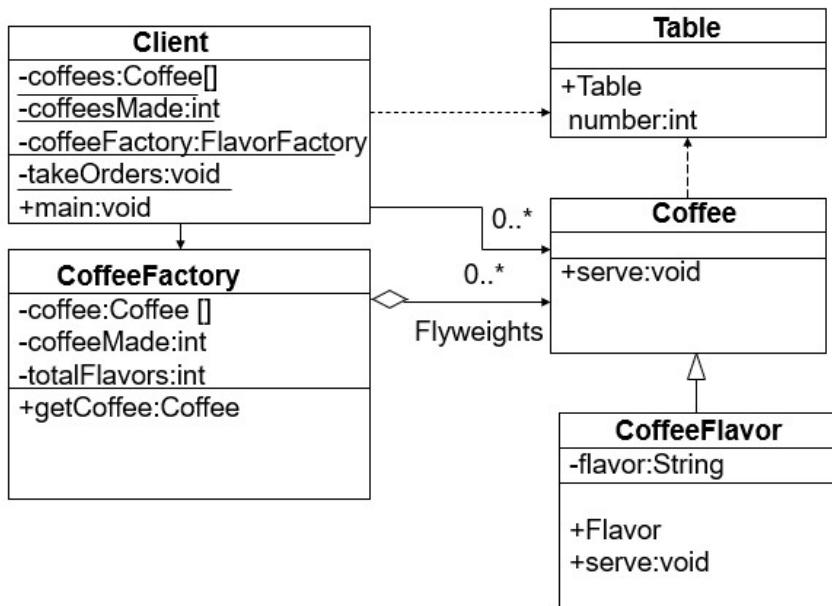
系统的要求: 在前面的咖啡摊项目里，由于没有供客人坐的桌子，所有的咖啡对象均没有考虑环境的影响。咖啡仅有内部状态，也就是咖啡的种类，而没有外部状态。下面考虑一个规模稍稍大一点的咖啡屋 (Coffee Shop) 项目。屋子里有很多桌子供客人坐，系统除了需要提供不同“口味”的咖啡之外，还需要跟踪咖啡被送到哪一个桌位上，因此，咖啡就有了桌子作为外部状态。

系统的设计: 由于外部状态的存在，没有外部状态的单纯享元模式不再符合要求。系统的设计可以利用有外部状态的单纯享元模式。在这里享元模式所涉及的角色与咖啡摊项目的角色都是一样的，除了 Table 角色之外：

- 环境角色：由 Table 类扮演，这就是所有的享元角色所涉及的外部状态。
- 抽象享元角色：此角色由 Coffee 类扮演，它是所有的具体享元类的超类，为这些类规定出

需要实现的公共接口.

- 具体享元 (咖啡“口味”) 角色: 该角色由 CoffeeFlavor 类扮演, 它的实例是可以共享的. 每一杯不同的咖啡都对应一个独立而且唯一的享元对象. 一个咖啡对象在创建出来之后, 其内部状态就不再改变.
- 享元工厂角色: 该角色由 FlavorFactory 类扮演, 它负责创建一个对应的咖啡对象. 在客户端提出请求后, 咖啡工厂就会检查是否已经有一个对应的对象存在, 如已有, 直接返回这个已有的对象; 反之, 创建一个新的对象, 并提供给客户端.
- 客户端 (Client) 角色: 客户端并不直接创建任何“口味”的咖啡对象, 而是向咖啡工厂提出请求, 由咖啡工厂提供一个相应的对象. 本角色代表咖啡屋的负责招待客人的侍者, 侍者代替客人向调制咖啡的工作人员请求咖啡, 后者将调制好的咖啡提供给侍者.



抽象享元角色的源代码如下:

```

1 public abstract class Coffee {
2     //将咖啡卖给客人
3     public abstract void serve(Table table);
4     //返回咖啡名字
5     public abstract String getFlavor();
6 }
  
```

“口味”角色的源代码如下:

```

1 public class CoffeeFlavor extends Coffee {
2     private String flavor;
3     //构造函数, 内部状态以参数方式传入
4     public Flavor(String flavor) { this.flavor = flavor; }
  
```

```

5 //普通方法，返回咖啡名字
6 public String getFlavor() { return this.flavor; }
7 //将咖啡卖给客人
8 public void serve(Table table) {
9     System.out.println("Serving table " + table
10        .getNumber() + " with flavor " + flavor );
11    }
12 }

```

“口味”工厂的源代码:

```

1 public class CoffeeFactory {
2     private Coffee[] coffees = new Coffee[20];
3     private int coffeesMade = 0;
4     private int totalFlavors = 0 ;
5     //工厂方法，根据所需的口味提供咖啡
6     public Coffee getCoffee(String flavorToGet) {
7         if (totalFlavors > 0) {
8             for (int i = 0; i < totalFlavors; i++) {
9                 if (flavorToGet.equals((coffees[i]).getFlavor())) {
10                     totalFlavors++;
11                     return coffees[i];
12                 }
13             }
14         }
15         coffees[totalFlavors] = new Coffee (flavorToGet);
16         coffeesMade++;
17         return flavors[totalFlavors++];
18     }
19     //辅助方法，返回创建过的口味种类的个数
20     public int getTotalFlavorsMade() {
21         return totalFlavors;
22     }
23 }

```

系统环境角色 Table 类的代码:

```

1 package com.javapatterns.flyweight.coffeeshop;
2 public class Table {
3     private int number; //桌子号码
4     public Table(int number) { //构造函数
5         this.number = number;

```

```
6    }
7    public void setNumber(int number) {
8        this.number = number;
9    }
10   public int getNumber() { return number; }
11 }
```

系统客户端角色代码如下:

```
1 public class Client {
2     //卖出的咖啡总数
3     private static Coffee[] coffee = new Coffee[100];
4     private static int coffeesMade = 0;
5     private static CoffeeFactory coffeeFactory;
6     //静态方法，提供一杯咖啡
7     private static void takeCoffees(String aFlavor) {
8         coffee[coffeesMade++] = flavorFactory.getCoffee(aFlavor);
9     }
10    public static void main(String[] args) {
11        //创建咖啡工厂对象
12        coffeeFactory = new CoffeeFactory();
13        //创建一个个咖啡对象
14        takeCoffees("Black Coffee");
15        takeCoffees("Capucino");
16        takeCoffees("Espresso");
17        takeCoffees("Espresso");
18        takeCoffees("Capucino");
19        takeCoffees("Capucino");
20        takeCoffees("Black Coffee");
21        takeCoffees("Espresso");
22        takeCoffees("Capucino");
23        takeCoffees("Black Coffee");
24        takeCoffees("Espresso");
25        //将所创建的咖啡对象卖给客人
26        for (int i = 0; i < coffeesMade; i++) {
27            coffee[i].serve(new Table(i));
28        }
29        //打印卖出的咖啡总数
30        System.out.println("\nTotal teaFlavor objects made:" + flavorFactory.
31                           getTotalFlavorsMade());
32    }
```

24 门面模式 (Facade)

门面模式: 外部与子系统的通信必需通过一个统一的门面 (facade) 对象进行. 门面模式是对象形式的结构型设计模式. 门面模式提供一个高层次的接口, 使得子系统更易于使用.

基于模块化的子系统划分:

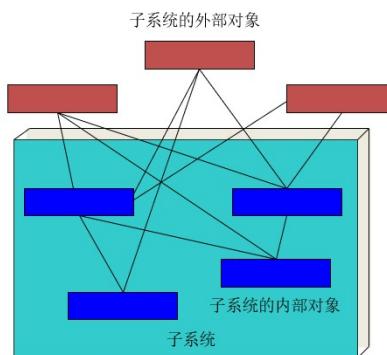
- 大型的软件系统都是比较复杂的, 软件设计的重要任务之一就是要合理控制软件复杂性.
- 处理复杂系统的一个常见方法便是将其“分而治之”, 把一个系统划分为几个较小的子系统.
- 而使用一个子系统的客户端往往需要同时与子系统内部的许多对象交互.

什么情况下使用门面模式:

- **为一个复杂子系统提供一个简单接口:** 子系统往往因为不断演化而变得越来越复杂, 使用门面模式可以使得子系统更具有可复用性.Facade 模式可以提供一个简单的操作视图.
- **提高子系统的独立性:** 一般而言, 子系统和其他的子系统之间、客户端与实现层之间存在着很大的依赖性. 引入 Facade 模式将一个子系统与它的客户端以及其他子系统分离, 可以提高子系统的独立性和可移植性.
- **形成层次化结构:** 在构建一个层次化的系统时, 可以使用 Facade 模式定义系统中每一层的入口. 如果层与层之间是相互依赖的, 则可以限定它们仅通过 Facade 进行通信, 从而简化了层与层之间的依赖关系.

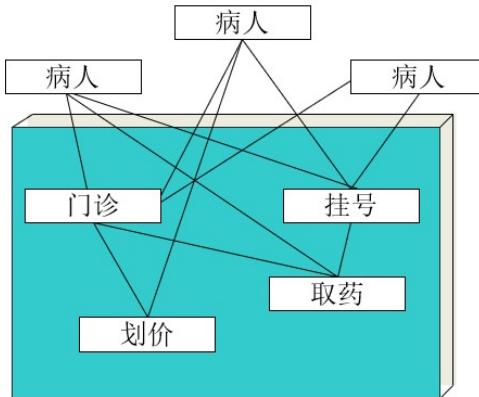
迪米特法则: 迪米特法则说:“只与你直接的朋友们通信”. 迪米特法要求每一个对象与其他对象的相互作用均是短程的, 而不是长程的. 只要有可能, 朋友的数目越少越好. 即一个对象只应当知道它的直接合作者的接口. 门面模式创造出一个门面对象, 将客户端所涉及的属于一个子系统的协作伙伴的数目减到最少, 使得客户端与子系统内部的对象的相互作用被门面对象所取代.

不同子系统间对象交互如图: 该图描述的是一个客户端必须与许多对象打交道才能完成一个功能, 图中的大方框代表一个子系统.



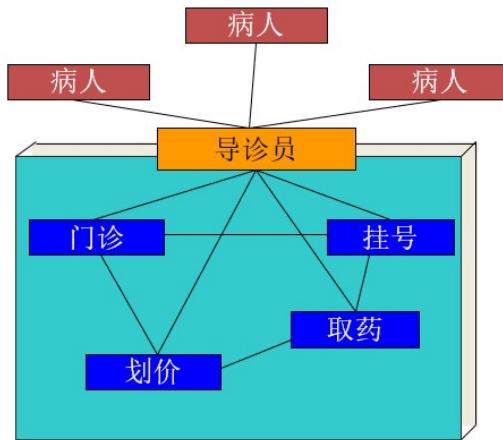
24.1 例子: 医院

若把医院作为一个子系统, 按照部门职能, 这个系统可以划分为挂号、门诊、划价、化验、收费、取药等部门. 看病的病人要与这些部门打交道, 就如同一个子系统的客户端与一子系统的各个类打交道一样, 不是一件容易的事情. 首先病人必须挂号, 然后门诊. 如果医生要求化验, 病人必须先划价, 然后缴款, 才能到化验部门做化验. 化验后, 再回到门诊室.



上图描述的是病人在医院里的体验, 图中的方框代表医院。

解决这种不便的方法就是引进门面模式. 仍然通过医院的例子说明, 可以设置一个导诊员的位置, 由接待员负责代为挂号、划价、缴费、取药等. 这个接待员就是门面模式的体现, 病人只接触接待员, 由接待员负责与医院的各个部门打交道, 如图所示.

