面向对象开发技术

山东大学计算机学院、软件学院

崔 立 真

目 录

第一篇 面向对象原理

第一章 面向对象思想

面向对象思想我们现在所说的面向对象编程思想虽然早在 20 世纪 60 年代就产生了，但是直到 20 世纪 80 年代，面向对象语言才真正引起计算领域的普遍关注。

1.1 什么是面向对象

面向对象的方法是一种分析方法、设计方法和思维方法。面向对象方法学的出发点和所追求的基本目标是使人们分析、设计与实现一个系统的方法尽可能接近人们认识一个系统的方法。使描述问题的问题空间和解决问题的方法空间在结构上尽可能一致。对问题空间进行自然分割，以更接近人类思维的方式建立问题域模型，以便对客观实体进行结构模拟和行为模拟，从而使设计出的软件尽可能直接地描述现实世界。构造出模块化的、可重用的、维护性好的软件，同时限制软件的复杂性和降低开发维护费用。

从程序设计方法的角度看，面向对象是一种新的程序设计范型(paradigm)，其基本思想是使用对象、类、继承、封装、聚合、关联、消息、多态性等基本概念来进行程序设计。

自八十年代以来，面向对象方法已深入到计算机软件领域的几乎所有分支。它不仅是一些具体的软件开发技术与策略，而且是一整套关于如何看待软件系统与现实世界的关系，用什么观点来研究问题并进行问题求解，以及如何进行系统构造的软件方法学。从这个意义上讲：面向对象方法是一种运用对象、类、继承、封装、聚合、关联、消息、多态性等概念来构造系统的软件开发方法。从现实世界中客观存在的事物出发来建立软件系统。强调直接以问题域（现实世界）中的事物为中心来思考问题、认识问

题，并根据这些事物的本质特征，把它们抽象地表示为系统中的对象，作为系统的基本构成单位。这可以使系统直接映射问题域，保持问题域中事物及其相互关系的本来面貌。

充分运用人类日常的思维方法。强调运用人类在日常的逻辑思维中经常采用的思想方法与原则，例如抽象、分类、继承、聚合、封装、关联等等。这使得软件开发者能更有效地思考问题，并以其他人也能看得懂的方式把自己的认识表达出来。

面向对象程序设计的主要特点：

- 从问题域中客观存在的事物出发来构造软件系统，用对象作为对这些事物的抽象表示，并作为系统的基本构成单位。（对象）
- 用对象的属性表示事物的静态特征；用对象的服务（操作）表示事物的动态特征。（属性与服务）
- 对象的属性与服务结合为一体，成为一个独立的、不可分的实体，对外屏蔽其内部细节。（封装）
- 对事物进行分类。把具有相同属性和相同服务的对象归为一类，类是这些对象的抽象描述，每个对象是它的类的一个实例。（分类）
- 通过在不同程度上运用抽象的原则可以得到较一般的类和较特殊的类。特殊类继承一般类的属性与服务，从而简化系统的构造过程及其文档。（继承）
- 复杂的对象可以用简单的对象作为其构成部分。（聚合）
- 对象之间通过消息进行通讯，以实现对象之间的动态联系。（消息）
- 通过关联表达对象之间的静态关系。（关联）

总的来说，面向对象程序设计就是用类和对象作为系统的基本构成单位。对象对应问题域中的事物，其属性与服务刻画了事物的静态特征和动态特征，它们之间的继承关系、聚合关系、消息和关联如实地表达了问题

域中事物之间实际存在的各种关系。因此，无论系统的构成成分，还是通过这些成分之间的关系而体现的系统结构，都可直接地映射问题域。

1.2 为什么 OOP 这么流行

为什么面向对象编程能够成为近 20 年来主要的编程方法呢？其中有几点重要原因：面向对象编程的适用范围非常广，从最细微的问题到最复杂的项目，都可以通过它来解决；面向对象编程为技术人员提供了一种行之有效的解决问题的抽象方法；同时大多数主流面向对象编程言都提供了大量的且越来越多的类库来支持各个领域应用程序的开发。

面向对象技术确实有利于构建复杂的软件体系，然而重要的是，不能认为 OOP 是万能的。计算机编程仍然是人们所必须担负的一项非常困难的任务。优秀的程序员需要天赋、创新、勤奋、逻辑思维、构建和提取抽象的能力以及经验——即使有最好的编程工具。

面向对象编程是一种新的思考问题的方法，它着重于面向对象对于计算的含义，以及如何构建信息才能把我们的意图与其他人和机器进行顺利的交流。要想成为优秀的面向对象技术人才，需要对传统软件开发方法进行彻底的重新认识。

1.3 语言和思维

我们所使用的语言直接影响我们观察世界的方式。这不仅适用于自然语言，而且也适用于人造语言，比如那些在计算机编程中使用的语言。

充分有效地利用面对象原则要求人们以一种新的方式来观察世界。简单地应用一种面向对象语言（比如 Java 语言或者 C++语言）本身并不能使我们成为一名真正的面向对象程序员。

1.3.1 爱斯基摩人和雪

在爱斯基摩人（因纽特人）的语言里，用不同的单词来表达雪的不同形式——湿的、绒毛似的、厚的、冰冷的等等。我们也能够区别不同类型的雪。但是我们所说的语言（英语），并不能迫使我们这样做，并且对我们来说，这样做也是不自然的。因此，不同的语言（比如 Inuktitut 言）能够引导人们（而不是要求人们）去用不同的方式观察世界。

1.3.2 关于计算机语言的一个例子

一个程序员选择哪种程序语言来考虑待解决的问题，会影响或改变算法的设计。

下面是一个证明计算机语言和问题解决方案之间关系的例子。七年前，一名进行基因研究的学生承担了一项分析 DNA 以用一个由 N 个整数值组成的矢量来表示，其中，N 非常大（大约好几万）。任务是判断 DNA 序列中是否包含重复的长度为 M 的序列片断，其中，M 为固定的小数值常量（比如 5 或者 10）。

当试运行程序时，令人非常失望的是，程序要花数小时才能完成。于是他和另外一名同学讨论这个问题，恰好这名同学精通 APL 编程语言。她建议用 APL 编写程序来解决这个问题。前面的这位同学表示怀疑。毕竟，FORTRAN 被公认为是一种非常“高效”的编程语言。它是编译执行的，而 APL 仅仅是解释执行的。尽管持有一定的怀疑态度，但他还是采纳了这个建议，结果发现用 APL 写出的算法只需几分钟而不是几小时就可以解决这个问题。用 APL 编程所做的事情就是需要重新考虑这个问题。把数据重新安排在一个 N 行 M 列的矩阵中，而不是前面的由 N 个元素组成的矢量。把矩阵按行排序（即把一行看作一个单元，在排序过程中移动整行）。如果

有任何序列片断被重复，那么，在排序后的矩阵中相邻的两行就会有相同的数值。检查如何满足这项条件很容易。APL 程序之所以快与 APL 语言本身无关，APL 程序之所以比 FORTRAN 程序快，是由于算法的不同；简单地讲，FORTRAN 程序使用了一个 0（MxN2）的算法，而 APL 程序使用的排序方案需要大约 0(M x logN)次操作。

这个故事并不是指 APL 语言在任何情况下，都比 FORTRAN 语言更加有效，而是指 APL 程序员自然而然地被引导发现一种完全不同的解决方式。原因是使用 APL 语言很难写循环，而排序却很简单——作为语言组成部分，排序被定义为内内部操作符。这样，由于排序运算很容易表达，优秀的 APL 程序员倾向于寻找关于它的创新性硅应用。这就是解决问题的编程语言是如何引导程序员采用不同的思维方式来看待问题。

1.4 一种观察世界的方式

假设有一位名叫 Chris 的人想送花给他的一位名叫 Robin 的朋友，Robin 生活在另外一个城市。因为距离太远，Chris 不能亲自把花送给他的朋友。不过，这却是一件非常容易解决的事情。Chris 来到附近的一家花店，这家花店由一位名叫 Fred 的花商经营。Chris 把打算送给 Robin 的花的种类和 Robin 的地址告诉给 Fred。Chris 确信，这样他的花就可以方便、自动地送到朋友那里了。

1.4.1 代理和团体

解决问题的方法就是找到一个合适的代理，并且把你的要求告诉他。代理有责任完成你的要求。Fred 会选择某种方式——一种算法或者一系列操作——来完成该项任务。Cluis 没有必要了解 Fred 使用什么方法来完成

任务，实际上，使用代理的人通常不想了解完成的任务细节。这些细节通常是隐蔽的。

一个面向对象程序可以组织一个团体，这个团体由一组互相作用的叫做“对象”的代理组成。每一个对象都扮演一个角色，并且为团体中的其他成员提供特定的服务或者执行特定的行为。

1.4.2 消息和方法

在面向对象编程中，行为的启动是通过将“消息”传递给对此行为负责的代理（对象）来完成的。消息对行为的要求进行编码，并且伴随着执行要求所需的附加信息（参数）来一起传递。 “接收器”就是消息发送的对象。如果接收器接受了消息，么同时它也接受了消息所包含的行为责任。然后，接收器响应消息，执行相应的“方法”以实现要求。

与消息传递相关的重要的信息隐藏原则——也就是说，提出要求客户不需要了解要求实现的具体方式。另外还有一条原则，我们所知道的所有人都是隐藏在消息传递过程中的。

消息传递与过程调用

首先，每一条消息都有一个指定的接收器相对应；接收器就是消息发送的对象。过程调用却没有指定的接收器。

其次，消息的解释（即用于响应消息的方法）由接收器来决定，并且随着接收器的不同而不同。例如，Chris 给他的一位名叫 Elizabeth 的朋友发了一条消息，她接收之后，立刻采取了相应的行动(即把花送给了他们共同的朋友 Robin)。然而，Elizabeth 完成要求的方法（很可能只是把消息发给 Fred）和 Fred 完成相同要求的方法是完全不同的。

1.4.3 责任

面向对象编程的一个基本概念就是用责任来描述行为。Chris 对行为的要求仅仅表明他所期望的结果(把花送给 Robin)。Fred 可以随意选择使用的方法来实现所期待的目标，并且在此过程中不受 Chris 的干扰。

通过用责任来讨论问题，我们提高了抽象的水平。使得对象之间更加独立，这正是解决复杂问题的关键。通常用协议来描述与一个对象相关的整个责任的集合。传统程序的执行通常是通过对数据结构进行操作——例如，改变数组或记录中的域。与之相反，面向对象程序却要求数据结构（即对象）提供服务。从传统的、结构化数据的角度来观察软件与从面向对象的角度来观察软件之间的区别可以用两句修改过的名言来总结：

- 不要问你能为数据结构做什么。
- 要问数据结构能为你做什么。

1.4.4 面向对象基本特征

被一些人称为“面向对象编程之父”的 Alan Kay，确定了以下关于 OOP 的基本特征：

1)任何事物都是一个对象。

2)通过互相联系的对象请求其他对象执行一定的行为来完成计算。对象之间通过发送和接收消息进行通信。消息是指对特定行为的请求，并且伴随着完成这项任务所需的参数。

3)每个对象都有自己的存储空间，用来存储其他对象。

4)每个对象都是一个类的实例。类用来代表一组相似的对象， 例如整数、链表等。

5)类可以看作是存储仓库，用来保存与一个对象相关的行为。也就是

说，同一个类的多个实例对象能够执行相同的行为。

6)类可以组织成一个单根树状结构，称为继承层次。在这个树状结构中，一个类实例的存储空间和行为自动地被其派生类使用。

1.5 面向对象简史

1.5.1 雏形阶段

一般认为，面向对象编程是一项计算机科学领域的新兴技术。实际上却刚好相反，我们现在所使用的关于面向对象编程的主要概念，例如对象、类和继承层次，早在 20 世纪 60 年代就已作为 Simula 语言（Simula 语言是由挪威计算中心的研究人员开发的）的一部分得到了发展。正如名称所暗示的那样，Simula 语言是受那些关于现实生活体系模拟的问题所启发的。但是，即使对 Simula 的开发者而言，这些语言结构的重要性也是慢慢才被认识到的。

20 世纪 70 年代，Alan Kay 在 Xerox PARC (Palo Alto 研究中心)成立了一个研究小纸组。他非常有先见地预言，未来的个人计算革命将在 10 年后发展起来（参见他于 1977 年发表在 Scierartific American 上的论文[Kay 1977]）。Kay 专注于开发一种编程语言，这种语言可以被非计算机专业人员或者没有经过任何计算机应用训练的普通人所理解。E 他发现通过类和模拟计算的概念，初学者很容易理解隐喻。然后他在 PARC 进行了一系列以孩子为程序员的试验，结果证明了这一现象。他的小组开发的这种编程语言叫做 Smalltalk，在随后的 10 年里发展了多个版本。1981 年 Byte 杂志上的一篇著名的论文使 Kay 和他的小组在 Xerox 开发的技术流行起来。

另外一项工程在这个国家的另一端几乎和 Kay 的工作同时进行着。Bjame Stroust 是贝尔实验室的一名研究者，早在英国剑桥大学完成博士学

位期间就学习了 Simula 语言，他当时正在进行着 C 语言扩展版本的开发工作，这个扩展版本将便于创建类和对象[Stroustrup 1982]。这项开发最终演化为 C++语言。

主要发展历程：

- 60 年代挪威计算中心开发的 Simula67——面向对象语言的先驱和第一个里程碑（首先引入了类的概念和继承机制）。
- 70 年代 CLU、并发 Pascal、Ada 和 Modula-2 等语言对抽象数据类型理论的发展起到重要作用（支持数据与操作的封装）。
- 犹他大学的博士生 Alan Kay 设计了一个实验性的语言 Flex。从 Simula 67 中借鉴了许多概念，如类、对象、继承等。
- 1972 年 Palo Alno 研究中心（PARC）发布了 Smalltalk-72，其中正式使用了“面向对象”这个术语。
- Smalltalk 的问世标志着面向对象程序设计方法的正式形成。但是这个时期的 Smalltalk 语言还不够完善。

1.5.2 完善阶段

PARC 先后发布了 Smalltalk-72，76，78 等版本，直至 1981 年推出该语言最完善的版本 Smalltalk-80。

Smalltalk-80 的问世被今认为是面向对象语言发展史上最重要的里程碑。迄今绝大部分面向对象的基本概念及其支持机制在 Smalltalk-80 中都已具备。它是第一个完善的、能够实际应用的面向对象语言。

1.5.3 繁荣阶段

自 80 年代中期到 90 年代，是面向对象语言走向繁荣的阶段。其主要

表现是大批比较实用的 OOPL 的涌现。OO 编程语言分为纯 OO 语言和混合型 OO 语言。混合型语言是在传统的过程式语言基础上增加 OO 语言成分，在实用性方面具有更大的优势。

随着这些信息的扩散，关于面向对象编程技术研究的其他类似项目蓬勃地开展起来。1986 年，关于面向对象编程的第一次重要会议召开时，已经有几十种可以使用的新的编程语言，其中包括 Eiffel 语言[Meyer 1988a]、Objective-C 语言[Cox 1986]、Actor 语言[Actor 1987]、ObjectPascal 语言和各种 Lisp 语言。

在 1986 年的 OOPSLA 会议后的近 20 年里，面向对象编程已经从一场革命技术转变成主流技术，在此过程中，它已发展成为整个计算机科学领域的重要组成部分。

1.5.4 面向对象开发方法简史

20 世纪 90 年代，主要面向对象开发方法有：

- Coad/Yourdon(OOAD)
- GradBooch(OOD)
- Ivar Jacobson(OOSE y Object-Oriented Software Engineering)
- Jim Rumbaugh(OMT Object Modeling Technique)

在 20 世纪 90 年代后期，Grady Booch、Ivar Jacobson 及 Jim Rumbaugh 三人尽管在面向对象领域各有各自的主张，但是通力合作将面向对象用合理的符号表示产生了统一建模语言 UML(Unified Modeling Language)，这个图形化的建模语言既有可视的表达形式又有严谨的语义支撑。

后来，面向对象思想的发展使得在软件开发中使用模式化的方法受到了重视，模式化的思想来源于建筑业。建筑大师 Alexander:每一个模式描述了一个在我们周围不断重复发生的问题以及该问题解决方案的核心，这

样你就可以一次又一次的使用该方案而不必做重复劳动。就像建筑业用预制的墙和窗来构筑房屋，在面向对象设计中我们使用对象和接口代替墙和窗来构筑系统，它们的核心都在于提供了相关问题的解决方案。

设计模式（design pattern）的提出，是面向对象程序设计演化过程中的一个重要里程碑。正如 Gamma，Helm，Johnson 和 Vlissides 在他们的经典著 作《设计模式》一书中所说的：设计模式使得人们可以更加简单和方便地去复用成功的软件设计和体系结构，从而能够帮助设计者更快更好地完成系统设计。

1.6 面向对象编程小结

- 面向对象编程不仅仅是在编程语言中加入一些新的特征。更重要的是，它是用来处理分解问题和开发程序解决方案的一种崭新的思考方式。
- 面向对象编程把程序看成是一个集合，这个集合由称为对象的松散连接的代理组成。：每一个对象都对特定的任务负责。通过对象的相互作用进行计算。因此，在某种程度上，编程就是一个对模型域的模拟。
- 对象是对状态（数据值）和行为（操作）的封装。因此，对象和专用计算机有很多相似之处。对象的行为由对象类来规定。每个对象都是某个类的一个实例。同一个类的所有实例都以相似的方式（即调用同一方法）来响应相似的要求。
- 对象通过调用方法表现其行为（类似于执行一个过程），以此来响应消息。对消息（即所使用的特定的方法）的解释由对象决定，并且随着对象类的不同而不同。类可以通过继承这种概念相互连接。通过继承，类被组织成一个层次化的继承树。树中较低层次的类，可以存取和使用与树中较高层次的类相关的数据和行为，这种情形称为子类继承了来自父类的行为。

- 设计面向对象程序就像组织一个团体，团体中的每位成员都被赋予一定的责任。团体目标的实现来自于每位成员的努力和他们之间的相互合作。
- 通过减少软件组件之间的相关性，面向对象编程可以实现可复用软件系统的开发。这样的组件与软件应用的其他部分是隔离的，可以作为独立的单元来创建和测试。
- 可复用软件组件允许程序员在更高的抽象层次上处理问题。我们可以简单地通过(被对象所理解的消息和对象需要执行的任务来定义和操作对象，而不必考虑执行细节。

第二章 面向对象编程与非面向对象编程

2.1 面向对象编程与非面向对象编程

对于非面向对象编程，程序往往是面向过程或者面向数据的。这些程序中通常有可全访问的数据及过程，由主程序或其子程序来控制及操作这些数据。程序的每个部分都可以访问全局数据，得到数据的一部分，操作这些数据，然后在在需要时，保存对数据的更改。

对于面向对象编程，程序被划分为一组通信的对象。每个对象均封装了关于某个概念所有行为和信息。实现功能的能力、实现功能所需的“知识”或数据被分布在对象之中。当一个对象对其他对象有需求时，就向该对象发出消息，这个对象接受到此消息后做出相应的动作并很有可能返回值给调用者。第一个对象甚至可以在第二个对象不存在的情况下创建该对象。因此，要开始进行面向对象编程时，通常就是先创建几个对象，然后让这些对象开始通信。

这种面向对象编程的观点，即对象分摊工作和责任，对我们而言是十分熟悉的，因为现实中人类也采用这样的交互方式。

例如一位企业主，并不需要对所有的事亲历亲为，事实上，该企业主只需要将任务分配给雇员。每位雇员不仅要完成给定的任务，并且还得负责维护和该任务相关的数据。比如，秘书不仅需要负责打印文件，也要负责将文件存放在适合的档案柜中；并且，如果文件中存放的是机密数据，秘书也要负责保护这些文件，并且负责允许或拒绝他人对文件的查看。在秘书的工作过程中，他可能还需要办公室内外其他人员的帮助。

面向对象的方式与非面向对象的方式相比：

- 由于所谓的“智能”被分布在了各个对象之中，每个对象都维护了各自实现任务所需的数据，将数据保存于较小的可管理单元就相对比较容易，
- 这种方式也便于理解这些单元是如何互相影响的。面向对象编程的分布性可以增加代码的可读性。
- 对于非面向对象程序而言，在全局数据结构上的一次小改动将有可能迫使所有访问该数据的过程进行相应的改变。
- 设计良好的面向对象程序只有少量的全局数据，而将大部分数据存储于对象中进行内部使用。
- 对象所属的类中的数据存储方式的改变通常意味着，程序中需要改变的部分只有该类内部的代码而已。
- 如果编程人员认为某个对象工作过于低效，就可以在不对系统的其他部分产生影响的情况下，重新设计该对象使其变得高效，从而支持了软件的可维护性。
- 由于每个对象都具有一个小的良好设计的角色，并独自包含所需的数据，在其他情况下重用该对象就会十分容易。

2.2 面向对象语言中的基本概念概述

2.2.1 类、对象、方法、消息

类(class)的概念可以从建模角度及编程语言的角度来理解。在设计软件应用的时候，类模型将对系统中重要的概念进行抽象处理，建立良好定义的责任及与其他类的良好关系。在面向对象编程语言中，类可以被视为对象的模板，而这些对象描述了某种行为、某些责任以及某些相关数据

对象(object)是类的实例。一个对象就类似于一位秘书或一位警察。对象所属的类定义了该对象拥有的数据类型、该对象的行为及该对象对这些数据的责任。但对于一位秘书而言，个人拥有用各自需要维护的数据（各自的状态）。对象之间通过消息传递的方式通信并命令彼此进行动作。

通过向其他对象传递消息，第一个对象让第二个对象执行某些代码。这些代码实际上就是一个过程浓面向对象语言中称为方法(method），该过程与第二个对象关联。因此，消息传递实际上就是一个对象向另一个对象发出的要求执行其某个方法的要求（或命令）。通过这个机制，对象可以被视为能对发出请求的任何客户端提供服务的服务器（通过消息的接受）。

需要注意的是，在 Java 中,只有简单数据类型的变量将数据存储于变量之中，对于所有对象数据类型的变量而言，变量存储对数据的引用（reference）。

在 Java 中，所有没有显式继承其他类的类都隐式继承了 Object 类。因此，所有的 Java 类要么直接继承了 Object 类，要么间接通过继承链中一个或多个中间类继承了 Object 类。这意味着，所有的类都自动继承了 Object 类的方法：clone. equals. finalize. getClass,hashcode. notify. notifyAII. toString 以及三个版本的 wait。

Java 同时也包含无对象(objectless)变量及无对象方法，称为类变量和类方法。实际上，通过使用 Java 的这些无对象特性，可以编写一个几乎完全非面向对象地程序。类变量的典型使用方式是定义常量。这样的常量不仅被该类的所有对象共享，而且通常被声明为公共变量，以供给该程序中所有的对象和类使用。除了用来定义常量以外，类变量也可以用来使得某类所有的实例（对象）共享一份数据。类方法可以被视为可独立于某类的所有对象而进行调用的方法，而非传递给该类的对象的消息。类方法是十分有用的，能在不引用 Math 对象的情况下，被 Java 代码中的任何主体存

取和执行。

2.2.2 继承

对象之间存在着一般和特殊的结构关系，也就是说它们存在继承关系。很多时候也称作泛化和特化关系。面向对象编程的最重要的特性之一便是“实现继承”(implementation inheritance)或“子类化”(subclassing)，这个特性显著地增强了类的可重用性并且减少了代码的冗余。

设想这样一个例子，某编程人员被分配编写一段绘图程序，要求在用户的控制之下，可以在画板上实现矩形的放大、缩小及移动。为了便于处理矩形，应该有一个 Rectangle 类来存储矩形的相关信息，例如大小和位置。

可能该编程人员稍微有点懒惰，他并没有重新编写一个 Rectangle 类，而是花了一两分钟在现存的代码库中查找是否已经存在可用的 Rectangle 类。当然，在 Java 类库中有很多 Rectangle 类，其中包括 java.awt.Rectangle,java.avixeom.Rectangle2D.Double 及 java.awt.geom. Rectangle2D.Float。通过了解这些类，这位编程人员认为 java.awt.Rectangle 是最接近他的需求的类。但是他需要的类必须包含一个 getCenter()方法及一个 setCenter(int x，int y)方法，而上述臵 Rectangle 类并没有包含这样的方法。那么，这位编程人员如何能通过最少的努力来满足他的需求呢？

如果现存的 Rectangle 类的源代码是可用的，这位编程人员可以通过修改这个类来满足需求，包括增加新方法，或者在可能的情况下，删除他不再使用的方法。或者，他可以复制这个 Rectangle 类的代码并将其插入一个新的类 EnhancedRectangle，然后在其中增加新的代码。

代码重用始终都是一个非常适合的举措，但却不是通过上述的两种方法。这两种种方法具有不优雅的方面。第一种方法会给使用原来的 Rectangle 类的代码带来问题——现在，同时存在两个版本的 Rectangle 类，

这会给使用者及编译器造成困扰。第二种方法要稍好一些，因为新的类并不会对使用原来的 Rectangle 类的代码部分产生影响。但是在这种情况下有着严重的代码冗余，而这种方法引入了不必要的复杂性（请记住优雅的一个特性是简无性，。比如，如果后来发现被复制的那部分代码存在错误，那么编程人员还需要去修改复制出的代码。

如果可用的只是 Rectangle 类的编译码而不是源代码，那么这两种方法则都不适用。这种情况下的一种解决方案是，让编程人员忽视编译码而定义并实现他自己的 EnhancedRectangle 类。然而，这种操作并没有利用现有的代码，所以导致了相当数量的代码冗余。在这种情况下，并不一定拥有完全一致的方法体的副本，但却拥有了完全一致的语义副本，这同样很糟糕。另外，Rectangle 类（假定）已经通过了彻底的测试，但实现新的类则可能会需要相当的努力，仅仅是用来保证该类的代码属于无错误级别的代码。

可以使用实现继承来避免所有这些问题。从子类指向父类的箭头在 UML 中称为泛化(generalization)关系。这样的声明使 EnhancedRectangle 类成为 Rectangle 类的子类，也使 Rectangle 类成为了 EnhancedRectangle 类的父类。因为 EnhancedRectangle 类是子类，它继承了 Rectangle 类的有方法及数据。注意，因为构造函数是不能被继承的，所以需要为新的子类创建构造函数。构造函数中的 super(x，y 真 w 真 h)将调用父类的构造函数来初始化所有 Rectangle 类的数据。用来实现两个新的成员方法的 getCenterX、getCenterY、setLocation、getWidth 及 getHeight 函数都由子类从父类中继承。通过这样的方式，子类化提供了一种创建新的类的方法来重用现存类的代码及数据，建出的新类除了具有更多属性（数据和 / 或行为）外，其他均和原来的类一样。这个通过增加新属性来扩充现存类的过程称为利用继承实现特殊化(specialization)。

2.2.3 方法绑定和改写

在简单的组织化结构中，鸭嘴兽代表了一类问题：哺乳动物都是靠生产幼子来繁殖后代，鸭嘴兽虽然也是一种哺乳动物，但是却产蛋。为了解决这类问题，需要寻找一项技术，能够处理一般规则以外的特例。

我们可以通过在子类中发布一条信息来解决此问题，这条信息可以改写(override)从父类继承的信息。通常，将子类中某一方法取与父类某一方法相同的名称，再结合寻找方法的规则（当响应特定消息时）来实现这个目的。

接收器类搜索并执行相应的方法以响应给定的消息。如果没有找到匹配的方法，搜索就会传导到此类的父类。搜索会在父类链上一直进行下去，直到找到匹配的方法，或者父类链结束。如果是前一种情况，就会执行方法；对于后一种情况，会产生错误信息。如果能在更高类层次找到相同名称的方法，所执行的方法就称为改写了继承的行为。虽然编译器在运行时不能确定调用哪个方法，但是许多面向对象编程语言(例如 Java)，却能够确定是否有合适的方法以供调用，并且能够产生错误消息作为编译时错误诊断，而不是运行时消息。

2.2.4 多态

多态性是指一般类中定义的属性和服务，在特殊类中不改变其名字，但通过各自不同的实现后，可以具有不同的数据类型或具有不同的行为。如不同花商送花方式。多态可以通过重载和类等级不同层次共享同一方法名字等多种方式实现。

第三章 抽象

通常，人们只使用几种简单的工具去创建、理解和管理复杂的系统。抽象就是最重要的技术之一。抽象是指对于一个过程或者一件制品的某些细节的有目的的隐藏，以便把其他方面、细节或者结构表达得更加清楚。

在形成抽象或模型时，我们有意识地避免理解很多细节，而是集中精力在几个主要特征上面。我们经常用另外一个术语来描述这样的行为，即信息隐藏。信息隐藏是指在抽象表现开发过程中有目的地忽略细节。

3.1 抽象层次

在一个典型的面向对象程序中，有很多抽象层次。更高层次的抽象部分地体现了面向对象程序面向对象的特征。

3.1.1 团体

在最高级别的抽象层次上，程序被看成是一个由很多对象组成的“团体”，为了实现共同目标，这些对象之间需要相互合作。

在面向对象开发中，团体这个概念有两种不同的形式。第一，程序员团体，在现实世界中为了完成应用程序，程序员之间必须互相合作。第二，程序员创建的对象团体，在虚拟世界里，为了实现共同目标，对象之间必须互相合作。像信息隐藏和抽象这样的面向对象的主要思想对这两种形式都适用。

团体中的每一个对象都为组织中的其他成员提供服务。在这个最高级别的抽象层次上，需要强调的最重要的特征就是交流和合作的渠道，以及

成员之间相互合作的方式。

3.1.2 单元

下一个级别的抽象不会在所有的面向对象程序中出现，也不被所有的面向对象语言所支持。但是，很多语言都允许一组对象一起工作，形成一个单元。关于这个思想的实例有 Java 语言中的包(packages)、C++语言名称空间（name spaces）和 Delphi 语言中的单元(units)。这种单元允许某些特定的而其他的特征隐藏在单元内部。

对于那些熟悉早期语言概念的读者，单元这个概念是继承了如 C 或 Modula 这样的语言的模块思想。在本章后面，我们会描述关于编程语言中抽象这个概念的简短历史，注意，面向对象编程思想应归功于早期的模块研究。

3.1.3 客户和服务器

下两个级别的抽象是用来处理两个独立对象之间的交互。我们常说的是一个对象能够为其他对象提供服务。这种直觉的建立来自于对客户和服务器之间通信的描述。这里的服务器只是表示提供服务的对象。这两个层次的抽象涉及了对这种关系的两种视角，一种来自于客户端，一种来自于服务器端。在一个优秀的面向对象设计中，我们可以描述和讨论服务器所提供的服务，而无需提及客户在使用这些服务时可能执行的任何行为。可以把这想像成广告牌。

这个广告牌描述了一个数据结构（如堆栈）所提供的服务。这个级别的抽象通常用接口来表示，接口是一个与类相似的结构，它定义了行为，但不描述如何来实现这些行为。

3.1.4 抽象行为的具体实现

下一抽象层次查看同一边界，但却不同于服务器端。这一抽象层次需要考虑抽象行为的具体实现。例如，有很多数据结构可以用来满足堆栈的要求。这一层次主要关注服务的具体实现方式。

3.1.5 方法的实现逻辑

最后，在抽象的最底层需要单独考虑一项独立的任务，即一个方法。这一层次的抽象主要注的是执行这一活动所需操作的精确顺序。例如，我们可能需要研究这样的技术，用来移走放人堆栈里的最近元素。

3.2 抽象的其他形式

抽象可以用来帮助理解复杂的系统。在某种程度上，抽象是系统结构的版图。我们所使用的结构可以反映出系统某些真实的方面 t 例如，汽车确实既有发动机又有传动装置，，或者它可能只是一种思维的抽象，用来帮助我们理解问题。

抽象的思想可以进一步划分为不同的形式：

3.2.1 部分分化

人们最常使用的用来帮助理解复杂系统的技术就是把抽象和分化胱成的各种组成部分结合起来。例如关于如何组织人体运动的信息。在某一层次上我们只关心各种组织，把我们的身体看作是由骨骼（保持刚性）、肌肉（产生运动）、眼睛和耳朵（产生感觉）、神经系统（传递信息）和皮肤（把各个部位连接起来）组成的。在下一个抽象层次上，我们可能会问肌肉是

如何工作的，并且会考虑一些如细胞结构和化学作用的问题。而化学作用又是由分子结构决定的，为了了解分子，又得把分子继续分解成原子。任何解释都必须基于正确的抽象层次来阐述。如果试图以原子层次的细节来解释一个人是如何行走的问题，即使不是不可能的，也将会是极其困难的。

3.2.2 封装和互换性

创建一个大系统的关键步骤就是将其分成各个组成部分。假设我们不是在编写软件，而是正在制造一辆新汽车的一组成员。通过将汽车分成发动机和传动装置两部分，就可以分配人员基本独立地进行这两方面的工{区分，那些制造发动机的小组成员对传动装置只需要有一个抽象的（也就是外部的）理解，而那些真正制造传动装置的小组成员则需要对传动装置有着更加详细的内部理解。

封装的一大优点在于允许我们考虑实现互换的可能性。当我们把一个系统分成各个部分的时候，理想的目标就是将各个部分之间的相互作用减少到最少。例如，通过封装发动机的行为，使其与传动装置相隔离，我们可以把一种类型的发动机换成另外一种类型，而不会影响系统其他部分的运行。

为了将这些思想应用于软件系统，我们需要一种方式来讨论软件组件执行的任务，而且这种方式要和组件满足责任的方式区别开来。

3.2.3 接口和实现

在软件中，我们使用接口和实现这两个术语来描述一项任务包含哪些内容和如何实现任务，以及外部视角和内部视角之间的区别。接口描述系统被设计用来做什么，这是抽象使用者必须理解的思想。接口并没有说明

所分配的任务应如何执行。因此，接口通过与完成抽象的实现相匹配来使系统正常工作。发动机的设计者会与传动装置的接口相联系，而传动装置的设计者必须实现这些接口。

同样，开发复杂的计算机系统的一个重要步骤是将一项任务分成各个部分。这些部分可以由小组的不同成员来开发。每一个部分都有两个方面：接口显示给外部世界，实现用来完成接口的要求。

接口和实现的分离不但使在较高层次上理解一项设计更加容易（因为接口的描述比任何特定实现的描述麦简单得多），而且使软件组件的互换性成为可能（因为可以使用任何实现，只有它能够满足接口规范）。

目录：当一个系统中组件的数量越来越多的时候，以“目录”的方式来组织它们通常是很有效的。在日常生活中我们要使用很多不同形式的目录，例如，电话号码簿、字典或者因特网搜索引擎等等。类似地，在软件中也使用各种各样的目录。一份关于系统中所有类的简单列表就是一个这方面的例子。另外一个目录的例子就是由类所定义的所有方法组成的清单。描述 Java 标准库的类参考手册是一个非常有用的目录。以上每一个实例都体现了以下思想，、目录为用户提供了一种机制，使用户可以从一个范围很大的集合中迅速地定位到某一特定的位置（可以是类、对象或者方法）。

3.2.4 服务视角

接口描述了软件组件所提供的服务，却不必描述完成服务所使用的技术，这个思想是理解和处理多季软件系统的核心手段。我们描述的关于送花的故事强调的正是这种抽象。那个故事的最后结果是有很多人都参与了送花的过程。团体中的每个成员为这个团体中磊華他人提供服务，没有一个成员能够自己独立地解决问题，只有通过互相协作，才能实现所期待的结果。

3.2.5 复合法

复合法是另外一种从独立的部分创建复杂结构的强大技术。这一思想从少量几个基本的形式开始，然后增加一些规则，将各个形式结合起来创建新的形式。复合法的主要原则是允许合并机制，既可以作用于新形式也可以用在原来的基本形式上。

很多计算机程序本身就可以看成是复合的产品，这时，方法或者过程的调用就是复合的机制。我们从语言的基本语句开始（赋值等），通过这些可以建立一个实用的函数库。以这些函数为基础，就可以形成更加复杂的函数。如此继续下去，每一层都建立在上一层的基础之上，直到最终完成所要求的应用程序。

3.2.6 特化分层

另外一个处理复杂问题的方法是使用特化分层来构建抽象，有时也把它称为分类系统。例如，在生物学上把生物分成动物和植物，动物又分成脊椎动物和无脊椎动物，脊椎动物包括哺乳动物，哺乳动物又包括猫、狗、鲸鱼等等。

在面向对象语言中，新的接口从现有的接口上形成。一个类从已有的类那里继承而来，这时，和原始类相关的所有特性（数据字段和行为）都同样适用于新类。

3.2.7 模式

当面对一个新问题时，大多数人会首先看一看他们曾经解决过的问题和现在的新任务有什么共同的特征。那些原来的问题可以作为一种模式，

新问题也将以相似的方式进行处理，只需针对不同的环境做一些必需的改动。

这一观察结果暗示了软件模式的思想。模式就是试图为问题提出一个已经被证实的解决方，以便于能够很容易地用类似的方式来解决以后的问题。在面向对象界，这种思想广泛地用来描述对象团体中成员之间的相互作用模式。

3.3 抽象机制的发展简史

每个抽象机制，都是在探究解决复杂问题的漫长过程中所获得的最终成果。我们回顾一下关于计算机语言处理复杂问题所使用的技术的历史。回顾完之后，就会发现，面向对象技术并不是革命性的，它只是从模块到抽象数据类型最终到对象这个发展过程的自然结果。

3.3.1 汇编语言

汇编语言是最早的抽象机制之——它是一种能够以更加人性且更加友好的方式进行编程的工具，把汇编语言编译成适于机器执行的语言。汇编程序允许使用符号名称。这一简单的过程就是漫长的抽象进程中的第一步。抽象使程序员把更多的精力放在定义要执行的任务上，并且可以使用更少的步骤来完成任务。