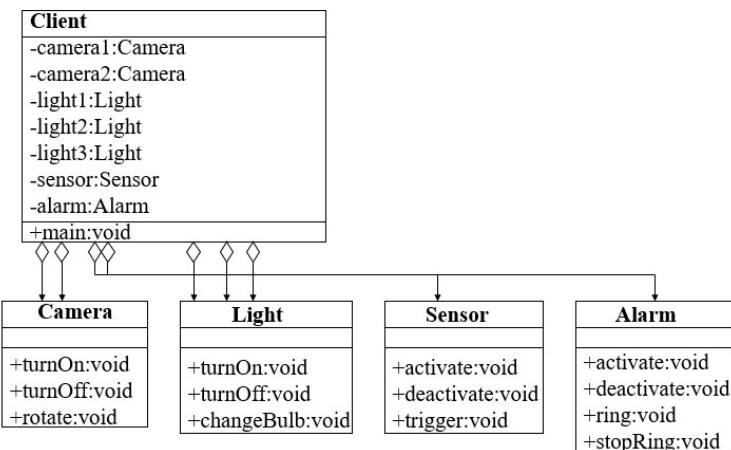


24.2 例子: 安全监控系统

一个安全监控系统由两个录像机、三个电灯、一个传感器和一个警报器组成. 安全监控系统的操作人员需要经常将这些仪器启动和关闭. 首先, 在不使用门面模式的情况下, 操作这个系统的操作员必须直接操作所有的这些设备.

24.2.1 不使用门面模式的设计

不使用门面模式的情况下，系统的设计图如图所示，可看出，Client 对象需要引用到所有的录像机 (Camera)、电灯 (Light)、感应器 (Sensor) 和警报器 (Alarm) 对象。Client 对象必须对安全监控系统全知全能，这是一个高耦合度的做法。



```
1 public class Camera {
2     public void turnOn() { //打开录像机
3         System.out.println("Turning on the camera.");
4     }
5     public void turnOff() { //关闭录像机
6         System.out.println("Turning off the camera.");
7     }
8     public void rotate(int degrees) { //转动录像机
9         System.out.println("rotating the camera by " + degrees + " degrees");
10    }
11 }
```

```
1 public class Light {
2     public void turnOn() { //打开灯
3         System.out.println("Turning on the camera.");
4     }
5     public void turnOff() { //关闭灯
6         System.out.println("Turning off the camera.");
7     }
8     public void changeBulb() { //换灯泡
9         System.out.println("changing the light-bulb.");
10    }
11 }
```

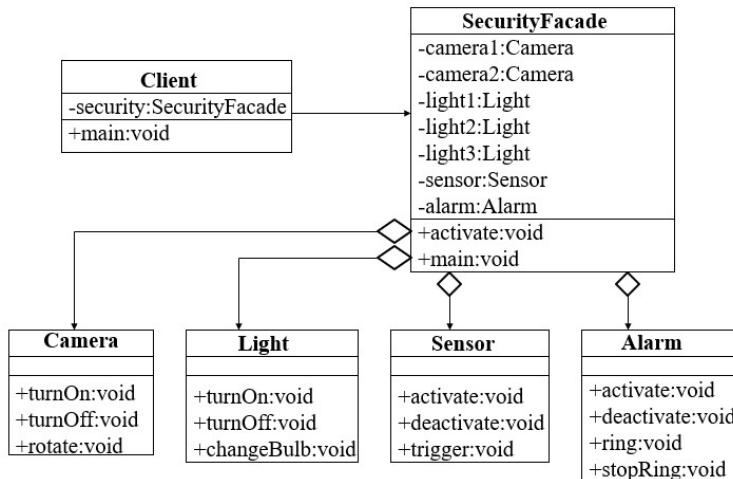
```
1 public class Sensor {  
2     public void activate() { //启动感应器  
3         System.out.println("Activating the sensor.");  
4     }  
5     public void deactivate() { //关闭感应器  
6         System.out.println("Deactivating the sensor.");  
7     }  
8     public void trigger() { //触发感应器  
9         System.out.println("The sensor has been triggered.");  
10    }  
11 }
```

```
1 public class Alarm {  
2     public void activate() { //启动警报器  
3         System.out.println("Activating the alarm.");  
4     }  
5     public void deactivate() { //关闭警报器  
6         System.out.println("Deactivating the alarm.");  
7     }  
8     public void ring() { //拉响警报器  
9         System.out.println("Ringing the alarm.");  
10    }  
11 }
```

```
1 public class Client {  
2     static private Camera camera1, camera2;  
3     static private Light light1, light2, light3;  
4     static private Sensor sensor;  
5     static private Alarm alarm;  
6     public static void main(String[] args) {  
7         camera1.turnOn();  
8         camera2.turnOn();  
9         light1.turnOn();  
10        light2.turnOn();  
11        light3.turnOn();  
12        sensor.activate();  
13        alarm.activate();  
14    }  
15 }
```

24.2.2 使用门面模式的设计

一个合理的改进方案就是准备一个系统的控制台，作为安全监控系统的用户界面。用户只需要操作这个简化的界面就可以操控所有的内部设备，这实际上就是门面模式的用意。使用门面模式对前面的设计方案进行改造后，就得到了如下页图所示的设计图，可以看出，门面 Security-Facade 对象承担了与安全监控系统内部各个对象打交道的任务，而客户对象只需要与门面对象打交道即可。这是客户端与安全监控系统之间的一个门户，它使二者之间的关系变得简单和易于管理。



```
1 public class SecurityFacade {
2     private Camera camera1, camera2;
3     private Light light1, light2, light3;
4     private Sensor sensor;
5     private Alarm alarm;
6     public void activate() {
7         camera1.turnOn();
8         camera2.turnOn();
9         light1.turnOn();
10        light2.turnOn();
11        light3.turnOn();
12        sensor.activate();
13        alarm.activate();
14    }
15    public void deactivate() {
16        camera1.turnOff();
17        camera2.turnOff();
18        light1.turnOff();
19        light2.turnOff();
```

```
20     light3.turnOff();
21     sensor.deactivate();
22     alarm.deactivate();
23 }
24 }
```

```
1 public class Client {
2     private static SecurityFacade security;
3     public static void main(String[] args) { security.activate(); }
4 }
```

可以看出, 客户端程序大大简化了, Client 类只有一个对 SecurityFacade 类的引用.

25 桥梁模式 (Bridge)

桥梁模式虽然不是一种使用频率很高的模式, 但是熟悉这个模式对于理解面向对象的设计原则, 包括开闭原则 (OCP) 以及组合聚合复用原则 (CARP) 都很有帮助. 理解好这两个原则, 有助于形成正确的设计思想和培养良好的设计风格.

桥梁模式的目的: 将抽象化 (Abstraction) 与实现化 (Implementation) 脱耦, 使得二者可以独立地变化.

- **抽象化:** 存在于多个实体中的共同的概念, 就是抽象化. 作为一个过程, 抽象化就是忽略一些信息, 从而把不同的实体当做同样的实体对待.
- **实现化:** 对抽象化给出的具体实现, 就是实现化.
- **脱耦:**
 1. 所谓耦合, 就是两个实体的某种强关联. 而将它们的强关联去掉, 就是耦合的解脱, 或称脱耦. 在这里, 脱耦是指将抽象化和实现化之间的耦合解脱开, 或者说是将它们之间的强关联改换成弱关联.
 2. 将两个角色之间的继承关系改为聚合关系, 就是将它们之间的强关联改换成为弱关联.

因此, 桥梁模式中的所谓脱耦, 就是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系, 从而使两者可以相对独立地变化. 这就是桥梁模式的用意.

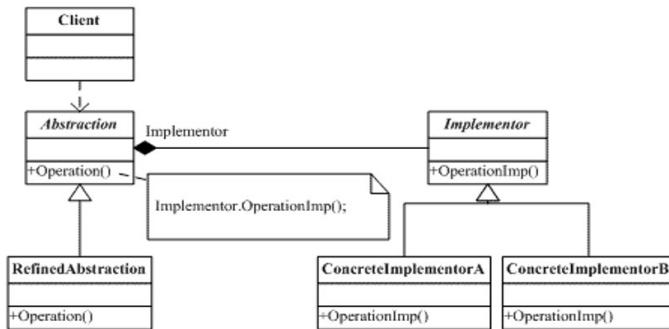
以下的情况下应当使用桥梁模式:

- 如果一个系统需要的抽象化角色和具体化角色之间增加更多的灵活性, 避免在两个层次之间建立静态关系.
- 设计要求实现化角色的任何改变不应当影响客户端, 或者说实现化角色的改变对客户端是完全透明的.
- 一个构件有多于一个的抽象化角色和实现化角色, 系统需要它们之间进行动态耦合.
- 虽然在系统中使用继承是没有问题的, 但是由于抽象化角色和具体化角色需要独立变化, 设计要求需要独立管理这两者.

继承关系的缺点: 继承存在不可避免的缺点, 即灵活性不够。继承是一种强耦合, 它在一开始便把抽象化角色和实现化角色的关系绑定 (binding), 使得两个层次之间互相限制, 无法独立地演化。那么, 能否使用一种弱耦合来实现抽象化角色和实现者层次之间的动态绑定呢? 桥梁模式就提供了这样一种用聚合关系实现的弱耦合解决方案。

25.1 桥梁模式的结构

桥梁模式是对象形式的结构模式, 又称为柄体 (Handle and Body) 模式或接口 (Interface) 模式。下图所示就是一个实现了桥梁模式的示意性系统的结构图。



这个系统含有两个等级结构:

- 由抽象化角色和修正抽象化角色组成的抽象化等级结构。
- 由实现化角色和两个具体实现化角色所组成的实现化等级结构。

桥梁模式涉及的角色:

- 抽象化 (Abstraction) 角色: 抽象化给出的定义, 并保存一个对实现化对象的引用。
- 修正抽象化 (Refined Abstraction) 角色: 扩展抽象化角色, 改变和修正父类对抽象化的定义。
- 实现化 (Implementor) 角色: 这个角色给出实现化角色的接口, 但不给出具体的实现。必须指出的是, 这个接口不一定和抽象化角色的接口定义相同, 实际上, 这两个接口可以完全不同。实现化角色应当只给出底层操作, 而抽象化角色应当只给出基于底层操作的更高一层的操作。
- 具体实现化 (Concrete Implementor) 角色: 这个角色给出实现化角色接口的具体实现。

```

1 abstract public class Abstraction {
2     protected Implementor imp;
3     //某业务逻辑方法
4     public void operation() { imp.operationImp(); }
5 }
  
```

```

1 public class RefinedAbstraction extends Abstraction {
2     //某业务逻辑方法在修正抽象化角色中的实现
  
```

```

3  public void operation() { // improved logic
4      super.operation();
5  }
6 }
```

```

1 abstract public class Implementor {
2     /*某业务逻辑方法的实现化声明*/
3     public abstract void operationImp();
4 }
```

```

1 public class ConcreteImplementorA extends Implementor {
2     public void operationImp() { //某个业务逻辑方法的实现化实现
3         // do something
4     }
5 }
```

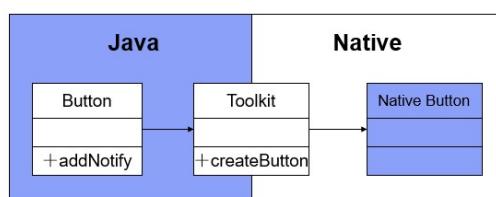
接口的“宽”与“窄”:

- 一般而言, 实现化角色中的每一个方法都应当有一个抽象化角色中的某一个方法与之相对应.
- 但是, 反过来则不一定成立, 即抽象化角色的接口比实现化的接口宽.
- 抽象化角色除了提供与实现化角色相关的方法外, 还有可能提供其他的业务逻辑方法; 而实现化角色则往往仅为实现抽象化角色的相关行为而存在.

25.2 Java 中的 Peer 结构

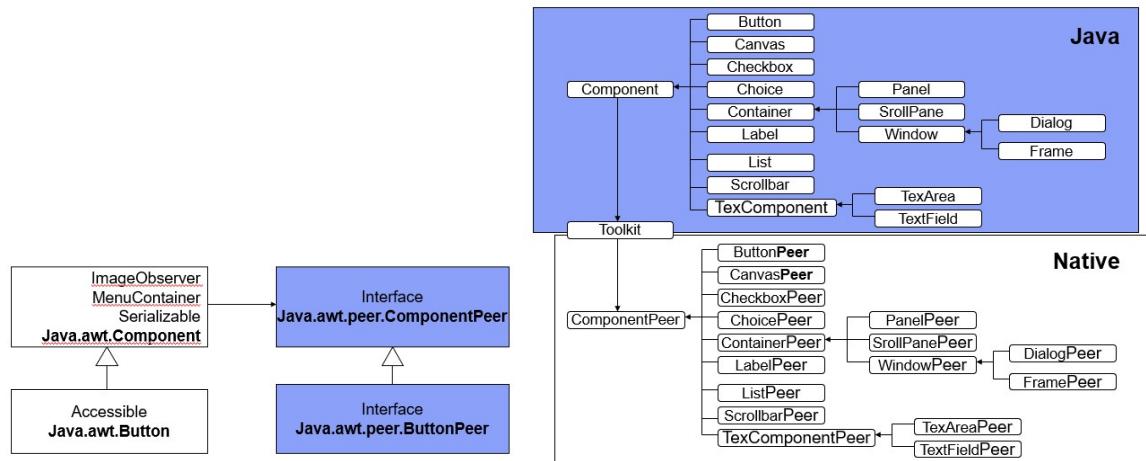
一个 Java 的软件系统总是带有所在的操作系统的视感 (Look and Feel). 如果一个软件系统是在 UNIX 上面开发的, 那么开发人员看到的是 Motif 用户界面的视感; 在 Windows 上面使用这个系统的用户看到的是 Windows 用户界面的视感; 而一个在 Macintosh 上面使用的用户看到的则是 Macintosh 用户界面的视感.

在一个 Java 应用程序和真正的显示界面之间还有几个软件架构层次. 假设现在考察的是一个 Button 构件, 那么在屏幕上看到的是一个本地的按钮, 与所在的操作系统的任何其他按钮并无区别. 因为这个与系统有关的按钮实际上是 Java 的 Button 对象的 Peer 对象所产生的, 这个对象的类型是 `java.awt.peer.Button`, 而不是 Java 应用程序所使用的类型为 `java.awt.Button` 的对象. 这个 Peer 对象会首先截获诸如鼠标单击这样的事件, 然后将相关信息传递给 java 的按钮对象.



这个将 Java 的 GUI 构件与本地环境的 Peer 构件联系起来的接口，就是所谓的 Peer 接口 (ComponentPeer). 一个 Peer 接口其实就是一个定义了 Peer 构件必须实现的各个方法的接口。而 AWT 提供的所有 Peer 接口都放在 java.awt.peer 库中。比如这个库里有一个叫做 ButtonPeer 的接口，这个接口仅含有一个 setLabel() 的方法。

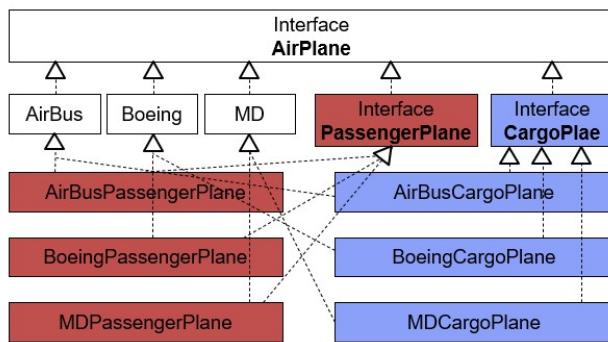
这意味着所对应的本地对象必须实现这个方法，才可以成为 AWT 的 Peer 对象。当然，正如 Button 本身还是 Component 的子类型，因而必须实现 Component 所规定的抽象方法一样，ButtonPeer 本身还是 ComponentPeer 的子类型，所以必须实现 ComponentPeer 接口所规定的各个方法。两种颜色代表两个不同的等级结构。



25.3 例子

问题: 空中客车 (Airbus)、波音 (Boeing) 和麦道 (donnell-Douglas) 都是飞机制造商，它们都生产载客飞机 (Passenger Plane) 和载货飞机 (Cargo Plane). 现在需要设计一个系统，描述这些飞机制造商以及它们所造的飞机种类。

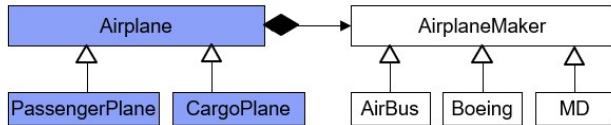
设计方案一: 首先，系统是关于飞机的，因此可以设计一个总的飞机接口，不妨叫做 AirPlane. 其他所有的飞机都是这个总接口的子接口或者具体实现。下图所示的就是这个设计方案的类图，可以看出，这是一个不大高明的设计，导致了一团理不清的关系。



可以看出，在这个设计方案里面，出现了两个子接口，分别表示客机和货机。所有的飞机又要继承自 Airbus,Boeing 和 MD 等超类。这样一来，每一个具体飞机都带有两个超类型：飞机

制造商类型, 客、货机类型.

设计方案二: 使用桥梁模式的关键在于准确地找出这个系统的抽象化角色和具体化角色. 从系统面对的问题不难看出, 代表飞机抽象化的是它的类型, 也就是“客机”或者“货机”; 而代表飞机的实现化的则是飞机制造商. 因此, 可以给出一个更有灵活性的设计, 这个使用了桥梁模式而做的新设计如下图所示.



首先 Airplane 扮演抽象化角色, 它给出所有修正抽象化角色所需的接口, 它的源代码如下所示.

```
1 abstract public class Airplane {  
2     public abstract public void fly() ;  
3     public protected AirplaneMaker airplaneMaker;  
4     public Airplane(AirplaneMaker maker) { airplaneMaker=maker; }  
5 }
```

载客飞机作为飞机种类, 属于修正抽象化角色. 它的示意性源代码如以下所示.

```
1 public class PassengerPlane extends Airplane {  
2     public void fly() {  
3         //Write your code here  
4         airplaneMaker.move();  
5         // ...  
6     }  
7 }
```

载货飞机作为飞机的另一个种类, 也属于修正抽象化角色. 它的示意性源代码如以下所示.

```
1 public class CargoPlane extends Airplane {  
2     public void fly() {  
3         //Write your code here  
4         airplaneMaker.move();  
5         // ...  
6     }  
7 }
```

飞机制造商 AirplaneMaker 扮演实现化角色, 它给出所有的修正抽象化角色所需要实现的接口. 它的源代码如以下所示.

```
1 abstract public class AirplaneMaker { abstract public void move(); }
```

空中客车是飞机制造商之一, 属于具体实现化角色. 这里给出的是它的象征性 Java 源代码, 如以下所示.

```
1 public class Airbus extends AirplaneMaker {  
2     public void move() { /* Write your code here */ }  
3 }
```

波音是另一个飞机制造商, 同样属于具体实现化角色. 这里给出的是它的示意性 Java 源代码, 如以下所示.

```
1 public class Boeing extends AirplaneMaker {  
2     public void move() { /* Write your code here */ }  
3 }
```

同样地, 飞机制造商麦道也属于具体实现角色, 它的示意性 Java 源代码如下所示.

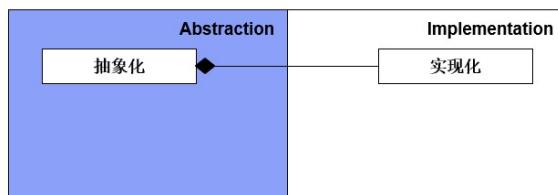
```
1 public class MD extends AirplaneMaker {  
2     public void move() { /* Write your code here */ }  
3 }
```

关于开闭原则 (OCP): 现在, 如果需要增加新的飞机种类, 或者新的飞机制造商的话, 只需要向系统引进一个新的修正抽象化角色, 或者一个新的具体实现化角色就可以了. 或者说, 系统的功能可以在不修改已有代码的情况下得到扩展. 换言之, 这个设计是符合开闭原则的.

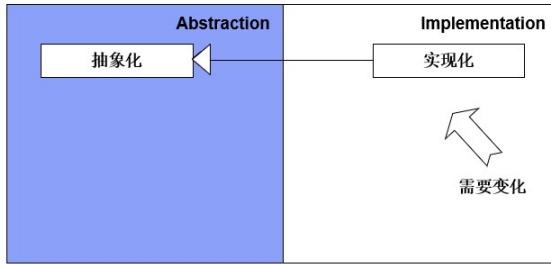
关于组合/聚合复用原则 (CARP) 由于这个新的系统设计适当地使用了继承关系来构建两个等级结构的内部结构, 并且使用聚合关系来构建两个等级结构之间的关系, 因此, 这是一个通过使用“组合/聚合复用原则来达到开闭原则要求的范例.

25.4 抽象与实现

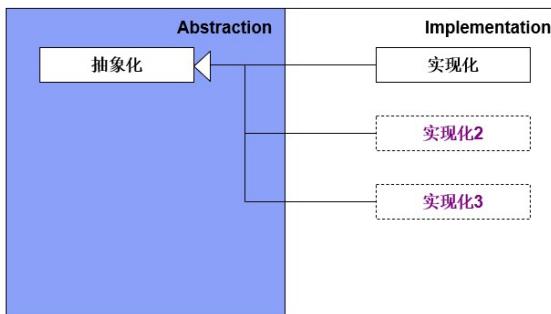
对变化的封装: “找到系统的可变因素, 将之封装起来”, 通常就叫做“对变化的封装”. 这实际上是达到“开 - 闭”原则的捷径, 与组合/聚合复用原则是相辅相成的. 抽象化与实现化的最简单实现, 也就是“开 - 闭”原则在类层次上的最简单实现, 如下图所示.



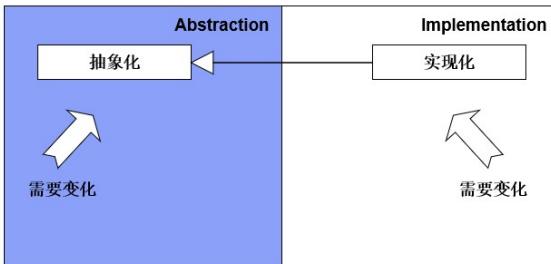
一般来说, 一个继承结构的第一层是抽象角色, 封装了抽象的业务逻辑, 这是系统中不变的部分. 第二层是实现角色, 封装了设计中会变化的因素. 这个实现允许实现化角色有多态性变化, 如下图所示.