3.3.2 过程

过程和函数代表编程语言抽象的进一步改进。过程将重复执行的任务或者只有轻微变化的任务的定义集中于一处，用来复用，而无需进行多次代码的复制。此外，过程首先为信息隐藏提供了可能，一名程序员可以为

其他程序员编写一个过程或者一系列过程，而其他程序员不需要知道完成的确切细节——他们只需要必需旨的接口。但是过程并不是所有问题的答案。尤其是，它们不是信息隐藏的有效机制，而只是部分地解决了多个程序员使用相同名称的问题。

3.3.3 模块

对于全局名称空间拥挤问题的解决方案是引入模块的思想。在某种程度上，模块可以简单地看作是用来改善建立和管理名称集合及其相关数值的一种技术。言之，模块提供了将名称空间划分成两个部分的能力。公有部分可以从模块外存取，而私有部分只能从模块内存取。类型、数据（变量）和过程可以在住任何部分定义。

1)模块必须为目标用户提供得以正确使用模块所需的所有信息，除此之外，无需提供其他任何信息。

2)模块必须提供完成模块所需要的所有信息的实现，除此之外，无需提供其他任何实现。

模块解决了一些软件开发过程中的问题，但并没有解决全部问题。模块本身提供了一种有效的信息隐藏的方式，但是模块不允许我们实现实例化(instantiation)，实例化是一种能够建立数据区域多份拷贝的能力。为了解决实例化这个问题，计算机科学家需要开发新的概念。

3.3.4 抽象数据类型

关于抽象数据类型概念的开发，义过，程序员应该能够定义他们自己的新的数据抽象，工作起来就像系统基本数据类型一样，这包括为用户提供能够创建多个数据类型实例的能力。但是同样重要的是，用户应该能够

使用这些实例，并且只需要知道实例所提供的操作，而无需关心这些操作是如何实现的。

抽象数据类型(abstract data type)是通过抽象规范来定义的。例如，我们所举的堆栈数据类型的规范包括入栈(push)、出栈(pop)和返回栈顶(top)三种操作。与 ADT 相匹配的将是一个或者多个不同的实现方式。对于我们举例所用的堆栈，可能有多种不同的实现技术，例如，一种技术使用数组，另外一平要求上，任何有效的实现都会执行得很好。

ADT 思想的重要进展就是最终把接口的概念和实现的概念分离开来。尽管我们强调模块是一种实现技术，抽象数据类型是更加理论化的概念，但是模块还是经常作为抽象数据类型的实现技术来使用。两者虽然相关，但是却不完全相同。要建立一个抽象数据类型，必须做到以下几点：

1)导出一个类型定义。

2)生成一套可用的操作，可以用来操纵类型的实例。

3)保护与类型相关的数据，使得只有通过所提供的例程才能操作这些数据。

4)建立类型的多个实例。

在某种程度上，一个对象就是一个简单的抽象数据类型。一个对象定义就是一个抽象数据类型，但是，面向对象编程的概念是建立在抽象数据类型思想上的，并且增加了代码共享和代码可复用性这一重要的创新。

3.3.5 面向对象编程

面向对象编程从以数据为中心来看待世界的观点开始，继续向前发展。不是数据抽象本身对计算重要，ADT 才是有用的抽象，因为它可以用服务(service)这个术语来定义，并且提给程序的其他部分。其他类型的抽象也可以用类似的方法进行定义，不是根据它们特定的行为或者数据值，而是根

据它们所提供的服务。

关于如何构建一个计算机程序的观点从以功能为中心开始，发展到以数据为中心，最后到以服务为中心。

除了计算以服务为中心的观点外，面向对象编程还在抽象数据类型的概念上增加了几个重要的新思想。，其中最重要的就是消息传递(message passing)。活动是通过对特定对象的请求(request)初始化的，而不是通过对特定功能的调用初始化的。

隐藏在消息传递中的思想是消息的解释(interpretation)可以随着对象的不同而改变，即消息所导致的行为和响应依赖于接收消息的对象。因此操作的名称不需要唯一，使用简单而直接的命名形式有助于创建更容易阅读和理解的代码。

最后，面向对象编程还增加了继承(inheritance)机制和多态(polymomorphism)机制。继承允许不同的数据类型共享同一代码，这将减少代码的数量，并且增加代码的功能。多态允许这些共享代码被定制成适合某一数据类型的特定环境。对组件独立性的啪强调有助于软件的增量开发，在这一过程中，每个软件单元在合并成一个大系统之前，都可以独立地进行设计、编码和测试。

第四章 类和方法

尽管不同的面向对象编程语言使用不同的术语，但是它们都具有共有的一些概念：类、实例、消息传递、方法和继承等。

4.1 类定义

看一下三种类似的语言（C++、Java 和 C#）中对类的定义，从表面上看，这三种语言有很多不同，例如，在 C++语言中类的定义以分号结束，而对于其他两种语言就不是这样。关于类的可视性修饰符（即 public）在 C++，捂着 ：Java 中标明了一整块说明语句，而在其他两种语言中，它却分别放在每一个说明语句前。在 C++和 C#语言中，程序员可以定义枚举数据类型来表示纸牌花色。在 C++语语言中，通过将定义语句放置在类中，可以清晰地表达两种数据类型之间的联系（这在 C#语言中是不可能的）。对于 C++语言，在类定义之外表示花色的常量必须以类名为前缀。

类名的首字母要大写，这是很多语言都遵循的惯例，但不是所有的语言都这样（尤其是，很多 C++程序员更愿意全部使首字母大写，目的是为了更容易区分类型名和实例变量名。一些语言，如 Delphi Pascal，还有其他的命名惯例。

Java 语言不提供枚举数据类型，因此程序员通常定义一系列的符号常量。符号常量需要使用两个修饰符 final 和 static 来修饰。在 Java 语言中，修饰符 final 表示这个名称所表示的值不能再改变。修饰符 static 的意义是不管对这个类创建多少实例，static 所修饰的变量实例都只有一个。这两个修饰符一起作用，就定义了一个不能改变的惟一变量——即常量。

4.2 方法

构造函数是一种特殊的方法，这种方法有着和类相同的名称，并且在对象中用来初始化数据字段。类必须对外提供一种存取数据字段的方法，通过定义于类中的方法来存取数据字段是一种好的面向对象编程方式。仅仅返回数据字段值的方法称为存取器(accessor)，有时也称为获取器(getter)。

为什么对这个简单的行为使用一个方法，页而不是直接存取数据字段来实现呢？一个原因就是方法可以使数据字段是只读的(read-only)。函数只能被调用，而数据字段既可以读又可以写。

一般说来，编程语言并不指定在类定义中方法的声明次序。然而，次序对可读性却有着相当大的影响，这种影响对程序员可能是至关重要的。很多编程方式指南在这一问题上提出了一些互相冲突的建议，下面就是一些最重要的观点。

- 在类定义中，首先列出一些主要的特征，一些次要的特征列在后面。
- 构造函数是对象定义的一个最重要的方面，因此应该位于类定义的顶部。
- 对方法的声明应该通过分组来表示，这样可以迅速便捷地找到对应于给定淀消息选择器的方法。分组的原则可以按照字母顺序排列，或者按照方法的职能进行分组。
- 私有数据字段只是对于类的开发者才非常重要，它们应该列在类定义中靠后的位置。

定义和实现的分离

很多语言（如 Java 和 C#语言）把方法的主体直接放在类定义中。他

语言，如 C++和 Object Pascal 语言，则将这两方面分离开来。在 C 卄语言中，程序员可以自由选择，小规模的方法可以在类内部定义，大规模的方法可以在类外部定义。

4.3 接口

一些面向对象语言，如 Java，支持一种称为接口的概念。接口为某种行为定义了一份协议，但是不提供任何实现。

如同类一样，接口也定义了一种新类型。这就意味着可以仅仅通过接口的名称来声明变量。接口的使用和继承的概念十分类似。

4.4 属性

Delphi、Visual Basic、C#语言和其他的编程语言（包括面向对象语言和非面向对象语言）都包含一种称为属性的概念。在语法上，属性是以数据字段的方式来进行操作的，但是像方法一样，它也是内部操作。也就是说，属性既可以作为表达式来读取，也可以对其进行赋值。

但是，无论是读取属性值还是对属性赋值，都必须通过函数来实现，而不是仅仅由一个简单的数据值来实现。在 Delphi 语言中，属性是通过关键字 property 以及修饰符 read 和 write 来声明的。紧跟 reard 和 write 关键字之后的可以是数据字段，也可以是方法名。当某一属性用在表达式的形式中时，read 属性就会被调用，而当属性作为赋值目标时，write 属性就会被调用。只能读取而不能写入的属性称为只读属性。

在 c#语言中，属性通过编写一个无参数的方法来定义，这个方法只包括一个 get 部分或者一个 set 部分。

public class PlayingCard {

```
public int rank {
    get
    {
        return rankValue;
    }
    set
    {
        rankValue = value;
    }
}
...
private int rankValue;
}
```

在 C#程序中，属性通常是无参数且返回惟一值的函数。

4.5 向前定义

程序有时需要两个或者更多的互相引用的类。这种形式称为相互递归(mutual recursion)。很多语言都支持这种相互递归。例如，Java 语言在生成代码之前会扫描整个文件，这样，如果程序文件的前面部分引用在文件后面声明的类，就不会产生任何冲突。其他的语言，例如 C++，会从前到后地依次处理程序文件中的类和方法去。名称在使用之前必须至少进行部分定义。这导致在 C++语言中必须实现向前定义(forward definition)。该定义的目的只是把这个名称置于循环中，关于定义的实现会在后面完成。当然，在读取类定义之前，对象不能做任何事情。要解决这种问题，需要对类定义和与其相关的方法的实现仔细地排序：首先是第一个类的定义，然后是

第二个类的定义，紧接着是第一个类的方法，最后是第二个类的方法。

4.6 内部类（或嵌套类）

在 Java 和 C++语言中，都允许程序员在一个类中定义另外一个类。这种定义方式在 Java 语言中称为内部类，在 C++语言中称为嵌套类。尽管这两个概念有相似的表现，但它们之间也有很大的语义差别。在 Java 语言中，内部类被链接到与其相关的外部类的一个具体实例上（内部类就创建在这个外部类实例中），并且允许存取这一对象的数据字段和方法。在 C++语言中，嵌套类只是一种简单的命名手段，它限制了和内部类相关的特征的可视性，除此之夕卜，两者再没有其他关系。

4.7 类的数据字段

使用被一个类的所有实例所共享的公共数据字段，在解决许多问题时都非常方便。然而，对这样的对象进行的操作却为面向对象语言的设计者创建了一个奇特的矛盾。为了理解这个问题，考虑一下当初创建类这个概念的原因——就是要减少创建相似对象的工作量。类的每一个实例都和这个类的其他实例有着完全一致的行为。现在，假设由于某种原因我们定义了一个公共数据字段，它可以被类的所有实例共享，然后，考虑如何对公共数据字段进行初始化。这看起来有两种选择，但是每种选择都不能令人满意。一种选择是每个实例都执行对公共数据字段的初始化任务（字段初始化后，还会被反复地初始化），另一种选择是没有实例执行丸行初始化任务（数据字段一直处于未初始化状态）。

为了解决这一矛盾，我们需要从类 / 方法 / 实例这个简单的范例中跳出来。另外一个机制就是，对象本身不必对共享数据进行初始化。如果对

象通过内存管理器自动将共享数据初始化为一个特定值（如 0），那么每一个实例都可以测试这一特定值，并且如果它们是第一个进行测试的实例，它们就会对数据进行初始化。但是，还有其他的更好的方法。

在 C++和 Java 语言中，共享数据字段都是使用 static 修饰符来创建的。在 Java 语言的符号常量创建中，我们已经看到了这样的应用。在 Java 语言中，静态数据字段的初始化是在加载类时，通过执行静态块(static block)来完成的。

在 C++语言中，有两种不同的方法。由基本数据类型表示的静态数据字段（或常量）可以在类的主体中进行初始化，就如同我们已经看到的那样。另外，也可以在类之外对静态数据字段进行全局的初始化。

在 C#语言中可以通过静态构造方法（声明为 static 的构造方法）来对静态数据字段进行初始化。构造方法不允许有任何参数。

在 Python 语言中，类的数据字段在方法层次上命名，而实例变量在方法内部命名（一般在构造方法内部）。

4.8 作为对象的类

在很多语言中（Smalltalk、Java 以及其他语言）类本身就是一个对象。当然，人们紧接着就会问什么类代表了对象所属的类别，即这个类是什么类？在多数情况下，有一个特殊的类，一般称为 Class，这就是类的类。既然对象是类，那么它们就具有相应的行为。我们可以对类做什么呢？通常，创建类的一个实例就是简单地给类对象传递一条消息。其他的一般行为包括返回类的名称、类实例大小或者类实例可识别的消息列表。

第五章 消息、实例和初始化

5.1 消息传递语法

我们使用消息传递（message passing，有时称为 method lookup，方法查询）这一术语来表示请求对象执行一项特定行为的动态过程。在第 1 章中，非正式地描述了消息传递，并且指明了如何区分消息和通常的过程调用。

- 消息总是传递给某个称为接收器的对象。
- 响应消息所执行的行为不是固定不变的，它们根据接收器类的不同而不同。也就是说，不同的对象可以接收相同的消息，但是却执行不同的行为。

对于任何消息传递表达式都有 3 个确定的部分。它们是接收器（receiver，消息传递的目的对象）、消息选择器(message selector，表示待传递的特定的消息文本)和用于响应消息的参数(argument)。

消息传递最通常的语法是使用句点来把接收器和消息选择器分离开来。还有一些次要的变化特征，比如在方法没有参数时，是否需要使用一对空的圆括号（在 Pascal 语言和一些其他的语言中可以省略这对圆括号）。

在 Smalltalk 和 Objective-C 语言中，表示消息传递的语法有一些细微的差异。在这些语言中，使用空格作为分隔符。一元消息（不含参数的消息）只是简单地写在接收器之后。含有参数的消息使用关键字表示法(keyword notation)。消息选择器被分成几个部分，每个参数之前都有一个部分。冒号跟在每个关键字之后。

aGame di8play: aCard atLocation: 45 and: 56,

Smalltalk 语言中，即使是二元操作，如加法，也被解释成将右值参数传递给左值的消息。

Z　<— x +y。　　“message to x to add y to itself and return sum　”

在 C++语言中定义的二元操作符也表示相似的含义。在 Objective-C 语言中，将一条类似于 Smalltalk 语言的消息包含在方括号中，乖括其他内容，例如，不包含存放消息结果的变量。

Int cardrank = [aCard getRank];

5.2 静态类型语言和动态类型语言

根据语言是动态类型还是静态类型，可以把语言分成两类。一般来说，静态类型语言把类型和变量联系在一起（这种绑定通常是通过声明语句来建立的），而动态类型语言只是把变量看作名称标识，把类型和数值联系在一起。Java、C++、C#和 Pascal 语言是静态类型语言，而 Smalltalk、CLOS 和 Python 语言是动态类型语言。

Objective-C 语言介于这两个阵营之间。在 Objective-C 语言中，如果变量是静态类型，则变量可以用一个固兑过这种方式声明的变量可以表示任何对象值，因此这种变量为动态类型。

PlayingCard aCard;　　/* a statically typed variable */

id anotherCard；　　／*a dynamically typed variable*／

在消息传递这方面，静态类型语言和动态类型语言之间存在显著的差异。一方面，静态类型语言在编译时使用接收器的类型来检查选择器，以理解它所接收的消息。另一方面，动态类型语言在编译时则没有办法去核实这一消息。因此在动态类型语言中，如果接收器不理解消息选择器，消息就可能产生运行时错误。而对于静态类型语言就从来不会发生这种运行时错误。

5.3 伪变量

大多数面向对象语言中，接收器并不出现在方法的参数列表中，而是隐藏于方法的定义之中。只有当必须从方法体内部去存取接收器的数值时，才会使用伪变量(pseudo-variable)。伪变量和通常的变量很相似，只是它不需要声明，也不能被更改（也许用伪常量这一术语更加合适，但是这一术语好像没有出现在任何语言的定义中）。

在 Java 和 C++语言中，指定接收器的伪变量命名为 this，在 Eiffel 语言中称为 Current，在 Smalltalk、Objective-C、Object Pascal 语言和许多其他的语言中称为 self。伪变量在使用时就好像是作为类的一个实例。

在很多编程语言中，对接收器伪变量的使用都可以忽略。如果在没有引用接收器的条件下，访问一个数据字段或者调用一个方法，那么这意味着接收器伪变量将作为消息的主体。一些 Java 编程方式指南建议，对于构造函数，使用出 IS 和构造函数的参数进行初始化数据成员。通过显式地使用 this，可以区分两个分别用作函数参数和数据成员的同名变量。

5.4 对象的创建

在大多数传统编程语言中，变量是通过声明语句来创建的，一些编程语言允许用户把变量声明和初始化结合起来，如下所示的是一个 Java 语言的实例。

Int sum = 0.0; //declare and initialize variable with zero

然而，大多数面向对象语言都把变量的命名过程和对象的创建过程分离开来。变量的声明只是创建一个标识变量的名称。为了创建对象，程序员必须执行另外的操作。通常这一操作是由 new 操作符来表示。

对象数组的创建

对象数组的创建涉及两个层次的问题。一是数组自身的分配和创建，然后是数组所包含的对象的分配和创建。

在 C++语言中，这些特征是结合在一起的。数组由对象组成，而每个对象则使用缺省（即无参数）构造函数来进行初始化。

另一方面，在 Java 中，表面上看来相似的语句却有着完全不同的效果。用来创建数组的 new 操作符只能用来创建数组。数组包含的每个数值必须独立创建，典型的方法是通过循环来实现。

从 C 或 C++程序员转向 Java 程序员常犯的一个错误就是，忘记把 Java 语言中数组的分配与数组所包含的元素的分配分离开来。

5.5 指针和内存分配

所有面向对象语言在它们的底层表示中都使用指针，但不是所有的语言都把这种指针暴露程序员。有的时候人们把“Java 语言没有指针”作为 Java 和 C++语言相比较时的特点，其实更加确切的说法是 Java 语言没有程序员可以看到的指针，因为所有的对象引用实际上就是存在于内部表示中的指针。

这一问题之所以重要，有三点原因。首先，指针通常引用堆分配的(heap allocated)内存，因此不符合传统的命令式语言中的与变量相关的通用规则。对于命令式语言，在一个过程中创建的变量值会随着过程的活动而存在，当从一个过程返回时，变量值也随之消失。另一方面，对于堆分配的变量值，只要存在对它的引用，就会一直存在，因此，变量值的生存期一般长于创建该变量过程的生存期。第二个原因是通过堆进行分配的内存必须通过某种方式进行回收。第三个原因就是，对于某些语言（特别是 C++语言），指针值和传统的变量值值是有区别的。在 C++语言中，对于以通常方式声明的变量，即所谓的自动(automatic)变量，其生存期总是绑定在创建该变

量的函数上。当退出过程时，变量的内存就会被回收。赋值给指针（或者是引用，指针的另外一种形式）的数值没有绑定到过程人口。这样的变量与自动变量有很多重要的区别。如同下面将要解释的那样，程序员必须显式地回收关于这些数值的内存。

内存回收

使用操作符 new 创建的内存称为基于堆(heap-based)的内存，或者简称为堆(heap)内存。与普通的变量不同，基于堆的内存没有绑定在过程的人口和出口处。尽管如此，内存仍然是有限的，因此，必须提供某种机制来回收内存空间。这样，就可以对已经分配给某个对象的内存空间进行再次使用，以满足后续的内存请求。

通常有两种方法可以完成内存回收这项任务。一些语言(如 C++和 Delphi Pascal)建议在程序中显式指定不再使用某个对象值，将对象所使用的内存回收并再次使用。为实现这一目的所使用的关键字根据语言的不同而有所不同。在 Object Pascal 语言中，使用 free 这个关键字，Objective-C 语言使用同样的关键字来回收内存，但是却写成消息的形式，接收器位于最前面。在 C++语言中，回收内存的关键字是 delete。；当删除一个数组时，deleted 关键字后要紧接着一对方括号。

作为程序员显式地管理内存的替代，另外一种回收内存的方法是垃圾回收机制。使用垃圾回收机制的语言（例如 Java、C#、Smalltalk 或 CLOS 语言）需要时刻监控对象的操作，当对象不再使用时，自动回收对象所占有的内存。通常，垃圾回收系统直到系统内存快要耗尽的时候才开始工作，在回收无用的内存时，需要将正在执行的应用程序挂起，回收完成后，程序恢复执行。垃圾回收过程需要一定的执行时间，因此，与程序员自己控制释放内存相比，要付出额外的代价，但是垃圾回收同时也避免了许多经常出现的编程错误。

- 对于一个程序，程序员忘记释放不再使用的内存一般是不会耗尽内存的（当然，如果任何情况下程序所需要的内存都超过所能提供的内存，那么程序还是可以耗尽内内存的）。
- 程序员不可能会去使用已经被释放的内存。释放的内存可以复用，因此内存的内容可以被重写。这样，使用释放后的内存的内容将产生不可预测的结果。
- 对于程序员来讲，不能对同一内存值释放多次，这样做将导致不可预测的结果。

当垃圾回收系统无法使用时，为了避免这些问题，通常有必要确保每一块动态分配的内存对象都有一个指定的属主(owner)。内存的属主负责保证内存位置的合理使用，以及当内存不再使用时得以释放。在大型程序中，难点之一就是争夺共享资源的所有权。

当一个对象不能指定为共享资源的属主时，通常可以使用的另外一项技术是引用计数(reference counts)。引用计数就是引用共享对象的指针的计数值。需要认真地确保计数的准确。无论何时，只要加人一个新指针，计数值就会减一。当计数值达到 0 时，就意味着没有指针引用该对象，它的内存可以被回收。

就像对动态类型有支持意见也有反对意见一样，支持垃圾回收以提高编程效率和反对垃圾回收以提高编程灵活性也形成了一对矛盾。自动垃圾回收的代价是昂贵的，因为它要求必须有一个运行时系统来管理内存。另一方面，内存使用错误的成本同样也是相当昂贵的。

5.6 构造函数

构造函数(constmctor)是用来初始化一个新创建对象的方法。把创建和初始化联系起来有很多优点。最重要的是，它确保对象在正确地初始化之

前不会被使用。当创建和初始化分离时（当使用没有构造函数的编程语言时），程序员在创建新对象之后，很容易忘记调用初始化例程，这样通常会导致不良后果。发生几率少一些的问题是，在同一个对象上调用两次初始化过程，这通常也会引起一定的麻烦。通过使用构造函数就可以避免这个问题。

在 Java 和 C++语言中，可以通过检查与类显示的名称是否相同来识别构造函数与普通方法的区别。构造函数与普通方法的另外一个细微的差异就是构造函数不声明返回值的数据类型。

当使用一操作符进行内存分配时，构造函数所需的参数紧随在类名之后。在 Java 和 C#语言中，数据字段可以初始化为特定的数值，这种赋值独立于构造函数中的参数赋值，数据字段可以在初始化时进行赋值，也可以在后来的构造函数中再次赋值。

在 C++、C#和 Java 语言中，只要每个函数参数的数目、类型或次序不同，就允许多个函数使用相同的名称定义。构造函数经常使用这种定义方式，例如，一个构造函数为无参数函数，而另外一个构造函数为有参数函数。参数数目、类型和次序的结合一般称为函数类型签名(type signature)。通过检查调用的类型签名可以决定选择哪个过载构造函数。

在 C++语言中，程序员必须注意，调用缺省构造函数时必须去掉括号。在这里使用括号虽然符合语法，但却有着完全不同的意义。另一方面，在 C++语言中，当使月号，而在 Java 或者 C#语言中则完全相反。

在 C++语言中，构造函数可以使用与其他语言稍有不同的语法初始化数据成员。这种冒号跟随着变量名称和括号括起来的初始化数值所组成的结构，称为初始化器(initializer)。

```
Class PlayingCard {
public:
```

```
PlayingCard (Suits is, int ir)
: suit(is), rank(ir), faceUp(true) { }
...
};
```

5.7 析构函数和终止器

当对象创建时（也可以说，当对象诞生时），构造函数允许程序员执行特定的行为。当变量即将消失并且它的内存将被在 C++语言中，这可以通过析构函数（destructor）这个方法来实现。只要从内存空间开始释放对象，析构函数就会自动调用。对于自动变量，当包含变量声明的函数返回时，变量的空间就会被释放。而对于动态分配的变量，空间的释放通过操作符 delete 进行。析构函数的名称为波浪字符“~”加上类的名称，它不需要任何参数，也不会被用户直接调用。

第六章 继承与替换

6.1 继承的描述

继承在程序语言中的含义如下：子类所具有的数据和行为总是作为与其相关的父类的属性的扩展(extension)（即更大的集合）。子类具有父类的所有属性以及其他属性。另一方面，由于子类是父类的更加特殊（或受限制）的形式，在某种程度上，子类也是父类的收缩(contraction)。作为扩展和收缩之间的张力，继承是许多技术的内在来源，但同时也造成了许多对其使用的混乱。继承总是可以传递的，这样类就可以继承各个级别的父类的特征。

6.2“是一个”检验

在检验两个概念是否为继承关系时，存在一项规则，这项规则称为是一个(is-a)检验。具体内容就是如果检验概念 A 与概念 B 是否为继承关系，那么就尝试着套用这个英语语句：“A (n) Ais a (n) B”，如果这个语句“听起来是对的”，那么这个继承关系很可能就是正确的。虽然偶尔也有一些合理的继承关系在进行“是一个”检验时失败。但对于大多数情污况，这种检验都能就继承技术使用的合理性给出一个可靠的信息。

6.3 使用继承的原因

- 继承作为代码复用的手段。由于子类可以继承父类的行为，这样，对于子类，一些代码就不必重新进行编写。当开发新程序时，这可以极大地减少代码的数量。

- 继承作为概念复用的手段。这出现在子类改写父类所定义的行为时，尽管此时并没有在父类和子类之间共享代码，但它们共享方法的定义。

6.4 不同语言中的继承

面向对象编程语言可以分为两类：第一类语言要求每个类都必须继承于已经存在的父类，另一类语言则没有此项要求。Java, Smalltalk, Objective-C 和 Delphi Pascal 语言都是前者的实例，而 c++和 Apple, Pascal 语言是后者的实例。

对于那些要求所有的类都必须继承于已存在的类的语言来说，一个好处就是存在一个关于所有对象的根类，这个根类在 Smalltalk 和 Objective-C 语言中表示为 Object，在 Delphi Pascal 语言中表示为 TObject。这个根类提供的任何行为都被所有的对象所继承。这样，每一个对象都必然拥有一套公共的最小的函数集合。

关于一个大的继承树的缺点是它将所有类结合成一个紧密耦合的单元。而对于 C++语言及其他语言的程序以拥有几个独立的继承层次，因此不必包含整个巨大的类库，每个程序只使用其中的一小部分。当然，这也意味着，程序员无法定义那种所有对象都必然包含的功能。

6.5 子类、子类型和替换

子类的实例必须拥有父类的所有数据成员。

子类的实例必须至少通过继承（如果不是显式地改写）实现父类所定义的所有功能（子类也可以定义新功能，但此时并不重要）。

这样，在某种条件十下，如果用子类实例来替换父类实例，那么将会

发现子类实例可以完全模拟父类的行为，二者毫无差别。

替换原则是指如果有 A 和 B 两个类，类 B 是类 A 的子类，那么在任何情况下都可以用类 B 来替换类 A，而外界则毫无察觉。

术语子类型(subtype)是指符合替换原则的子类关系，以区别于一般的可能不符合替换原则的子类关系。所有面向对象编程语言都支持替换原则，尽管某些语言在改写方法时需要附加的语法。大多数语言都以一种非常直接的方式来支持替换原则，父类只包含从子类得来的值。对此，一个主要例外出现于 C++语言中，对于 C++语言，只有指针和引用真正地支持替换原则，声明为值（不是指针）的变量不支持替换原则。

语法和语义：在某种程度上，子类和子类型之间的关系非常类似于关于语言的语法和语义之间的关系，虽然并不完全一致。语法用来处理如何编写一条语句，就像子类用来处理如何声明一个类一样。语义用来表示一条语句的含义，就像子类型用来处理子类如何保留父类的含义一样。

看一下堆栈这个例子。在 Java 语言中，ADT 的描述可能会以接口的形式出现。例如，堆栈可能会描述成下面这样

```
Interface Stack {
    Public void push (Object value);
    Public Object top ();
    Public void pop ();
}
```

下面的类定义代码符合 Stack 接口，但是对于除了最近放进堆栈的元素之外的所有元素，它违反了 UFO 特征，因此这段代码并不符合我们所希望的堆栈所具有的属性。

```
class NonStack implement8 Stack {
    public void push (Object value) { v = value; }
```

```
    public Object top () { return v; }
    public void pop () { v = null; }
    private Object v = null;
}
```

从这个实例可以看出，属性对于堆栈的含义非常重要，并且属性无法通过接口定义来指定。这并不是因为我们不愿意这样做，而是由于 Java 语言（像许多其他语言一样）并没有提供一种可以通过接口来指定属性的方法。因此，大多数程序员（以及大多数编程语言）都要求助于一些非正式的信息，例如关于类需求的英语描述。

子类型与子类之间的差异：如果说新类是已存在类的子类型，那么这个新类不仅要提供已存在类的所有操作，而且还要满足与这个已存在类相关的所有属性。例如，新类的任何一个与旧类类型签名一致的方法都必须符合与这个方法相关的属性（当然，新类也可以增加新的操作）。

子类型关系是通过行为这个术语来描述的，它与新类的定义或构造无关。正如 NonStack 类所说明的那样，一个新类符合子类远比符合子类型容易。同样也很容易举一个例子来说明一个子类型并不是一个子类。例如，SmalITalk 语言中的 Array 所定义的数据类型大致如下。

```
At: index put: value
At：    index
Size
```

这种数据类型的属性包含这样一个事实，如果数值通过给定的索引放入集合，那么当后来使用同一索引来存取数组时，将返回同一数值。Dictionary 数据类型支持相同的接口，但不限制索引值必须为整数。由于 Dictionary 类支持与 Array 类相同的接口，因此即使 Dictionary 类与 Array 类之间并不存在继承关系，但是也可以说 Dictionary 是 Array 的子类型。

6.6 改写和虚拟方法

子类有时为了避免继承父类的行为，需要对其进行改写(override)。从语法上讲，这意味着子类将定义一个与父类有着相同名称且类型签名相同的方法。当改写与替换相结合时，就会出现这样一种情况：变量声明为一个类，它所包含的值来自于子类，与给定消息相对应的方法同时出现于父类和子类。几乎在所有有这种情况的案例中，我们想要执行的方法都是来自于子类的方法，而忽略父类的同名方法。

对于许多面向对象语言（如 Smalltalk、Java 语言）来说，只要子类通过同一类型签名来改写父类的同名方法，自然便会发生所期望的行为。另一方面，还有一些语言需要程序员来说明是否允许这种替换。许多语言使用关键字 virtual 来表明这一含义。在 C++语言中，必须在父类中使用这一关键字。

6.7 接口和抽象类

与类一样，接口可以继承于其他接口，甚至可以继承于多个父接口。虽然继承类的规范和实现接口的规范并不完全相同，但它们非常相似，因此将使用继承这一术语来描述这两种行为。

有些面向对象语言支持一种称为抽象方法(abstract method)的术语，它是一种介于类和接口之间的概念。例如，在 Java 和 C#语言中，类可以使用 abstract 关键字定义一个或多个方法，但不对这些方法进行实现。而在创建类的实例之前，子类必须实现父类的每一个抽象方法。因此，抽象方法的行为由父类进行指定，但是必须由子类来提供这些行为的实现。

也可以将整个类命名为抽象类，而不管这个类是否包含抽象方法。禁止创建关于抽象类的实例，抽象类只能用于继承，来作为其他类的父类。

在 C++语言中，使用纯虚方法(pure virtual method)这个术语来表示抽象方法这个概念，并且通过赋值操作符来表示。

类可以同时包含抽象（或纯虚）方法和非抽象方法。所有方法都声明为抽象（或纯虚）方法的类相当于 Java 语言中的接口的概念。

6.8 继承的形式

6.8.1 特化子类化（子类型化）

继承和子类化最常用的场合可能就是特化。在特化子类化时，新类是父类的一种特殊形式，但是它符合父类的各种规定。因此，这种形式显然支持替换原则。这种分类方式（特化子类化）是使用继承最常用的思维方式，也是一个良好的设计所要努力达到的目标。

6.8.2 规范子类化

另外一个继承常用的场合就是保证使子类和父类维持同一个特定的公共接口——也就是说，它们实现同样的方法。父类是由一部分已实现的操作和一部分需推迟到子类来实现的操作组成的结合体。在父类与子类之间通常不存在接口的变化——子类仅实现父类所描述但却没有实现的行为。

实质上，规范子类化是特化子类化的一种特殊情况，只是这种子类化不是对已有类型的改进，而是实现了父类中未完成的抽象规范定义。在这种情况下，父类有时称为抽象规范类(abstract specification class) 。

实现接口的类经常是继承的一种完全实现形式。然而，规范子类化也可以通过其他方式来实现。

一般来说，规范子类化的使用场合是，父类仅仅对行为进行定义，却

没有对其实现，而由子类来实现这些行又为。

6.8.3 构造子类化

子类一般通过父类就可以继承需要实现的几乎所有行为，只是需要对一些对应于接口的方法名称进行改变，或者以一种特定的方式对方法参数进行修改。即使父类与子类之间不符合这种是关系，上述的情况也依然存在。

构造子类化通常用于创建对持久存储系统的二进制文件进行写操作的类。父类仅仅实现原始二进制数据的写操作。子类主要用于存储每种结构。子类通过存储过程，使用父类型的行为来完成数据类型的实际存储。

由于构造子类化经常违反替换原则（形成的子类并不是子类型），因此静态类型语言不支持构造子类化。由于它通常可以快速简单地开发新的数据抽象，因此在动态类型语言中得到了广泛的应用。在 Smalltalk 语言标准库中有许多通过构造子类化形成的实例。

6.8.4 泛化子类化

在某种意义上，通过继承进行的泛化子类化与特化子类化恰好相反。这里，子类将父类的行为进行扩展，建立了一种更泛化的对象。泛化子类化的使用场合通常是我们打算基于已存在的类来建立一种不想修改或者不能修改的类。

让我们来考虑一下图形显示系统，在这里，类 Window 定义为在简单的黑白背景下进行显示。可以建立一个子类型 ColoredWindow，通过增加一个存储颜色的数据字段，我们可以改写继承之后的窗口，使之显示相应的背景颜色，来代替原来的黑白颜色。

泛化子类化通常用于基于数据值的整体设计，其次才是基于行为的设计。这可以通过彩色窗口这个实例加以说明，彩色窗口就包含了简单窗口所不需要的数据字段。

6.8.5 扩展子类化

泛化子类化是对一个对象已存在的功能进行修改或扩展，而扩展子类化则是对对象增加新功能。扩展子类化与泛化子类化之间的区别在于，后者必须至少改写来自父类的一个方法，而且子类的功能需要与父类紧密联系，而扩展子类化只是对父类增加新的方法，并且子类的功能与父类的联系并不那么紧密。

一个扩展子类化的实例就是继承于 Set 类用于存储字符串值的 StringSet 类。这个类将提供关于字符串操作的附加方法——例如，“通过前缀进行查找”的功能将返回关于所有元素的集合的一个子集，这个子集中的字符串都以指定的前缀开头。这些操作对于子类都很有意义，但对于父类则关系不大。

对于扩展子类化，父类的功能是可用的，而且是未用的，扩展子类化并不违反替换原则，因此，扩展子类化中的子类也是子类型。

6.8.6 限制子类化

限制子类化用于子类的行为少于或限制于父类。与泛化子类化一样，限制子类化经常用于这种场合，就是程序员需要创建新类，这种新类继承于已存在的不应该或不能进行修改的类。

例如，一个已存在的类库提供一个双向队列(deque)数据结构，可以从队列的两端添加或删除元素。但是，现在程序员希望写一个堆栈类，需要

的功能是只能从堆栈的一端添加或删除元素。

由于限制子类化显然违反替换原则则，通过它来创建的子类不是子类型，因此应该尽可能避免使用限制子类化。

6.8.7 变体子类化

在两个或多个类需要实现类似的功能，但它们的抽象概念之间似乎并不存在层次关系的情况下，可以使用变体子类化。例如，用来控制鼠标的代码可能与用来控制图形输入板的代码几乎完全相同。但是，在概念上，没有理由让其中任何一个类来作为另外一个类的子类。因此，可以选择其中任何一个类作为父类，另外一个类继承于这个父茎类，并改写与设备相关的代码。

但是，通常使用的更好的替代方法是将两个类的公共代码提炼成一个抽象类，比如类 PointingDevice，并且让这两个类都继承于这个抽象类。与泛化子类化一样，当基于已存在的类创建新类时，就无法使用这种方法了。

6.8.8 结合子类化

结合子类化使用的场合是，需要一个子类，可以表示两个或更多的父类的特征的结合。例如，一个助教，既有教师的特征，也有学生的特征，因此在逻辑上可以表示成任何一个。可以继承于两个或多个父类的能力称为多重继承(multiple inheritance)。多继承的概念非常微妙复杂。

6.9 一些继承的变体

6.9.1Java 语言中的匿名类

偶尔会有这样的情况，就是程序员需要创建一个简单的、不多于一个实例的类。这样的对象称为单件(singleton)。Java 编程语言提供了创建这样的对象的一种机制，甚至不需要对这样的对象所涉及的类进行命名，因此这样的类称为匿名类(anonymous class)。

为了创建匿名类，需要以下几个条件：

1) 只能创建一个匿名类的实例。

2) 匿名类必须继承于父类或接口，并且不需要构造函数进行初始化。

6.9.2 继承和构造函数

对于 Java、C 卄和其他编程语言，只要父类构造函数不需要附加参数，父类的构造函数和子类的构造函数就都会自动地执行。当父类需要参数时，子类必须显式地提供参数。在 Java 语言中，这通过关键字 super 来实现。

6.10 继承的优点

6.10.1 软件可复用性

当继承另外一个类的行为时，提供这些行为的代码将不需要重新编写。虽然这是显而易见的事情，但它所隐含的意义却十分重要。许多程序员都要花费大量的时间来重写那些已经写过很多次的代码——例如，寻找符合特定模式的字符串，或者为表格增加新元素等常用操作。通过面向对象技

术，这些函数可以只编写一次就获得复用。

可复用代码的另外一个优点就是增加了程序的可靠性（代码复用的比例越高，发现错误的几率就越大），因此降低了由于多个用户共享同一代码所产生的维护成本。

6.10.2 代码共享

代码共享涉及面向对象技术的各个层面。其中一个层面就是，许多用户或项目可以使用同一个类。另外一种代码共享就是一个程序员在开发项目的一部分时，通过继承一个父类来创建两个或者多个子类。

6.10.3 接口的一致性

当两个或更多的类继承于同一个超类时，我们可以确保它们所继承的行为是相同的。这样，就更容易保证相似的对象具有相似的接口，并且用户可以避免将一些表面相似但行为和相互作用方式截然不同的对象组成一个混乱的集合。

6.10.4 软件组件

继承为程序员提供了一种构造可复用软件组件的手段，以实现几乎不需要编写多少代码就可以开发出新的应用程序这样的目标。现在已经有几个这种商用类库，预计将来会有更多的专用系统类库供我们使用。

6.10.5 快速原型法

当软件系统主要是由一些可复用的组件构成的时，主要的开发时间将

集中在了解系统的不常用的和新的组成部分。这样，软件系统就可以快速容易地完成。这种编程形式称为快速原型法编程或探索编程。开发完原型系统后，用户对其进行实验，然后基于第一个系统再创建第二系统，再进行新系统的实验等等，不断地进行多次的迭代实验。当开始一个软件项目时，如件系统的目标和需求只能进行模糊的把握时，这种开发方式将非常有用。

6.10.6 多态和框架

对于传统的软件开发方式，虽然在软件设计时可能是自上而下的，但在编码时通常都是邑自下向上的。即先要编写好低级例程，在此之上再建立稍高一级的抽象，然后再建立更高级的抽象。这个过程就像构建一堵墙，每块砖都必须放置在已经砌好的砖上。

通常，代码的可移植性随着抽象级别的提高而降低。也就是说，最低级别的例程可以用于多个不同的项目，下一级别的抽象可能还会得到复用，但是更高级别抽象的例程将与特定的应用程序紧密地结合在一起。低级别抽象的程序代码段可以用于新系统，并且一般仍然有意义。而更高级别的组件则只有构建在特定的低级别单元上时才有意义（由于声明或数据依赖关系）。

编程语言中的多态允许程序员创建高级别的可复用组件，通过对这些组件进行裁剪，可以适应各种不同的应用程序的变化。

6.10.7 信息隐藏

使用可复用组件的程序员只需了解组件的性质和接口，而不必了解用于实现组件的技术的详细信息。这样可以减少软件系统的相互关联。前面

我们曾提到，这种相互关联的特性是导致传统软件变得复杂的一个主要原因。

6.11 继承的代价

6.11.1 程序执行速度

继承方法由于必须处理相应的子类，因此程序运行速度通常比专用代码慢。但是，一味地关心程序运行效率则是错误的 O。首先，这种速度的差异通常很小。其次，执行速度的下降通常会通过软件开发速度的提高得以平衡。最后，大多数程序员实际上很少会了解他们所使用程序的执行时间主要花费在哪些地方。在项目开发的早期花费大量的时间来考虑程序的运行效率是不明智的，更好的做法是开发完可以投入运行的系统，并对其进行监控，找到程序的执行时间主要耗费在什么地方，再对这部分代码进行改进。

6.11.2 程序大小

除了为特定的项目构造的系统之外，任何软件库的使用都会增加程序的大小。虽然这种现象是实际存在的，但随着内存价格的降低，程序的大小变得越来越不重要了。现在，控制开发的费用和快速构建高质量、无错误代码比限制程序的大小更加重要。

6.11.3 消息传递的开销

很多事实已经证明，消息传递是一项在本质上比过程调用代价更高的操作。但是，与整体执行速度一样，过多地考虑消息传递的开销通常是一

种大头不算小头算的做法。

6.11.4 程序复杂性

尽管面向对象编程通常被吹捧为软件复杂度的解决方案，但实际上，对继承的过度使用通常会使软件由一种形式的复杂转向另一种形式的复杂。为了真正理解使用继承的程序的流程控制，需要对程序的整个继承图进行多次的遍历扫描。

第七章 静态行为和动态行为

面向对象语言的强大之处在于对象可以在运行时动态地改变其行为。因此，想要理解面向对象编程的原理，需要首先理解静态行为与动态行为之间的差异，以及这种差异的内在涵义。

对于编程语言来说，术语静态(static)几乎总是用来表示在编译时绑定于对象并且不允许以后对其进行修改的属性或特征。例如，静态类型变量意味着这个变量的类型在编译时得以确定，并且在程序的执行期间不能改变。另一方面，术语动态(dynamic)则用来表示直到运行时绑定于对象的属性或特征。这样，动态类型变量所拥有的类型决定于当前它所表示的数值，并且可以随着程序的执行而改变。

7.1 静态类型化和动态类型化

编程语言中最显著的分歧来自于静态类型语言与动态类型语言之间的差异。所有的语言都考虑了类型(type)这个概念。动态类语言与静态类型语言之间的差异在于变量或数值是否具备类型这种特性。

对于静态类型语言，类型在编译时绑定于变量。这通常发生在使用下面的声明语句时。

int a；//variable a will tnaintain integer values

double m； //wbrle variable m holds floating point quantities

一些静态类型语言则使用更微妙的类型推断(type inference)过程。非面向对象语言 ML 是使用这项技术的最著名的实例。对于舰，变量的类型是根据程序来决定的，这样，程序员就不需要提供变量类型的显式声明语句。

```
val a = (b + 2) / n;
//    b must be integer, since it is being added
//    to the integer constant 2.
//    therefore n must also be integer,
// as must be a
```

对于动态类型语言（有时也称为非类型语言，untyped language），类型决定于数值，而与变量无关。变量仅仅代表一个名称。在程序执行期间，不仅变量所代表的数值可以改变，而且变量所代表的类型也可以改变。

自从最早的编程语言出现开始，静态类型语言与动态类型语言之间就开始存在分歧。同时产生于 20 世纪 50 年代的 Fonran 语言和 Lisp 语言就是这两种类型的早期例子。关于这两种类型语言各自所具备的优点的争论几乎从未停止过。一般来说，争论的焦点主要是语言的目标应该实现高效性还是实现灵活性。

静态绑定一项语言特征意味着在编译时做太多的工作，因此在运行时只需完成少量的工作。例如，对于声明为静态的变量，可以在编译时就做出内存如何分配的决定，这样做一般比在每次变量改变数值时都重新进行内存分配效率要高一些。同样，静态编译时类型化禁止在程序执行前有任何的程序错误出现。动态类型语言的支持者则认为程序的灵活性比高效性更重要。

静态类和动态类

关于变量的静态类是指用于声明变量的类。静态类（就像名称所暗示的那样）在编译时就确定下来，并且再也不会改变。关于变量的动态类是指与变量所表示的当前数值相关的类。同样，如名称所暗示的那样，动态类在程序的执行过程中，当对变量赋新值时可以改变。

静态类型与动态类型之间的最重要区别如下：对于静态类型面向对象

编程语言，在编译时消息传递表达式的合法性不是基于接收器的当前动态数值，而是基于接收器的静态类来决定的。

```
Class Animal {
      Public void speak () { System.out.println(“Animal Speak ll’); }
}
Class Dog extends Animal {
      Public void speak () { bark(); }
      Public void bark () { System.out.println(“Woof !”); }
}
Class Seal extends Animal {
      Public void bark () {System.out.println(“Arf !”); }
}
Class Bird extends Animal {
      Public void speak () { System,out.println(“hreet l”);
}
```

```
Dog fido;
fido = new Dog();
fido . speak() ;    //    will bark
Woof !
fido.bark() ;    //    will bark
Woof !
Animal pet;
pet    = fido;    // legal to assign Dog to Animal
pet.speak(); // willwork
Woof !
pet .bark() ;    //    error, pets do not know bow to bark
```

7.2 运行时类型决定

替换原则可以通过提升数值在继承层次上的位置来体现。将 Dog 类型的数值赋值给类型为 Animal 的变量的同时，也将数值的类型从子类提升到父类。

有时也需要做与上面相反的事情：判断一种类变量目前所包含的数值是否导出类层次中的低层次类。例如，判断类型为 Animal 的变量所包含的数值是否为 Dog。

每种面向对象编程语言都具有执行这种测试的能力，但用于不同语言的语法则大不相同。这些不同类型的测试即使在动态类型语言中也非常重要，因此这些动态类型语言也具有这种检查能力。

```
C++
    Animal * aPet = …; // a pointer to an animal
    Dog * d =: dynamic_cast<Dog *>(aPet);
    If (d 1= 0) { // null if not legal, nonnull if ok
Delphi Pascal
    (if (typep aPet ‘Dog) … )
Java
    If (aPet instanceof Dog)
```

7.3 向下造型（反多态）

做出数值是否属于指定类的决定之后，通常下一步就是将这一数值的类型由父类转换为子类。这一过程称为向下造型(down casting)，或者反多态(reverse polymorphism)，因为这一操作所产生的效果恰好与多态赋值的效果相反。关于各种语言的向下造型操作的语法如下所示。需要注意的是，几种语言的函数在造型转换成功时返回有效结果，在造型转换非法时返回

空值，因此，在造型转换的同时也实现了类型测试。

```
C++
    Animal * aPet = …;
    Dog * d = dynamic_cast<Dog *>(aPet);
    If (d !=0) { // null if not legal, nonnull if ok
Java
    Animal aPet;
    Dog d;
    d = (Dog) aPet;
```

7.4 静态方法绑定和动态方法绑定

对于几乎所有的面向对象编程语言来说，在响应消息时对哪个方法进行绑定是由接收器当前所包含的动态数值来决定的。

在有些编程语言中，改写的规则要比其他语言更复杂一些。C 卄语言在响应消息时所进行的方法绑定不论在关键字的数目上，还是在声明方法上都比其他语言复杂。

```
Class Animal {
Public :
      virtual void speak () { cout<< “Animal Speak ! \n”; }
      void reply () { cout<<   “Animal Reply ! \n”; }
};
Class Dog : public Animal {
Public:
      Virtual void speak () { cout<< “woof ! \n”; }
      Void reply () { cout<<   “woof again! \n”; }
};
Class Bird : public Animal {
Public :
      Virtual void speak () { cout 岌  “t\reet l\n”; }
};
```

如果方法所执行的消息绑定是由最近赋值给变量的数值的类型来决定的，那么我们就称这个变量是多态(polymorphic)的（对于 Smalltalk、Java 和大多数其他面向对象语言来说，从这种意义上来讲，所有变量都是多态的）。而对于 C++语言的声明为简单类型的变量，在这种意义上则不是多态的。C++语言支持引用的概念，引用是一种可以保证指向有效对象的简单的指针类型。引用也是多态的。Virtual 这个关键字是非常重要的，如果省略这个关键字，那么通过指针所实现的对象引用就将不再是多态的了。、在 C++语言中，只有使用指针或引用，并且只有当相关的方法声明为 virtual 时，才可以实现多态消息传递。为了理解 C++语言的多态规则比其他编程语言更加复杂的原因，需要对面向对象多态变量与内存管理之间的关系进行研究。

尽管 C#和 Delphi Pascal 语言不支持引用，用来决定消息表达式含义的规则也不如 C++语言那样复杂，但是这两种语言也需要关键字 virtual。

第八章 替换的本质

继承和替换原则的引人影响了几乎所有编程语言的重要概念，包括类型系统、赋值的含义、等价的测试、复制的建立、存储分配等各个方面。

8.1 内存布局

让我们研究一下这个看起来似乎很简单的问题，不同的回答将导致不同的方向：当局从特定的类实例化对象时需要多少存储空间？举一个具体的例子：如果变量win声明为特定类Window的实例，如果我们创建Window类的子类，例如 TextWindow，由于，TextWindow 是一个 Window，因此通过 win 变量来保存类型为 TextWindow 的数值也一定是合法的。创建类 Window 的实例化变量 win 需要分配多少存储空间？

关于这个问题至少有三个可能的答案。

1)只分配基类所需的存储空间。即，对于变量 win，只分配由类 Window 所声昨明的数据区域所占用的空间，忽略那些由 Window 的子类所声明的数据区域所占用的空间。

2)无论是基类还是子类，都分配可以用于所有合法的数值的最大的存储空间。

3)只分配用于保存一个指针所需的存储空间。在运行时通过堆来分配数值所需的存储空间，同时将指针设为相应的合适值（即存储空间的地址）。

8.1.1 最小静态空间分配

C 语言被设计成运行时高效的语言。通常认为，基于堆栈的内存地址

分配的程序的执行速度比基于动态变量的程序的执行速度要快。顺理成章地，C++语言作为 C 语言的继任者，同时保留了动态变量（运行时分配内存）和非动态变量的概念。

C++语言在变量的声明方式和指针存取变量值的方式上与 C 语言不同。例如，在下面的代码中，变量 wm 在堆栈中进行内存分配。当进入声明这个变量的过程时，程序会为这个变量分配相应大小的堆栈空间。这一存储空间的大小为基类 Window 所占用的存储空间的大小。另一方面，变量 tWinPtr 则只是包含一个指针。当程序执行到 new 语句时，会为 tWinPtr 指针指向的数值动态分配存储空间。由于此时已经知道 TextWindow 类占用的存储空间的大小，因此通过堆分配相应大小的存储空间，就可以保存一个 TextWindow 类型的对象。

```
Window win;
Window *tWinPtr;
tWinPtr = new TextWindow;
```

当执行下面的语句时，结果会怎样？

```
win = *tWinPtr;
```

分配给 win 变量的存储空间只适合于容纳一个 Window 对象，而 tWinPtr 所指向的对象的尺寸却大于 win 变量的尺寸（如图 12-1 所示）。显而而易见，以上的赋值操作无法将 tWinPtr 所指向、 的数值完全复制给变量 win。缺省的行为是仅仅复制相应的数据字段（在 C 廾语言中，程序员可以改写赋值操作符的含义并提供所希望的语义，因此这里所说的行为是指在程序员没有改写赋值操作符条件下的缺省行为）。这样显然会丢失一部分信息（保存于 tWinPtr 的额外字段中）。由于在上面的赋值过程中，右边变量的某些字段被舍弃，不再出现于左边变量的数值中，因此，有些学者使用切割（slicing）这个术语来描述这个过程。

丢失这些信息的后果严重吗？C++语言的语义保证了变量 win 只能调用定义于 Window 类中的方法，不能调用定义于 TextWindow 类中的方法。定义并实现于 Window 类中的方法无够存取或修改定义于子类中的数据，因此不可能出现父类存取子类的情况。

C++语言设计者对这种进退两难的问题的解决方案是改变与过程相关的虚拟方法的调用规则。新规则总结如下：

- 对于指针（和引用）变量：当消息调用可能被改写的成员函数时，选择哪个成员函数决于接收器的动态数值。
- 对于其他变量：关于调用虚拟成员函数的绑定方式取决于静态类（变量声明时的类），而不取决于动态类（变量所包含的实际数值的类）。

8.1.2 最大静态空间分配

对于声明对象需要分配多大的存储空间这个问题的另外一种解决方案是，不论是父类变量还是子类变量，都分配变量值可能使用的最大存储空间。这种方法类似于传统语言中所使用的内存重叠分布的数据类型，例如 Pascal 语言中的记录数据类型和 C 语言中的联合数据类型。

这种方案看起来好像很理想，但它忽视了这样一个问题：只有当整个程序运行完之后才会知道所有对象的大小。这样，在决定一个对象的存储空间前，必须先对模块（即 Object Pascal 语言中的单元、C++语言中的文件）以及整个程序进行扫描。由于这种要求太严格了，因此在主要的面向对象编程语言中，都没有使用这种方法。

8.1.3 动态内存分配

第三种方法是在堆栈中根本不保存对象值。在开始执行过程时，只是在堆堆栈中通过一个指针大小的空间来保存一个标识变量。数值则保存于另外一个数据区间——堆，堆存储空间不支持堆栈存储空间的先进后出的分配规则。由于所有指针变量都具有恒定不变的大小，因此当把子类变量值赋值给父类变量时，不会产生任何问题。

Object Pascal、Smalltalk. Java 和 Objective-C 语言都使用这种方法。通过 Object Pascal 语言中对象和指针的相似性，读者可能已经猜到了这一点。对于指针和对象，在操作对象之前，都必须调用标准的 new 过程来分配存储空间。同样，程序员必须通过显式地调用 free 过程，才能释放对象所占用的存储空间。这种技术除了需要用户显式地分配内存以外，还涉及到变量赋值时指针语义(pointer semantics)的应用。当使用指针语义时，通过赋值语句传递的数值为指针的数值，而不是指针所指向的数值。

8.2 赋值

编程语言所使用的内存分配方法会影响到赋值的含义，下面我们对各种编程语言中赋值操作符的确切含义进行一下总结。对于赋值可以给出两种解释。

- 复制语义（copy semantics）：赋值会将操作符右侧的变量值复制给操作符左侧的变量。此后，这两个变量值是互相独立的，其中一个变量值的改变不会影响到另郊外一个变量值。复制语义有时用在 C++语言中，有时则不是这样。
- 指针语义（pointer semantics）：赋值会将操作符左侧变量的参考值改变成右侧变量的参考值(这种方法有时也称为指针赋值

(pointer assignment))。这样，两个变量不仅具有相同的数值，而且还指向存储数值的同一内存地址。一个变量值的改变会同时改变两个变量的数值，这可以通过不同的变量名称得以反映。Java、CLOS、Object Pascal 语言以及许多其他的面向对象语言都采用指针语义。

编程语言在使用指针语义时，通常都会提供某种方法来实现变量值的完全复制。而且，对于对象内存的分配，与堆栈中自动分配内存的方式相比较，通过堆来为对象动态分配内存更多地采用指针语义。在使用指针语义时，常常会出现变量值存活于创建这个变量的上下文之外的情况。各种面向对象编程语言在使用这两种语义的方式上各有不同：有的使用其中的一种，有的使用另外一种，还有的使用两者相结合的方式。

8.3 复制和克隆

当语言使用指针语义进行赋值时，几乎总是存在一种方法来对一个变量值进行复制或克隆。当对指向其他对象的变量值进行复制时，有两种可能的方案。一种是与原来变量共享实例变量的浅复制(shallow copy)，即原有变量和复制产生的变量引用相同的变量值。另一种方案就是深复制(deep copy)，这种方式将建立实例变量的新的副本。

8.3.1C++语言中的拷贝构造函数

C++运行时系统经常创建变量值的副本来作为过程的临时变量或参数。用户可以自己通过定义拷贝构造函数(copy constructor)来控制这个过程。拷贝构造函数的参数是一个与这个类本身类型相同的引用参数。为类创建拷贝构造函数通常被认为是一项良好的编程习惯。

8.3.2Java 语言中的克隆

在实际编程过程中，我们常常要遇到这种情况：有一个对象 A，在某一时刻 A 中已经包含了一些有效值，此时可能会需要一个和 A 完全相同新对象 B，并且此后对 B 任何改动都不会影响到 A 中的值，也就是说，A 与 B 是两个独立的对象，但 B 的初始值是由 A 对象确定的。在 Java 语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但实现 clone（）方法是其中最简单，也是最高效的手段。

Java 的所有类都默认继承 java.lang.Object 类，在 java.lang.Object 类中有一个方法 clone()。JDK API 的说明文档解释这个方法将返回 Object 对象的一个拷贝。要说明的有两点：一是拷贝对象返回的是一个新对象，而不是一个引用。二是拷贝对象与用 new 操作符返回的新对象的区别就是这个拷贝已经包含了一些原来对象的信息，而不是对象的初始信息。

Java 语言中的 Object 类定义了名为 clone 的方法，但是，这个方法声明为 protected。为了使用这个方法来使对象可以被复制，程序员必须做两件事情。首先，程序员必须将他编写的类声明为支持 Cloneable 接口。其次，必须改写 clone 方法，并且将其定义为 public。这个方法有可能会抛出 CloneNotSupportedException 错误，这也必须作为方法声明的一部分。实际上，这个方法除了调用继承于父类的方法之外，不需要再做其他任何事情。

下面的例子包含三个类 UnCloneA，CloneB，CloneMain。CloneB 类包含了一个 UnCloneA 的实例和一个 int 类型变量，并且重载 clone()方法。CloneMain 类初始化 UnCloneA 类的一个实例 b1，然后调用 clone()方法生成了一个 b1 的拷贝 b2。最后考察一下 b1 和 b2 的输出：

```
package clone;
class UnCloneA {
```

```
    private int i;
    public UnCloneA(int ii) { i = ii; }
    public void doublevalue() { i *= 2; }
    public String toString() {
        return Integer.toString(i);
    }
}
class CloneB implements Cloneable{
    public int aInt;
    public UnCloneA unCA = new UnCloneA(111);
    public Object clone(){
        CloneB o = null;
        try{
            o = (CloneB)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return o;
    }
}
public class CloneMain {
    public static void main(String[] a){
        CloneB b1 = new CloneB();
        b1.aInt = 11;
        System.out.println("before clone,b1.aInt = "+ b1.aInt);
```

```
        System.out.println("before clone,b1.unCA = "+ b1.unCA);


        CloneB b2 = (CloneB)b1.clone();
        b2.aInt = 22;
        b2.unCA.doublevalue();
        System.out.println("================================");
        System.out.println("after clone,b1.aInt = "+ b1.aInt);
        System.out.println("after clone,b1.unCA = "+ b1.unCA);
        System.out.println("================================");
        System.out.println("after clone,b2.aInt = "+ b2.aInt);
        System.out.println("after clone,b2.unCA = "+ b2.unCA);
    }
}
        /** RUN RESULT:
before clone,b1.aInt = 11
before clone,b1.unCA = 111
================================
after clone,b1.aInt = 11
after clone,b1.unCA = 222
================================
after clone,b2.aInt = 22
after clone,b2.unCA = 222
*/
```

输出的结果说明 int 类型的变量 aInt 和 UnCloneA 的实例对象 unCA 的 clone 结果不一致，int 类型是真正的被 clone 了，因为改变了 b2 中的 aInt

变量，对 b1 的 aInt 没有产生影响，也就是说，b2.aInt 与 b1.aInt 已经占据了不同的内存空间，b2.aInt 是 b1.aInt 的一个真正拷贝。相反，对 b2.unCA 的改变同时改变了 b1.unCA，很明显，b2.unCA 和 b1.unCA 是仅仅指向同一个对象的不同引用！从中可以看出，调用 Object 类中 clone()方法产生的效果是：先在内存中开辟一块和原始对象一样的空间，然后原样拷贝原始对象中的内容。对基本数据类型，这样的操作是没有问题的，但对非基本类型变量，我们知道它们保存的仅仅是对象的引用，这也导致 clone 后的非基本类型变量和原始对象中相应的变量指向的是同一个对象。大多时候，这种 clone 的结果往往不是我们所希望的结果，这种 clone 也被称为“影子 clone”。要想让 b2.unCA 指向与 b2.unCA 不同的对象，而且 b2.unCA 中还要包含 b1.unCA 中的信息作为初始信息，就要实现深度 clone。 默认的克隆方法为浅克隆，只克隆对象的非引用类型成员。

把上面的例子改成深度 clone 很简单，需要两个改变：一是让 UnCloneA 类也实现和 CloneB 类一样的 clone 功能（实现 Cloneable 接口，重载 clone()方法）。二是在 CloneB 的 clone()方法中加入一句 o.unCA = (UnCloneA)unCA.clone();

8.4 相同

与赋值一样，决定一个对象与另外一个对象之间是否相同比我们想像的要更复杂。关于这一问题，有两点事项需要考虑。其一，“相同”和“同一”之间的区别。其二，如果允许类重定义“相同”的含义，那么，在这种情况下，会遇到由替换上下文所引起的悖论。

8.4.1 相同和同一

有一个古老的笑话，大概情形是这样的：一个男人走进一家比萨饼店坐了下来，一名服务员来到他的桌旁询问他想要点什么。这个人环顾了一下整个房间，然后，指着坐在相邻餐桌旁的一位妇女说：“我想要一份她正在吃的比萨。”然后，服务员走到那位妇女的桌旁，拿起她面前已经吃了一半儿的比萨饼，送到这个人的面前，服务员的举动令他感到非常震惊。

这个故事令人发笑的原因在于它混淆了两个相关但却截然不同的概念：相同(equality)和同一（identity）。当询问关于同一的问题时，我们想要知道两个对象是否严格表示同一个实体。这就像有人问：“早晨的星星和晚上的星星是否一样呢？”（早晨的星星和晚上的星星都是金星的名字。）

另一方面，当那个男人说他想要一份那位妇女正在吃的比萨时，他并不是想要那位妇女正在吃的比萨，而是想要各种特征都和她所吃的比萨相同的另一份比萨。

尽管相同与同一这个问题，在面向对象语言中似乎显得更为普遍，但是，它并非起源于面向对象语言。

例 C 语言：

```
char *x=“abc”;
char *y=“abc”;
if (x == y) …
```

这个错误是因为，虽然两个指针地址所存储的数值相同，但是，这两个指针引用的内存地址却不同。

C 语言和 C++语言用来检测同一的双等号操作符也适用于 Java、C#和 Objective-C 语言。

8.4.2 相同检验的悖论

虽然很容易定义同一，但是同一的意义却是与特定领域相关的。例如，如果两个字符串有着相同的特征表示，它们是否应该看作是相同呢？对于两个三角形会怎样呢？如果两个不同的对象有着相同的边长，它们是否应该看作是相同呢？关于这些问题，正确答案并不惟一，并且每个问题（和每个程序员）要根据具体的情况来判断。

为了适应这一灵活性，许多面向对象语言允许程序员定义相同检验操作符的含义。相同检验只不过是这样一条消息：将另外一个值作为参数传给一个给定值。由于消息传递是非对称的，因此，如果 a 等于 b，无法保证 b 等于 a。

对于那些所有类都继承自公共的根类的语言来说，相同检验操作符通常都在根类中定义，并且可以在子类中改写。什么类型应该与参数值相联系？通常总是通过将参数声明为根类来解决这一问题。这样，子类被强迫使用同一类型签名，因此，在进行有意义的比较之前，必须首先检验参数类型并进行类型转换。例如，在下面 Java 语言的类定义中，方法 equajs（继承自 Object 类的方法）截的参数必须声明为 Object。但是，程序员只关注纸牌的检验。因此，在进行正式的比较之前，必须对其进行转换。

```
Class PlayingCard extends Object {
    Public Boolean equals (Object right) {
        If (right instanceof PlayingCard) {
            PlayingCard rightCard = (PlayingCard) right;
            // do card-card comparison
            return (rank = rightCard.rank) &&
            (suit == rightCard.suit);
```

```
            }
            Return false;   // false on a/l otber comparrsons
        }
    }
```

由于可以在类层次的不同级别实现对相同操作符的改写，因此，产生了许多奇怪的悖论。例如，假定有两个类：Parent 和 Child，它们都提供验证相同的消息。因为每个类都可以为任何方法自由地提供它所希望的含义，因此，大体上，我们可以说，不关心这两个方法之间的关系。这样，就很容易发生下面所述的几种情况。

- 假定 p 代表 Parent 类的一个实例，c 代表 Child 类的一个实例。因为验证 p 是否等于 c 是由父类的方法所实现的，而验证 c 是否等于 p 是由子类的方法所实现的，因此，可能会出现一个为真而另外一个为假的结果。
- 类似地，假定有两个子类的实例：cl 和 c2。可能会出现这种情况，通过父类的方法得出 p 既等于 cl 又等于 c2，但是，使用子类的方法比较 cl 和 c2 却可能会返回假。
- 反之，假定有另外一个父类的实例皿。可能会出现这种情况，p 等于 c，c 等于 p2，但是，p 不等于 p2。

第九章 多重继承

继承概念的核心就是“是一个”关系。从某个角度来看，“是一个”关系是分类的一种形式。但是，当我们观察现实世界中的对象分类时却发现，这些对象很少会符合我们所构造的单一继承关系。这是因为，现实世界中的对象几乎总是以一种多重、互相不重叠的方式进行分类的。例如，本书的作者就可以描述成一位男性、一位计算机科学教授、一个父亲等等。这些分类中的每一种都揭示了作者的某一方面特性，而且把任何一个分类作为另外一个分类的子集都不恰当。如果我们尝试着将这些分类构造成一、个继承层次，结果将会形成无效的分类。例如，将所有的男性归属于教授之下，或者将所有的教授归属于男性之下，或者其他一些同样离奇的分类方式都是不科学的。

9.1 多重继承带来的问题

9.1.1 名称歧义

使用多重继承所带来的最常见的困难就是名称可以用来表示多项操作。为了证明这一点，我们来探讨程序员开发的纸牌游戏。假定已存在一个数据抽象 CardDeck，它提供了与一副纸牌相关的各种功能（例如洗牌、从一副纸牌当中抽取一张纸牌），但却不包含图形功能。假定还存在另外一套可以实现图形对象的类。图形对象需要在二维显示界面上占据一定的位置。而且，所有图形对象都必须知道如何通过 draw 这个虚拟方法来显示自己。为了最大限度地利用这两个已经存在的类，程序员决定为这个新的抽象创建 GraphicalCardDeck 类，它继承于 CardDeck 和 GraphicalObject。很

明显，从概念上来看，类 GraphicaICardDeck 是一个 CardDeck，相应地，从逻辑上也派生自 CardDeck。同时，类 GraphicalCardDeck 也是一个 GraphicalObject。惟一的问题是 draw 这个命令的意义在两个类之间的冲突。

一种解决方案是总是使用全限定名。程序员特意指定自己希望使用的绘图类型，而布不是询问 GraphicaICardDeck 的实例该如何绘制。

```
GraphicalCardDeck gcd;
Card *aCard=gcd->CardDeck::draw();
gcd->GraphicalObject::draw();
```

然而，这种解决方案还不够理想，因为这种语法与其他函数调用的语法不同，并且程序员必须记住哪个方法来自于哪个类。

更常用的解决方案是重命名(renaming)和重定义(redefinition)的结合。通过重定义，我们可以改变命令操作的意义，例如在子类中对虚拟方法进行改写。通过重命名，我们可以改变方法的名称，这样，调用方法时就不会出现与该命令相关的命令功能警告。

```
class GraphicalCardDeck : public CardDeck, public GraphicalObject {
public:
   virtual Card*draw () { return CardDeck::draw(); }
   virtual void draw(Graphics *g) { GraphicalObject::draw(g); }
}
GraphicalCardDeck gcd;
Graphis g;
gcd->draw(); // selects CardDeck draw
gcd->draw(g); // selects GraphicalObject draw

class GraphicalCardDeck : public CardDeck, public GraphicalObject {
```

```
public:
    virtual void draw () { return CardDeck::draw(); }
    virtual void paint () { GraphicalObject::draw(); }
}

    GraphicalCardDeck gcd;
    gcd->draw(); // selects CardDeck draw
    gcd->paint(); // selects GraphicalObject draw
```

9.1.2 对替换的影响

对函数 draw 进行的名称重定义仅仅解决了单独使用 GraphicaICardDeck 类时的部分问题，当考虑替换原则的含义时，就会进一步产生问题。假定 GraphicaICardDeck 处于一个由图形对象组成的列表中，而这个列表的作用是显示窗口图像。当图形对象调用方法 draw 时，结果是执行 CardDeck 中对应的 draw 方法，而不是希望的图形操作。

```
GraphicalObject * g = new GraphicalCardDeck();
g->draw(); // opps, doing wrong method!
```

在 C++语言中，解决这一问题的典型方法就是引入两个新的辅助类。这两个类都继承自一个父类，并且使用不同的方法名称来重定义 draw 操作。Class CardDeckParent : public CardDeck {

```
Public :
    virtual void draw () { cardDeckDraw ();}
    virtual void cardDeckDraw () { CardDeck :: draw ();}
};
```

```
Class GraphicalObjectParent : public GraphicalObject {
Public :
    virtual void draw () { goDraw ();}
    virtual void goDraw () {GraphicalObject :: draw ();}
};
```

子类继承这些新的父类，改写相应的新方法。当独立使用子类时，新的子类对两个行为都可以访问。当以替换的方式对该对象赋值给任何一个父类的实例时，都会产生所希望的行为。

```
Class GraphicalCardDeck : public CardDeckParent, GraphicalObjectParent
{
Public :
    virtual void cardDeckDraw () {    }
    virtual void goDraw () {      }
};
```

注意，对于子类来说，方法 draw 仍然是歧义的，但是，当任何一个父类的实例引用这个对象时，方法 draw 则是明确的。

9.2 接口的多重继承

Java 和 C#语言都不支持类的多重继承，但它们都支持接口的多重继承。虽然接口的多重继承和类的多重继承有时作用很相似，但它们之间有着很重要的差异。两种机制都可用于分类化（对两者来说，是否符合“是一个”关系都是有效的验证继承的手段），但是，对于子类来说，接口不为其提供任何代码。正是由于这一原因，所以不会产生两部分继承代码之间的冲突。或者，两个父类接口中的方法具有相同的类型签名，此时，两个

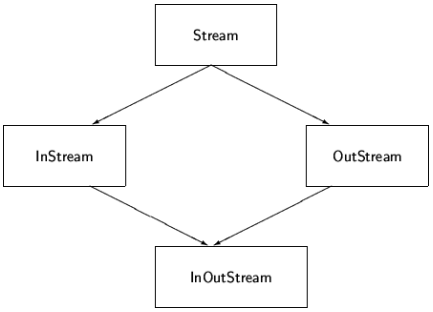
方法将得以合并，子类只需实现一个方法。

关于接口继承的惟一限制与重载时所遇到的问题很相似。即不能通过返回值的类型或抛出异常的类型来区分两个方法，因为此时编译器无法通过函数调用过程来判断究竟调用了哪个方法。在这种情况下，编译器会检测到错误，并发出诊断消息。

```
Interface CardDeck {
    Public void draw () throvs EmptyDeckException;
}
Interface GraphicalObject {
    public void draw ();
}
Class GraphicalCardDeck implements CardDeck, GraphicalObject {
    Public void draw  ()  {  …  } // which one is this?
}
```

9.3 继承于公共祖先

关于多重继承的最复杂问题发生于当两个或更多的类继承自公共的父类时，例如，如图所示。 Scott Meyers 戏剧化地称之为“消失的钻石”[Meyers 1998]。如果我们考虑一下这个公共祖先类所定义的数据成员，将有助于理解这两个图示之间的差别。子类应该拥有这些数据成员的一份还是两份拷贝呢？这个问题没有正确的答案，因为在实际中两种现象都存在。



例如，程序员想要创建一个链表，他首先创建了一个父类 Link，然后将链表中的对象类型声明为 Link 类的子类。

```
class Link{
public:
   Link *nextLink;
}
class CardDeck:public Link{..};
class GraphicalObject:public Link{..}
```

当创建同时继承自 CardDeck 类和 GraphicalObject 类的子类时，这个子类应该包含多少个 nextLink 字段？假如纸牌列表和图形对象列表是不同的，那么每种类型的列表都应该有各自的链接。因此，子类用于两个独立的链接字段看起来更为恰当。

现在，我们来考虑一下一种不同类型的类层次。程序员开发了一套基于流(stream)概念的输入，输出库。除了具有更多的结构外，流是文件的泛化。例如，可以有整数流，也可以有实数流。InStream 类为输人流提供了一套规则。用户可以通过附加到一个文件的方式打开一个输入流，并获取流的下一个元素，等等。OutStream 类为输出流提供了相似的功能。这两个类都继承自同一父类 Strcam。指向实际的幕后文件（例如文件指针）的信息包含于父类之中。现在，假设用户想要创建一个输入输出相结合的流。从感觉上来看，输入输出流既是输人流的派生，也是输出流的派生。

但是，在这种情况下，只存在一个对象，因此，我们希望两个文件指针（和其他公共数据）引用相同的数值。即只需要公共祖先的数据的一份拷贝。

在 C++语言中，这个问题通过在父类列表中使用 viItual 修饰符来解决。关键字 virtual 意味着在当前类的派生类中，超类可以出现多次，但只包含超类的一份拷贝。

```
class Stream{
        File *fid
}
class InStream : public virtual Stream{
        int open(File *)
..};
class OutStream : public virtual Stream{
        int open(File *)
..};
class InOutStream : public InStream, public OutStream {
        int open(File *)
..};
```

构造函数与多重继承

当多个父类定义了构造函数时，影响数据字段初始化的各个构造函数的执行顺序非常重要。用户可以通过在子类构造函数中直接调用基类构造函数来控制它。关于这条规则的一个例外发生在虚拟基类中。在其他任何初始化发生之前，虚拟基类总是通过无参数构造函数（由系统提供，而不是用户提供）进行一次初始化。这样，在调用非虚拟祖先的构造函数之前，首先调用虚拟基类的构造函数。