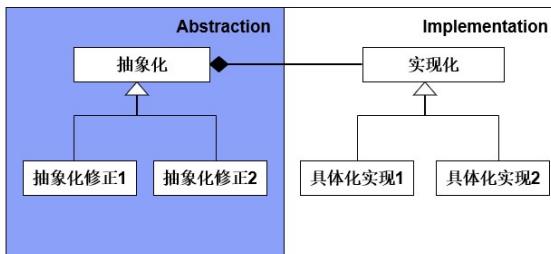
换言之，客户端可以持有抽象化类型的对象，而不在意对象的真实类型是“实现化”、“实现化 2”还是“实现化 3”，如下图所示。



显然，每一个继承关系都封装了一个变化因素，而一个继承关系不应当同时处理两个变化因素。这种简单实现不能够处理抽象化与实现化都面临变化的情况，如下图所示。



上图的两个变化因素应当是彼此独立的，可以在不影响另一者的情况下独立演化。比如，下面的两个等级结构分别封装了自己的变化因素，由于每一个变化都是可以通过静态关系表达的，因此分别使用继承关系实现，如下图所示。



26 策略模式 (Strategy)

策略模式属于对象形式的行为模式. 其用意是针对一组算法, 将每一个算法行为封装到具有共同接口的独立的类中, 从而使得他们可以相互替换. 策略模式使得算法可以在不影响到客户端的情况下发生变化.

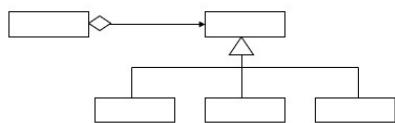
26.1 介绍

问题: 假设现在要设计一个贩卖各类书籍的电子商务网站的购物车 (Shopping Cart) 系统. 在计算价格时, 一个最简单的实现就是把所有的货品的单价乘上数量, 但是实际情况肯定比这要复杂. 比如, 网站可能对所有的教材类图书实行每本一元的折扣; 对其余的图书没有折扣. 对连环画类图书提供每本 7% 的促销折扣, 而对非教材类的计算机图书有 3% 的折扣, 对其余的图书没有折扣. 由于有这样复杂的折扣算法, 使得价格计算问题需要系统地解决. 那么怎么解决这个问题呢?

解决方案: 这里提供三种方案如下所示:

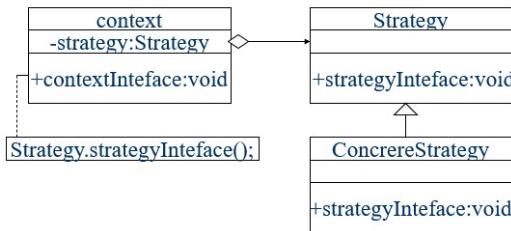
1. 所有的业务逻辑都放在客户端里面. 客户端利用条件选择语句决定使用哪一个算法. 这样一来, 客户端代码会变得复杂和难以维护.
2. 客户端可以利用继承的办法在子类里面实现不同的行为, 但是这样会使得类和行为紧密耦合在一起. 强耦合会使两者不能单独演化.
3. 使用策略模式. 策略模式把行为和其应用环境割开来. 环境类负责维护和查询策略类, 各种算法则在具体策略类 (ConcreteStrategy) 中提供.
 - 将算法和算法的使用环境 (context) 独立开来, 使算法的增减、修改对其使用者透明.
 - 策略模式正是解决这个问题的系统化方法. 当出现新的促销折扣或现有的折扣政策出现变化时, 只需要实现新的策略类, 并被客户端使用即可. 策略模式相当于“可插入式 (Pluggable) 的算法”.
 - 当准备在一个系统里使用策略模式时, 首先必须找到需要包装的算法, 看看算法是否可以从使用环境中分割开来, 最后再考察这些算法是否在以后发生变化.

策略模式的用意是针对一组算法, 将每一个算法封装到具有共同接口的独立的类中, 从而使得它们可以相互替换. 策略模式使得算法可以在不影响到客户端的情况下发生变化. 策略模式的简略类图如下所示.



26.2 策略模式的结构

策略又称做政策 (Policy) 模式. 下面是一个示意性的策略模式结构图:



模式涉及到三个角色：

- 环境（Context）角色：表示策略的使用环境，它持有一个 Strategy 对象的引用.
- 抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或抽象类实现. 此角色给出所有的具体策略类所需的接口.
- 具体策略（ConcreteStrategy）角色：包装了相关的算法或行为.

```

1 public class Context {
2     private Strategy strategy;
3     public void contextInterface() {
4         strategy.strategyInterface();
5     }
6 }
```

```

1 abstract public class Strategy {
2     public abstract void strategyInterface();
3 }
```

```

1 public class ConcreteStrategy extends Strategy {
2     public void strategyInterface() { /*write you algorithm code here*/}
3 }
```

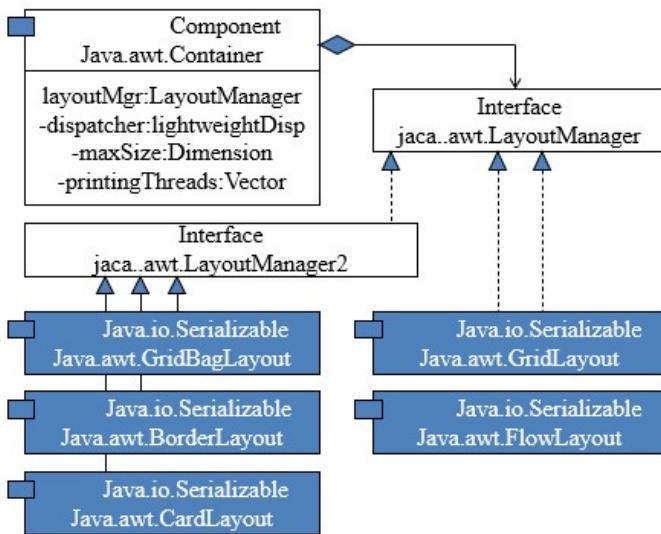
这里所给出的仅仅是策略模式的最简单实现，因此具体策略角色才只有一个。一般而言，有意义的策略模式的应用都会涉及到多个的具体策略角色。我们可以把这里给出的源代码当作策略模式的骨架，在将它使用到自己的系统中时，还需要添加上与系统有关的逻辑。

26.3 例:Java 语言内部的例子

Java 语言使用了很多的设计模式，包括策略模式。使用策略模式的例子可以在 `java.awt` 库和 `Swing` 库中看到。

AWT 中的 LayoutManager: `java.awt` 类库需在运行期间动态地由客户端决定一个 Container 对象怎样排列它所有的 GUI 构件。Java 语言提供了几种不同的排列方式，包装在不同的类里：`BorderLayout`,`FlowLayout`,`GridBagLayout`,`GridLayout`,`CardLayout`.

LayoutManager 的类图如下所示：



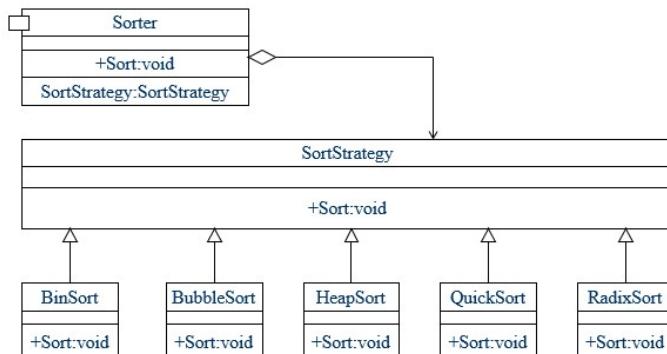
图中没有阴影部分的类是具体策略角色,`java.awt.Container` 类是环境角色, 而 `java.awt.LayoutManager` 则是抽象策略角色. 任何人都可以设计实现自己的 Layout 类. 比如 Borland 公司就在 Jbuilder 里面提供了 XYLayout, 作为对几种 JDK 提供的 Layout 类的补充.

系统涉及到三个角色:

- 环境角色: 在这里由 `Container` 类扮演.
- 抽象策略 (Strategy) 角色: 这是一个抽象角色, 由 `LayoutManager` 类扮演. 此角色给出所有的具体 `Layout` 类所需要的接口.
- 具体策略 (ConcreteStrategy) 角色: 由 `BorderLayout`、`FlowLayout`、`GridLayout`、`GridBagLayout`、`CardLayout` 等扮演 (请见图中没有阴影的类), 他们包装了不同的 `Layout` 行为.

26.4 例: 排序策略系统

假设要设计一个排序系统, 可以动态地使用二元排序、堆栈排序、快速排序、基数排序等排序算法. 显然, 采用策略模式把几种排序算法包装到不同的算法类里面, 让所有的算法类具有相同的接口, 就是一个很好的设计. 排序策略系统的设计如下图所示.



客户端/环境类必须决定在何时使用哪个排序算法, 即这个决定不是在模式内部决定的.

26.5 例: 图书折扣的计算

```
1 abstract public class DiscountStrategy {  
2     private double price = 0;  
3     private int copies = 0;  
4     abstract public double calculateDiscount();  
5     public DiscountStrategy(double price,int copies) {  
6         this.price=price;  
7         this.copies=copies;  
8     }  
9 }
```

```
1 public class NoDiscountStrategy extends DiscountStrategy {  
2     private double price = 0.0;  
3     private int copies = 0;  
4     public NoDiscountStrategy(double price, int copies) {  
5         this.price = price;  
6         this.copies = copies;  
7     }  
8     public double calculateDiscount() { return 0.0; }  
9 }
```

```
1 public class FlatRateStrategy extends DiscountStrategy {  
2     private double amount;  
3     private double price = 0;  
4     private int copies = 0;  
5     public FlatRateStrategy(double price, int copies) {  
6         this.price = price;  
7         this.copies = copies;  
8     }  
9     public double getAmount() { return amount; }  
10    public void setAmount(double amount) { this.amount = amount; }  
11    public double calculateDiscount() { return copies * amount; }  
12 }
```

```
1 public class PercentageStrategy extends DiscountStrategy {  
2     private double percent = 0.0;  
3     private double price = 0.0;
```

```

4  private int copies = 0;
5  public PercentageStrategy(double price, int copies) {
6      this.price = price;
7      this.copies = copies;
8  }
9  public double getPercent() { return percent; }
10 public void setPercent(double percent) { this.percent = percent; }
11 public double calculateDiscount() { return copies * price * percent; }
12 }
```

```

1 class Book {
2     private DiscountStrategy strategy;
3     private float price;
4     void setStrategy(DiscountStrategy st) { this.strategy=st; }
5     // ...
6     float getPrice(int copies) {
7         return price*copies-strategy.calculateDiscount();
8     }
9 }
```

26.6 使用场景

在下面的情况下应当考虑使用策略模式:

1. 如果在一个系统里面有许多类, 它们之间的区别仅在于它们的行为, 那么策略模式可以动态地让一个对象在许多行为中选择一种行为.
2. 一个系统需要动态地在几种算法中选择一种. 那么这些算法可以包装到一个个地具体算法类里面, 而这些具体算法类都是一个抽象算法的子类. 换言之, 这些具体算法均有统一的接口, 由于多态性原则, 客户端可以选择使用任何一个具体算法类, 并只持有一个数据类型是抽象算法类的对象.
3. 一个系统的算法使用的数据不可以让客户端知道. 策略模式可以避免让客户端涉及到不必要接触到的复杂的和只与算法有关的数据.
4. 如果一个对象有很多的行为, 如果不用恰当的模式, 这些行为就只好使用多重的条件选择语句来实现. 此时, 使用策略模式, 把这些行为转移到相应的具体策略类里面, 就可以避免使用难以维护的多重条件选择语句, 并体现面向对象设计的概念.

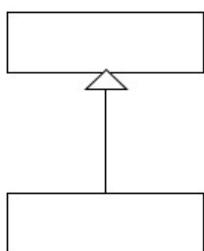
策略模式与其他模式的关系: 策略模式与很多其它的模式都有着广泛的联系.Strategy 很容易和 Bridge 模式相混淆. 虽然它们结构很相似, 但它们却是为解决不同的问题而设计的.Strategy 模式注重于算法的封装, 而 Bridge 模式注重于分离抽象和实现, 为一个抽象体系提供不同的实现.Bridge 模式与 Strategy 模式都很好的体现了”Favor composite over inheritance”的观点.