9.4 内部类

Java 语言和 C++语言所提供的嵌套类机制在某些场合的使用非常类似

于多重继承，但这种机制可以避免如前所述的许多语义问题。Java 语言中的嵌套类可以访问外部类的方法（C++语言禁止这样使用，但可以通过构建内部类时，传递外部类的引用参数来模拟这一行为）。为了创建一个继承于两个父类的对象，可以使外部类继承自第一个父类，而内部类继承自第二个父类。

```
Class GraphicalCardDeck extends CardDeck {
    public void draw ( ) { //外部类改写 CardDeck 方法
    }
    private drawingClass drawer= new drawingClass();
    public GraphicalObject myDrawingObject () {return drawer;}
    private class drawingClass extends GraphicalObject {
        public void draw ( ) {//内部类改写 GraphicalObject 方法
        }
    }
}
```

第十章 多态及软件复用概述

多态这个术语有着希腊词根，它的意思大概就是“多种形式”（poly = “许多”，morphos = “形式”。Morphos 与古希腊神 Morphus 有关，他在睡觉的时候，会展现多种不同的形态，因此，是名副其实的多态）。在生物学上，多态物种是指在一个独立的生物中或者不同的生物之间，可以表现出不同的形态或者颜色类型，例如现代人。寄生虫是另外一种多态物种。在化学上，多态化合物是指一种可以结晶成两种以上不同形式晶体的化合物，例如碳可以结晶成石墨、金刚石或球壳状碳分子。

10.1 编程语言中的多态

四种形式的多态。

- 术语重载(也称为专用多态(ad hoc polymorphism))用于描述一个函数名称（或者方法名称）有多种不同实现方式的情况。通常可以在编译时基于类型签名来区分各个重载函数的名称。
- 在某种意义上，改写(或者包含多态(inclusion polymorphism))是重载的一种特殊情况，但是，只发生在有关父类与子类之间关系的上下文中。与专用多态一样，对于改写，有两种使用同一方法名称的定义。然而，与重载不同的是，这两种定义必须具有相同的类型签名，并且，这两个方法必须出现在具有继承关系的两个类中。
- 多态变量(polymorphic variable) （或者赋值多态（assignment polymorphism））是指声明为一种类型，但实际上却可以包含另一种类型数值的变量。当多态变量用作参数时建立的函数将展现纯

多态(pure polymorphism)。

- 泛型（或模板）提供了一种创建通用工具的方法，可以在特定的场合将其特化。

10.2 软件复用机制

两种最常用的软件复用机制：继承(inhentance)和组合(composition)。

为了说明这两项技术，我们使用已存在的 List 类（包含整数值的列表）来构造一系列抽象。假定已经开发了具有下列接口的 List 类。

```
Class List {
    List( );
    void add (int);
    int firstElement ( );
    int size ( );
    int includes (int);
    void remove (int);
};
```

即，我们所建立的列表抽象允许在列表的起始位置增加新的元素，返回列表的起始元素，计算列表中元素的数目，检查列表是否包含某个数值，以及从列表中移走某个元素。

我们想要创建一个集合抽象，来执行这样一些操作，如增加数值到集合，确定集合中元素的数目，以及检查集合中是否包含某个特定的数值。

10.2.1 使用组合

我们先来研究如何通过组合(有时也称为分层(layering))来实现集合抽

象。从前面的讨论可知，对象仅仅是数据（数据值）和行为的封装。在通过使用组合复用已存在的数据抽象来创建新数据类型时，新数据结构的部分状态只是已存在的数据结构的一个实例。数据类型 Set 包含一个名称为 theData 且类型声明为 List 的实例字段。

10.2.2 使用继承

```
Class set:public List{
    public:
        //constructor
        Set();
        //operations
        void add(int);
        int size();
};
```

10.2.3 组合和继承的比较

- 组合是这两种技术中较为简单的一种。它的优点是，可以更清楚地指示出在特定的数据结构中需要执行哪些操作。通过观察 Set 数据抽象的声明，可以清晰地看到，列表这个数据类型所提供的操作仅仅是增加一个元素、包含检验和泛化集合大小等。我们无需考虑列表类所定义的所有操作。
- 在使用继承时，新数据抽象的操作是使其得以构建的原数据结构的操作的超集，这样，为了确切了解哪些操作对新数据结构是合法的，程序员必须检查原数据结构的声明。例如，通过检查 Set

类的声明，无法立即知道 includes 方法是否可以合法地应用于集合。只有通过检查更早的 List 数据抽象的声明，才可以确定可以用于集合的合法操作。但是，当程序员想要理解一个通过继承方式构建的类时，会有一些困难。程序员必须经常在两个（或更多个）类声明之间切换。当程序员在不同的类定义之间移动时，就像一种跳来跳去的舞蹈一样，因此称之为“yo-yo”问题。

- 使用继承构建数据抽象的代码的简洁性是继承的另一个优点。使用继承，我们不必为子类编写任何访问父类功能的代码。由于这个原因，对于实现一个案例来说，使用继承来：实现的代码几乎总是要短于使用组合实现的代码，并且，继承还通常提供更多的功能。例如，在前面的实例中，继承不但实现了 includes 检测，还实现了 remove 这个功能。
- 继承无法防止用户使用父类的方法来操纵新的数据结构，即使这个方法是不正确的。例如，当我们使用继承通过 List 类来建立 Set 类时，无法防止用户使用 frsirstElement 方法来获取第一个元素。
- 在使用组合时，List 类是作为集合的存储机制来使用的，这仅仅是一个技术实现的细节。通过组合这项技术，我们可以很容易地使用另一种不同的技术来重新实现 Set 类（例如使用哈希表），并且对 Set 抽象的用户影响很小。但是，如果用户更加看重这一事实，即 Set 仅仅是 List 的一种特殊形式，那么这样的改变就很难实现了。
- 继承允许将新的抽象作为已存在的多态(polymorphic)函数的一个参数来使用。
- 很难对可理解性和可维护性进行取舍。继承具有代码简练的优点，但规则不简练。组合的代码虽然更长，但是在其他程序员理解这

个抽象时，只需研究这部分代码即可。

- 由于可以避免附加函数调用（虽然在 C 升语言中，通过使用内联函数，可以消除这项开销），因此通过继承实现的数据结构在执行时会比通过组合构造的数据结构稍具优势。

第十一章 重载

11.1 类型签名和范畴

有关理解重载的一个非常重要的概念就是函数（或方法）类型签名（type signature）。前面已经讲过，函数签名是关于函数参数类型、参数顺序和返回值类型的描述。对于类定义的方法，类型签名通常不包括接收器类型。因此，我们可以说，父类中方法的类型签名可以与子类中方法的类型签名相同。

为了理解函数重载而需要了解的第二个概念就是范畴。范畴定义了能够使名称有效使用的一段程序，或者能够使名称有效使用的方式。本地变量，例如前面过程中的 result 和 i，只能用于声明该变量的函数内部。超出函数的反括号范围之外，该变量名称则不再有意义。对于一个类的数据成员，只有声明为 public，该变量名称在类定义的外部通过类实例名称限定（或者，对于静态变量，通过类名称限定）才有意义。

通过继承创建的新类将同时创建新的名称范畴，该范畴是对父类的名称范畴的扩展。因此，对于不同的编程语言，形成名称范畴的方式也各不相同。

我们可以通过类型签名和范畴这两个概念将重载进行两种形式的分类。一种分类是基于具有不同范畴的方法，而不考虑其类型签名。例如，同一名称可以用于两个或更多个互不相关的类中。另外，重载的分类也可以基于具有不同类型签名的方法，而不考虑其范畴。

11.2 基于范畴的重载

两个不同的函数可以使用相同名称的本地变量，由于它们的范畴之间并不重叠，因此不会产生任何混乱。方法之间使用相同名称的本地变量，也不会产生混乱或歧义。

11.3 基于类型签名的重载

另外一种方式的重载发生在这样一种情况下：多个过程（或者函数、方法）允许共享同一名称，且通过该过程所需的参数数目、顺序和类型（对于静态类型语言）来对它们进行区分时。即使函数处于同一上下文，这也是合法的。惟一的要求就是每种实现方式都必须具有不同的类型签名。这意味着，通过函数参数的不同，可以在编译时明确地确定用户将要调用的函数。

在 Delphi Pascal 语言中，处于同一范畴的两个方法可以重载同一名称，但是程序员必须使用 overload 指令显式地声明这一事实。

需要重点指出的是，关于重载的解析（即选择与特定调用相匹配的方法体），是在编译时基于参数值的静态类型完成的。与改写不同，重载不涉及运行时机制。

强制和转换

对于所有的编程语言（不仅是对于面向对象语言），都会经常看到基于类型签名的重载。最常见的一个例子可能就是对相加操作符“+”的重载。虽然编译器为整数相加所生成的代码与为浮点数相加所生成的代码截然不同，但是程序员仍然倾向于将这一操作符看作是一个实体——“相加”函数。

强制(coercion)、转换 Iconversion)和造型(cast)

由于术语“强制”和“转换”都表示类型的变化，因此它们之间很容易混淆。强制是一种隐式的类型转换，它发生在无需显式引用的程序之中。关于这种类型转换的一个典型例子就是两个变量（一个声明为实数，一个声明为整数）之间的相加操作。整数变量值在相加之前会自动转换成实数值。

与前者相反，术语“转换”通常用来表示程序员所进行的显式类型转换。在许多语言中，用来执行这种转换的操作称为“造型”。C 卄或 Java 语言中的造型如下所示。

x=（(double）i) + x;

通常，造型和转换既可以寥实现基本含含义的改变（例如将整数转换成实数），也可以实现类型的转换，而保持含义不变（例如将指向子类的指针转换成指向父裴类的指针）。

如果允许混合类型的算术运算，那么两个数值之间的相加操作可以至少有以下三种解释方式：

- 存在四种不同的函数，分别对应于“整数+整数”、“整数+实数”、“实数+整数”和“实数+实数”。此时，使用的是重载而非强制。
- 存在两种不同的函数：“整数+整数”和“实数+实数”。在“整数+实数”和“实数+整数”时，整数值将强制转换成实数值。此时，使用的是重载和强制的结合。
- 只存在一种函数：　“实数+实数”。所有参数都强制转换成实数值。此时，只使用强制，不使用重载。

作为转换的替换。

如果我们将强制(coercion)这个词广义地解释为“类型的改变”，那么，替换原则将引入一种对于传统语言所不存在的另外一种形式的强制。这发生在类型为子类的数值作为实参用于使用父类类型定义对应的形参的方法

中时。类似的类型改变也可能发生于接口与实现这个接口的类的实例之间。

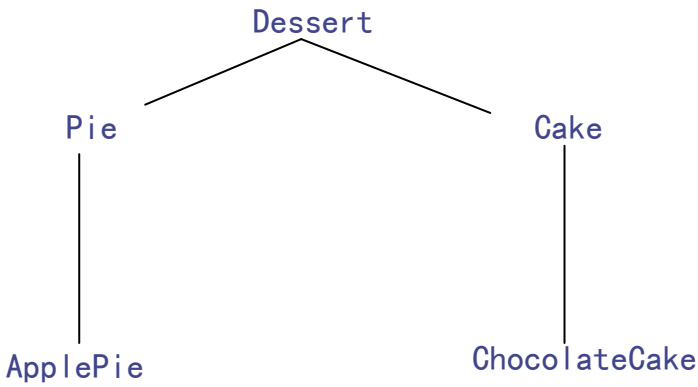
下面这个 Java 实例说明了关于继承与重载之间的某些细微的相互作用。对于 Java 语言，如果两个或者更多的方法具有相同的名称和相同的参数数目，则编译器将使用下面的算法来确定如何匹配。

1) 找到所有可能进行调用的方法一即，各个参数可以合法地赋值给各个参数类型的所有方法。如果找到一个在调用时可以完全匹配所使用的参数类型的方法，那么就执行这个方法。

2) 如果第一步所产生的集合中的任何方法的参数类型都可以赋值给集合中的任何其他方法，那么就将第二个方法从集合中移走。重复以上操作，直至无法实现进一步的缩减为止。

3) 如果只剩下一个方法，那么这个方法就非常明确了，调用这个方法即可。如果剩余的方法不止一个，那么调用就产生歧义了，此时编译器将报告错误。

为了说明这一过程，我们举一个例子，假如有如图所示的继承层次的 5 个类。假设在某一段代码中包含下列三个重载方法，每个方法都需要一对类型为 Dessert 的参数。



Void order (Dessert d, Cake c);

Void order (Pie p, Dessert d);

```
Void order (ApplePie a, Cake c);


order (aDessert, aCake);//执行方法一
order (anApplePie , aDessert);//执行方法二
order (aDessert , aDessert);//错误
order (anApplePie , aChocolateCake);//执行方法三
order (aPie , aCake);//错误
order (aChocolateCake, anApplePie );
order (aChocolateCake, aChocolateCake);
order (aPie , aChocolateCake);
```

由于 Dessert 和 Cake 与第一个方法的类型签名直接匹配，因此第一个函数调用是合法的。类似地，由于只有第二个方法的参数可以合法地接受第二个函数的实参的赋值，因此第二个函数调用能够与第二个重载方法相匹配（在第一个和第三个方法中出现的将 Dessert 类型的实参赋值给 Cake 参数是非法的）。

第三个函数调用实例是非法的。通过第一步筛选后，由于 Dessert 类型的数值不能任意地向下造型为更特殊的类型，因此可能调用的候选方法的集合就为空了，同时报告编译错误。

第四个函数调用就更加微妙了。经过第一步处理后，存在着三个候选方法。因此，选择方法的算法进行到第二步。在这一步中，首先，由于 ApplePie 是一个比 Dessert 更特殊的类(ApplePie 可以赋值给 Dessert，反之则不行)，并且第二个参数是一致的，因此第一个重载方法就被排除了。基于同样的原因（两个参数都不是更特殊的类型，Pie 不如 ApplePie 特殊，Dessert 不如 Cake 特殊），第二个方法也被排除了。由此，只剩下一个方法，因此，第四个函数调用将执行第三个方法。

与第三个方法类似，最后这个函数调用也是无效的。经过第一步处理之后，只有最后一个方法得以排除，可能调用的方法集合中还剩下两个方法。由于 Dessert 不能赋值给 Pie，因此无法将第一个方法归人第二个方法。同时，由于 Dessert 无法转换成 Cake，因此也无法将第二个方法归入第一个方法。这样，在第二步处理之后，留下两个方法，编译器无法决定使用它们之中的哪一个方法，于是就产生错误。

11.4 重定义

当子类定义了一个与父类具有相同的名称但类型签名不同的方法时，发生重定义。类型签名的变化是重定义区别于改写的主要依据。如果不是因为语言使用两种不同的技术解析重定义名称，使得粗心的程序员因不能识别这一差异而大吃一惊，重定义是不会引人注意的。这两种技术模型分别称为重定义融和(merge)模型和分级(hierarchical)模型。

Java 编程语言使用融和模型。此时，处于当前活动范畴的各种不同的含义都被融和到一起形成一个集合。为了符合使用重载的方法名称对特定方法体的调用，所有可能的方案都要进行检查，以选择最匹配的方案。

C++程序语言使用另外一种方法。在该语言中，每个范畴都维护自己独立的列表。为了使名称与方法匹配，需要依次对各个范畴进行检查，因此将其标识为分级的。然而而，当找到一个名称定义所存在的范畴时，就选择这一范畴中最合适的匹配。

```
class Parent {
    public void example (int a)
    {System.out.println(“in parent method”);}
}
class Child extends Parent {
```

```
        public void example (int a,int b)
        {System.out.println(“in child method”);}
    }
```

使用时

```
Child aChild = new Child();
aChild.example(3);
```

对于 C++语言来说，则会产生编译错误

C++每个范畴都维护自己独立的列表。

为了使名称与方法匹配，需要依次对各个范畴进行检查，因此将其标识为分级的。

通过在子类中重定义两个方法来实现与 Java 模型相同的效果

```
class Parent {
        public void example (int a)
        {System.out.println(“in parent method”);}
    }
    class Child extends Parent {
        public void example (int a)
       {Parent::example(a);}
        public void example (int a,int b)
        {System.out.println(“in child method”);}
    }
```

第十二章 改写

如果子类的方法具有与父类的方法相同的名称和类型签名，我们就说子类的方法改写(override)了父类的方法。当改写（也称为包含多态（inclusion polymorphism））与替换结合起来时，这项技术就非常重要了。使用替换原则的多态变量声明时具有一种类型（例如，父类型），但是实际上可以包含另外一种类型（通常是子类型）的数值。当对应于改写后的方法的一条消息传递给这样一个数值时，执行的过程将是子类提供的过程，而不是父类提供的过程。

在某种意义上，由于重载也涉及一个方法名称和两个或更多个方法体，因此改写可以看成是重载的一种特殊情况。然而，重载和改写之间的相似之处仅此而已。另一方面，重载和改写之间还存在着以下差异：

- 对于改写来说，方法所在的类之间必须符合父类／子类继承关系，而对于简单的重载来说，并无此要求。
- 如果发生改写，两个方法的类型签名必须匹配。
- 重载方法总是独立的，而对于改写的两个方法，有时会结合起来一起实现某种行为。
- 最后一点也是最重要的一点，重载通常是在编译时解析的，而改写则是一种运行时机制。这意味着，对于任何给定的消息，都无法预言将会执行何种行为，而只有到程序实际运行的时候才能对其进行确定。

12.1 标识改写

各种语言在如何通过代码实现标识改写这方面存在着差异。某些语言，

例如，Smamalltalk、Java 和 Objective-C 等完全不需要标识。在这些语言中，仅通过父类和子类方法类型签名的相似性来指示改写的存在。对于其例如 C++语言，则必须对父类进行标识，来预示着有可能发生改写（尽管这种标识不能保证一定会发生改写）。还有一些语言，例如 Object Pascal 语言，标识必须置于子类。在 Delphi Pascal 语言和 C#语言中，父类和子类都需要用关键字进行标识。

改写并不能改变方法的可存取性。如果一个方法在父类中为 public，那么不允许在子类中将该方法声明为 private，反之亦然（关于这项规则的一个例外就是 C++语言中的私有继承，此时，父类中的 public 特征在子类中可以变为 private）。

12.2 代替与改进

实际上，存在两种不同的关于改写的解释方式：

- 代替　在程序执行时，实现代替(replacement)的方法完全覆盖父类的方法。即，当操作子类实例时，父类的代码完全不会执行。
- 改进　实现改进(refinement)的方法将继承自父类的方法的执行作为其行为的一部分。这样，父类的行为得以保留且扩充。

由于改写含义的第一种解释通常与源自美国的语言（例如 Smalltalk 和 C++语言手）相关，因此一般将其称为美国语义(American semantic)。第二种解释则主要与最初的面向对象语言 Simula 以及后来的 Beta 语言这两种来自斯堪的纳维亚的语言相关，因此称之为斯堪的纳维亚语义(Scandinavian semantic)。

实际上，这两种形式的改写都很有用，并且，它们经常在一种编程语言内同时出现。例如，几乎所有的语言在构造函数中都使用改进语义。即，子类构造函数总是调用父类的构造函数，来保证父类的数据字段和子类的

数据字段都能够正确地初始化化。

改进与子类 / 子类型之间的差异

由于改进的使用保证了父类的行为得以保留，使得父类所执行的行为也必然是子类行为的一部分，因此，通过这种机制所创建的子类几乎不可能不是子类型。由此，支持使用改进语义语言的人们认为这种机制非常优雅。而对于使用代替语义的语言来说则无法保证这一点。

支持使用代替语义的人们则认为，这样的错误是很罕见的，而且很容易检测。如果采用代替语义作为缺省手段，就无法通过这种机制来实现代码复用。

12.3 延迟方法

如果方法在父类中定义，但并没有对其进行实现，那么我们称这个方法为延迟(deferred)方法。延迟方法有时也称为抽象(abstract)方法，并且，在 C++语言中通常称之为纯虚方法。

延迟方法的一个优点就是通过使用这些技术，可以使程序员在比挚际对象的抽象层次更高的级别上考虑与之相关的活动。例如，在表示几何形状的类的集合中，我们可以为每个子类 Circle. Square 和 Triangle 都定义一个方法 draw 来绘制自身的形状。我们可以在父类廷 Shape 中午定义类似的方法，但是由于 Shape 类并没有足够的信息来绘制其形状，因此这个方法姜实际上并不执行任何有意义的行为。然而，这个方法的存在允许用户将 draw 这个概念与 Shape 笼这个独立类关联起来，而不是将其与三个独立的概念 Square、Triangle 和 Circle 联系起来。

使用延迟方法还有第二个更具实际意义的原因。在静态类型面向对象语言（例如 C++、Object Pascal 等）中，对于给定对象，只有当编译器可以确定与给定消息选择器相匹配的响应方法时，才允许程序员发送消息给

这个对象。假定程序员希望定义一个关于类 Shape 的豹多态变量，以便在不同的时候保存不同的形状。根据替换规则，这样的赋值是可能的。然而，只有确偽保消息可以被与这个变量相关的任何数值所理解，编译器才允许该变量使用 draw 这条消息。将寄 draw 这个方法赋给 Shape 类则有效地提供了这种保证，虽然 Shape 类的这个方法实际上从不执行。

延迟方法几乎总是要与其他并;）<舀被改写的方法结合起来使用。这一思想平衡了代码复用（非改写方法）与特化（改写方法）之间的关系。非改写方法(有时称为基础(foundational)方法)提供的功能在不同的形势下可以通过延迟方法对其进行改写。

12.4 改写与遮蔽

由于改写机制与遮蔽这个与编程语言相关的概念之间存在着外在的语法相似性，因此此很容易将它们混淆。下面考虑遮蔽是如何与继承相联系的。一个很好的例子就是关于实例变量的。虽然并不鼓励如下面代码所示的这种使用方式，但它们在 Java 语言中是合法的。

子类中关于 x 的声明将遮蔽父类中对同名变量的声明。类似于重载，改写区别于遮蔽的最重要的特征就是，遮蔽是在编译时基于静态类型解析的，并且不需要运行时机制。

```
Class Parent {
Public int x = 12;
}
Class Child extend Parent {
Public int x = 42;    //sbadows variable from parent class
}
```

```
Parent p = new Parent();
System . out .println(p . x) ;
12
Child c = new Child();
System. Out .println(c .x) ;
42
p = c;    //    be careful bere!
System . out .println (P . x) ;
12
```

几种语言需要对改写显式指明，如果不使用关键字，将产生遮蔽。这可以通过下面的 C++语言中的类来说明。

```
Class Parent {
    Public :
    // note, no virtual keyword bere
    Void example () { cout<< “Parent”<<endl; }
};
Class Child : public Parent {
    Public :
    Void example () { cout<< “Child”<< endl; }
};
```

类似于 C++语言，如果忽略 override 指令，Delphi 语言将会遮蔽方法，但是编译器会发出警告。这种警告可以通过使用 reintroduce 指令来消除，这条指令显式地指示了程尾序员想要使用这个新方法来遮蔽原来的旧方法。

```
Type
```

```
TParent = class
    Public
        Procedure example (); virtual;
    End;
TChild = class (TParent)
    Public
        Procedure example (); reintroduce;
    End;
```

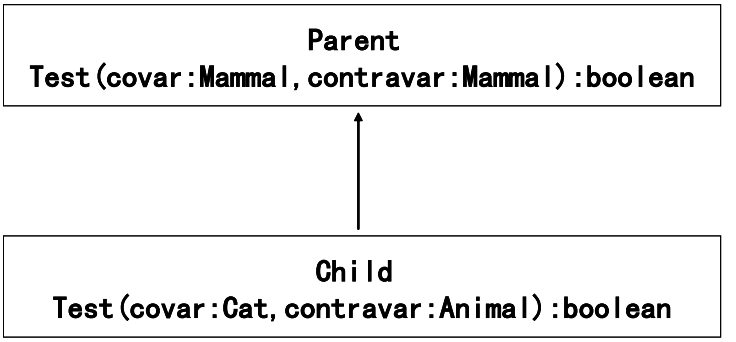
需要注意的是，不要将改写、遮蔽和重定义这三个概念混淆。以下总结了它们之间的差异。

- 改写　父类与子类的类型签名相同，并且在父类中将方法声明为虚拟的。
- 遮蔽　父类与子类的类型签名相同，但是在父类中并不将方法声明为虚拟的。
- 重定义　父类与子类的类型签名不同。

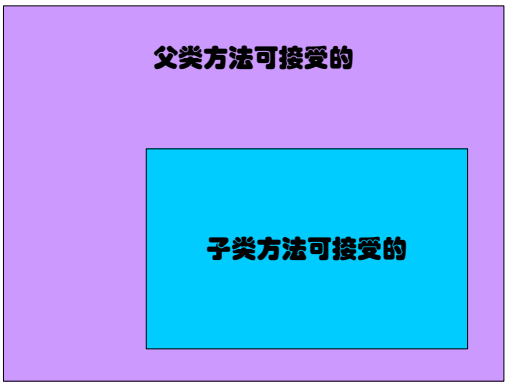
12.5 协方差与反协方差

当一个子类方法的类型签名不同于父类方法的类型签名时，如果可以使用改写，那么可能会很有用处。一个常见的例子就是用于比较两个对对象的 equals 方法。由于一个对象通常都是与同类型的对象进行比较，因此子类方法的参数类型应该为子类类型才有意义。然而，类型签名的改变会遇到很多困难。程序员很少会有任意改变类型签名的需求，通常都是将类型在其继承层次上提升或降低。当一个类型降低类型层次作为子类时，将使用协方差（covariant）变化这个术语。反之，当一个类型由子类化反向提升类型层次时，将使用反协方差(contravariant)这个术语。这两种情况显

示于下图所示的类层次关系中，此时，子类方法的第一个参数(名称为 covar)从 Mammal 类型变化成更特定的 Cat 类型，而第二个参数则以反协方差的方式变化成为更一般的 Animal 类型。



我们首先考虑如果子类可接受的数值集合范围小于父类可接受的数值集合范围（见下图），将会怎样。这是有可能的，例如，协方差变化作用于值传递参数时（父类有一个类型为 Mammal 的参数，子类限制同一个参数为 Cat 类型）。



当协方差变化遇到替换原则时，问题就产生了。根据替换原则，我们应该能够声明一个类型为父类型的变量，但是却将类型为子类型的数值赋值给它。

```
Parent aValue = new Child();
aValue.test( new Dog(),new Mammal());
```

在相反的情况下，则不会发生错误。此时，关于值传递参数的反协方差变化增加了子类可处理的数值范围。在这种情况下，父类和子类都无法将不可接受的数值传递给方法。

当我们考虑改变方法的返回值类型时，形势就完全不同了。假如父类的一个方法返回类型为 Mammal 的数值，并且子类包含一个将返回值类型扩展到 Animal 的反协方差变化，于是子类方法可以接受的数值集合就大于父类方法可以接受的数值集合。于是，我们就又遇到了关于替换原则的问题。对于子类，由于返回一个鸟类完全符合方法的类型限制条件，因此如果子类返回一个类型为 Bird 的数值是完全合法的。

```
class Parent {
    Mammal test ( ) {
        return new Cat ();
    }
}
class Child extends Parent {
    Animal test () {
        return new Bird ();
    }
}
Parent aValue = new Child();
Mammal result = aValue.test();
```

大多数语言设计者都通过使用一种称为非方差(novariance)的技术来避免这种问题。即，子类不允许以任何方式改变关于改写方法的类型签名。

12.6 改写的变体

12.6.1Java 语言中的 final 方法

有时父类则希望禁止子类对其方法的改写。即，程序员可能希望父类所定义的方法就是该方法的最终确定版本，不希望子类再对其进行任何改变。在 Java 语言中可以使用 final 修饰符来实要现这一目的。

C#语言也有一个类似的关键字 sealed，该关键字可以用于类，但不能用于单独的方法。

12.6.2C#语言中的版本化

如子类自己实现了虚拟方法，即，该方法不是继承自父父类。

```
class Parent{
    …
}
class Child : Parent{
    public virtual void aMethod(int){
    …
    }
}
```

后来，发布了一个新版本的父类，并且这个新版本（开发这个新版本父类的团体可能完全不同于开发原有子类的团体）包含了一个使用相同名称的方法。

```
class Parent{
    public virtual void aMethod(int){
```

```
        …
        }
    }
    class Child : Parent{
        public virtual void aMethod(int){
        …
        }
    }
```

这两个方法是不同的方法，但是几乎所有面向对象语言都认为它们是同一个方法。在 C#语言中，程序员可以在子类中增加 new 关键字。这个关键字的出现将重载改为重定义（即，子类的方法被看作是与父类的同名方法不同的另一个方法）。

```
    class Parent{
        public virtual void aMethod(int){
        …
        }
    }
    class Child : Parent{
        public new virtual void aMethod(int){
        …
        }
    }
```

第十三章 多态变量

多态变量(polymorphic variable)是指可以引用多种对象类型的变量。这种变量在程序执行过程可以包含不同类型的数值。对于动态类型语言，所有的变量都可能是多态的。对于静态类型语言，多态变量则是替换原则的具体表现。

13.1 接收器变量

包含接收器的变量没有被正常地声明，因此，通常称之为伪变量（pseudo-variable）。不同的语言以不同的名称来表示它。在 SmaDtalk 语言和 Object Pascal 语言中，用 self 来表示它。在 C 卄语言、C#语言和 Java 语言中，用 this 来表示。在 Eiffel 语言中，用 current 来表示。

13.2 多态变量在框架中的作用

多态接收器功能的强大之处表现在消息传递与改写相结合时。这种结合是软件框架开发的关键所在。例如，考虑一下典型的窗口化系统。我们将采用 Java 语言的 AWT 作为实例，而同样的原则几乎适用于所有的框架。这一系统中的方厂法可以分为两大类。在父类中定义基础方法。这些方法被多个子类所继承，但不被改写。同时，父类定义了关于多种活动的方法，但是要延迟到子类才实现。

由于基础方法被子类所继承，因此它们可以用于各个子类实例。接收器变量多态性的展现。当执行基础方法时，接收器实际上保存的是一个子类实例的数值。当执行一个改写方法时，执行的是子类的方法，而不是父

类的方法。

```
class Window{
    public void repaint(){
        …paint(graphicsContext);…
    }
    abstract public void paint(Graphics g); private Graphics graphicsContext;
}
class GraphicsWindow extends Window{
    public void paint(Graphics g){
        //do the appropriate painting job
    }
}
```

基础方法执行延迟方法的模式。该结合允许在不修改原始代码的条件下，裁剪延迟方法以适应新的形势。是解决软件复用问题的关键。

13.3 纯多态

许多作者都保留多态方法(或者纯多态(pure polymorphism))这个术语，用于一个支持可变参数的函数，同时，也保留重载(overloading)这个术语，用于一个名称对应多个函数定义的情况。这样的技术并不仅限于面向对象语言。例如，在 Lisp 语言和 ML 语言中，很容易编写可以处理列表或者任意元素的函数。由于在函数定义时参数类型是未知的，因此该函数为多态的。实现多态函数的能力是面向对象编程最强大的技术之一。它支持代码只编写一次、高级别的抽象以及针对各种情况所需的代码裁剪。通常，程序员都是通过给方法的接收器发送延迟消息来实现这种代码裁剪的。

关于纯多态的一个简单实例就是用 Java 语言编写的 StringBuffer 类中的 append 方法。这个方法的参数声明为 Object 类型，因此可以表示任何对象类型。该方法的大致定义如下所示。

```
Class Stringbuffer{
    String append(Object value){
        return append(value,toString();}
    …
}
```

方法 toString 被延迟实现。方法 toString 在子类中得以重定义。toString 方法的各种不同版本产生不同的结果。所以 append 方法也类似产生了各种不同的结果。Append;一个定义，多种结果。

第十四章 泛型

还有另外一种形式的多态，这种多态是通过泛型（generic）这个工具来提供的（在 C++语言中称为模板(template)）。泛法，与通常的函数参数一样，泛型提供了一种无需识别特定数值的定义算法的方法。

泛型将名称定义为类型参数。虽然在编译器读取类描述时，无法知道类型的属性，但是该类型参数可以像类型一样用于类定义内部。在将来的某一时刻，会通过具体的类型来匹配这一类型参数，这样，就形成了类的完整声明。在各种面向对象语言中，C++语言、Beta 语言和 Eiffel 语言中的泛型最为常见。

14.1 模板函数

```
Template <class T>
T max(T left,T right){
    //return largest argument
    if(left<right)
        return right;
    return left;
}
```

名称 T 是一个参数，但是它不同于函数的两个参数。在函数的代码体中，T 可以用于任何合适的类型。

对于哪些类型可以用来代替 T 这个参数有限制，但是这些限制并不是函数头强加的，勺，．1：而是函数代码体要求的。尤其需要注意的是，无论 T 为什么类型，都必须可以使用小于操作符来比较这一类型的两个实例。

因此任何符合这一条件的类型都可以使用。尤其是是对于基本类型型，能支持关系操作符。

为了调用模板函数，程序员只需编写普通的函数调用代码即可。模板参数通过参数类型来延迟实现。

14.2 模板类

尽管模板函数很有用，但是更常见的是模板参数用于整个类。假如存在一个模板类，它定义了一个几乎可以容纳任何类型数值的盒子的泛型。

```
Template <class T> class Box{
      public:
        Box(T initial):value(initial){}
        T getValue(){return value;}
        setValue(T newValue){value=newValue;}
      private:
        T value;
};
```

为了创建模板实例，模板参数必须与具体类型联系起来。

```
Box(int) iBox(7);
Cout<<iBox.getValue(); ???
iBox.setValue(12);
Cout<<iBox.getValue(); ???
```

参数必须与接收器的类型相匹配

```
iBox.setValue(3.1415);//ERROR,invalid type
```

模板类一般用于开发容器类

```
Template <class T> class List {
```

```
Public:
    void add ( T );
    T firstElement( );
    T value;
Private:
    List<T> * nextElement
```

14.3 案例研究

在开发实践中，我们经常会遇到这样的问题，就是在不允许改变原有类的情况下，如何将来自于两个或者更多个不同类的元素结合起来。例如，原有类可能是由两个独立的的软件厂商以二进制形式来提供的。但是，你却需要以统一的表示方式来处理它们，并且对其执行相同的操作。这个问题的解决方案是关于如何使用继承、模板、重载函数的一个非常好的说明，同时也是这些技术之间相互作用的一个说明。

例如，假如现有一些 Apple 和 Orange，它们来自于不同的公司。Apple 类能够使用 printOn(ostream＆)方法将自身输出到输出流，而 Orange 类也能使用名称为 writeTo(ostream&)的方法执行类似的操作。现在，要将 Apple 对象和 Orange 对象保存在同一个列表中，并且使用一个多态函数将它们写出到输出流。

第一步：定义一个公共的父类，具有共同行为

```
Class Fruit {
    public:
        virtual void print (ostream &)=0;//纯虚方法
};
```

第二布：使用模版，创建一个 fruit adapter 类，将以 Apple 或者 Orange

为参数，同时符合水果的定义

```
Template <class T>
Class FruitAdapter : public Fruit {
Public:
    FruitAdapter (T & f):theFruit (f){ }
    T & value ( ){ return theFruit ;}
    virtual void print ( ostream & out ){print(theFruit,out);}
Public:
   T & theFruit;
};
```

第三步：使用模版函数简化适配器的创建过程

```
Template <class T>
Fruit * newFruit ( T & f )
{
    return new FruitAdapter <T>(f);
};
```

最终方案：

```
Apple anApple("Rome");
Orange anOrange;

List<Fruit *> fruitList;

fruitList.insert(newFruit(anApple));
fruitList.insert(newFruit(anOrange));
```

```
List<Fruit *> ::iterator start = fruitList.begin();
List<Fruit *> ::iterator stop = fruitList.end();

For ( ;start!=stop; ++start) {
      Fruit & aFruit = *start;
    aFruit.print(cout);
}
```

第十五章 框架

面向对象软件框架(software framework)这一概念说明了应用继承和改写思想所具有的强大能力，以及面向对象界中的软件复用与传统编程语言中的有限的软件复用之间的差异。如果使用最基本的术语来表达，软件框架就是对于一类相似问题的骨架解决方案。框架的结构是通过类的集合形成的，这些类互相紧密地结合起来，共同实现针对问题的可复用解决方案。使用最广泛的应用程序框架就是用于创建图形用户界面(GUI)的框架。框架这一概念所能实现的功能远远超过了用户界面的开发。例如，框架可以用来构建用于不同场合的编辑器、编译器的构建以及创建财务模型应用程序等等。

15.1 复用和特化

框架开发的一个重要基础就是能够以两种完全不同的方式来使用继承。首先，它可以作为软件复用的工具，将代码抽象从一个项目带到另外一个项目。其次，继承同时提供了一种在父类和子类之间共享代码的能力，改写可以作为一种特化的工具来使用——即将一个通用的工具用于特定的任务。为了使各种情况适合于框架的建立，我们可以将一个类的方法分为两大类。

- 基本方法体现了对问题的现存的解决方案。它定义于父类，通过继承但却不对其改写的方式成为子类的一部分。
- 特化方法为了适合于子类的特定环境，改变了其在父类中的行为。这些方法通常都是延迟的方法。正是这些方法将父类的通用解决方案改变成用于特定应用程序的解决方案。

代码复用，概念复用：基本方法与特化方法之间的差异正是继承的各种功能的另外一种体现。推动继承技术发展的两个主要动力就是代码复用和概念复用。基本方法是指那些继承自父类的方法，这体现了对父类所提供的代码的复用。特化方法是指那些定义于父类但在子类中实现的方法，这种方法是概念复用的例子。

实例：雇员排序

```
class Employee {
public:
    string name;
    int salary;
    int startingYear;
}
```

如果存在关于这些记录的数组，那么可以通过他用例如插入排序这样的方法来对其进行排序。

```
void sort (Employee * data[ ], int n) {
    for (int i = 1; i < n; i++) {
        int j = i-1;
        while (j >= 0 &&
            v[j+1]->startingYear < v[j]->startingYear) {
            // swap elements
            Employee * temp = v[j];
            v[j] = v[j+1];
            v[j+1] = temp;
            j = j - 1;
        }
```

```
    }
}
```