设计原则的讨论: 策略模式是开闭原则的一个极好的应用范例。在采用策略模式之前，设计师必须从“开-闭原则”出发，考察这个图书销售系统是否有可能在将来引入新的折扣算法。如果有可能，那么就应当将所有的折扣算法封装起来，因为他们是可变化的因素。

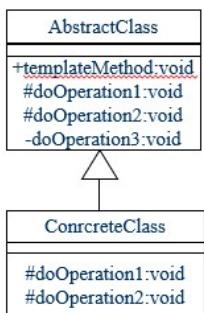
27 模板方法模式 (Template Method)

模板方法模式是最为常见的几个模式之一。模板方法模式是基于继承的代码复用的基本技术，模板方法模式的结构和用法也是面向对象设计的核心。模板方法模式为设计抽象类和实现具体子类的设计师之间的协作提供了可能：一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责实现这个算法的各个逻辑步骤。

模板方法所代表的行为称为顶级行为，其逻辑称为顶级逻辑。模板方法模式的简略类图如下图所示。



模板方法模式的静态结构如下图所示：



```
1 abstract public class AbstractClass {  
2     public void templateMethod() {  
3         doOperation1();  
4         doOperation2();  
5         doOperation3();  
6     }  
7     protected abstract void doOperation1();  
8     protected abstract void doOperation2();  
9     private final void doOperation3() { /*do something*/ }
```

```
10 }
```

```
1 public class ConcreteClass extends AbstractClass {  
2     protected void doOperation1() {  
3         System.out.println("doOperation1();");  
4     }  
5     protected void doOperation2() {  
6         System.out.println("doOperation2();");  
7     }  
8     //The following should not happen:  
9     //doOperation3();  
10 }
```

27.1 好莱坞原则

在好莱坞工作的演艺界人士都了解，在把简历递交给好莱坞的娱乐公司以后，所能做的就是等待。这些公司会告诉他们“不要给我们打电话，我们会给你打”。这便是所谓的“好莱坞原则”。好莱坞原则的关键之处，是娱乐公司对娱乐项目的完全控制。应聘的演员只是被动地服从总项目流程的安排，在需要的时候完成流程中的具体环节。虽然“好莱坞原则”很早的时候在不同的题材里被讨论过，使用这个比喻描写模板方法模式则是由【GoF】给出的，他们认为“好莱坞原则”体现了模板。

模式的关键：子类可以置换掉父类的可变部分，但是子类却不可以改变模板方法所定义的顶层逻辑。每当定义一个新的子类时，不要按照控制流程的思路去设计其职责，而应当按照“责任”的思路去设计。换言之，应当考虑有哪些操作是必须置换掉的，哪些操作是可以置换掉的，以及哪些操作是不可以置换掉的。使用模板方法模式可以使这些责任变的清晰。

27.2 例：计算账户利息

```
1 abstract public class Account {  
2     protected String accountNumber;  
3     public Account() { accountNumber = null; }  
4     public Account(String accountNumber) {  
5         this.accountNumber = accountNumber;  
6     }  
7     final public double calculateInterest() {  
8         double interestRate = doCalculateInterestRate(); //计算利率  
9         String accountType = doCalculateAccountType(); //获得账户类型  
10        double amount = calculateAmount(accountType, accountNumber);  
11        return amount * interestRate;  
12    }
```

```
13 abstract protected String doCalculateAccountType();
14 abstract protected double doCalculateInterestRate();
15 final public double calculateAmount(String accountType,
16     String accountNumber) {
17     //retrieve amount from database...here is only a mock-up
18     return 7243.00D;
19 }
20 }
```

```
1 public class MoneyMarketAccount extends Account {
2     public String doCalculateAccountType() { return "Money Market"; }
3     public double doCalculateInterestRate() { return 0.045D; }
4 }
```

```
1 public class CDAccount extends Account {
2     public String doCalculateAccountType() {
3         return "Certificate of Deposite";
4     }
5     public double doCalculateInterestRate() { return 0.065D; }
6 }
```

```
1 public class Client {
2     private static Account acct = null;
3     public static void main(String[] args) {
4         acct = new MoneyMarketAccount();
5         System.out.println("Interest earned from Money Market account = "
6             + acct.calculateInterest());
7         acct = new CDAccount();
8         System.out.println("Interest earned from CD account = "
9             + acct.calculateInterest());
10    }
11 }
```

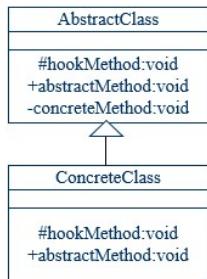
运行结果:

- Interested from Money Market account 325.935
- Interested from CD account 470.795

显然，货币市场帐户和定期存款帐户的不同利息数额，是由于基本方法在不同的具体子类中有不同的实现所造成。

27.3 模板方法模式中的方法

- **模板方法:** 一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。这个模板方法一般会在抽象类中定义，并由子类不加以修改地完全继承下来。一个抽象类可以有任意多个模板方法，而不限于一个。每一个模板方法都可以调用任意多个具体方法。
- **基本方法**有可以分为三种: 抽象方法、具体方法、钩子/回调方法.



27.4 应用: 代码重构

- 模板方法模式可以作为方法层次上的代码重构的一个重要手段。
- 不良的代码设计常常会将过多的代码放在一个方法里面，造成一个具有几千行代码的大方法。这样的方法应当拆分成一些较小的方法，拆分的策略往往可以使用模板方法模式。
- 将大方法打破

假设下面就是这个需要重构的过大的方法:

```
1 public void bigMethod() { //需要重构的源码
2     // ...
3     //代码块1
4     // ...
5     //代码块2
6     // ...
7     //代码块3
8     // ...
9 }
```

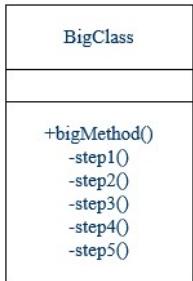
```
1 //初步重构后的源代码
2 public void bigMethod() {
3     step1();
4     step2();
5     if(something) {
6         step3();
7     } else if() {
```

```

8     step4();
9 } else if () {
10    step5();
11 }
12 }
13 private void step1() { /*原来代码块1*/ }
14 // ...

```

划分后的类图如下图所示。



以多态性取代条件转移: 以下代码说明其三个方法作为一个新方法 newMethod () 的多态性体现，划分到三个不同的子类中去。

```

1 //重构的最后结果
2 public abstract class AbstractClass {
3     public void bigMethod() {
4         step1();
5         step2();
6         newMethod();
7     }
8     private final void step1() {/*原来的代码块1*/}
9     private final void step2() {/*原来的代码块2*/}
10    protected abstract void newMethod();
11 }
12
13 public class ConcreteClass extends AbstractClass {
14     public void newMethod() {/*原来的代码块3*/}
15 }
16
17 public class ConcreteClass2 extends AbstractClass {
18     public void newMethod() {/*原来的代码块4*/}
19 }
20

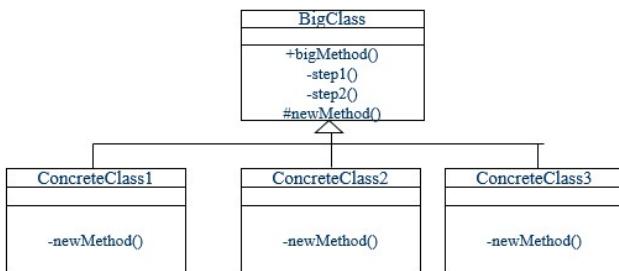
```

```

21 public class ConcreteClass3 extends AbstractClass {
22     public void newMethod() /*原来的代码块5*/
23 }

```

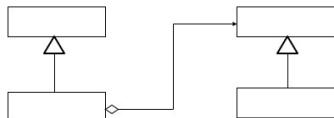
下图所示是代码重构后的最终结果。



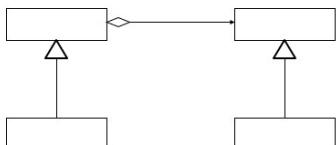
28 观察者模式 (Observer)

观察者模式是对象行为模式，又称为发布-订阅模式、模型-视图 (model-view) 模式、源-监听器模式或者从属者模式。观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

根据观察者对象引用的存储地点，观察者模式的类图有微妙的区别。观察者模式类图（一）如下图所示：



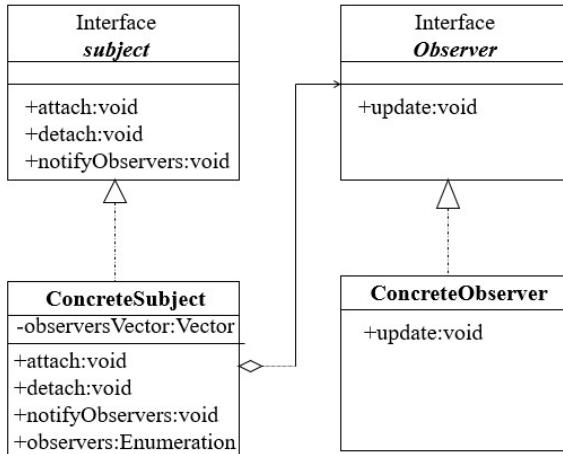
观察者模式的简略类图（二）如下图所示



Java 语言提供的观察者模式的实现属于此种结构。

28.1 观察者模式的结构

以一个简单的示意性实现为例，讨论观察者模式的结构。如下图：



可以看出，在这个观察者模式的实现里有下面这些角色：

- 抽象主题（Subject）角色：主题角色把所有对观察者对象的引用保存在一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，主题角色又叫做抽象被观察者（Observable）角色。
- 具体主题（Concrete Subject）角色：负责管理具体主题的相关状态；在具体主题的状态改变时，给所有的观察者发出通知。具体主题又叫做具体被观察者（Concrete Observable）。具体主题角色负责实现对观察者引用的聚集的管理方法。
- 抽象观察者（Observer）角色：为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。这个接口叫做通知/更新接口。
- 具体观察者（Concrete Observer）角色：实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。

```

1 package com.javapatterns.observer;
2 public interface Subject {
3     //调用这个方法注册|登记一个新的观察者对象
4     public void attach(Observer observer);
5     //调用这个方法删除一个已经登记的观察者对象
6     public void detach(Observer observer);
7     //调用这个方法通知所有登记过的观察者对象
8     void notifyObservers();
9 }

```

```

1 package com.javapatterns.observer;
2 import java.util.Vector;
3 import java.util.Enumeration;
4 public class ConcreteSubject implements Subject {
5     private Vector observersVector = new java.util.Vector();
6     public void attach(Observer observer) { //注册

```

```

7     observersVector.addElement(observer);
8 }
9 public void detach(Observer observer) {
10    observersVector.removeElement(observer);
11 }
12 public void notifyObservers() {
13    java.util.Enumeration enumeration = observers();
14    while (enumeration.hasMoreElements()) {
15        ((Observer)enumeration.nextElement()).update();
16    }
17 }
18 public Enumeration observers() {
19    return ((java.util.Vector) observersVector.clone()).elements();
20 }
21 }
```

```

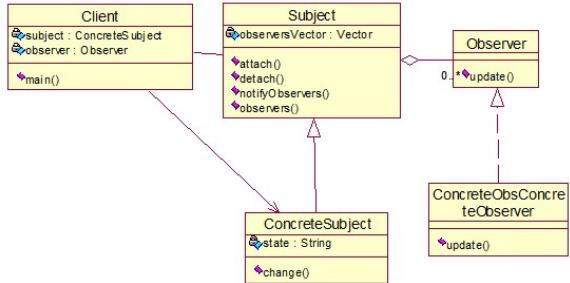
1 package com.javapatterns.observer;
2 public interface Observer {
3     void update(); //调用这个方法会更新自己
4 }
```

```

1 public class ConcreteObserver implements Observer {
2     public void update() { //调用这个方法会更新自己
3         System.out.println("I am notified.");
4     }
5 }
```

28.1.1 另一种方案

- 考察主题对象的功能时，发现它必需使用一个 java 聚集来维护一个所有的观察者对象的引用。前一个方案中，管理这个聚集的是具体主题，因此在类图中存在从抽象观察者角色到具体主题角色的聚合连线。
- 但是所有具体主题管理聚集的方法都相同，因此可以将它移到抽象主题中。
- 第二种实现方案和第一种实现方案的主要区别就是代表存储观察者对象的聚合连线是从抽象主题到抽象观察者。



```

1 import java.util.Vector;
2 import java.util.Enumeration;
3 abstract public class Subject {
4     //这个聚集保存了所有对观察者对象的引用
5     private Vector observersVector = new java.util.Vector();
6     //调用这个方法登记一个新的观察者对象
7     public void attach(Observer observer) {
8         observersVector.addElement(observer);
9         System.out.println("Attached an observer.");
10    }
11    //调用这个方法删除一个已经登记过的观察者对象
12    public void detach(Observer observer) {
13        observersVector.removeElement(observer);
14    }
15    //调用这个方法通知所有登记过的观察者对象
16    public void notifyObservers() {
17        java.util.Enumeration enumeration = observers();
18        while (enumeration.hasMoreElements()) {
19            System.out.println("Before notifying");
20            ((Observer)enumeration.nextElement()).update();
21        }
22    }
23    //这个方法给出观察者聚集的Enumeration对象
24    public Enumeration observers() {
25        return ((java.util.Vector) observersVector.clone()).elements();
26    }
27}

```

```

1 public class ConcreteSubject extends Subject {
2     private String state;
3     //调用这个方法更改主题的状态
4     public void change(String newState) {

```

```

5     state = newState;
6     this.notifyObservers();
7 }
8 }

```

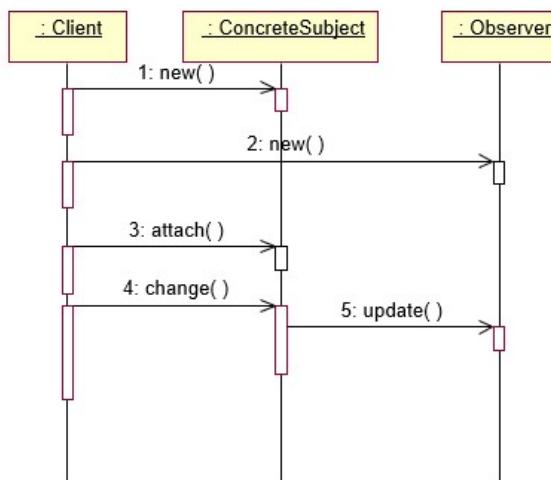
抽象观察者角色的源代码没有变化, 不再列出.

```

1 package com.javapatterns.observer.variation;
2 public class Client {
3     private static Subject subject;
4     private static Observer observer;
5     public static void main(String[] args) {
6         subject = new ConcreteSubject();
7         observer = new ConcreteObserver();
8         subject.attach(observer);
9         subject.change("new state");
10    }
11 }

```

在运行时, 客户端首先创建了具体主题的实例, 以及一个观察者对象. 然后, 它调用主题对象的 attach() 方法, 将这个观察者对象向主题对象登记, 也就是将它加入到主题对象的聚集中去. 这时, 客户端调用主题的业务逻辑方法, 改变了主题对象的内部状态. 主题对象在状态发生变化时, 调用超类的 notifyObservers() 方法, 通知所有登记过的观察者对象. 其时序图如下:



28.2 Java 提供的对观察者模式的支持

在 java 语言的 java.util 库里面, 提供了一个 Observable 类以及一个 Observer 接口, 构成 Java 语言对观察者模式的支持。

- Observer 接口: 这个接口只定义了一个方法, 即 update(Observable o, Object arg) 方法。当被观察者对象的状态发生变化时, 被观察者对象的 notifyObserver() 方法就会调用这一方法。
- Observable 类: 被观察者类都是 java.util.Observable 类的子类。java.util.Observable 提供公开的方法支持观察者对象, 这些方法中有两个对 Observable 子类非常重要:
 1. setChanged():setChanged() 被调用后会设置一个内部标记变量, 代表被观察者对象的状态发生了变化.
 2. notifyObservers(): 这个方法被调用时, 会调用所有登记过的观察者对象的 update() 方法, 使这些观察者可以更新自己.
 3. addObserver(): 将观察者对象加入到聚集中. 当有变化时, 这个聚集可以告诉 notifyObservers() 方法哪些观察者对象需要通知.

```

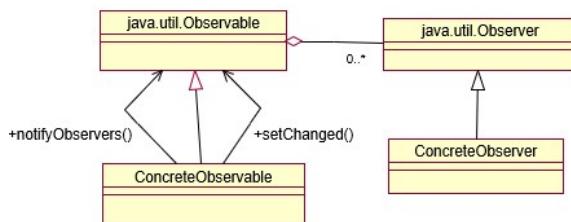
1 package java.util;
2 public class Observable {
3     private boolean changed = false;
4     private Vector obs;
5     //构造子,用零个观察者构造一个被观察者
6     public Observable() { obs = new Vector(); }
7     //将一个观察者加到观察者聚集上面
8     public synchronized void addObserver(Observer o) {
9         if (o == null) throw new NullPointerException();
10        if (!obs.contains(o)) {
11            obs.addElement(o);
12        }
13    }
14    //将一个观察者对象从观察者聚集上删除
15    public synchronized void deleteObserver(Observer o) {
16        obs.removeElement(o);
17    }
18    //相当于notifyObservers(null)
19    public void notifyObservers() { notifyObservers(null); }
20    /**
21     * 若本对象有变化(那时hasChanged方法会返回true)
22     * 调用本方法通知所有登记的观察者,即调用它们的update()方法,
23     * 传入参数this和arg作为参数
24     */
25    public void notifyObservers(Object arg) {
26        //临时存放当前的观察者状态,参见备忘录模式
27        Object[] arrLocal;
28        synchronized (this) {

```

```

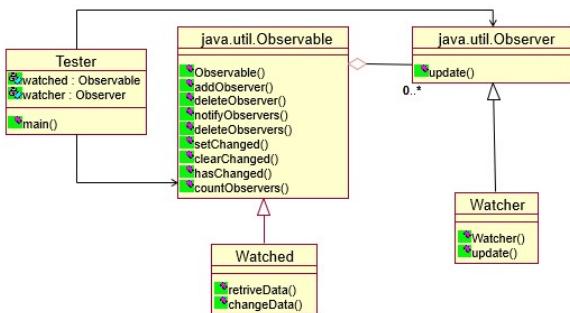
29     if (!changed) return;
30     arrLocal = obs.toArray();
31     clearChanged();
32   }
33   for (int i = arrLocal.length-1; i >= 0; i--)
34     ((Observer)arrLocal[i]).update(this, arg);
35   }
36   //将观察者聚集清空
37   public synchronized void deleteObservers() { obs.removeAllElements(); }
38   //将“已变化”设为true
39   protected synchronized void setChanged() { changed = true; }
40   //将已变化重置为false
41   protected synchronized void clearChanged() { changed = false; }
42   //检测本对象是否已变化
43   public synchronized boolean hasChanged() { return changed; }
44   //返回被观察对象的观察者总数
45   public synchronized int countObservers() { return obs.size(); }
46 }
```

Observable 也被称为主题对象。一个被观察者对象有数个观察者对象，每个观察者对象实现 Observer 接口。被观察者对象发生变化时，会调用 Observable 的 notifyObservers 方法，此方法调用所有的具体观察者的 update() 方法，从而使所有的观察者都被通知更新自己。使用 java 语言提供的观察者模式的支持类图如下：



28.3 怎样使用 Java 对观察者模式的支持

本节给出一个非常简单的例子。在这个例子中，被观察者对象叫做 Watched；而观察者对象叫做 Watcher。Watched 对象继承自 java.util.Observable 类；而 Watcher 对象实现了 java.util.Observer 接口。另外有一个对象 Tester，扮演客户端的角色。这个系统简单的结构图如下：



```

1 package com.javapatterns.observer.watching;
2 import java.util.Observable;
3 public class Watched extends Observable {
4     private String data = "";
5     public String retrieveData() { return data; }
6     public void changeData(String data) {
7         if (!this.data.equals(data)) {
8             this.data = data;
9             setChanged();
10        }
11        notifyObservers();
12    }
13}

```

```

1 package com.javapatterns.observer.watching;
2 import java.util.Observable;
3 import java.util.Observer;
4 public class Watcher implements Observer {
5     public Watcher(Watched w) { w.addObserver(this); }
6     public void update( Observable ob, Object arg) {
7         System.out.println("Data has been changed to: '" +
8             ((Watched)ob).retrieveData() + "'");
9     }
10}

```

```

1 package com.javapatterns.observer.watching;
2 import java.util.Observer;
3 public class Tester {
4     static private Watched watched;
5     static private Observer watcher;
6     public static void main(String[] args) {

```

```

7   watched = new Watched();
8   watcher = new Watcher(watched);
9   watched.changeData("In C, we create bugs.");
10  watched.changeData("In Java, we inherit bugs.");
11  watched.changeData("In Java, we inherit bugs.");
12  watched.changeData("In Visual Basic, we visualize bugs.");
13 }
14 }

```

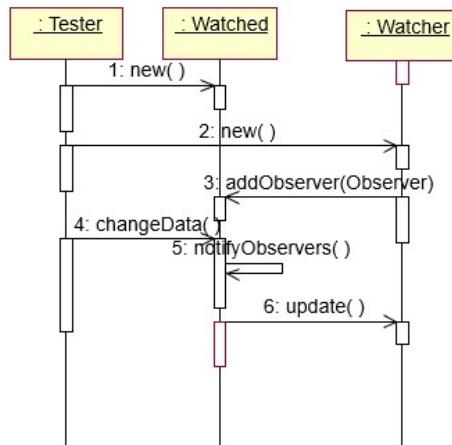
可以看出，虽然客户方将 Watched 对象的内部状态赋值了四次，但是值的改变只有三次，代码清单如下：

- watched.changeData("In C, we create bugs.");
- watched.changeData("In Java, we inherit bugs.");
- watched.changeData("In Java, we inherit bugs.");
- watched.changeData("In Visual Basic, we visualize bugs.");

相应地，Watcher 对象响应了三次改变。运行结果如下：

- Data has been changed to: ‘In C, we create bugs.’
- Data has been changed to: ‘In Java, we inherit bugs.’
- Data has been changed to: ‘In Visual Basic, we visualize bugs.’