现在，假如我们想要使用与其同名的函数，但目的稍有不同。首先，想不再根据起始年份遨行排序，而是根据收入进行排序。解决方案很简单，只需编辑这个方法的代码，将 startingYear 改为 salary 即可——但是，必须确保将该字段所出现的两个地方都进行替换。假如只改变了其中的一处，却没有改变另一处，将不会产生编译错误，但结果却完全错诺误。现在，假如在下一个项目中，将不再对雇员记录进行排序，而是需要对一个浮点数组进行排序。为了实现这种功能，需要改变函数标题声明、名称为 element 的内部数据值以及用于比较的存取操作。

需要注意的关键特征是，这两种变化都需要对源程序进行源代码(source code)级的修改。实际上，我们能够复用的是插入排序的思想，而不是其真正的实现。

现在，我们来考虑一下对于同一问题的面向对象解决方案。可能需要进行源代码改变的特征包括元素类型、元素数目、两个元素数值的比较、交换两个元素的过程等。由于想要改变这些特征，因此我们将它们封装到多个方法中。

```
class InsertionSorter {
public:
    void sort () {
        int n = size();
        for (int i = 1; i < n; i++) {
            int j = i - 1;
            while (j >= 0 && lessThan(j+1, j)) {
                swap(j, j+1);
```

```
                    j = j - 1;
                }
            }
        }
    private:
        virtual int size() = 0; // abstract methods
        virtual boolean lessThan(int i, int j) = 0;
        virtual void swap(int i, int j) = 0;
    }
```

为了使代码适合于特定问题，需要建立一个子类并实现延迟的方法。可以通过如下所示的方法来解决关于雇员资历的排序问题。

```
    class EmployeeSorter : public InsertionSorter {
    public:
        EmployeeSorter (Employee * d[], int n)
            { data = d; sze = n; }
    private:
        Employee * data[];
        int sze = n;
        virtual int size () { return sze; }
        virtual bool lessThan (int i, int j)
            { return data[i]->startingYear < data[j]->startingYear; }
        virtual void swap (int i, int j) {
            Employee * temp = v[i];
            v[i] = v[j];
            v[j] = temp;
```

```
        }
    }
```

基类不再需要改变。特化子类满足不同的需求。如：改变为对收入进行排序只需改变子类，无需改变父类。对浮点数进行排序也只需创建一个新的子类，而无需改变父类。继承允许进行高级别算法细节的封装。还允许在不改变原始代码的情况下修改或特化这些细节。

15.2 倒置库

框架改变了应用程序(开发者定义的代码)与库代码之间的关系。传统的应用程序中，应用程序特定的代码定义了程序执行的总体流程。在框架中，控制流是由框架决定的,并且随应用程序的不同而不同。新的应用程序的创建者只需改变供框架调用的例程即可,而无需改变总体结构。框架占主导地位,而应用程序特定的代码处于次要位置。

预言变化

面向对象设计艺术，为应用程序预言将来可能发生的变化,并对应用程序进行相应的设计。做到这点不容易，需要程序员认识到可以通过类似于以前解决问题的方式或者现存软件系统的方式来解决新问题时，才能够将该问题泛化，使其适合于更广泛的应用程序。像 C++,需要程序员区分哪些方法可以改写，以及哪些方法不能改写，这种方式过于僵硬了。由于程序员在最初无法预言改写某个方法的需求。

第十六章 对象互连

一种考虑对象互连的方式就是研究可视性(visibility)和依赖性(dependency)这两个概念。可视性这个软件工程术语描述了关于名称的特性——通过该名称句柄可以存取对象。如果对象的名称是合法的且代表该对象，那么在这个特定的环境下，该对象就是可见的。通常，用于描述可视性的相关疼聒还包括标识符的范畴（scope）。

可视性与连通性的关系体现在：如果我们能够控制并降低作为标识符的名称的可视性，那么就能更加容易确定应该如何使用标识符。例如，在Smalltalk 语言中，实例变量具有限制在方法内部的可视性，只能在方法的内部对其直接存取。虽然这并不意味着无法在类的外部对这种数值进行存取或修改，但是所有这种做法也都必须通过至少一个方法作为中介。另一方面，在 Apple Object Pascal 语言中，无论是否知道类名称，实例变量都是可见的。因此，编程语言将不提供保证，使实例变量的修改只能通过方法来实现。此时，我们必须依赖用户对变量的正确使用。

依赖性这个概念将两个对象或者类联系起来。在不存在另外一个对象的条件下，如果一个对象的存在无任何意义，我们就说该对象依赖于另外那个对象。例如，子类几乎总是依赖于它的父类。

16.1 耦合和内聚

耦合(coupling)和内聚(cohesion)的思想提供了一个框架，用于评价对象和类的应用是否有效。耦合描述了类之间的关系，而聚合描述了类内部的关系。如果想要降低类之间的互连性，可以通过减少类之间的耦合来实现。另一方面，设计良好的类应该具有特定的目的；所有元素都应该与一

个任务相关。这意味着在一个良好的设计中，类内部的元素应该具有内部的内聚性。

16.1.1 耦合的种类

从最差的耦合到较好的耦合如下面所示。

- 内部数据耦合
- 全局数据耦合
- 控制（或顺序）耦合
- 组件耦合
- 参数耦合
- 子类耦合

16.1.2 内聚的种类

从最弱的内聚（最少期望的）到最强的内聚（最期望的）依次排列的结果如下所示。

- 随机内聚
- 逻辑内聚
- 时间内聚
- 通信内聚
- 顺序内聚
- 功能内聚
- 数据内聚

第二篇 UML 基础与应用

第一章 UML 基础

1.1UML 的诞生

统一建模语言(Unified Modeling Language, UML)，如同它的名字所表明的那样，是一些早期面向对象建模语言的统一。UML 的三个主要设计师（Grady Booch.James Rumbaugh 和 Ivar Jacobson）在 UML 之前都曾经发表过他们各自的方法，UML 原来是为了通过将这三种方法的深入理解结合为一体，并发展可用的普遍公认的统一表示法来促进而向对象技术的传播而准备的。这些原有方法共享的公共框架，以及这三个当事人加了同一家公司，也促进了这种结合（1994 年，Rumbaugh 加入 Rational 软件公司，而 Booch 早已经在那里工作。第二年 Jacobson 也加入了 Rational 公司。）UML 草案版开始在软件工业界流传开来，并且根据大量的反馈信息做了大幅度修改。由于许多公司感到 UML 能够适应它们的战略目标，因此一个 UML 联盟蓬勃发展起来。联盟的成员包括 DEC、Hewlett-Packard. Intellicorp. Microsoft. Oracle. Texas Instruments. Rational 和其他一些公司。1997 年，应“对象管理组”(Object Management Group，OMG)向外界征求标准建模语言的建议，联盟制订了 UML l.0 版并提交给 OMG。后来联盟继续发展，产生了 UML l.1 版，提交给 OMG 后，于 1997 年被 OMG 采纳为标准。1998 年 OMG 接管了 UML 标准的维护工作，并且又制订了两个新的 UML 修订版。UML 成为软件工业界事实上的标准，并且仍在不断发展。UML 13 版、1.4 版和 1，5 版先后诞生，最近 OMG 正式批准了 2.0 版。

由于 UML 令人印象深刻的起源，以及对象管理组织（Object Management Group，OMG）将其作为标准采用所带来的促进，甚至在 UML

以权威性的形式公布之前，就在软件行业引发了极大的兴趣，而且可以预料，在可预知的将来，UML 将会作为主要的面向对象建模表示法继续下去。

UML 是一种可视化的建模语言，它能让系统构造者用标准的、易于理解的方式建立起能够表达出他们想象力的系统蓝图，并且提供一种机制，以以便于不同的人之间有效地共享和交流设计结果。

交流思想是极为重要的。在 UML 出现以前，系统开发往往是无计划的议题。系统分析员尽力去获取客户的需求，用某种他自己能够理解（但客户不一定总能理解）的表示法来产生需求分析文档，然后将这个分析文档转交给一个程序员或者一个程序员小组，并且期待着最后所开发出的系统正是客户所需要的。

由于系统开发需要人与人之间的交流，因此在开发过程的每个阶段中都很可能潜伏着错误。系统分析员可能没有正确地理解客户的需求。他编制的文档客户可能不能理解。系统分析员经常编写出语句冗长、内容庞大的需求文档，项目组的其他成员很难用上这些文档，这真是添乱。可笑的是，这些无足轻重的文档常常把重要的需求（以及需求之间的相关性）挤出人们的脑海之外。因此，系统分析的结果对程序员来说可能很不明确，随后程序员据此构造出的程序很可能不仅难以使用而且根本不是客户所需要的最初问题的解决方案。

1.2 多种视图

UML 可以通过称为 4+1 视图模型的软件系统结构来了解。这个模型的 UML 版本如图 1.1 所示，从图中可以清晰地看到，该模型得名于系统的结构通过五个视图描述的事实，挚宁用例视图具有将其他四个视图的内容结合到一起的特殊作用。

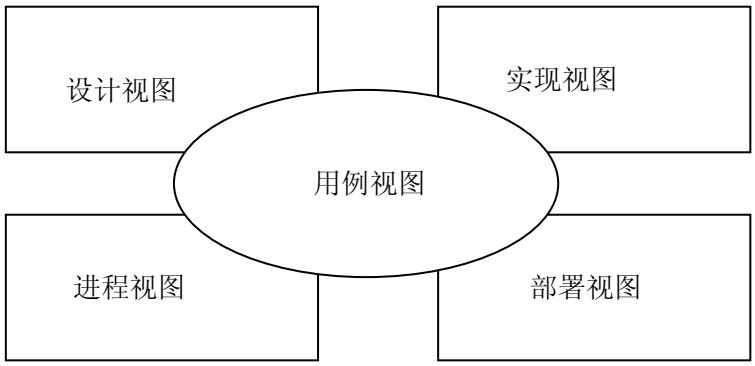


图 1.1　4+1 模型视图

图 1.1 中的五个视图并不对应于 UML 中描述的特定的形式构造或图，更恰当的是每个视图对应一个特定的研究系统的观点。不同的视图突出特定的参与群体所关心的系统的不同方面，通过合并所有五个视图中得到的信息就可以形成系统的完整描述，而为了特殊的目的，只考虑这些视图的子集中包含的信息可能就足够了。

用例视图(use case view)定义了系统的外部行为，是最终用户、分析人员和测试人员所关心的。该视图定义了系统的需求，因此约束了描述系统设计和构造的某些方面的所有其他视图。这正是用例视图具有中心作用的原因，也是通常所说的用例驱动开发过程。

设计视图（design view）描述的是支持用例视图图中规定的功能需求的逻辑结构。它由程序构件的定义，主要是类、类所持有的数据、类的行为以及类之间交互的说明组成。包含在这个视图中的信息是程序员特别关心的，因为如何实现系统功能的细节都在这个视图中描述。

实现视图（implementation view）描述构造系统的物理构件。这些构件不同于设计视图中描述的逻辑构件，这些构件包括如可执行文件、代码库和数据库等内容。这个视图中包含的信息与配置管理和系统集成这类活动有关。

进程视图 process view，涉及系统中并发性的问题，部署视图‘deployment view’描述物理构件如何在系统运行的实际环境（如计算机网络）中分布。这两个视图处理的是系统的非功能性需求，例如容错性和性能等问题。进程视图和部署视图在 UML 中相对地未充分开发，尤其是与设计视图相比。在设计视图中包含了大量非正式地被当作是与设计有关的符号。

1.3UML 的组成

模型通常作为一组图呈现给设计人员。图(Diagrams)是一组模型元素的图形化表示。不同类型的图表示不同的信息，一般是它们描述的模型元素的结构或行为。各种图都有一些规则，规定哪些模型元素能够出现在这种图中以及如何表示这些模型型元素。

UML 定义了 9 种不同类型的图，在表 I.I 中列出了这些图并指出了与它们相关的典型视图。

表 1.1　UML 的图的类型

序号	图	视图
1	用例图(Use case diagram)	用例视图(Use case view)
2	对象图（Object diagram）	用例和设计视图(Use case and design view)
3	顺序图(Sequence diagram)	用例和设计视图(Use case and design view)
4	协作图（Collaboration diagram）	用例和设计视图(Use case and design view)
5	类图(Class diagram)	设计视图(Design view)

6	状态图(Statechart diagram)	设计和进程视图（Design and process view）
7	活动图(Activity diagram)	设计和进程视图（Design and process view）
8	构件图(Component diagram)	实现视图(lmplementation view)
9	部署图(Deployment diagram)	部署视图(Deployment view)

1.3.1 类图

考虑一下你周围的世界。你周围的事物大部分都可能具有某些属性（特性），并且它们以某种方式体现出各自的行为。我们可以认为这种行为是一组操作。

UML 类的表示，矩形方框代表类的图标，它被分成 3 个区域。最上面的区域中是类名，中间区域是类的属性，最下面区域里列出的是类的操作。类图就是由这些类框和表明类之间如何关联的连线所组成。

下图表示了几个重要的类，如 Customer Reservation Ticket 和 Performance。顾客可多次订票，但每一次订票只能由一个顾客来执行，有两种订票方式，个人票或套票，前者只是一张票，后者包括多张票。每一张票不是个人票就是套票中的一张。但是不能又是个人票又是套票中的一张。每场演出都有多张票可供预定。每张票对应一个唯一的座位号，每次演出用剧目名、日期和时间来标识。

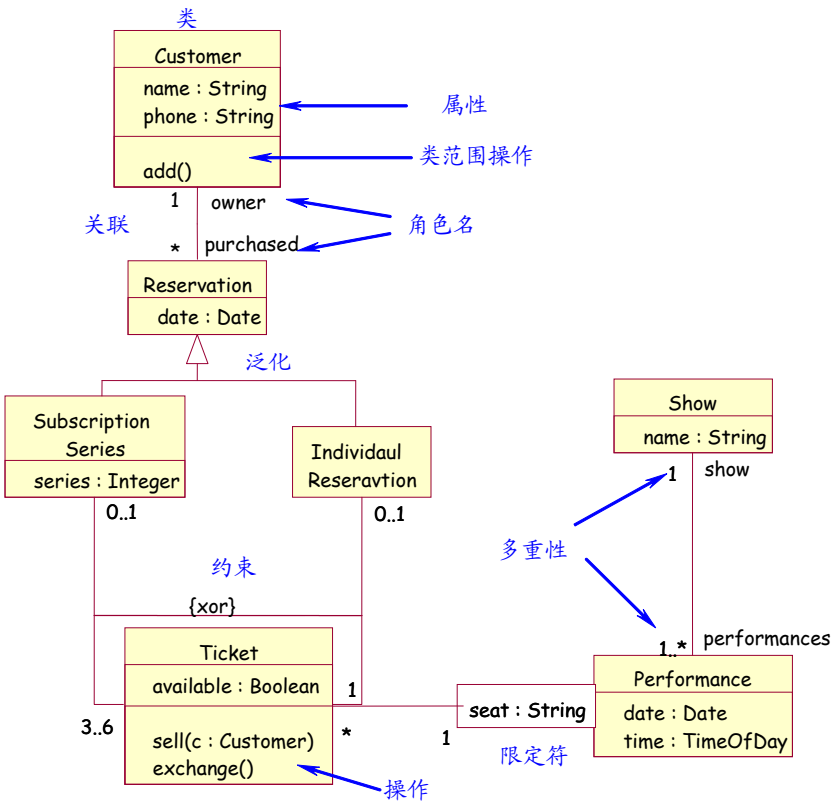


图 1.2　UML 类图标

1.3.1.1 关系

关联

当类之间在概念上有连接关系时，类之间的连接叫做关联(association)。关联的可视化表示方法是用一条线连接两个类，并把关联的名字放在这个连接线之上。表示出关联的方向是很有用的，关联的方向用一个实心三角形箭头来指明。.以在图中靠近每个类的地方的关联线上标明每个类的角色。

关联上的约束

有时，两个类之间的一个关联随后就有一个规则。可以通过关联线附

近加注一个约来说明这个规则。例如，一个 Bank Teller（银行出纳员）为一个 Customer（顾客）服务(serve)，但是服务的顺序要按照顾客排队的次序进行。在模型中可以通过在 Customer 类附近加上一个花括号括起来的“ordered（有序）”来说明这个规则（也就是指明约束），

另一种类型的约束是 Or（或）关系，通过在两条关联线之间连一条虚线，虚线之上标注{or}来表示这种约束。

关联类

和类一样，关联也可以有自己的属性和操作。此时，这个关联实际上是个关联类(association class)。关联类的可视化表示方式与一般的类相同，但是要用一条虚线把关联和对应的关联线连接起来。关联类也可以与其他类关联。例如 Player 类和 Team 类之间的 Plays On 关联对应的关联类：Contract（契约）关联类。

链

正如对象是类的实例一样，关联也有自己的实例。如果我们想象要一个特定的队员效力一个特定的球队，那么两者之间的 Plays On 关系就叫做一个链(link)，可以用两个对象之间的连线来表示它。和对象的名字要加下划线一样，链的名字也要加下划线。

多重性

某个类有多个对象可以和另一个类的单个对象关联。表示多重性的方法是在参与关联的类附近的关联线上注明多重性数值。

UML 使用星号(*)来代表许多(more)和多个(many)。在一种语境中，两点代表 Or（或）关系，例如“1. . *”代表一个或者多个。在另一种语境中，Or 关系用逗号来表示，例如“5，10”代表 5 或者 10。

继承和泛化

在 UML 中，用父类到子类之间的连线来表示继承关系。父类连线部

分，指向父类的一端带有一个空心三角形箭头。这种连接类型的短语的含义为 is a kind of（属于…中的一种）。

抽象类

表明一个类是抽象类的方法是类名用斜体书写。

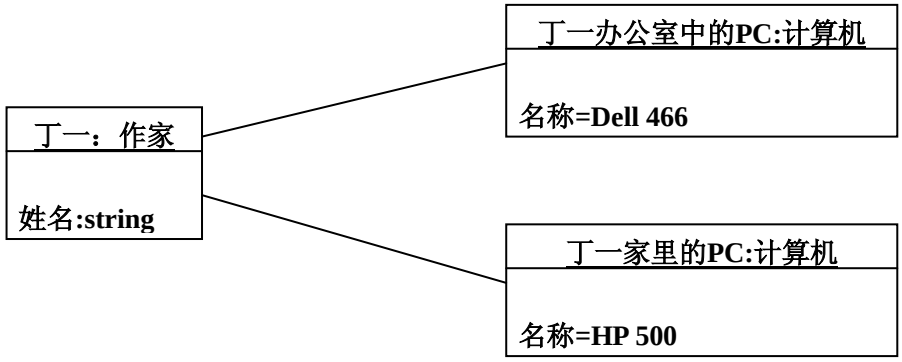
依赖

另一种类间关系是一个类使用了另一个类。这种关系叫做依赖(dependency)。最通常的依赖关系是一个类操作的型构中用到了另一个类的定义。UML 表示法是在有仃依赖关系颇的类之间画上一条带箭的虚线。

1.3.2 对象图

对象(object)是一个类的实例，是具有具体属性值的一个具体事物。对象的图标也是一个矩形，和类的图标一样，但是对象名下面要带下划线。实例的名字以一个小写字母开头。也可能是一个匿名的对象，这仅仅意味着尽管你指明了对象所属的类，但你并没有提供一个具体的对象名。

(a) 类图



(b) 对象图

图1.3 对象图示例

1.3.3 用例图

用例(use case)是从用户的观点对系统行为的一个描述。对于系统开发人员来说，用例是一个有价值的工具：它是用来从用户的观察角度收集系统需求的一项屡试不爽的技术。这对于那些试图建立一个供人使用的（而不是计算机设备使用的）系统是很重要的。

图 1.4 是售票系统的用例图。执行者包括售票员，监督员和公用电话亭。公用电话亭是另一个系统，它接受顾客的订票请求，在售票处的应用模型中，顾客不是执行者者，因为顾客不直接与售票处打交道。用例包括通过公用电话亭或售票员购票，购预约票，只能通过售售票监督。应监督员的要求，购票和预约票包括一个共同的部分，即通过信用卡来付钱。对售票系统的完整描述还要包括其他一些用例，例如换票和验票等。

用例也可以有不同的层次，用例可以用其他更简单的用例进行说明。

在交互视图中，用例做为交互图中的一次协作来实现。

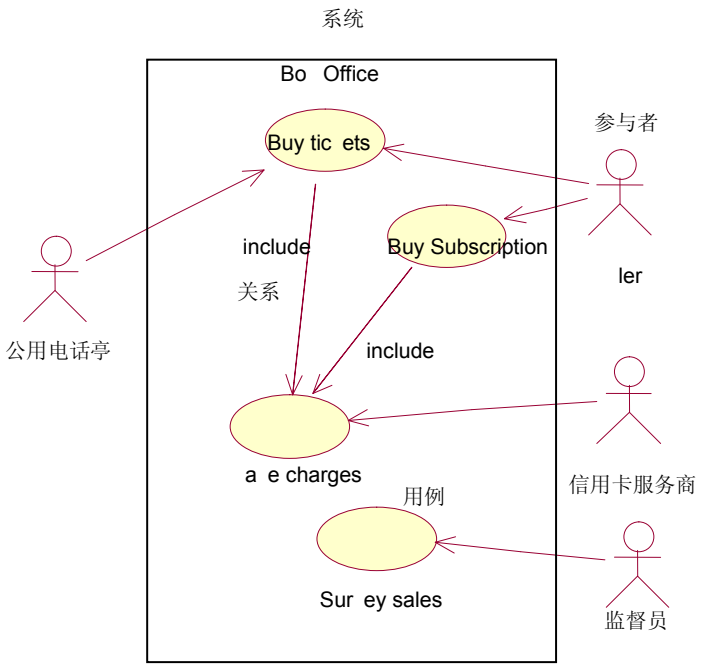


图 1.4 用例图

1.3.4.1 用例之间关系

包含

即在一个用例中重用另一个用例中的步骤。要表达用例的包含关系，可以使用类之间依赖关关系的表示符号，也就是连接两个类之间的虚线，箭头指向被依赖的类。在线上要加一个关键字，也就是用双尖括号扩起来的“include”。

扩展

新用例扩展(extencf)了原来的用例，因为它在原用例的基础上增加了新的步骤序列，因此原用例被称作基用例(base use case)。扩展只能发生在基用例的序列中的某个具体指定点上。这个点叫做扩展点(extension

points)。与包含关系相似，可视化扩展关系也是用一条依赖线（带箭头的虚线），沿线上加一个用双尖括号括起来的“extend”关键字。箭头是从扩展的用例指向被扩展用例。

泛化

类可以继承另一个类，用例也是如此。在用例继承中，子用例可以从父用例继承行为和含义，还可以增加自己的行为。任何父用例出现的地方子用例也可以出现。用例之间的泛化关系建模与类之间泛化关系建模方法相同，用一条带空心三角形箭头的实线从子用例指向父用例。

参与者之间也像用例一样可能存在泛化关系。

分组

在一些用例图中，用例的数目可能非常多，这时就需要组织这些用例。这种情况在一个系统包含很多个子系统时就会出现。最直接的方法是把相关的用例放在一个包中组织起来。包用一个一边突起的文件夹形的矩形框表示。一组用例可以出现在一个文件夹框中。

1.3.4 状态图

图 1.5 是票这一对象的状态图。初始状态是 Available 状态，在票开始对外出售前，一部分票是给预约者预留的，当顾客预定票，被预定的票首先处于锁定状态，此时顾客仍有是否确实要买这张票的选择权，故这张要票可能出售给顾客也可能因为顾客不要这张票而解除锁定状态。如果超过了指定的期限顾客仍未做出选择，此票被自动解除锁定。状态预约者也可以换其他演出的票，如果这样的话，最初预约票也可以对外出售。

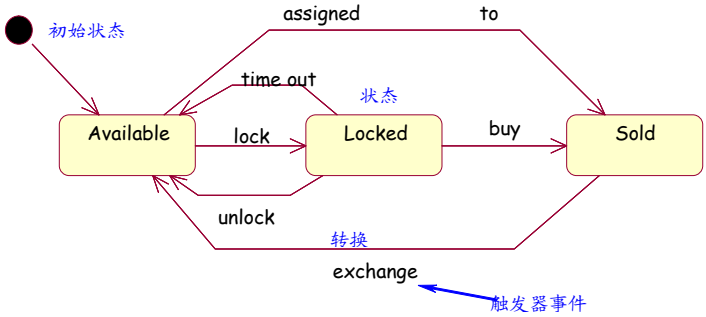


图 1.5 状态图

状态图可用于描述用户接口，设备控制器和其他具有反馈的子系统，它还可用于描述在生命期中跨越多个不同性质阶段的被动对象的行为，在每一阶段该对象都有自己特殊的行为。

1.3.5 顺序图

类图和对象图表达的是系统的静态结构。在一个运行的系统中，对象之间要发生交互，并且这些交互要经历一定的时间。UML 顺序图所表达的正是这种基于时间的动态交互。

图 1.6 是描述购票这个用例的顺序图。顾客在公共电话亭与售票处通话触发了这个用例的执行。顺序图中付款这个用例包括售票处与公用电话亭和信用卡服务处的两个通信过程，这个顺序图用于系统开发初期，未包括完整的与用户之间的接口信息。例如，位是怎样排列的，对各类座位的详细说明都还没有确定，尽管如此，交互过程中最基本的通信已经在这个用例的顺序图中表达出来了。

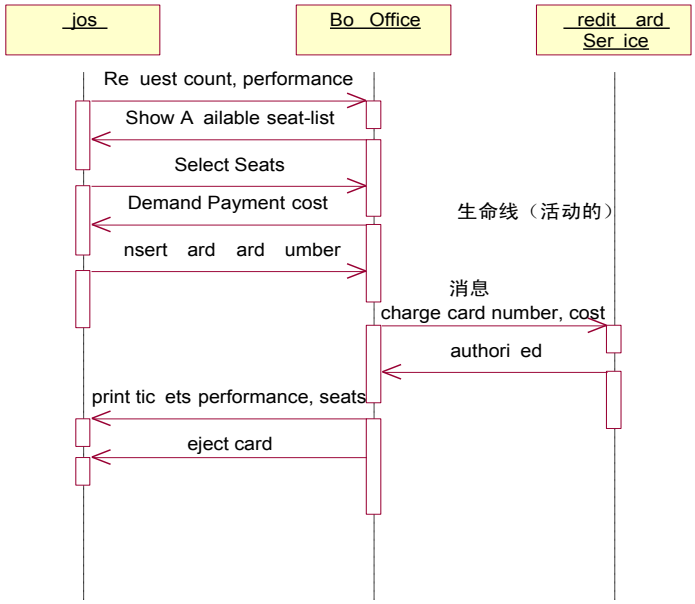


图 1.6 顺序图

1.3.6 活动图

活动图是状态机的一个变体，用来描述执行算法的工作流程中涉及的活动，活动状态代表了一个活动，一个工作流步骤或一个操作的执行。活动图描述了一组顺序的或并发的活动，活动视图用活动图来体现。

图 1.7 是售票处的活动图它表示了上演一个剧目所要进行的活动。箭头说明活动间的顺序依赖关系。例如，在规划进度前，首先要选择演出的剧目，加粗的横线段表示分叉和结合控制，例如，安排好整个剧目的进度后，可以进行宣传报道，购买剧本，雇用演员，准备道具，设计照明，加工戏服等，所有这些活动都可同时进行，在进行彩排之前，剧本和演员必须已经具备。这个例说明了活动图的用途是对人类组织的现实世界中的工作流程建模，对事物建模是活动图的主要用途，但活动图也可对软件系统中的活动建模，活动图有助于理解系统高层活动的执行行为，而不涉及建

立协作图所必须的消息传送细节。用连接活动和对象流状态的关系流表示活动所需的输入输出参数。

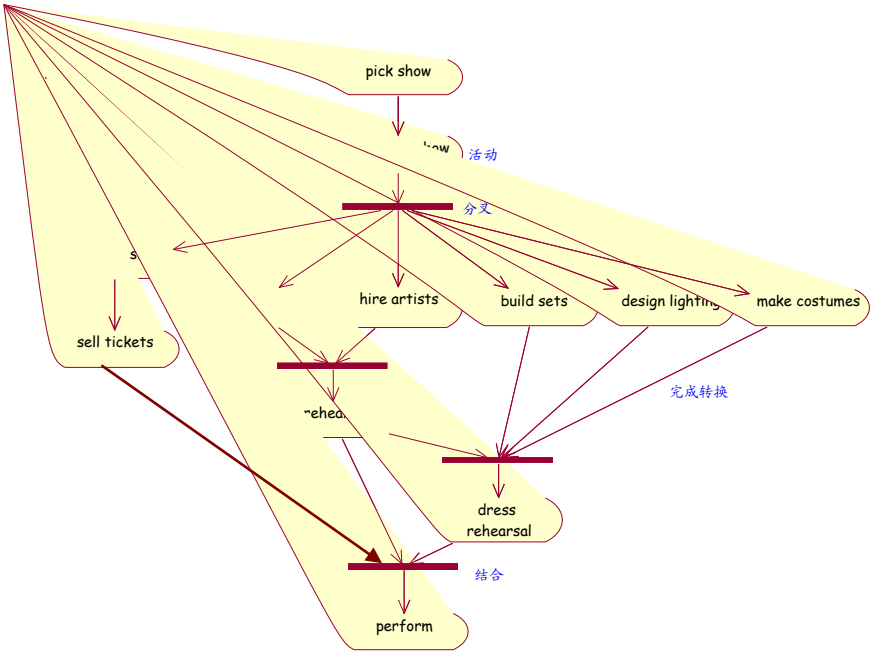


图 1.7 活动图

泳道

活动图中还可以增加角色的可视化维数。要对角色可视化，应该将图分割成多个平行的段，这些段被称为泳道（swimlane）。每个泳道的顶部可以显示出角色名，每个角色负责的活动放在各个角色的泳道中。一个泳道到另一个泳道之间可以发生转移。

1.3.7 协作图

协作图对在一次交互中有意义的对象和对象间的链建模，对象和关系只有在交互的时候才有意义。协作图用几何排列来表示交互作用中的各角色。

如图 1.8，消息的发生顺序用消息箭头处的编号来说明。协作图的一个用途是表示一个类操作的实现协作图可以说明类操作中用到的参数和局部变量以及操作中的永久链。当实现一个行为时，消息编号对应了程序中嵌套调用结构和信号传递过程。图 1.8 是开发过程后期订票交互的协作图 这个图表示了订票涉及的各个对象间的交互关系，请求从公用电话亭发出，要求从所有的演出中查找某次演出的资料，返回给 ticketseller 对象的指针， db 代表了与某次演出资料的局部暂时链接，这个链接在交互过程中保持，交互结束时丢弃，售票方准备了许多演出的票，顾客在各种价位做一次选择，锁定所选座位，售票员将顾客的选择返回给公用电话亭，当顾客在座位表中做出选择后，所选座位被声明，其余座位解锁。

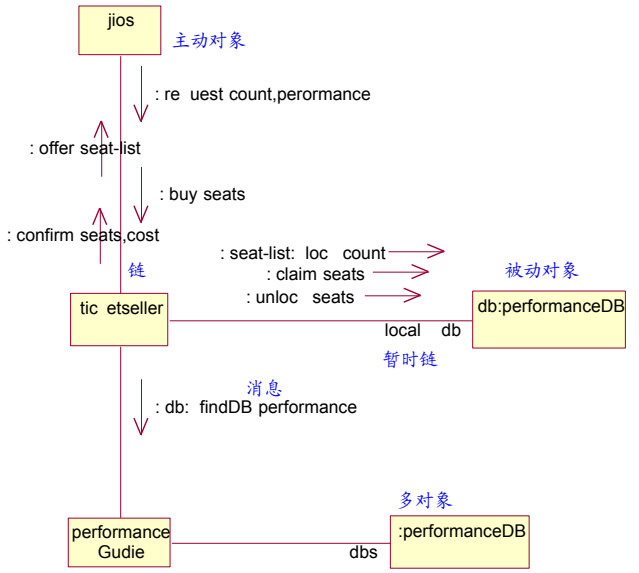


图 1.8 协作图

顺序图和协作图都能够表示对象之间的交互。因此，UML 中中它们被合称为交互图(interaction diagram) 。

1.3.8 构件图

鉴于很多建模工作者反映这个图标很糟糕，UML 2.0 使用一个劝讲的标记。图 1.9 给出了表示一个软件构件的新的方式。

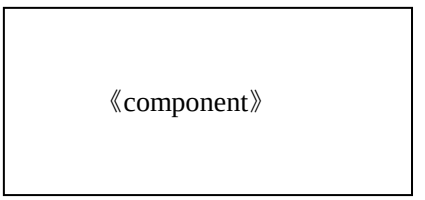


图 1.9　UML 2.0 中的软件构件图标记

1.3.9 部署图

UML 部署图显示了基于计算机系统的物理体系结构。它可以描述计算机，展示它们之间的连接，以及驻留在每台机器中的软件。每台计算机用一个立方体来表示，立方体之间的连线表示这些计算机之间的通信关系。图 1.10 是部署图的一个例子。

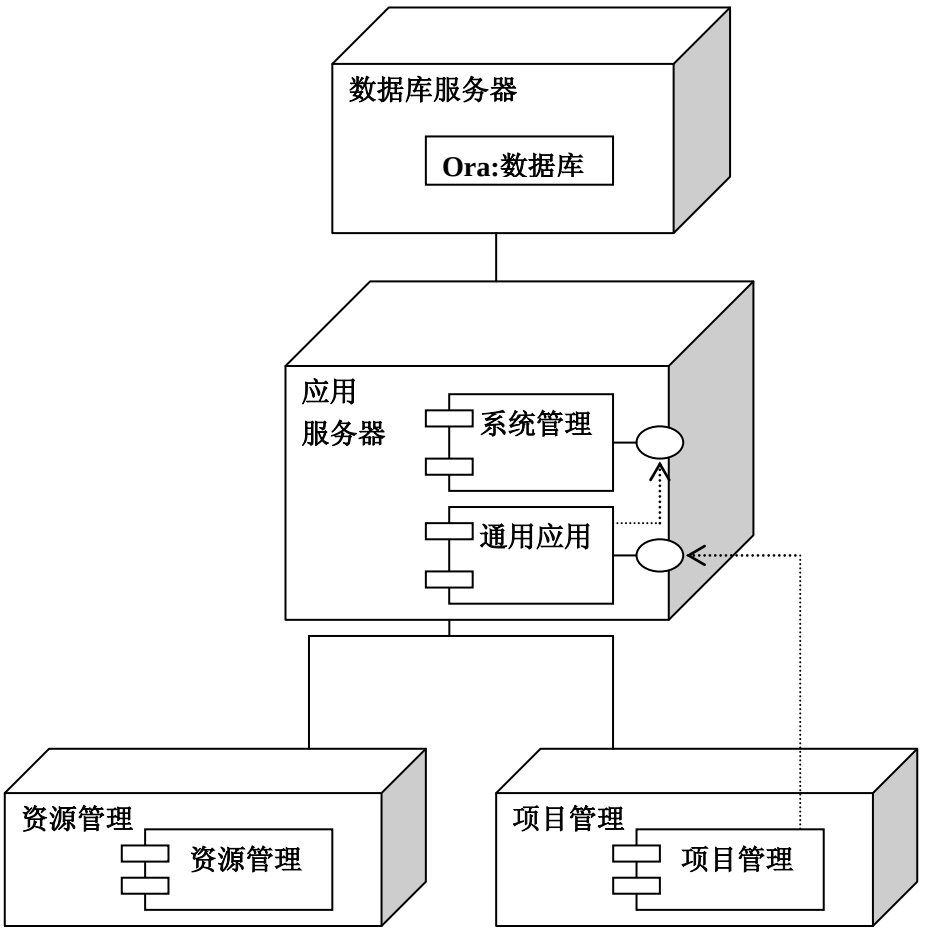


图 1.10　UML 部署图

1.3.10 包图

将许多类集合成一个更高层次的单位，形成一个高内聚、低耦合的类的集合在 UML 中这种聚集机制称为包(package)。不仅仅是类可以运行包的机制，任何模形元素都可以运行包的机制 包以及其间依赖的图称为包图(package diagram)。

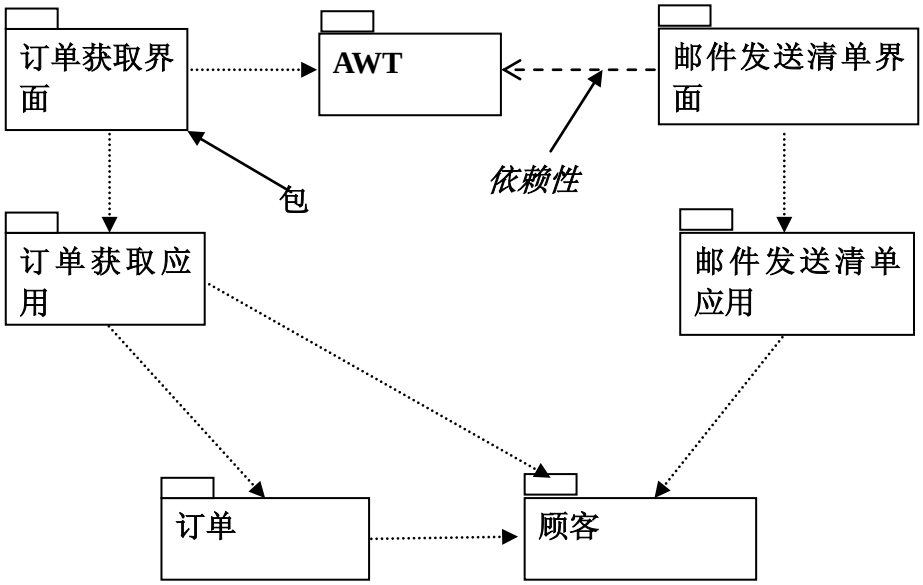


图 1.11　包图

第二章 UML 在统一过程中的运用

统一过程考虑了各种 UML 图在一个开发项目环境中的典型使用，在 UML 图之间存在着的某些关系是逻辑上的必然结果，即使将这些图独立地用于一个结构化过程时，也值得记住这些关系。

统一过程被描述为“用例驱动”的过程，这就意味着在统一过程后面的各个阶段要系统地使用用例。在分析和设计中对用例的主要使用如图 2.1 所示。

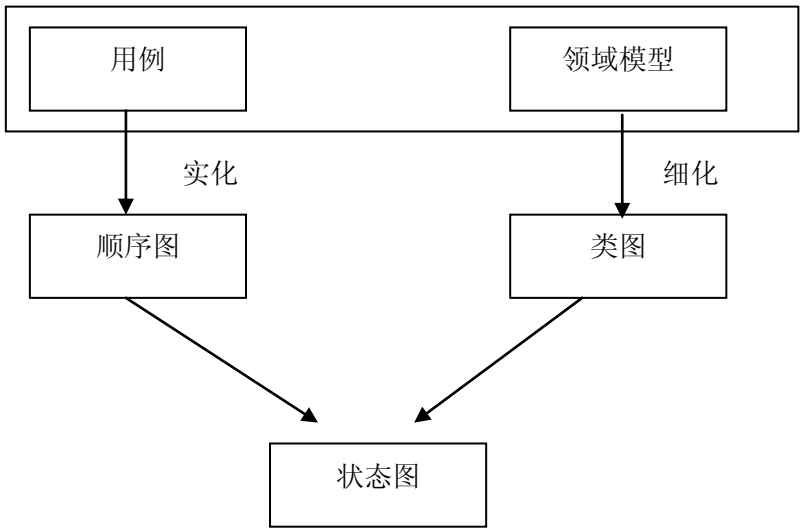


图 2.1　使用用例进行实化和细化

这里最重要的活动是用例的实化(realization)。实化是用例的高层实现，用 UML 交互图表示。它表示了系统如何将用例的功能作为对象之间的一系列交互来提供。实化中的对象来自领域模型中定义的类。

领域模型自然不会定义实化用例所需要的一切。实化一般揭示了对添加的类和重新确定类之间关系的需要，并强制设计人员更详细地指定支持

交互所需要的属性和操作。这个细化（refinement）过程把最初的领域模型发展成为一个更全面的类图，其中包含了足够的细节以形成实现的基础。

这里重要的一点是，类模型的开发不是孤立地进行的，对类模型需要做出的修改完全是由实化用例的过程推动的。这意味着，类图是以这种使设计人员能够确信、详细指定的类实际上将会支持所需要的功能的方式演进的。如果脱离对象交互的考虑而产生复杂的类图，情况将不会是这样。

最后，以用例的实化和详细的类图为基础，可以为需要状态图的类开发出状态图，作为那些类的实例的生命周期的文档。图 2.1 概括地表示了用例模型的内容如何直接渗透到主要的 UML 模型产品中。

一旦产生了图 2.1 所示的模型，可以直接用这些模型来指导系统的实现。用例模型对开发过程的最后一个重大影响是产生系统的测试用例。为了准备对系统的验收测试，可以从用例中系统地导出测试：能够成功执行这些用例，是系统实现了某些对用户有用的业务功能的合理保证。

第三章 UML 案例研究

项目的大部分时间都应该花费在系统的分析与设计上。经过了充分的分析与设计之后，编码过程才会高效率地平稳推进，并且系统安装和部署出现问题的可能性也会大大降低。现在应用 UML 来分析设计一个餐馆系统.

首先要做的是需求收集和了解餐馆领域。需求收集段包括如下几个动作：

- 发现业务过程。
- 执行领域分析。
- 识别协作系统。
- 发现系统需求。
- 结果提交给客户。

3.1 发现业务过程

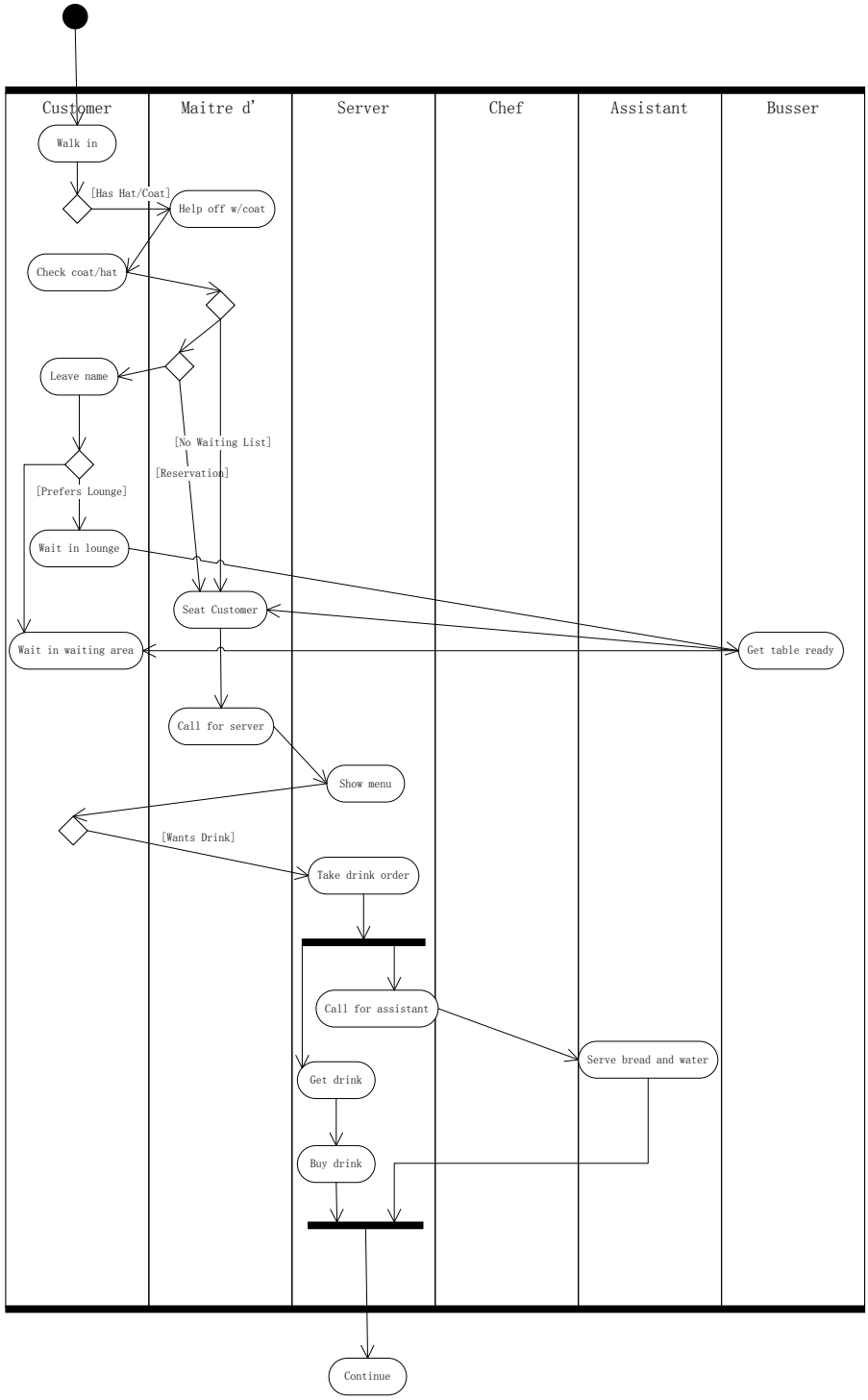
3.1.1 招待一位顾客

当一名顾客走进餐馆时，要做些什么呢：

- 如果顾客穿着外套，我们会帮助他脱下外套，将外套存放在存衣间里，并给顾客一张取衣票。假设顾客比较多，需要排队等候。我们会询问顾客是否要预订席位，并尽可能处理顾客的预订，让顾客尽快入席。如果没有空缺的席位可供预订，顾客可以登记下他的名字，并可以选择先到我们安排的休息室里喝点饮料，休息一会。当然顾客也可以不去休息室，也可以到一个指定的候餐区坐下来等。

- 轮到某个排队的顾客或者已经预订了席位的顾客来到餐馆后，在顾客入座就餐之前，餐桌必须要提前准备好。清洁师要事先清理桌面，除去旧的桌布，换上新的桌布，还要调整好桌子和座位。当一切准备就绪后，领餐员将顾客领到餐桌前就座，并叫一名服务员来招待顾客。每名服务员都被分派了一个指定的服务区，乡并且通常服务员都在各自的服务区附近活动，而且知道哪个餐桌已经准备就绪。领餐员一打手势，服务就能看见。
- 然后服务员就接管了这一桌的顾客。他给每位就餐者一份菜单，并询问顾客是否要预订酒水。然后服务员会叫一名‘助手’，助手来的时候会给顾客端来面包、黄油，并给每位顾客倒一杯水。如果顾客订了饮料，则服务员立刻去取来。
- 如果一名顾客因为等待席位而在休息室里，并且叫了饮料在喝，没喝完的话他可以带到就餐席位去。
- 每天我们都有当日的特色菜点，这些特色菜点菜单上都没有，服务员在顾客点菜时会向顾客背诵出这些菜点。
- 顾客通常会让服务员给他们推荐一些菜点，并且服务员也很诚实——服务员通常会告诉顾客哪个好吃，哪个不太好吃等等。
- 现在服务员取回了顾客订的饮料并给顾客，顾客可以边喝边点菜。服务员暂时离开,给顾客 5 到 10 分钟的时间选择,然后再回来招待顾客。顾客选的越快，服务员回来的也应越快。
- 每个服务员都被指派一个服务区，里面有几张餐桌。有一个服务区是专门为勺吸烟者设置的，其余是非吸烟者就餐的服务区。
- 服务员轮流负责各个服务区。
- 顾客点了菜，服务员记录下顾客所点的菜点，接着知厨师。服务员将顾客的选择填在一张表格中并交给厨师。

- 表格中都要登记桌号、顾客点的饭菜、酒水，还有（也是最重要的）菜单送到厨房的时间。
- 大部分顾客吃主菜前都要吃一些小菜，并且都希望主菜是刚出锅的热菜。因此厨师就得先做小菜（通常这些小菜都是事先已经做好了的，如一些沙拉、凉菜），服务员将小菜端给一组顾客。问题是一组顾客中有人吃的快有人吃的慢，但是主菜还得为多个人同时上，这样吃的慢的顾客在吃主菜时可能菜就有点凉了。因此为每个顾客做小菜的先后次序必须要能够很好地协调。
- 当就餐完毕后，服务员会上前询问顾客是否再来些甜食。如果顾客要吃，服务员就会给顾客一份甜食菜单，并记录下顾客的选择。如果顾客不吃甜食，服务员还会询问顾客要不要喝咖啡，如果要，服务员则端来咖啡和杯子为顾客冲咖啡。如果顾客就餐后什么也不需要，服务员就拿来账单让顾客确认，过几分钟后服务员再回来收钱（现金或信用卡支付），找零钱，给顾客收据。顾客离开座位，取回衣帽，离开餐馆。
- 顾客走后，服务员还要叫清洁师清理餐桌，并重新布置餐桌和座椅以备下一批顾客使用。
- 可以将活动图改绘为泳道图。当建立了一十个业务过程模型后，很有必要做这件事情，因为泳道图可以反映出参与业务过程的各个角色。


Customer
Maitre d'
Server
Chef
Assistant
Busser
Walk in
[Has Hat/Coat]
Help off w/coat
Check coat/hat
Leave name
[No Waiting List]
[Reservation]
[Prefers Lounge]
Wait in lounge
Seat Customer
Wait in waiting area
Get table ready
Call for server
Show menu
[Wants Drink]
Take drink order
Call for assistant
Serve bread and water
Get drink
Buy drink
Continue

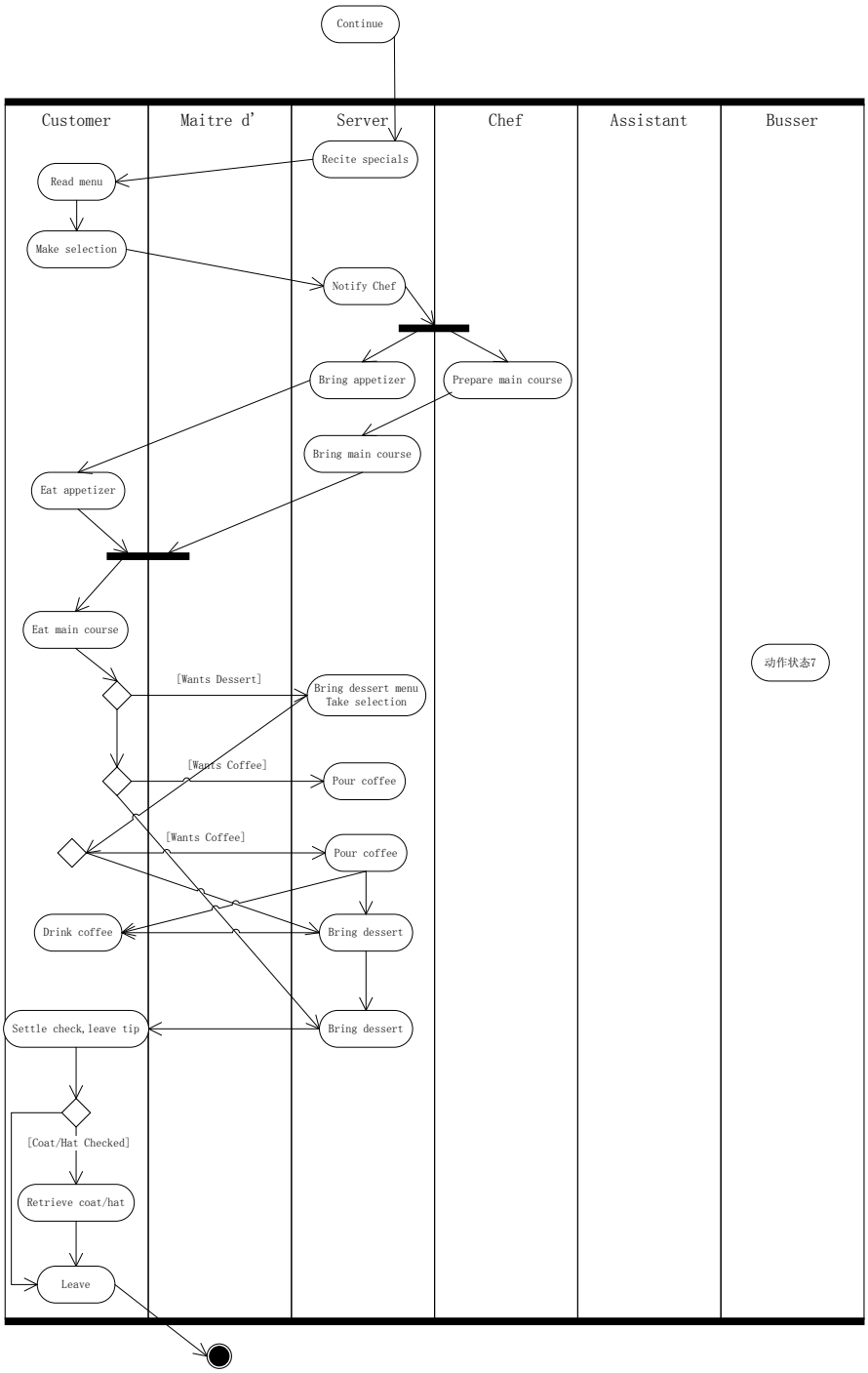


图 3.1　餐馆业务过程“招待一名顾客”的泳道图

3.1.2 准备饭菜

厨师收到服务员拿来的菜单后就开始做小菜，同时也要开始做主菜。当顾客吃完小菜后，服务员到厨房去端来主菜，送到顾客的餐桌上。服务员要不时地去餐桌旁检查。这时就要服务员和厨师间的协作：厨师将做好的小菜交给服务员后，要等服务员回来通知他顾客马上就要吃完小菜时才做烹饪主菜的最后手续。服务员呆在各自的服务区内，不停地监视顾客的餐桌，在合适的时候，服务员会到厨房通知厨师，顾客即将吃完小菜，就要上主菜。厨师得知这一消息后，完成主菜烹饪的最后工序。厨师的技术都很熟练，并有一些助手在旁协助，能够同时协调好多组顾客的上菜要求。目标是尽量让想吃主菜的顾客尽快吃上主菜。

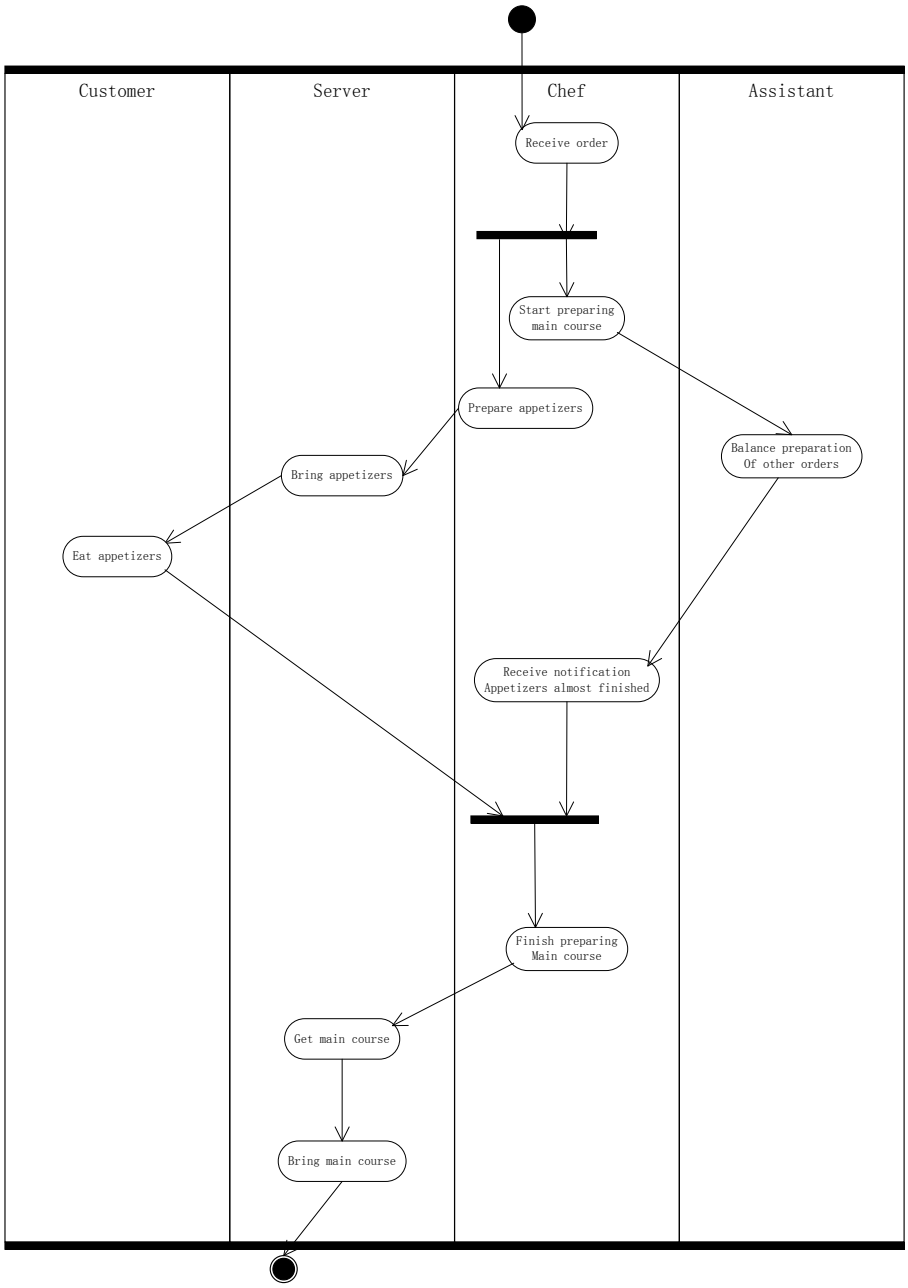


图 3.2　餐馆业务过程“准备饭菜”的泳道图

3.1.3 清理餐桌

首先服务员要确认顾客已经离开，然后叫来清洁师料理一下餐桌。如

果餐馆业务繁忙的话，这个过程必须快速完成。我们没有像服务员一样多的清洁师，有时这个过程显得有些杂乱无章。清洁师不一定总在附近，服务员可能需要去寻找他们。清洁师必须取下旧桌布，换上干净桌布。并把取下的旧桌布叠好放在厨房后面的仓库里。第二天我们会将旧桌布打包并派人把桌布送到洗衣店去清洗。

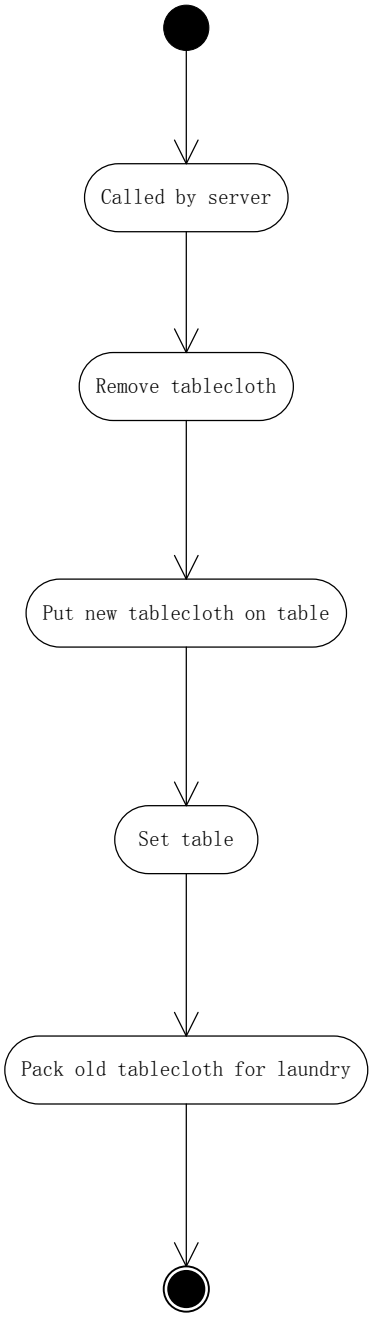


图 3.3　　“清理餐桌”的活动图

3.2 领域分析

3.2.1 开发初步类图

可能成为系统中的类的名词列表：

顾客(customer)、外套(coat)、. 储衣室(cloakroom)、取衣票(coat-check ticket)、帽子(hat)、队(line)、等候队列(waiting list)、休息室(cocktaillounge)，饮料(drink)、正餐(dinner)、候餐区(waiting area)、餐桌(table)、清洁师(busser)、桌布(tablecloth)、领餐员(maitred 信)、服务区(serving area)、菜单(menu)、助手(assistant)、碟(tray)、面包(bread)、黄油(butter)、玻蒋坏（glass）、水(water)、一组顾客(pany)、服务员(server)、选择(selection)、每日特色菜点(daily special)、餐馆(restaurant)、厨师(chef)、盘装菜(dish)、厨房(kitchen)、定单(order)、吸烟区(smoking area)、表单(form)、小菜(appetizer)、主菜(皿血 course)、甜食(dessert)、甜食菜单(dessert menu)、咖啡(coffee)、杯子(cup)、账单(check)、现金(cash)、信用卡(credit card)、找回的钱(change)、信用卡收据(credit card receipt)、小费（卹）、餐具(silverware)、餐巾纸(napkin,)、房间(room)、经理（manager）、预订的事物(reservation)。

3.2.2 对类分组

现在我们将设法形成一些有意义的组。人组成的一组：Customer、Busser、MaitreD、Assistant、Chef、Party. Server 和 Manager。除了 Customer 和 Party 以外，其余的类代表的人都是餐馆中的雇员(Employee)，因此可进一步将这些类分为 3 组：Customer、Party 和 Employee 组。

第二组由餐馆中的食物组成：Drink、Diner、Bread、Butter、Water、Daily Special、Dish、Appetizer. Main Course. Dessert 和 Coffee。

第三组由餐馆中的用具组成：Glass、Silverware、Tray、Cup、Napkin 和 Tablecloth。

第四组包含与支付有关的项目：CoatCheckTicket、Check、Cash、Change、CreditCard、CreditCardReceipt 和 Tip。

还有一组由餐馆中的区域组成：WaitingArea、SmokingArea、CocktaiILounge、Cloakroom、Kitchen、ServingArea、Table 和 Room。“Room”指的是存放脏桌布的房间（当然也可以存放其他东西），这些桌布第二天要被送到洗衣店中清洗。为了使这个词意义更明确，不妨称其为“LaundryRoom（待洗衣物存放间）”。

最后，将餐馆中用到的各种表单划为一组：Menu、DessertMenu、CoatcheckTicket、Check 和 Form。最后一个词代表的是当定单送到厨房后服务员给厨师的表单。为了更明确起见，这里将它称为“OrderForm（定单表）”。

注意，最后一组还可以再细分为两组：Form（表单）和 Paymentltem（支付项）。这样分组也是可以接受的。

每个组名可以成为一个抽象类名一抽象类用做其他的类的超类，自己并不产生实例的类。抽象类 RestaurantArea 有 CocktailLounge. ServingArea、Table、WaitingArea. Cloakroom 和 Kitchen 子类。

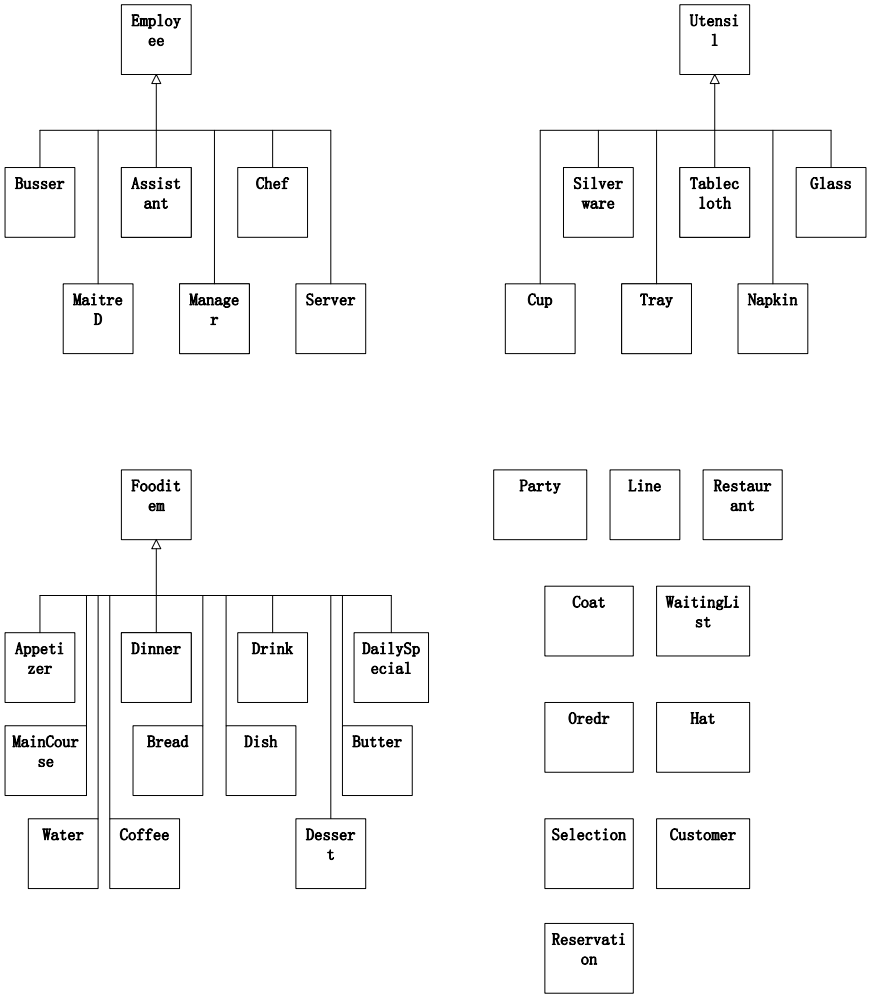


图 3.4　用抽象类将类图划分为有意义的组

3.2.3 形成关联

下一步，要建立和标记出类之间的关联。动词和动词短语可以帮助我们标记关联。

一种策略是先从几个类开始，找出与这几个类存在关联的其他类，然后再寻找另外一组类与其他类的关联，直到穷尽了所有的类为止。在标记出类之间的关联后，进一步找出类之间的聚集和组成关系。最后使用一些动词或动词短语来表示类的操作。

3.2.3.1Customer 参与的关联

先从 Customer 类开始寻找关联。哪些类与 Customer 类有关联呢? Reservation 是明显的一个。另一个是 Server。另几个是 Menu. Meal. DessertMenu. Dessert. Order. Check. Tip.Coat 和 Hat。

下面用一些表示关联的动词短语来标记上面产生的关联。

- The Customer makes a Reservation
- The Customer is served by a Server
- The Customer eats a Meal
- The Customer eats a Dessert
- The Customer places an Order
- The Customer selects from a Menu
- The Customer selects from a DessertMenu
- The Customer pays a Check
- The Customer leaves a Tip
- The Customer checks a Coat with a CoatCheckClerk
- The Customer checks a Hat with a CoatCheckClerk

接着我们的注意力将转移到关联的多重性。重性是关联的一部分：它指明类 B 的多少个实例与类 A 的一个实例发生关联。

怎样对下面的关联建模呢？

- The Customer checks a Coat with a CoatCheckClerk
- The Customer checks a Hat with a CoatCheckClerk

这种关联被称为三元关联(ternary associatioron)。“三元”意味着三个类同时参与一个关联。在模型中，三元关联用一个菱形框表示，在菱形框附近写上关联的名字。三元关联的多重性含义为当一个类的实例数量固定

时，另外两个类的多少个实例参与这个三元关联。在本例中，一个 Customer 可以从一个 CoatCheckClerk 那里取回多于一件的 Coat。参与一个关联的类也可能超过 3 个。由于通用性的缘故，在 UML 中这种关联被称为 n 元关联 (n-ary association)。

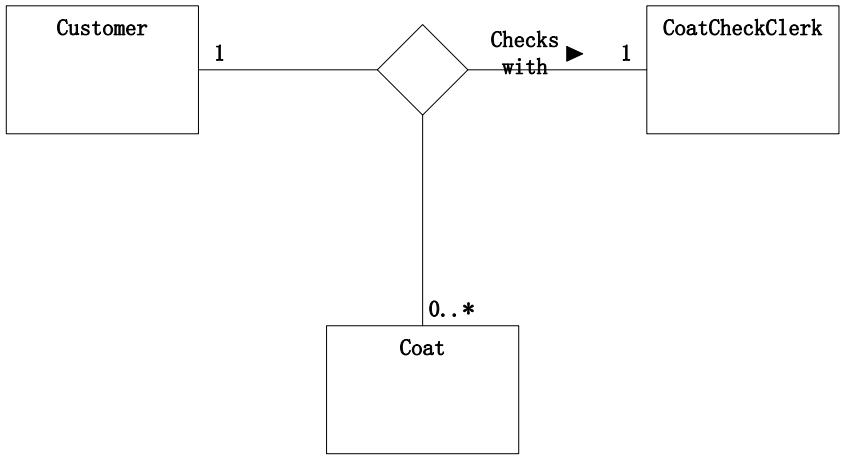


图 3.5　一个三元关联

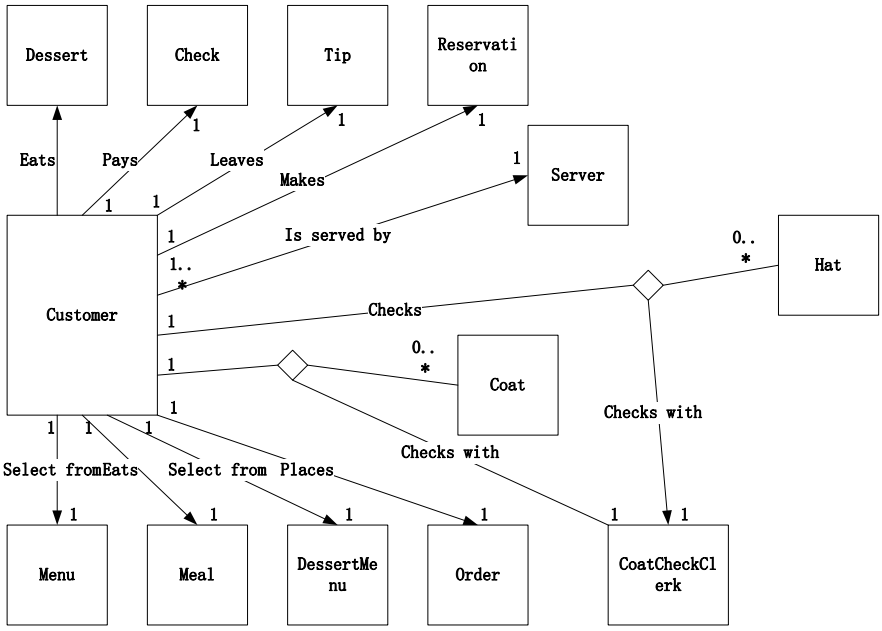


图 3.6　添加了多重性后 Customer 参与的关联

3.2.3.2 Server 参与的关联

让我们用 Customer-Server 之间的关联作为出发点，继续寻找 Server 类参与的关联。对 Server 参与的关联的一种建模方式是将这些关联作为三元关联：

- The Server takes an Order from a Customer
- The Server takes an Order to a Chef
- The Server serves a Customer a Meal
- The Server serves a Customer a Dessert
- The Server brings a Customer a Menu
- The Server brings a Customer a DessertMenu
- The Server collects Cash from a Customer
- The Server brings a Customer a Check
- The Server collects a CreditCard from a Customer

检查这些关联，使用最少数量的关联标记，并将一些关联表示为恰当的关联类。

Server 的工作显然可概括成“take”和“bring”。 “∞llect”是一种“take”， “serve”是一种“bring”。我们可以将 Server 参与的关联标记为“take”或“bdng”。再在这些关联上附加一个关联类，在这个类中可以指明“take”或“bring”的是什么。为了达到这样的目的，我们给关联类中设置一个枚举类型的属性 itemType。这个属性可以取的值是 Server 可能“bring”或“take”的东西。

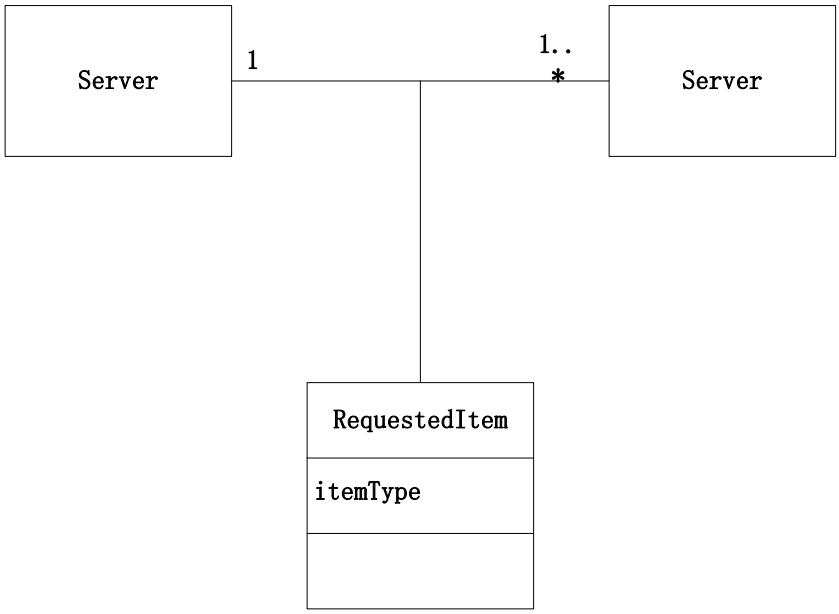


图 3.7　在 Server 参与的关联中使用关联类

Server 还同时与 Assistant 和 Busser 关联

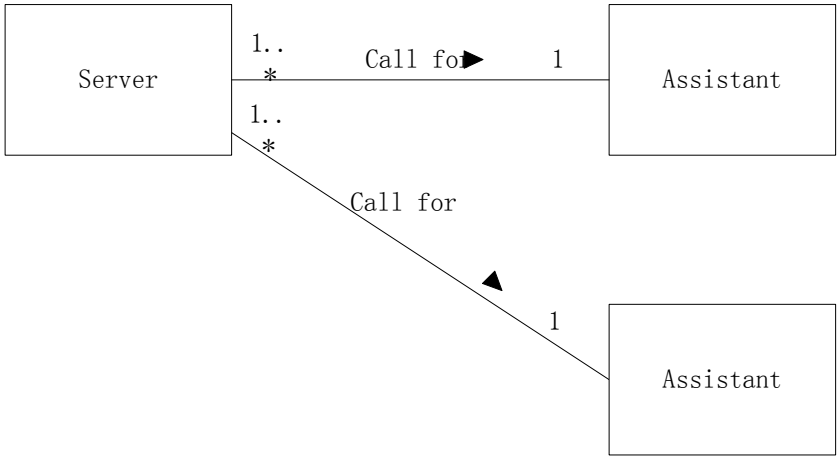


图 3.8　Server 参与的其他关联

3.2.3.3Chef 参与的关联

Chef 与 Server、Assistant 以及 Meal 关联，如图所示。关联类 Order 对 Server 带给 Chef 的点菜单建模，这个类的属性（可以是一个枚举类型）