系统的活动时序图如下：



Tester 对象首先创建了 Watched 和 Watcher 对象。在创建 Watcher 对象时，将 Watched 对象作为参数传入；然后 Tester 对象调用 Watched 对象的 changeData() 方法，触发 Watched 对象的内部状态变化；Watched 对象进而通知实现登记过的 Watcher 对象，也就是调用的 update() 方法。

28.4 Java 中的 DEM 事件机制

AWT 中的 DEM 机制: 在 AWT1.1 版本以及以后的各个版本中，事件处理模型均为基于观察者模式的委派事件模型 (Delegation Event Model 或 DEM)。在 DEM 中，主题角色负责发布事件，观察者角色向特定的主题订阅它所感兴趣的事件。当一个具体主题产生一个事件时，它就会通知所有感兴趣的订阅者。在 DEM 模型中发布者叫做事件源，而订阅者叫做事件监听器。

```
1 new java.awt.Button(new ActionListener(){  
2     void actionPerformed(){  
3         // ...  
4     }  
5 });
```

28.5 观察者模式的优缺点

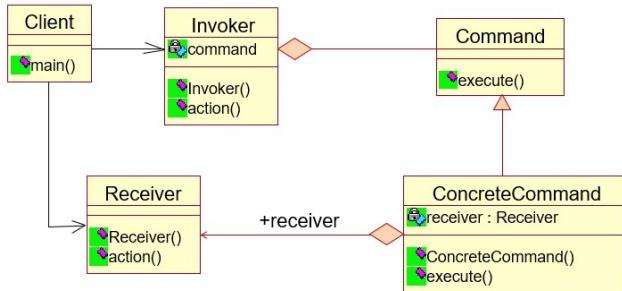
- 优点：观察者模式在被观察者和观察者之间建立一个抽象的耦合。观察者模式支持广播通信。
- 缺点：如果一个被观察者有很多的观察者，将所有的观察者都通知到会花费很多时间。如果在被观察者之间有循环依赖，被观察者会触发它们之间进行循环调用，导致系统崩溃。虽然观察者随时知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎样发生变化的。

29 命令模式 (Command)

命令 (Command) 模式属于对象的行为模式。命令模式又称为行动 (Action) 模式或事务 (Transaction) 模式。命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行，何时被执行，以及是怎么被执行的。

29.1 命令模式的结构

命令模式的类图: 下面以一个示意性的系统为例，说明命令模式的结构。



命令模式涉及到 5 个角色：

- 客户端 (Client) 角色：创建了一个具体命令 (ConcreteCommand) 对象并确定其接收者。
- 命令角色 (Command)：声明了一个给所有具体命令类的抽象接口。
- 具体命令(ConcreteCommand)角色：定义一个接收者和行为之间的弱偶合；实现 execute() 方法，负责调用接收者的相应操作。execute() 方法通常叫做执行方法。
- 请求者 (Invoker) 角色：负责发出一个请求。
- 接收者 (Receiver) 角色：任何一个类都可以成为接收者；实施和执行请求的方法叫做行动方法。

```

1 public class Invoker {
2     private Command command;
3     public Invoker(Command command) { this.command=command; }
4     public void action() { /* 行动方法 */
5         command.execute();
6     }
7 }

```

Receiver 类是命令的接收者，在命令的控制下执行行动方法。这里仅给出一个 action() 方法的示意性实现。代码如下：

```

1 public class Receiver {
2     public Receiver() { }
3     public void action() {
4         System.out.println("Action has been taken.");
5     }
6 }

```

```

1 public interface Command { void execute(); }

```

```

1 public class ConcreteCommand implements Command {
2     private Receiver receiver;
3     public ConcreteCommand(Receiver receiver) {
4         this.receiver=receiver;

```

```

5 }
6     public void execute() { receiver.action(); }
7 }

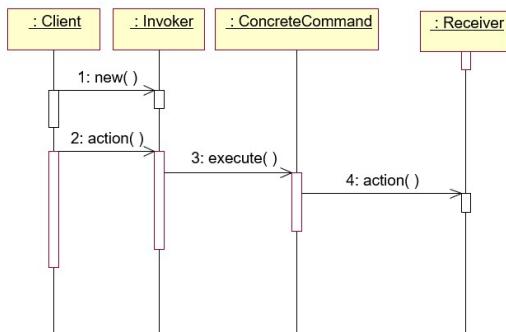
```

```

1 public class Client {
2     public static void main(String[] args) {
3         Receiver receiver=new Receiver();
4         Command command=new ConcreteCommand(receiver);
5         Invoker invoker=new Invoker(command);
6         invoker.action();
7     }
8 }

```

命令模式的活动序列:



29.2 命令模式的优劣

- 优点:
 1. 命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。
 2. 命令类与其他任何别的类一样，可以修改和扩展。
 3. 可以把命令对象聚合在一起，合成为合成命令。比如上面的例子所讨论的宏命令便是合成命令的例子。合成命令是合成模式的应用。
 4. 由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。
- 缺点：使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

29.3 例:AudioPlayer 系统

系统的描述: User 有一个录音机，该录音机有 play、rewind 和 stop 功能，录音机的键盘是请求者角色 (Invoker)，User 是客户端类，录音机是接收者角色。Command 类是抽象命令角色，PlayCommand、StopCommand 和 RewindCommand 是具体命令类。User 不需要知道 play、

rewind 和 stop 功能是怎么执行的，这些命令执行的细节由键盘 Keypad 具体实施。User 只需要在键盘上按下相应的键就行了。

系统的角色：

- 客户端（Client）角色：User
- 请求者（Invoker）角色：Keypad
- 抽象命令（Command）角色：Command
- 具体命令角色：PlayCommand、RewindCommand、StopCommand
- 接收者角色：录音机 AudioPlayer

请求者 Invoker 角色 Keypad 源代码：

```
1 public class Keypad {  
2     private Command playCmd;  
3     private Command rewindCmd;  
4     private Command stopCmd;  
5     public Keypad(Command play, Command stop, Command rewind) {  
6         playCmd=play;  
7         stopCmd=stop;  
8         rewindCmd=rewind;  
9     }  
10    public void play() { // 具体action方法  
11        playCmd.execute();  
12    }  
13    public void stop() { stopCmd. execute(); }  
14    public void rewind() { rewindCmd. execute(); }  
15 }
```

抽象命令角色 Command 源代码：

```
1 public interface Command { public abstract void execute(); }
```

具体命令类 PlayCommand 源代码：

```
1 public class PlayCommand implements Command {  
2     private AudioPlayer myAudio;  
3     public PlayCommand(AudioPlayer a) { myAudio=a; }  
4     public void execute() { myAudio.play(); }  
5 }
```

具体命令类 StopCommand 源代码：

```
1 public class StopCommand implements Command {  
2     private AudioPlayer myAudio;  
3     public StopCommand(AudioPlayer a) { myAudio=a; }
```

```
4 public void execute() { myAudio.stop(); }
5 }
```

具体命令类 RewindCommand 源代码:

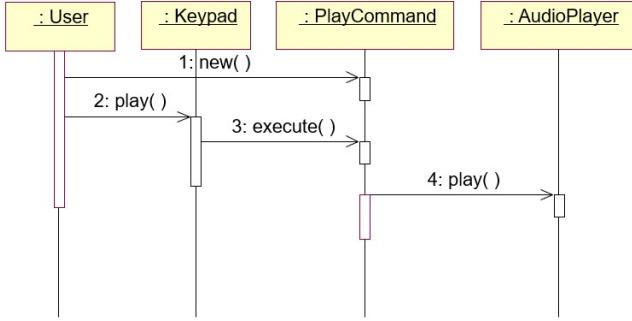
```
1 public class RewindCommand implements Command {
2     private AudioPlayer myAudio;
3     public RewindCommand(AudioPlayer a) { myAudio=a; }
4     public void execute() { myAudio.rewind(); }
5 }
```

接收者类 AudioPlayer 源代码:

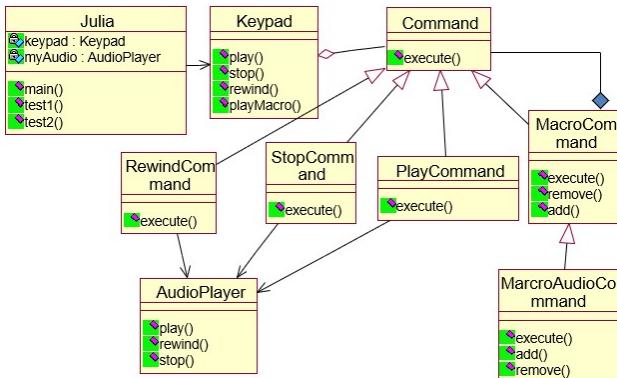
```
1 public class AudioPlayer {
2     public void play() { System.out.println("Playing"); }
3     public void stop() { System.out.println("Stopped"); }
4     public void rewind() { System.out.println("Rewinding"); }
5 }
```

```
1 public class User {
2     private static Keypad keypad;
3     private static AudioPlayer myAudio = new AudioPlayer();
4     public static void main(String[] args) { test1(); }
5     private static void test1() {
6         Command play=new PlayCommand(myAudio);
7         Command stop=new StopCommand(myAudio);
8         Command rewind=new RewindCommand(myAudio);
9         keypad=new Keypad(play,stop,rewind);
10        keypad.play();
11        keypad.stop();
12        keypad.rewind();
13        keypad.stop();
14        keypad.play();
15        keypad.stop();
16    }
17 }
```

系统的时序图如下:



增加宏命令功能: 设想 User 的录音机有一个记录功能, 可以把每一次执行的命令记录下来, 再在任何需要的时候重新把这些记录下来的命令一次性执行, 这就是所谓的宏命令集功能。因此, User 的录音机系统现在有四个键, 分别为播音, 倒带, 停止和宏命令功能。此时系统的设计与前面的设计相比有所增强, 主要体现在 User 类现在有了一个新的方法, 用以操作宏命令键。带有宏命令集功能的录音机模拟系统的设计图如下:



```

1 public interface MacroCommand extends Command {
2     void execute(); /* 执行方法 */
3     /* 宏命令聚集的管理方法, 可以删除一个成员 */
4     void remove(Command toRemove);
5     /* 宏命令聚集的管理方法, 可以添加一个成员 */
6     void add(Command toAdd);
7 }

```

具体宏命令 MacroAudioCommand 类负责把个别的命令合成宏, 代码如下:

```

1 import java.util.Vector;
2 public class MacroAudioCommand implements MacroCommand {
3     private Vector commandList=new Vector();
4     public void add(Command toAdd) { commandList.addElement(toAdd); }
5     public void remove(Command toRemove) {
6         commandList.removeElement(toRemove);

```

```

7   }
8   public void execute() {
9     Command nextCommand;
10    for (int i=0; i<commandList.size(); i++) {
11      nextCommand = (Command)commandList.elementAt(i);
12      nextCommand.execute();
13    }
14  }
15 }
```

Julia 类的方法 test2() 源代码如下：

```

1 public static void test2() {
2   Command play=new PlayCommand(myAudio);
3   Command stop=new PlayCommand(myAudio);
4   Command rewind=new PlayCommand(myAudio);
5   MacroCommand macro=new MacroCommand();
6   macro.add(play);
7   macro.add(stop);
8   macro.add(rewind);
9   macro.add(stop);
10  macro.add(play);
11  macro.add(stop);
12  KeyPad keypad=new KeyPad(play,stop,rewind,macro);
13  keypad.playMacro();
14 }
```

30 状态模式 (State)

状态模式，又称状态对象模式。状态模式是对象的行为模式 [GOF95] 状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它所属的类一样。

很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的对象。这样的对象状态通常是从一系列预定义的值中取出的。

```

1 class Elevator {
2   int level; // -2 表示最底层
3   int state; // 1 表示正常, 2 表示故障, 3 表示检修
4   void onDownButtonPressed() {
5     if(level== -2 || state!=1) {
6       // do nothing
7     } else {
```