显示了点菜单的状态。

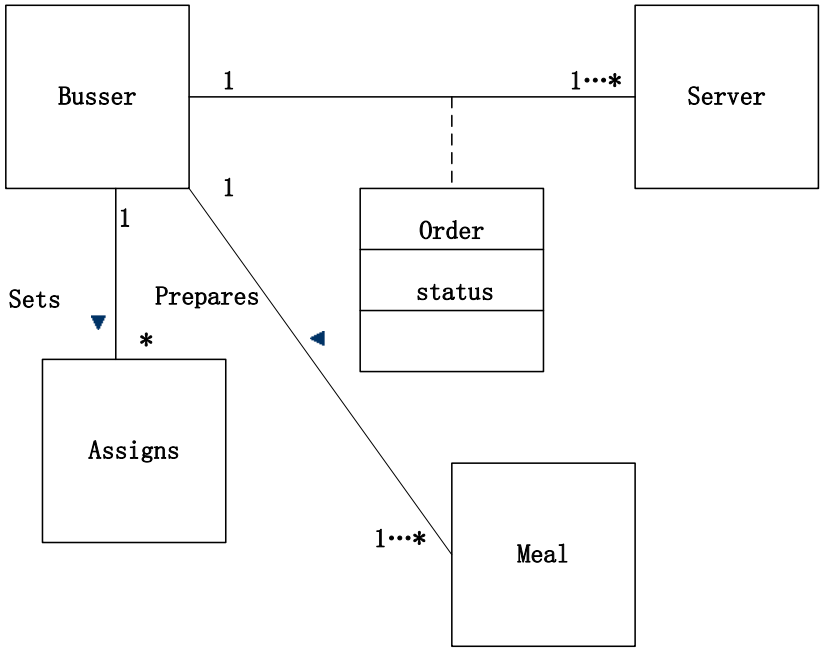


图 3.9　Chef 参与的关联

3.2.3.4Busser 参与的关联

Busser 有两个关联，如图所示。其中一个关联表示 Server 招呼 Busser，并且通过多重关系表示有多个 Server 在招呼同一个 Busser。另一个关联表示一个 Busser 摆好了多张桌子。

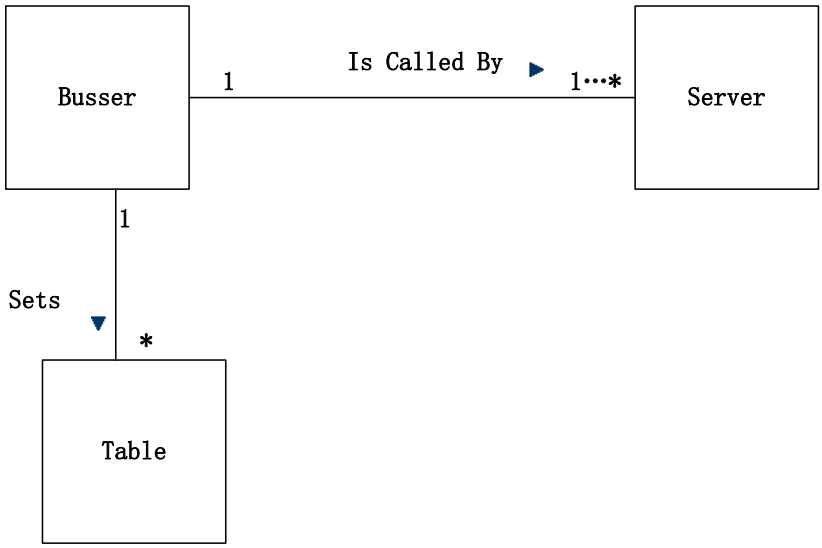


图 3.10　Busser 参与的关联

3.2.3.5Manager 参与的关联

Manager 是我们在领域分析中引进的新类。这个类与许多其他类都有关联，这些关联短语可表示如下：

- The Manager operates the Restaurant
- The Manager monitors the Employees
- The Manager monitors the Kitchen
- The Manager interacts with the Customer

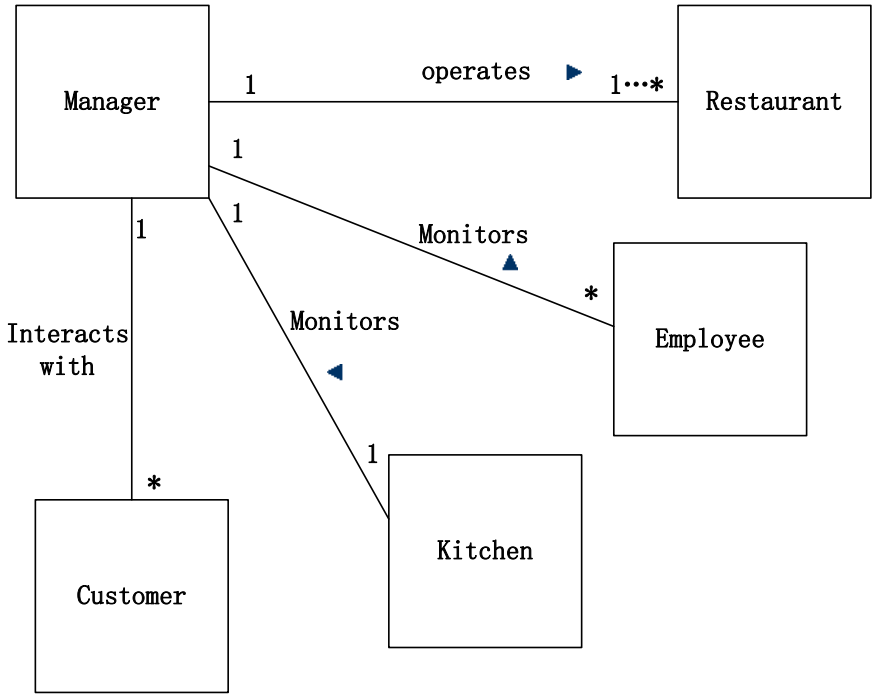


图 3.11　M anager 参与的关联

3.2.4 形成聚集和组成

我们已经用抽象类对类进行了分组，还找出了类之间的主要关联关系。下一步是要找出类之间的包含关系，即聚集关系。在餐馆这个领域中，此项工作不是很难。例如，一个 Meal 对象是由一个 Appetizer、一个 MainCourse、一个 Drink、一个 Dessert 组成的。Appetizer 和 Dessert 是可选的。并且这些成员对象代表的事物在 Meal 中是按照一定时间顺序出现的，应当在模型中反映出这种时间顺序。

如下是其他一些组成关系：

- 一个 Order 由一个到多个 MenuSelection 组成。
- 一个 Restaurant 由一个 Kitchen、一个到多个 ServingArea、一个 WaitingArea、一个 CocktaiILounge 和一个 LaundryRoom 组成。

- 一个 ServingArea 由一个到多个 Table 组成。
- 一个 Party 由一名到多名 Customer 组成。

在每一种情况中，一个组成体只属于一个聚集体，因此下图是这些组成关系的模型。

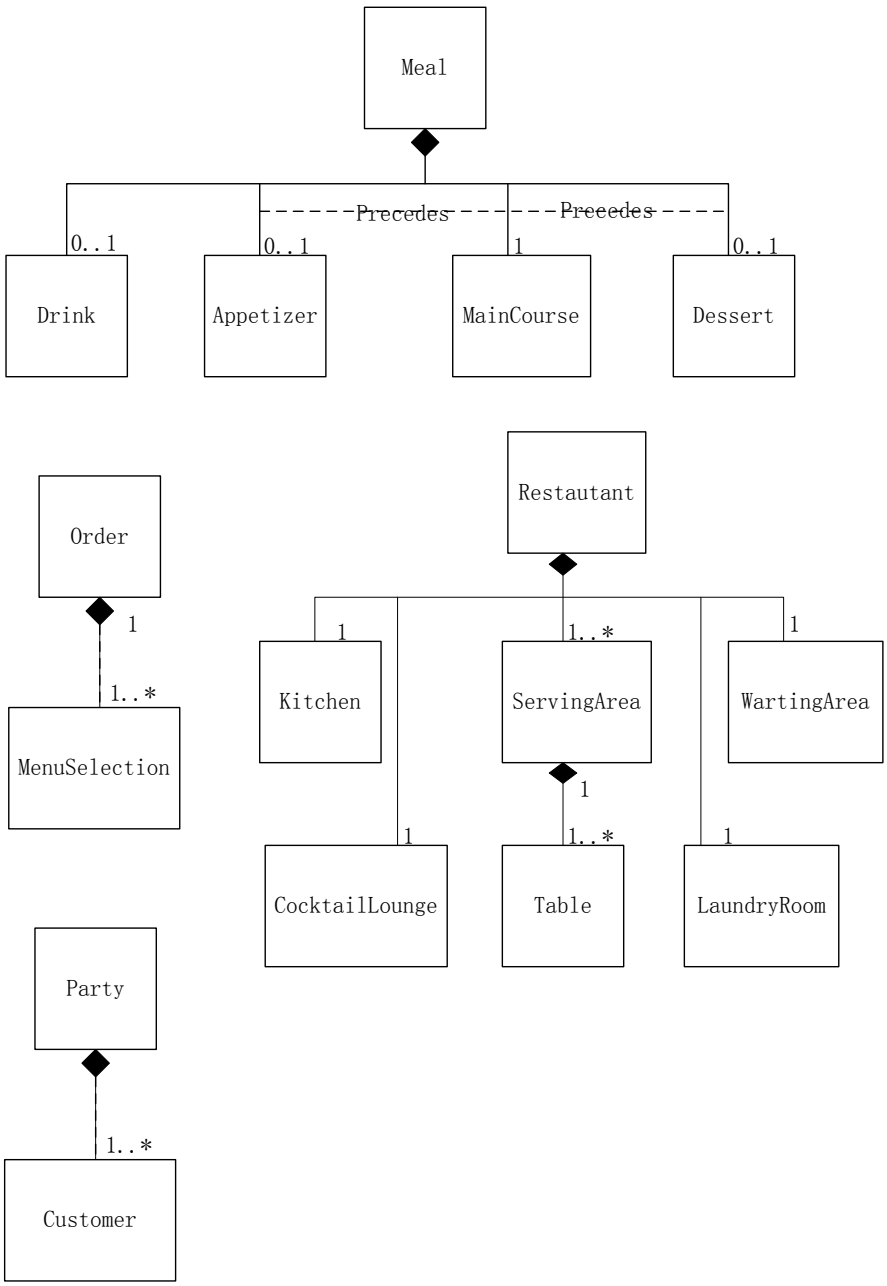


图 3.12　餐馆业务领域中的组成关系

3.2.5 填充类的信息

3.2.5.1Customer 类

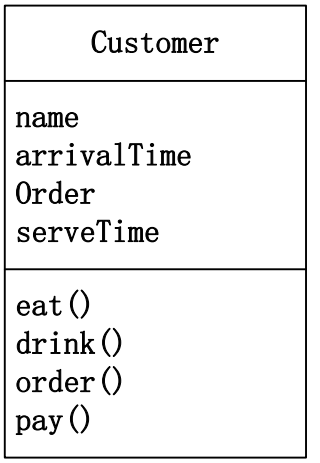


图 3.13　Customer 类

3.2.5.2 Employee 类

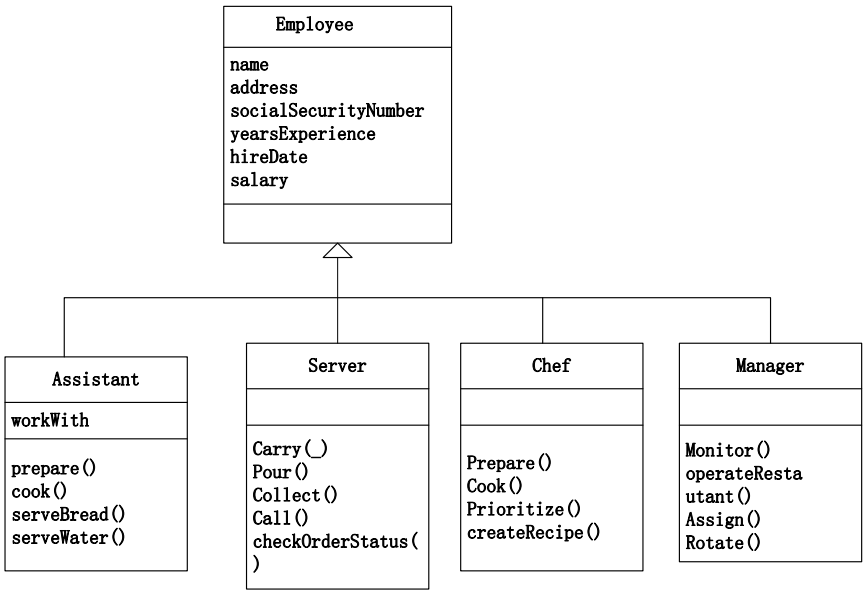


图 3.14　Employee 类和它的子类

3.2.5.3 Check 类

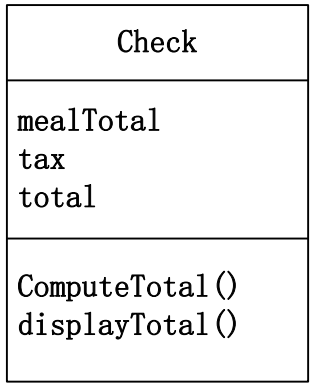


图 3.15　Check 类

3.3 收集系统需求

必须要开发出一个系统的视图。开发组将使用他们所了解到的业务过

程知识和新获取的领域知识，为的是看看外出就餐的哪些地方可以使用技术来改善。

假设每名服务员和清洁师都携带一台终端——一台手提式个人计算机。进一步假设我们在这些计算机之间建立一个无线网络。厨房和经理办公室里可以分别放一卜台桌面电脑终端。另一种可选的方案是让服务员和清洁师使用掌上电脑。但手提式个人计算机一般带有显示器和键盘，这种特征可以为以后的设计增加灵活性。

收集系统需求，目标是产生能反映系统功能的包图，每个包代表系统的一个功能模块，其中包含了详细说明该功能模块的若干个用例。需求编制成文档，有了需求文档在手，就可以对项目耗费的时间和金钱做出估算。

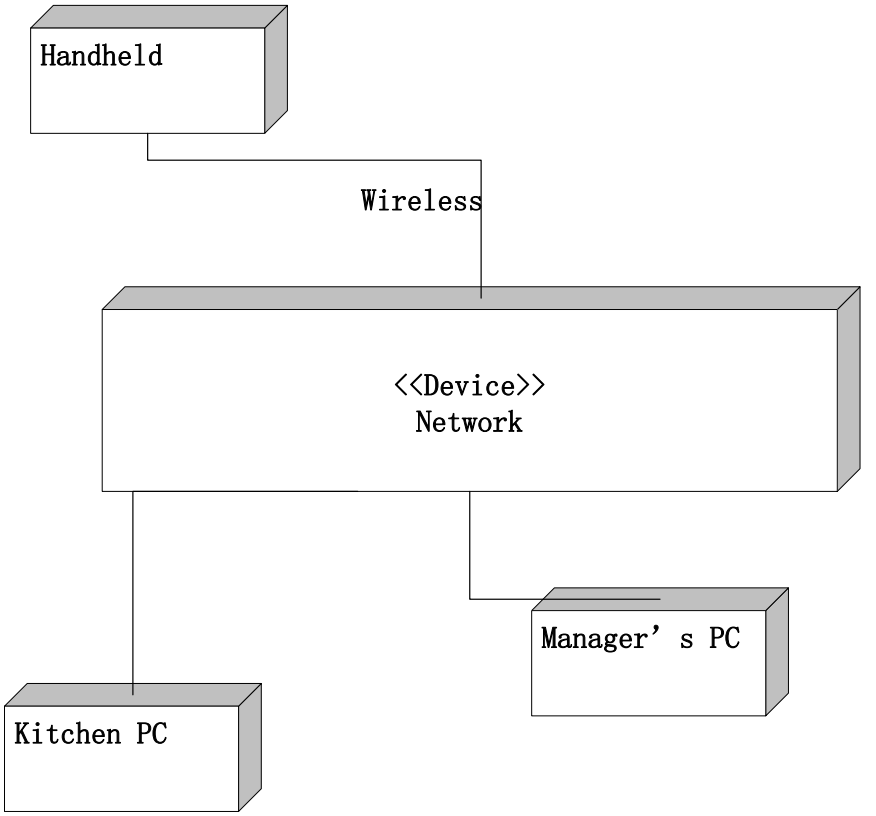


图 3.16　WIN 系统

按照角色。厨师必须做某些事情，服务员也必须做自己的一些事情，

等等。据业务过程图和类图，人员角色有 Server、Chef、Busser、Assistant 和 Manager。还应该有 Coat-check Clerk 和 B artender。使用 UML 包图表示需求。

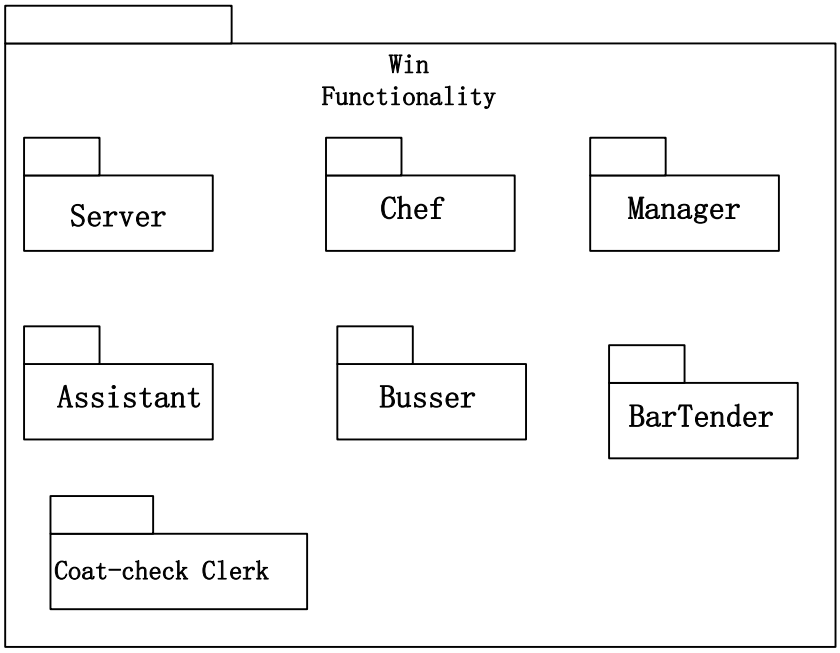


图 3.17　WIN 系统的功能包

最后产生了一组需求，这些需求通过用例来表达，若干个相关用例被组织进一个包中。

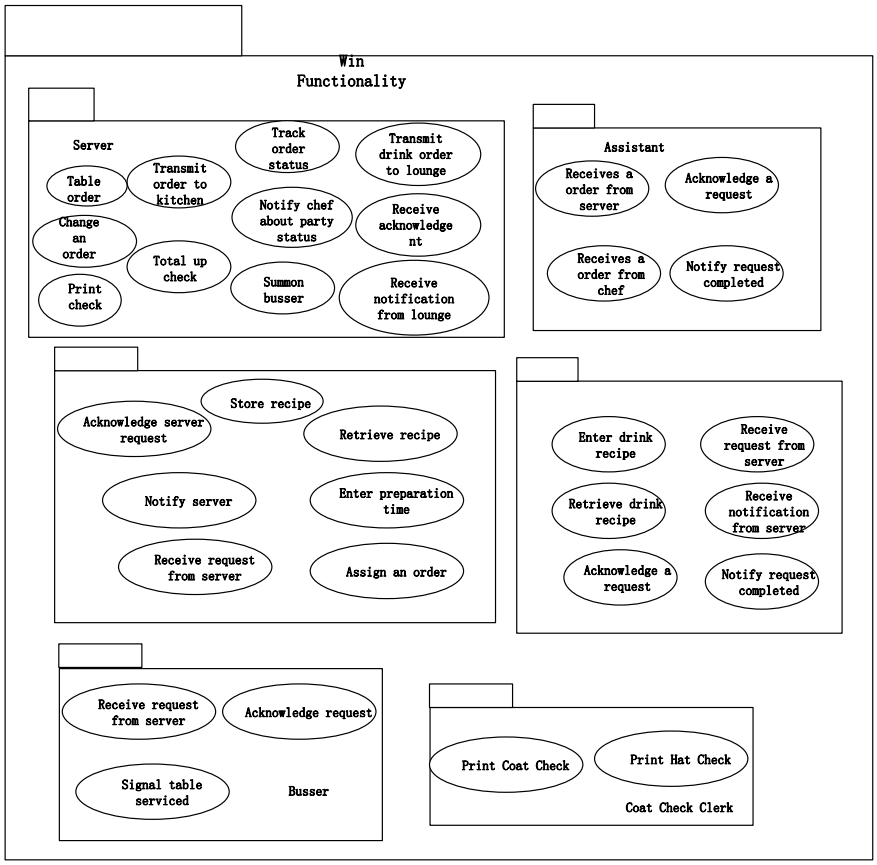


图 3.18　系统的功能包图

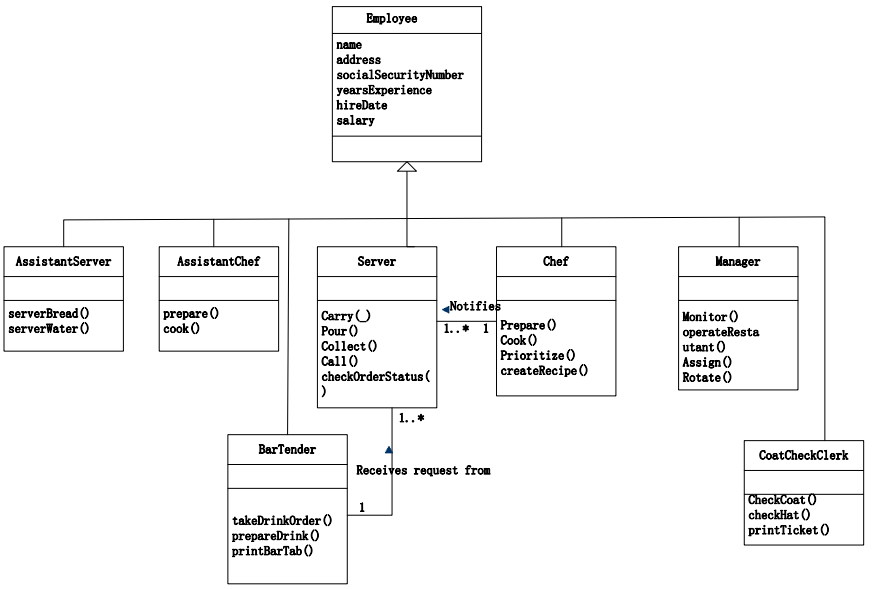


图 3.19　新增加类后的类图

3.4 开发用例

3.4.1 用例“Take an Order”

让我们从用例“Take an order”开始。我们必须根据服务员提供的用例描述、假设条件、前置条件、步骤序列和后置条件来描述用例。功能包和子包早已清楚地指明这个用例的发起参与者(Server)和受益参与者(Customer)。

对这个用例的一句话的叙述可以是“服务员将顾客的定单信息输入到他的手提式个人计算机中并将定单信息传递到厨房”。假设条件是顾客想就餐，顾客已经阅读了菜单并做出了选择。另一个假设条件是服务员的手提式个人计算机已经出现了“输入定单”用户界面。

前置条件是顾客已经就坐并阅读了菜单。后置条件是定单被输入进 WIN 系统中。

用例的步骤序列是：

1．服务员激活他的手提式个人计算机的“输入定单”用户界面。

2．“输入定单”用户界面出现在显示器屏幕上。

3．服务员将顾客的菜单选项输入到 WIN 系统中。

4．系统将定单发送到厨房的桌面电脑。

3.4.2 用例“Transmit the Order to the Kitchen”

这个用例至少应被包含在（也就是被使用）两个用例当中——前一个用例和用例“Change an order”。

用例的叙述是:“将输入到手提式个人计算机中的定单通过无线网络传送到厨房的桌面电脑”。假设条件是已经具备了通信手段（通过无线网络）以及具备了“输入定单”用户界面。与其他用例的假设条件重复了的假设条件还要叙述出来吗？是的。每个用例在设计文档中都占单独的页，页号可以作为用例的索引。为了清晰起见，即使与其他用例的假设条件相同，每个用例的假设条件都应该完整地叙述出来。

前置条件是定单信息已经录入到手提式个人计算机中。后置条件是定单被正确传递到厨房的桌面电脑。受益参与者是 Customer。

步骤序列如下：

1．点击“定单”用户界面上的“send to kitchen”按钮。

2．WIN 系统将定单发送进入无线局域网。

3．定单到达厨房的桌面电脑。

4．“输入定单”用户界面出现提示信息，提示定单已经被正确传递到厨房的桌面电脑。

修改 Server 包中的用例图。必须在“Take an order”、“Change and ordrder”这两个用例与本用例之间添加<<include>依赖关系。

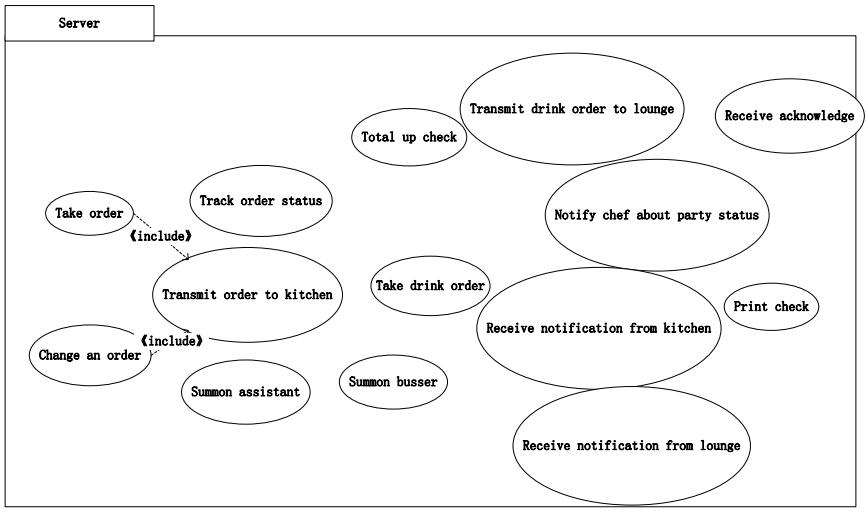


图 3.20　修改后的 Server 包中的用例

3.4.3 用例“Change an Order”

前置条件是定单已经被传给厨房的桌面电脑。后置条件是修改后的定单传到厨房的桌面电脑。受益参与者是 Customer。

本用例的步骤序列是：

1．服务员激活他的手提式个人计算机中的“修改定单”用户界面。

2．屏幕上出现了这名服务员已经发送了的定单列表。

3．服务员在定单列表中选择要修改的定单。

4．服务员录入要修改的定单的修改信息。

5．系统将修改后的定单发送到厨房的桌面电脑。

步骤 5 包含了前面分析过的用例“Transmit the order to the kitchen”。

3.4.4 用例“Track Order Status”

前置条件是定单已被发送到厨房。后置条件是定单的状态信息被传送

到服务员的手提式计算机中。受益参与者是 Customer。

步骤序列包括：

1. 服务员激活他的手提式个人计算机中的“跟踪定单状态”用户界面。

2．屏幕上出现这名服务员已经发送了的定单列表。

3．服务员选择欲跟踪的定单。

4．系统产生一个跟踪消息到厨房的桌面电脑。

5．厨房的桌面电脑收到了这条消息。

6．在厨房的桌面电脑上厨师把要跟踪定单的界面激活。

7．厨师在用户屏幕界面中输入一个时间估计值。

8．系统将厨师输入的时间估计值传给服务员的手提式个人计算机。

3.4.5 用例“Notify Chef about Party Status”

用例叙述

服务员通过无线网络告诉厨师：顾客马上就要吃完小菜。

假设条件

- 服务员呆在顾客所在的服务区。
- 服务员能够端详出顾客的下一步要干什么。
- 系统提供了“顾客状态”用户屏幕界面。
- 系统可以在服务员的手提式个人计算机和厨房的桌面电脑之间双向传递信息

前置条件

- 顾客几乎就要吃完小菜。

后置条件

- 厨师做主菜的最后工序。

步骤序列

1．服务员激活他的手提式个人计算机上的“顾客状态”用户界面。

2．用户界面上出现了服务员所在的服务区中的餐桌列表。

3．服务员在列表中选择餐桌。

4．服务员发送有关这个餐桌的一条“almost finished with appetizer”消息给厨房的桌面电脑。

5．厨房的桌面电脑接收到了这条消息。

6．服务员收到了来自厨房桌面电脑的一个确认应答消息。

最后一步使用了 Server 包中的用例“Receive acknowledgement”。

3.4.6 用例“Total Up a Check”

用例叙述

在定单中添加定单条目。

假设条件

- 系统中有一个能够通过服务员手提式计算机访问的定单数据库。
- 定单中的每个条目都有标价。

前置条件

- 一组顾客就餐完毕。

后置条件

- 计算出账单的总价格。

步骤序列

1．服务员激活有关的用户界面，使一个活动的定单条目列表出现在他的手提式个人计算机的屏幕界面上。

2．服务员选择要结账的菜单。

3．服务员点击屏幕上的一个按钮，计算账单总价格。

4．系统根据每个定单条目的价格计算出账单总价，并显示在屏幕上。

受益参与者

- Customer

3.4.7 用例“Print a Check”

用例描述

打印计算出总价的账单。

假设条件

每个服务区都有一台（无线）网络打印机。

前置条件

- 账单总价格已被计算出。

后置条件

- 打印出一份账单。

步骤序列

1．服务员点击一个按钮。

2．网络打印机开始打印。

3.服务员再点击一个按钮将这个账单对应的定单从活动定单列表中删除。

受益参与者

- Customer

3.4.8 用例“Summon an Assistant”

用例叙述

要求一名助手清理餐桌，迎接下一组顾客的到来。

假设条件

- 系统中允许两名雇员之间进行无线通信。
- 系统提供了用于向一名助手发送消息的用户界面。

前置条件

- 存在一个要被清理的餐桌。

后置条件

- 助手及时赶来，并清理和调整餐桌。

步骤序列

1．服务员激活用来给助手发送消息的用户界面，并给他发消息。

2．服务员接收到来自助手的一个确认应答消息。

要包含用例“Notify chef about party status”，最后一步也要使用“Receive Acknowledgment”用例。

受益参与者

- Assistant

通过分析这个用例和 Assistant 包中的其他用例，可以发现将 Assistant 类进一步划分为丙两个类 AssistantServe 和 AssistantChef 是必要的（可使用例的描述更清晰）。那么需不需要这两个类的一个抽象超类 Assistant 呢？也许需要，但这样做很可能不会带来多大益处，反而使问题复杂化。

必须回到领域分析的结果中创建这两个新类。重新调整后的类图，特别是调整了 Employee 类后的类图如图 3.21 所示。

此外，还要修改包图，它要包括 Assistant Server 包和 Assistant Chef 包。

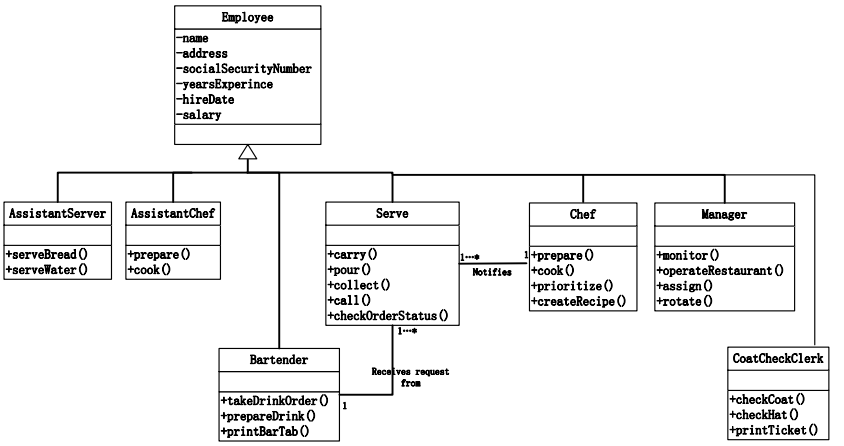


图 3.21　　调整后的 Employee 类图

3.5 系统中的构件

用例分析的一个重要方面是揭示出组成系统的构件。每个用例的“假设条件”、段可以找到这些构件。

从软件构件来看，显然要包括一组用户界面。WIN 系统需要包括胎『员手提式计算机上现过的“输入定单”、“修改定单”、“跟踪定单状态”、“顾客状态”以及“发送消息给助手”等用户屏幕界面。为了便于组织这些屏幕界面，还可以设计一幅“主屏幕”。WIN 系统还需要厨房的桌面电脑上的用户界面，以帮助厨师察看和跟踪定单状态。通常，这些用户界面都该显示“主屏幕”，接收用户输入并显示信息。如果餐馆真想取悦顾客，所有的用户界面都应该能够跟踪定单以及某个顾客的状态。这样，任何人只需要访问 WIN，就可以回答顾客的问题，并及时注意到顾客的状态。

另一个明显的软件构件是存储和管理所有定单信启、的数据库。数据库的记录要包括餐桌、定单号、定单录入时间、服务员姓名、定单是否处于活动状态等数据字段。

当然，我们还需要一个定单处理程序，它工作在创建定单的接口后面，

把定单发送到指定的地方并在数据库中登记下来。

从硬件构件来看，需要一个无线局域网络、可移动雇员（服务员、助手、清洁师）使用的手提式个人计算机以及经理办公室、厨房和休息室里要安装的桌面电脑。每个服务区还要安装一台网络打印机。也许衣帽保管员还需要配备电脑和打印机。

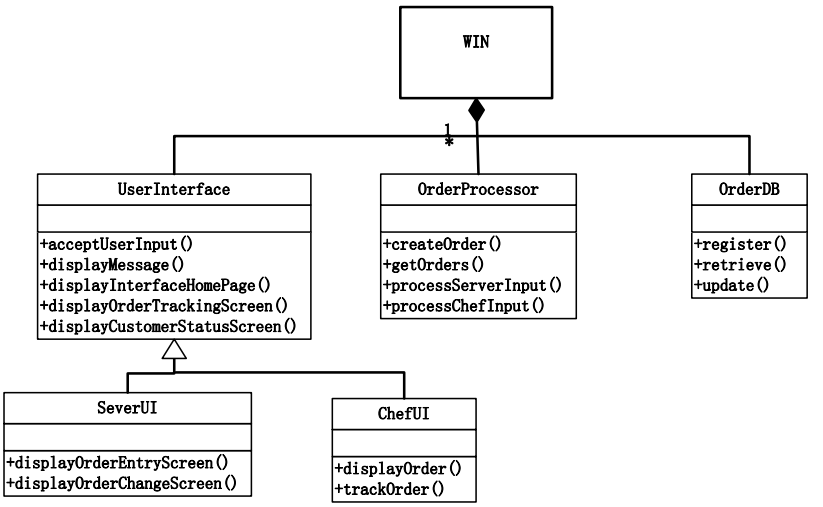


图 3.22　WIN 系统的构件的建模

定单处理程序和定单数据库必须驻留在计算机上。一种可能的方案是，定单处理程序和数据库存储在一台中央计算机上，而网络上所有其他的机器都能够访问到它们。无线网络则使得手持计算机、台式机和中央计算机之间可以实现无线通信。

3.6 系统中的交互

下面要说明系统中的构件如何交互以实现用例所代表的功能，每个用例的背后都隐藏一张顺序图。

3.6.1 用例“Take an Order”

这个用例包含了“Transmit the order to the kitchen”，此用例的步骤序列为：

1．点击“定单”用户界面上的“send to kitchen”按钮。

2．WIN 系统将定单发送到无线局域网。

3．定单到达厨房的桌面电脑。

4．“输入定单”用户界面出现提示信信息，提示定单已经被正确传递到厨房的桌面电脑。

一张顺序图将会精确地表示出用例中的交互（协作图也起同样的作用）。

为了绘制顺序图我们必须要考虑几方面的问题。

首先，当服务员接到了顾客的定单，实际上，服务员要创建某种事物——一份定单！它是 WIN 系统中的一个对象（同样也是在前面的领域分析中识别出的 Order 类的一个实例）。厨师将使用这份定单作为行动的依据。服务员还要计算定单中的账目总额。顾客根据总额支付账单。因此这个新创建的定单对象是系统中的重要事物。

其次，如果察看用例“Change an order”和“Track order status”（后面将做这件事），你将看到这两个用例要引用一张定单列表。必须要有一个定单数据库来存放定单列表。定单中的信息如何输入到数据库中？这件事必须在用例中完成。

还要考虑其他方面。在被包含的用例中，“kitchen”这个词的词义比较含糊。因为我们现在要对软件构件建立模型，所以必须在此将它的词义澄清。用我们的常识知识设想一下，就可得出：定单必须要出现在厨师电脑的屏幕界面上。