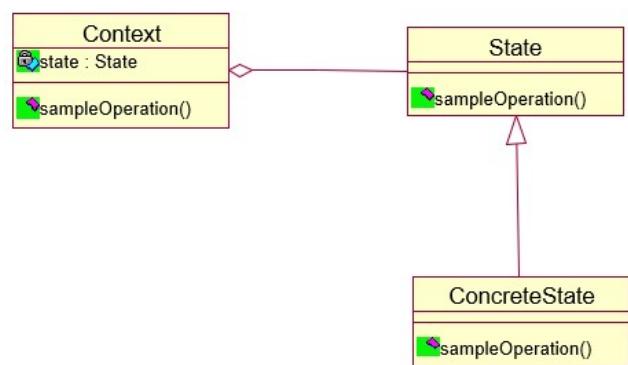
```

8     //电梯向下
9 }
10 }
11 }

```

30.1 状态模式的结构

状态模式把对象的行为包装在不同的状态对象里，状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式的示意性类图如下：



模式所涉及的角色有：

- 抽象状态角色：定义一个接口，用以封装环境（Context）对象的一个特定的状态所对应的行为。
- 具体状态角色：每一个具体状态类都实现了环境（Context）的一个状态所对应的行为。
- 环境角色：定义行为将随状态变化的对象，并持有一个具体状态对象的引用。

下面是环境角色 Context 的源代码，可以看出，环境类持有一个 State 对象，并把所有的行为委派给此对象。

```

1 public class Context {
2     private State state;
3     public void sampleOperation() { state.sampleOperation(); }
4     public void setState(State state) { this.state=state; }
5 }

```

```

1 public interface State { void sampleOperation(); }

```

```

1 public class ConcreteState implements State {
2     public void sampleOperation() { }
3 }

```

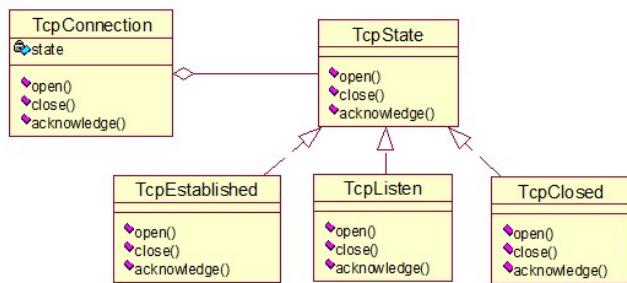
30.2 使用场景

在以下各种情况下可以使用状态模式：

1. 一个对象的行为依赖于它所处的状态，对象的行为必须随着其状态的改变而改变。
2. 对象在某个方法里依赖于一重或多重的条件转移语句，其中有大量的代码。状态模式把条件转移语句的每一个分支都包装到一个单独的类里。这使得这些条件转移分支能够以类的方式独立存在和演化。

30.3 例:TCP

考虑由 TcpConnection 代表一个 TCP/IP 网络的连接。一个 TcpConnection 对象可以具有以下状态之一：Listening（监听）、Established（已建立连接）、Closed（关闭）。当 TcpConnection 对象接到其它对象的请求时，会根据其状态不同而给出不同的回应。TCP 系统的 UML 类图如下所示：



在此，使用状态模式的关键是引进了抽象类或接口 **TcpState** 来代表网络的连接状态。它的子类实现了由状态决定的行为。**TcpConnection** 类持有一个状态对象，**TcpState** 子类的实例，来表示 TCP 连接的现在状态。**TcpConnection** 类把所有与状态有关的请求都委派给它的状态对象。**TcpConnection** 使用它的 **TcpState** 的子类实例来执行特定的连接状态所对应的操作。当连接的状态改变时，**TcpConnection** 对象就改变它所用的状态对象。例如，当网络连接从“已建立”改为“关闭”时，**TcpConnection** 对象会把它的状态对象从 **TcpEstablished** 的实例改为 **TcpClosed** 的实例。

系统有如下的角色：

- 环境角色：**TcpConnection**。此类定义了客户端感兴趣的接口，并持有有关具体状态类的实例，以定义它当前所处的状态。
- 状态角色：**TcpState**。此类定义了把依赖于一个特定的状态所对应的行为包装起来所需要的接口。
- 具体状态角色：**TcpEstablished**、**TcpListen**、**TcpClosed**。实现了环境类的状态所对应的行为。

```
1 public class TcpConnection {
2     private TcpState state;
3     public void open() { state.open(); }
```

```

4   public void setState(TcpState state) { this.state=state; }
5   public void close() { state.close(); }
6   public void acknowledge() { state.acknowledge(); }
7 }
```

```

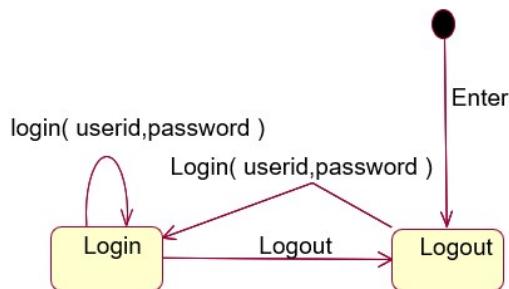
1 public interface TcpState {
2     void open();
3     void close();
4     void acknowledge();
5 }
```

```

1 public class TcpEstablished implements TcpState {
2     public void open() { }
3     public void close() { }
4     public void acknowledge() { }
5 }
```

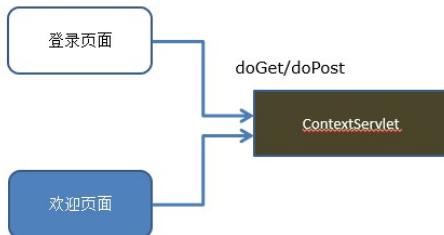
30.4 例：用户登录子系统

用户进入系统时首先遇到登录页面，输入用户名和密码，单击〔log on〕键，系统会向数据库查询。登录成功后，就会见到欢迎页面。登录失败，仍然要面对登录页面。下图描述了用户的状态变化情况。

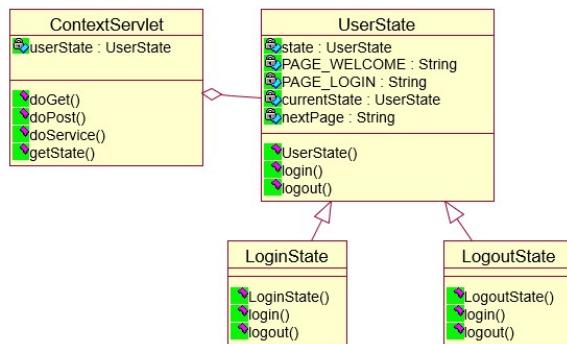


从上面的状态图可以看出，用户进入系统时，处于 Logout 状态，用户输入正确的用户名和密码可以过渡到 Login 状态。当用户处于 Login 状态时，可以发出 Logout 命令，使系统过渡到 Logout 状态。用户在 Login 状态时，重新发出 Login 的命令不会改变状态。

不同的状态会产生不同的网页，对应于 Login 状态的是欢迎页面，对应于 Logout 状态的是登录页面。登录页面提交后的 Servlet 是 ContextServlet。这个 servlet 在 doGet 或 doPost 方法中接收用户名和密码，并且创立对应的状态对象。



登录子系统的静态结构图如下：



登录页面的源代码如下：

```

1 <HTML>
2   <BODY>
3     <FORM action= “/Servlet/com.javapatterns.state.login.ContextServlet”
4       method=get>
5     User ID<INPUT TYPE=text name=userid value= “jeff”><br>
6     Password < INPUT TYPE=password name=password value= “pass”><br>
7       <input type=submit name= “btnAction” value= “Log On”>
8     </FORM>
9   </BODY>
10 </HTML>

```

登录后的页面（欢迎页面）。其源代码如下：

```

1 <HTML>
2   <BODY>
3     <FORM action= “/Servlet/com.javapatterns.state.login.ContextServlet”
4       method=get>
5     Welcome to website!<br>
6       <input type=submit name= “btnAction” value= “Log Out”>
7     </form>
8   </body>
9 </html>

```

环境类 ContextServlet 的源代码如下：

```
1 package com.javapatterns.state.login;
2 import java.io.IOException;
3 import javax.servlet.ServletException;
4 import javax.servlet.ServletRequest;
5 import javax.servlet.http.*;
6 public class ContextServlet extends HttpServlet {
7     private UserState userState = new LogoutState();
8     public void doGet(HttpServletRequest request,
9             HttpServletResponse response)
10    throws ServletException, IOException {
11        doService(request, response);
12    }
13    public void doPost(HttpServletRequest request,
14            HttpServletResponse response)
15    throws ServletException, IOException {
16        doService(request, response);
17    }
18    public void doService(HttpServletRequest request,
19            HttpServletResponse response)
20    throws ServletException, IOException {
21        String userId=request.getParameter("userid");
22        String password=request.getParameter("password");
23        String btnAction=request.getParameter("btnAction");
24        if (btnAction.equalsIgnoreCase("log on")) {
25            this.getState().login(userId,password);
26        } else if (btnAction.equalsIgnoreCase("log out")) {
27            this.getState().logout();
28        }
29        response.sendRedirect(this.getCurrentState().getNextPage());
30    }
31    /*状态的取值方法*/
32    public UserState getState() { return userState.getCurrentState(); }
33 }
```

环境类首先判断客户端的请求是登录还是退出。当接到登录请求时，环境类调用状态类的 login() 方法。当接到退出请求时，环境类调用状态类的 logout() 方法。

抽象类 UserState 代码清单如下：

```
1 public abstract class UserState {
```

```

2  private UserState state;
3  private String nextPage;
4  protected static String PAGE_WELCOME=
5      "/javapatterns/state/welcome.html";
6  protected static String PAGE_LOGIN=
7      "/javapatterns/state/login.html";
8  public UserState() { this.nextPage= PAGE_LOGIN; }
9  /*行为方法*/
10 public abstract boolean login(String userId,String password);
11 public abstract void logout();
12 /*设置状态*/
13 public void setCurrentState(UserState state) { this.state=state; }
14 /*获取当前状态*/
15 public UserState getCurrentState() {
16     if (this.state==null) {
17         this.state=new LogoutState();
18     }
19     return this.state;
20 }
21 public String getNextPage() { return nextPage; }
22 public void setNextPage(String nextPage) { this.nextPage=nextPage; }
23 }

```

子系统的具体类 LoginState 包装了系统在登录成功后的 Login 状态。其源代码如下：

```

1 public class LoginState extends UserState {
2     public LoginState() { }
3     public boolean login(String userId, String password) {
4         setPage(UserState.PAGE_WELCOME);
5         setCurrentState(new LoginState());
6         return true;
7     }
8     public void logout() {
9         setPage(UserState.PAGE_LOGIN);
10        setCurrentState(new LogoutState());
11    }
12 }

```

子系统的具体类 LogoutState 包装了系统在登录失败后的 Logout 状态。其源代码如下：

```

1 public class LogoutState extends UserState {
2     public LogoutState() { }

```

```
3  public boolean login(String userId, String password) {
4      StringBuffer sql=new StringBuffer(50);
5      sql.append("select count(*) from user_info where user_id='");
6      sql.append(userId);
7      sql.append("' and password='");
8      sql.append(password);
9      sql.append("'");
10     int counting=DBManager.query(sql.toString());
11     if (counting > 0) {
12         this.setNextPage(UserState.PAGE_WELCOME);
13         this.setCurrentState(new LoginState());
14         return true;
15     } else {
16         this.setCurrentState(new LogoutState());
17         setNextPage(UserState.PAGE_LOGIN);
18         return false;
19     }
20 }
21 public void logout() {
22     this.setCurrentState(new LogoutState());
23     setNextPage(UserState.PAGE_LOGIN);
24 }
25 }
```