在考虑到这些方面之后，用例“Take an order”的步骤序列就可以修改成下面的序列：

1．服务员激活他的手提式个人计算机的“输入定单”用户界面。

2．“输入定单”用户界面出现在显示器屏幕上。

3．服务员将顾客的菜单选项录入到定单输入界面中。

4．定单处理程序创建一个定单对象。

5．定单处理程序将定单传送到厨师的界面。

6．定单处理程序将定单输入到定单数据库中。

7．定单处理程序告知服务员，定单已经被传送到厨房并且已经在定单数据库中登记过了。

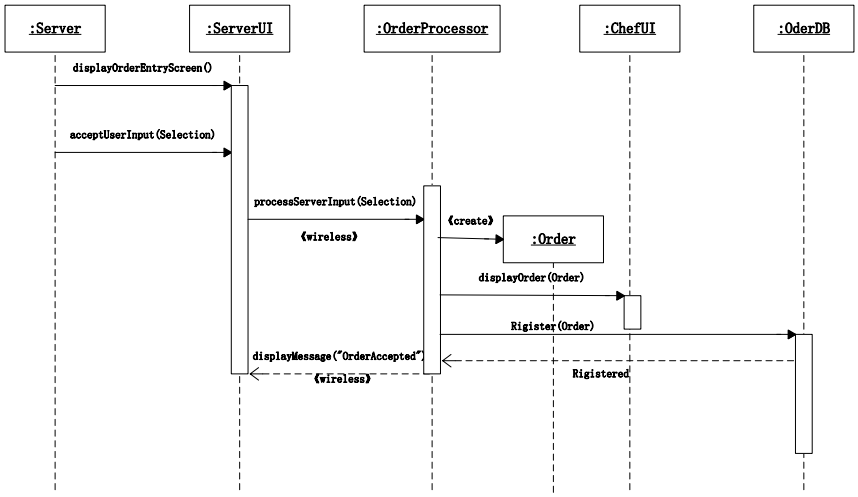


图 3.23　用例“Take an order”的顺序图

图的顶部列出的对象图标代表了用例中的构件。Order 对象是在用例执行期间被创建的，因此它的图标的位置比其他对象的位置低，并且指向它的消息上带有构造型《create》。每个对象向下引出的虚线是该对象的“生命线”，从上到下代表时间的流逝。在生命线上的小矩形框代表该对象的“激活”。每个激活代表了对象正在执行某个动作的持续时间。一条生命线到另

外一条生命线的箭头表示从一个对象到另一个对象的一条消息。箭头的类型代表着消息的类型。Order 对象在这个用例进行过程中创建。因此，它在图中的位置比其他的对象要低，并且指向它的消息都具有《create》构造型。

3.6.2 用例 “Change an Order”

同样，绘制该用例的顺序图可以帮助我们细化和修改用例。在步骤 4 之后，毫无疑问我们想要系统创建一份修改后的定单。在步骤 5 之后，系统应该将修改后的定单信息保存到数据库中。

因此，新的用例步骤序列应当是：

1．服务员激活他的手提式个人计算机中的“修改定单”用户界面。

2．屏幕上出现了这名服务员已经发送了的定单的列表。

3．服务员在定单列表中选择要修改的定单。

4．服务员重新录入要修改的定单的修改信息。

5．定单处理程序将修改后的定单发送到厨房的桌面电脑。

6．定单处理程序输入新的定单到定单数据库中。

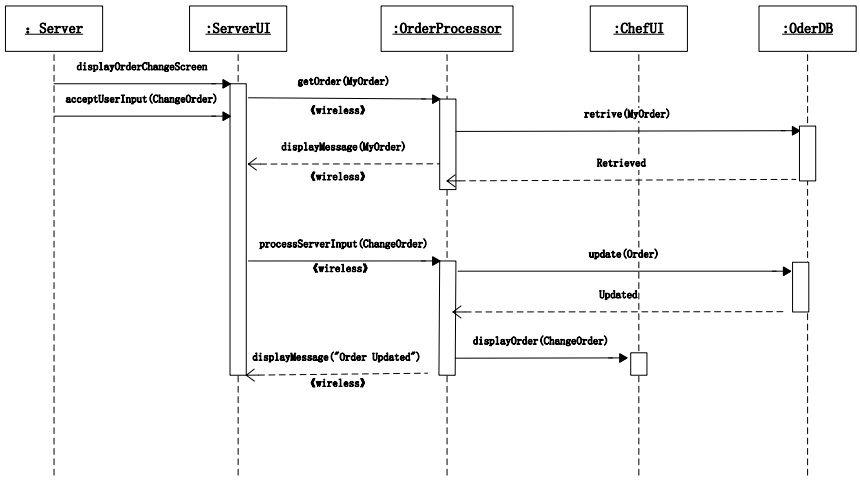


图 3.24　用例“Change an order”的顺序图

3.6.3 用例　“ Track Order Status “

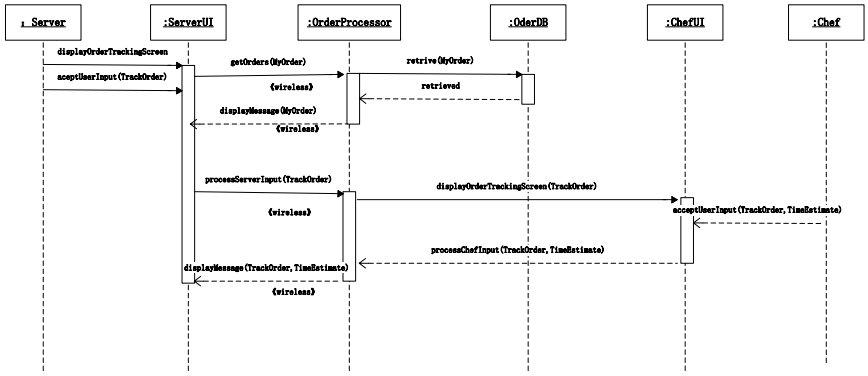


图 3.25　用例“Track an order”的顺序图

3.7 设计外观、感觉和部署

3.7.1GUI 设计的一般原则

主要设计原则：

1．理解用户要做什么。典型的用户界面设计都要进行任务分析（task analysis）来理解用户任务的性质。前面介绍过的用例分析大致相当于任务分析。

2．让用户在与系统的交互过程中有掌握控制权的感觉。无论何时用户发起的交互都应该可以被取消。

3．要提供给多种方式来完成每个与界面相关的动作（例如关闭一个窗口或者文件）并且能够友好地容忍用户操作中的错误。

4．由于受习惯影响，我们的眼睛通常对屏幕的左上角最敏感，可以将重要的信息放在屏幕左上角。

5．充分利用空间关系。屏幕上的图形构件之间的距离不要太远，必要的时候可以用一个框将它们包围起来。

6．重视可读性和可理解性（文字是我们赖以生存的东西）。系统应使用主动语气与用户交流。

7．即使能够在屏幕上添加很多种颜色，也要限制颜色的数量。颜色的使用要慎重。太多的色彩会分散用户对手头任务的注意力。另外，一个好的做法是让用户选择和修改颜色。

8．如果想用颜色来表达某种含义，别忘了对用户来说，要理解某种颜色的含义不是很容易的一件事。还要记住，有一些用户（大约 10%左右）对一些颜色容易搞混淆，分色有一定困难。

9．和使用颜色的限制一样，字体也不能滥用。要避免使用斜体或者过

于华丽的字体。

10．尽量保持界面构件（例如按钮和列表框）的尺寸相同。如果使用了大小不一的类似构件、太多的颜色和字体，这种设计被 GUI 设计专家称为“小丑—喘气”式设计。

11．构件和数据域应当左对齐——按照左边界将这些构件对齐。这样在用户浏览屏幕时可以减少眼球的移动范围。

12．当用户阅读和处理信息并单击按钮时，按钮应该放在信息框的右边并排成一列，或者放在信息框的右下方的一行中。这样的布局符合我们从左到右的阅读习惯。如果其中有一个按钮是默认按钮，应该加亮显示它，并将它设置在第一个按钮的位置。

3.7.2 从用例到用户界面

用例的顺序图只是用例的一个角度的视图。如果我们能够将顺序图“旋转”成三维的，使顺序图最左边面向读者，从这个角度观察，就得到了用户界面。

让我们考察 Server 包中的用例，并看看这些用例如何映射到 WIN 用户界面。

Server 包的接口必须考虑到所有这些用例。

一种方式是将这些用例划分为不同的组。对于上述用例来说，3 个组就足够了。一组处理定单 (“Take an order“、‘‘Change an order”、 “Track order status’． “Take a drink order”)o 另一组处理账单(“Total up a check”、“Print check”)。第三组针对发送和接收消息（“Notify chefabout party status”、“Summon an assistant“、“Sununon a busser”、“Transmit drink order tolounge “、 “Receive acknowledgment”、‘‘ Receive notification from lounge “） 。

应当从一个主屏幕开始，然后开始讨论各个用例组所对应的屏幕。应当可以从一组用例的界面导航到另一组用例。在一组中的用例所对应的界面之间应当能相互导航。图 3.26 是最开始的主屏幕。这个屏幕用于手提电脑，因此它的尺寸要进行必要的压缩。

图 3.27 显示了 Server 包接口中的第一屏，也就是显示与定单有关舵的用例的一屏。. 这个屏幕以 Take Order 模式打开。屏幕中自色框是可滚动的显示菜单的区域，服务员可以在这个区域中查看顾客所选择的菜肴品种。点击“OK”按钮就生成定单并且将定单发送到厨房的 PC 机中。点击屏幕右部的按钮就可以导航到其他窗口。

点击底下的一排按钮可以引发单独的一组功能。例如，“Message”按钮，可以产生如图 3.28 所示的屏幕。另外，用户界面不一定要完全可视，界面可以包含一个语语音信号通知服务员一个消息的到来。用户可以点击“Read”按钮读取消息的信息。

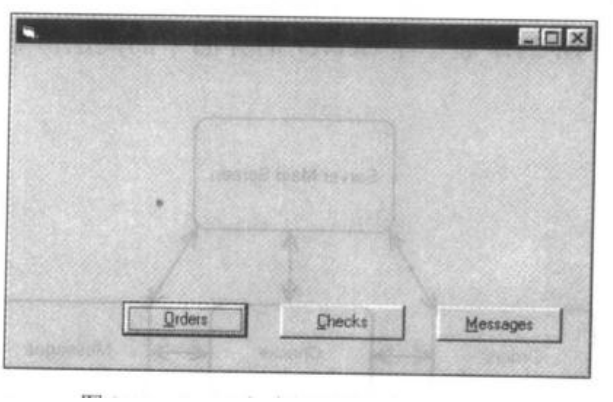


图 3.26　Sever 包中用例的最初的主屏幕

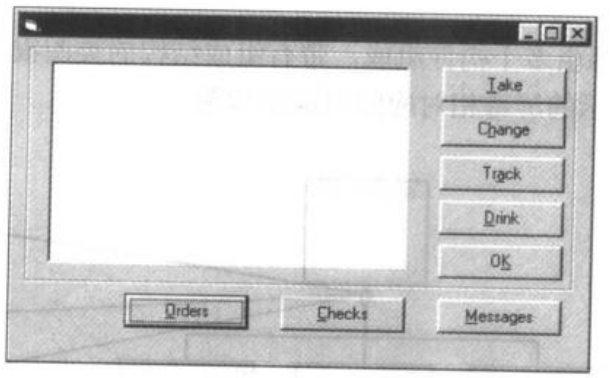


图 3.27　与定单有关的用例的屏幕界面

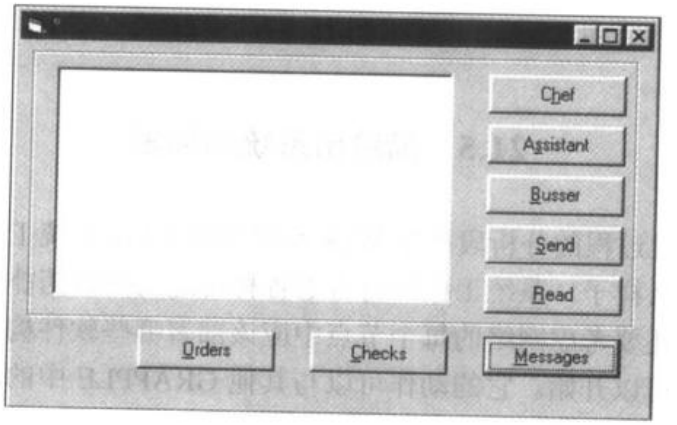


图 3.28　与消息有关的用例的屏幕界面

3.7.3 用于 GUI 设计的 UML 图

UML 并没有给出具体的有关如何进行 GUI 设计的建议。可以使用状态图来描述用户界面的切换流程。

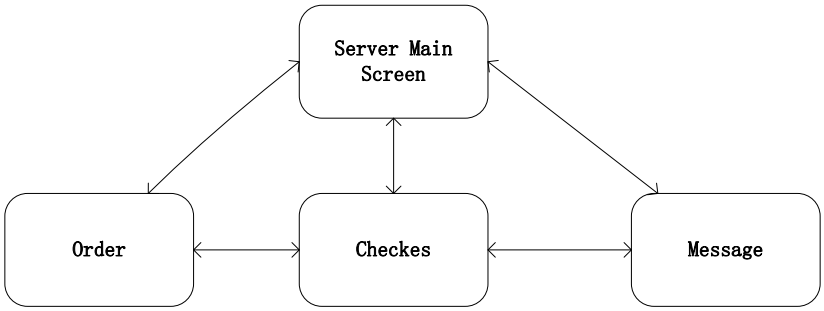


图 3.29　Server 的界面中高层屏幕的切换流程

由于一个特定的屏幕由多个构件组成，带有组成结构的类图很适合用来对屏幕建模。图 3.30 是一个和图 3.28 所示的屏幕相对应的组成结构图。

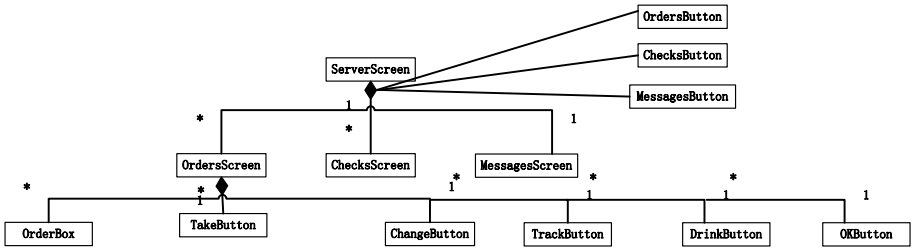


图 3.30　对应于图 3.28 的界面构件类的组成关系图

3.7.4 描绘出系统的部署

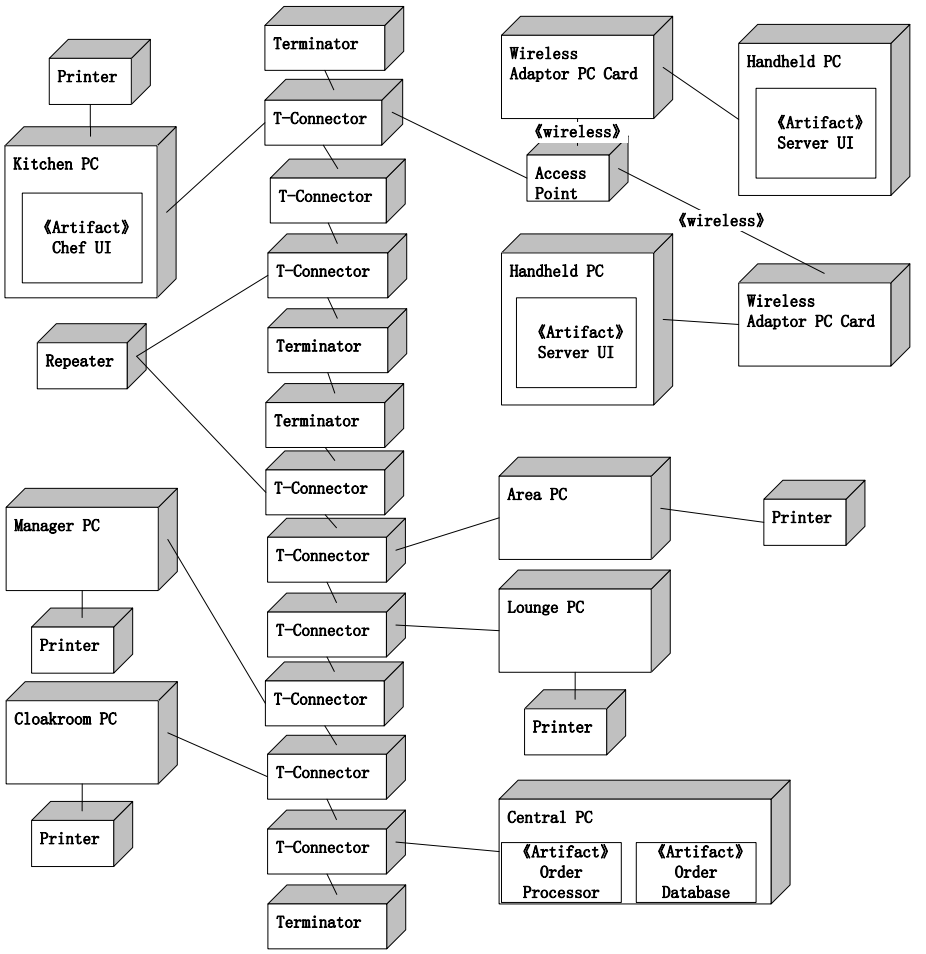


图 3.31　WIN 系统的原始部署图

设计模式和面向对象设计的基本原则

- 设计模式的思想根源是基本原则的宏观运用,本质上是没有任何模式的
- 发现模式的人永远是大师,而死守模式的人,最多只能是一个工匠.

设计模式 Design Pattern

- 面向对象研究的新领域
  - 20世纪90年代，面向对象方法与技术在国内软件业界十分火爆，人们热衷于谈论“对象”并引以为荣。十多年来，人们发表、出版了无数的文章和书籍。现在，该写的似乎都写完了，没有新花样玩了，真是一片无聊
  - 设计模式(Design Pattern)及时问世，面向对象爱好者们终于有了新的追求

3

设计模式：起源

- 起源
  - **Christopher Alexander**
  - 当代著名建筑大师
  - 加州大学伯克利分校建筑学教授、环境结构研究所所长、美国艺术与科学院院士
  - 在建筑、室内、计算机、家具设计甚至哲学方面都卓有建树
  - 著作：《A Pattern Language》、《The Timeless Way of Building》

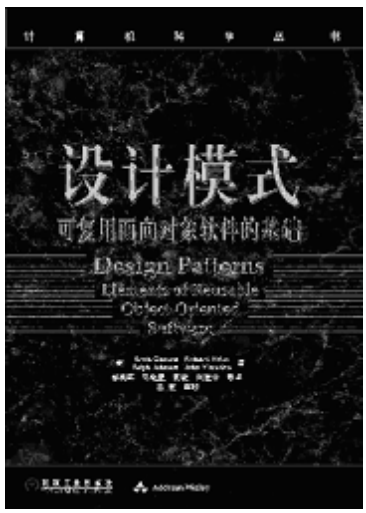
4

设计模式：起源

- **Gof (Gang Of Four, “四人帮”)**
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - **1995**年出版了《Design Patterns: Elements of Reusable Object-Oriented Software》
  - 该书确立了设计模式这个术语，创导了一种新的面向对象设计思潮。从此，参与设计模式研究的人数爆炸性地增长

5

设计模式：起源



6

设计模式

- 什么叫模式？
  - “每一个模式描述了在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该解决方案而不必重复劳动”
  - 尽管软件技术发展非常快，但是仍然有非常多的设计模式可以让我们套用
  - 设计模式可以帮助人们简便地复用以前成功的设计方案，提高工作效率

7

设计模式：研究现状

- 设计模式的研究现状
  - **pattern** 与 **Java**、**C#**
  - **pattern** 与 组件技术(如**CORBA**)
  - **pattern** 与 系统结构
  - **pattern** 与 泛型编程(**generic programming**)相结合
  - 其他(例如**UML**等)

8

- 模式的分类(gof提出的23个)

	创建型	结构型	行为型
类	Factory Method	Adapter (类)	Interpreter Template Method
对象	Abstract Factory Builder Prototype Singleton	Adapter (对象) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

9

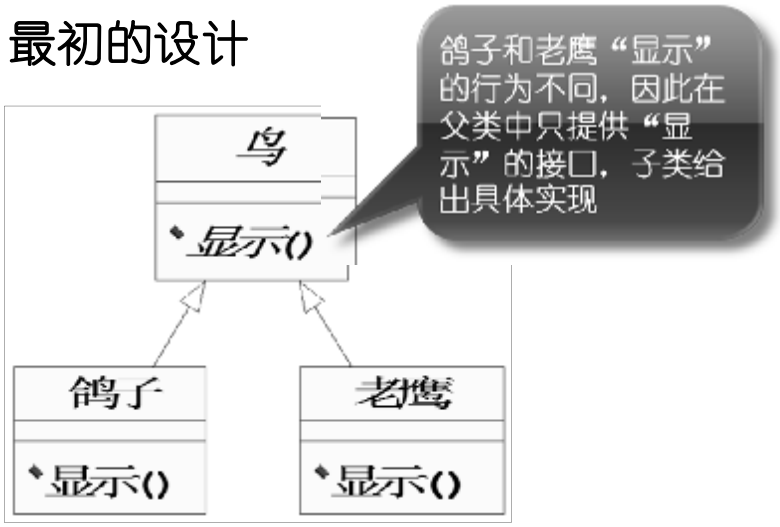
Bridge(桥梁)模式

- 案例
  - 有一个叫做**HuntBird**的游戏，里面需要表示各种各样的鸟类

10

Bridge(桥梁)模式

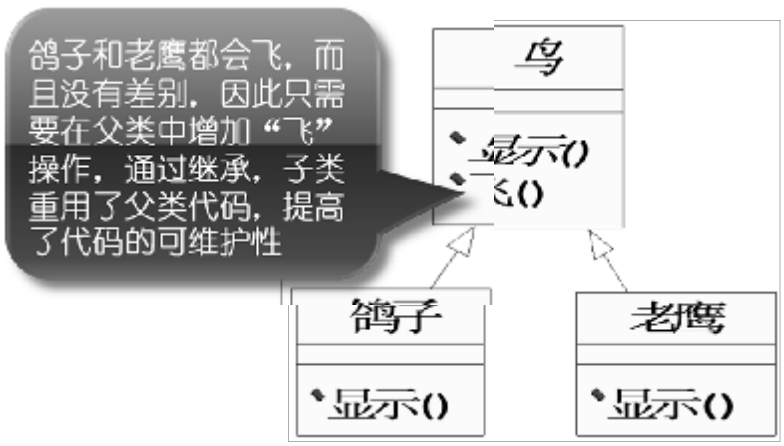
- 最初的设计



11

Bridge(桥梁)模式

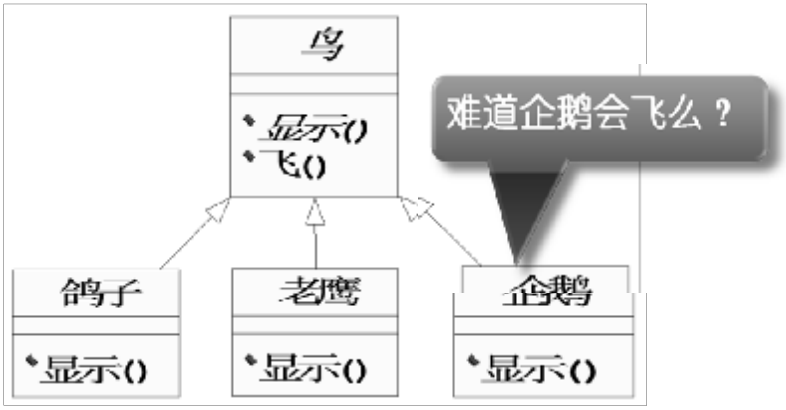
- 需求变化：鸟类要会飞



12

Bridge(桥梁)模式

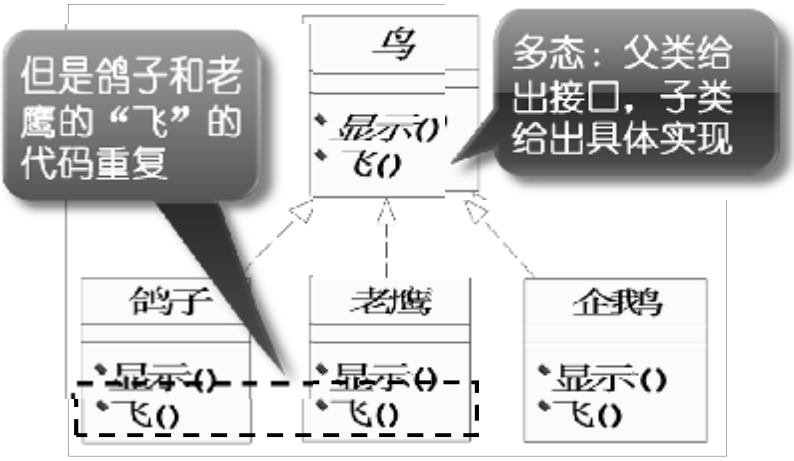
- 如果增加一种鸟类“企鹅”呢？



13

Bridge(桥梁)模式

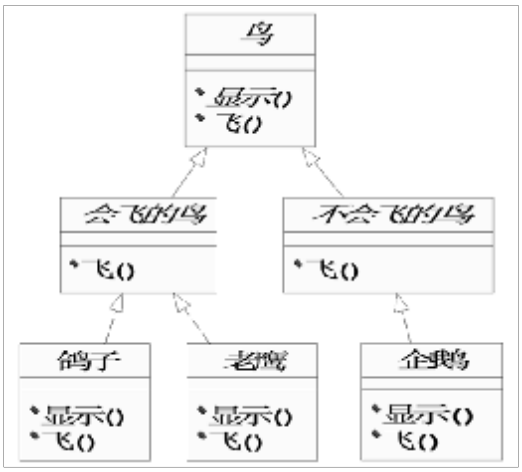
- 改进方法：对“飞”使用多态



14

Bridge(桥梁)模式

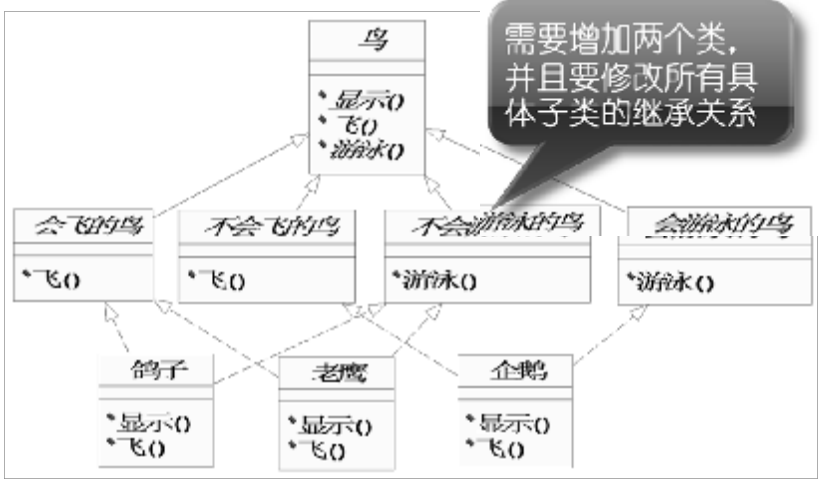
- 改进方法：再次使用继承



15

Bridge(桥梁)模式

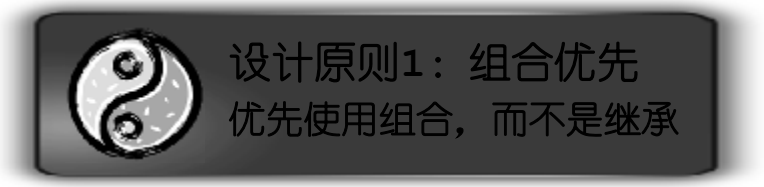
- 如果增加“游泳”行为呢？



16

Bridge(桥梁)模式

- 继承只会使得问题越来越复杂

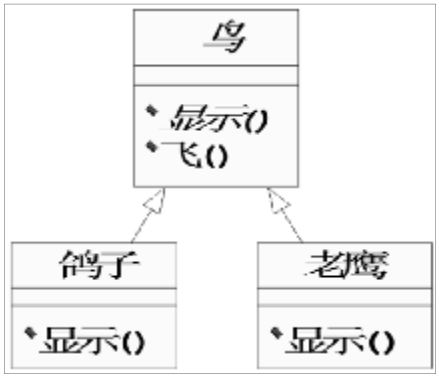


- 继承是面向对象的基本法宝啊？？
  - OO=类＋对象＋继承＋消息通信

17

设计原则：组合优先

- 继承复用的优点
  - 可以很容易的修改或扩展父类的实现



18

设计原则：组合优先

- 继承复用的缺点
  - 继承破坏封装，因为父类的实现细节完全暴露给子类(白盒复用)
  - 父类的实现发生改变，则子类必受牵连
  - 继承是静态的，不能在运行时发生改变，不灵活

19

设计原则：组合优先

- 组合复用的优点
  - 不破坏封装，这种复用是黑盒复用，因为成员对象的内部细节对新对象保密
  - 所需依赖少(只依赖接口)
  - 是动态的，可以把成员对象动态替换为另一个类型相同的对象
- 组合复用的缺点
  - 对象数量会增加
  - 使用委托(delegation)会使得系统复杂

20

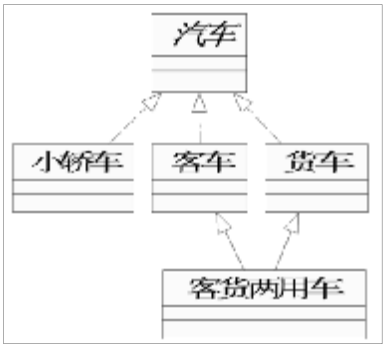
设计原则：组合优先

- 组合优先
  - Favor composition over inheritance
  - 当需要应对变化的时候，应该首先使用组合的方式，而不是继承
  - 因为组合更加灵活
- 例1：
  - 汽车有很多种，小轿车、货车、客车，有的车是客货两用，有的车水陆两用

21

设计原则：组合优先

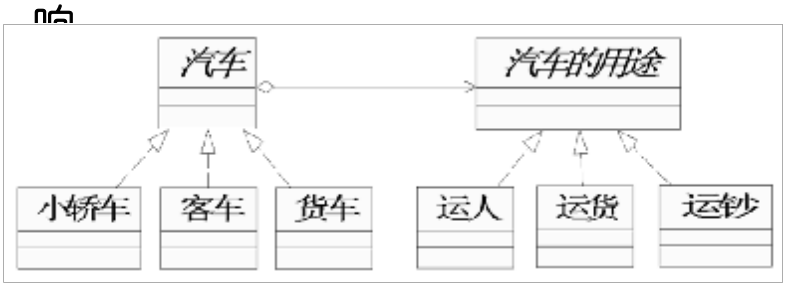
- 如果使用继承来描述：
  - 一旦增加新的汽车种类或用途，都需要大量改动原有代码



22

设计原则：组合优先

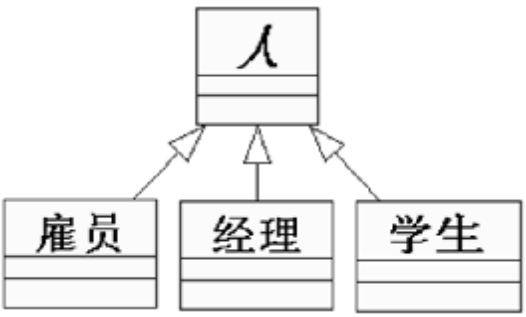
- 使用“组合”思路考虑问题
  - “汽车”拥有某种或某些“用途”
  - “汽车”和“用途”独立变化，互不影响



23

设计原则：组合优先

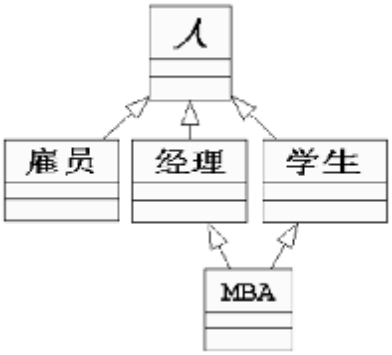
- 区分“Is-A”与“Has-A”
  - 有一个系统需要描述经理、雇员和学生
  - 它们都是人，所以：



24

设计原则：组合优先

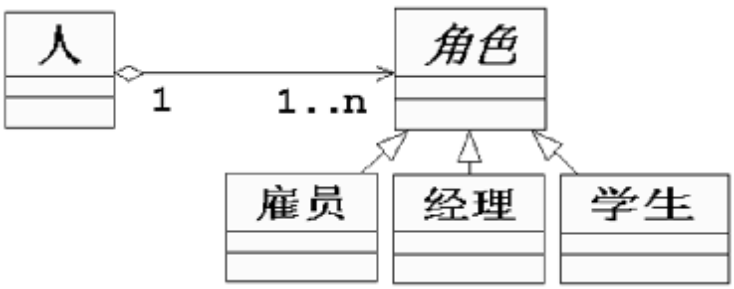
- 问题
  - 有些人既是经理，又是学生，比如某位在读**MBA**的老总



25

设计原则：组合优先

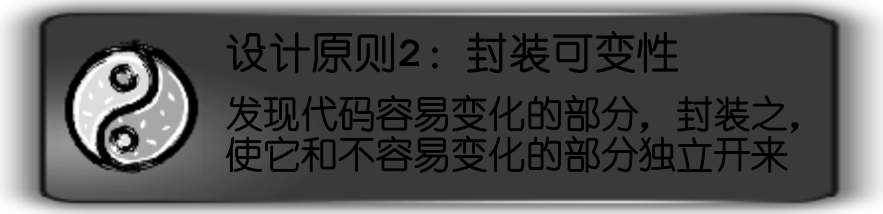
- 换一个角度看问题
  - 雇员、经理、学生其实都是角色的一种
  - 人拥有角色



26

Bridge(*桥梁*)模式

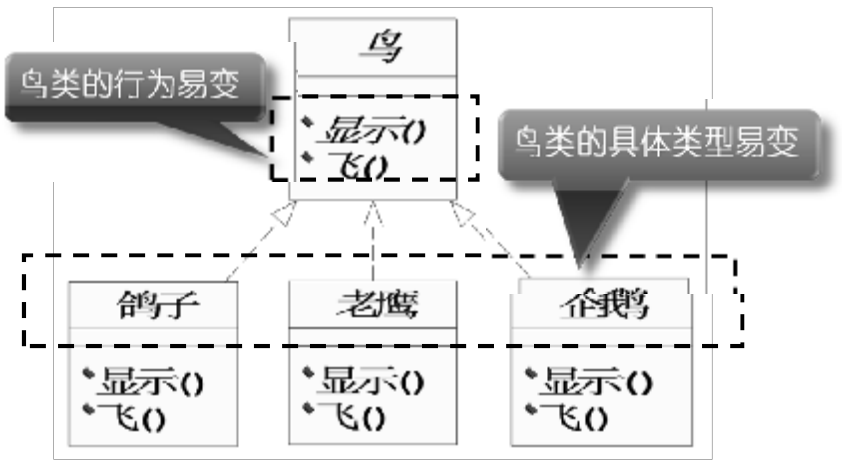
- 是什么导致设计的不完美？
  - 变化，无法避免的、经常的需求变化
- 设计者的理想
  - 当需求变化的时候，尽可能少的修改代码就可以满足新的需求



27

Bridge(*桥梁*)模式

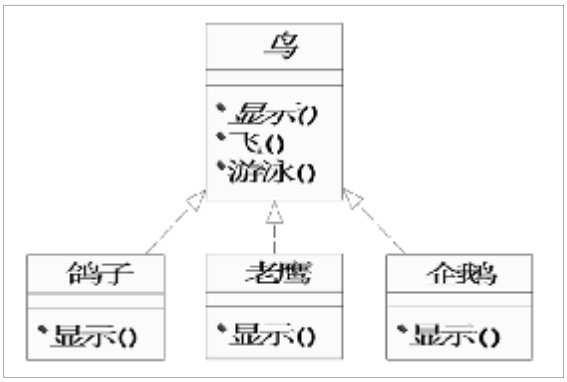
- “发现变化点”



28

Bridge(桥梁)模式

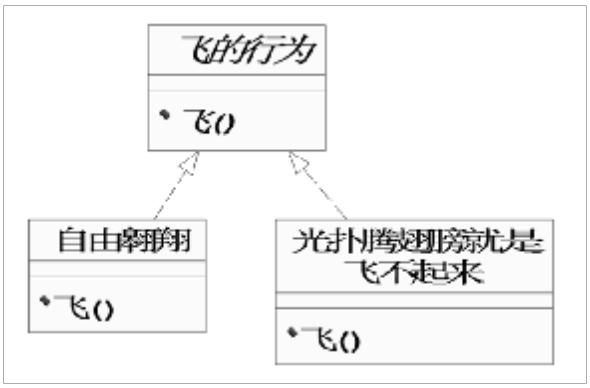
- “封装变化点”
  - 变化点1：小鸟一家



29

Bridge(桥梁)模式

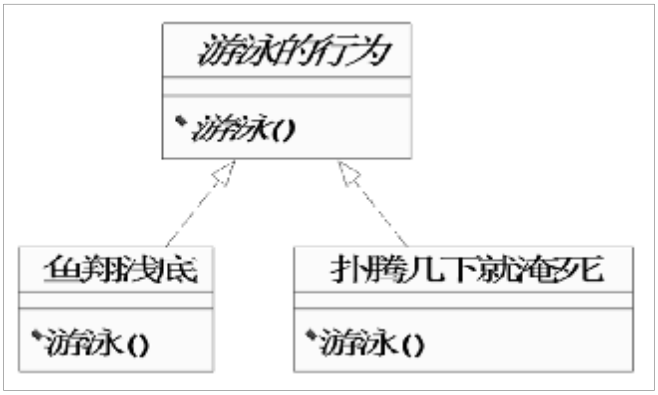
- “封装变化点”
  - 变化点2：鸟类的行为——飞



30

Bridge(桥梁)模式

- “封装变化点”
  - 变化点2：鸟类的行为——游泳



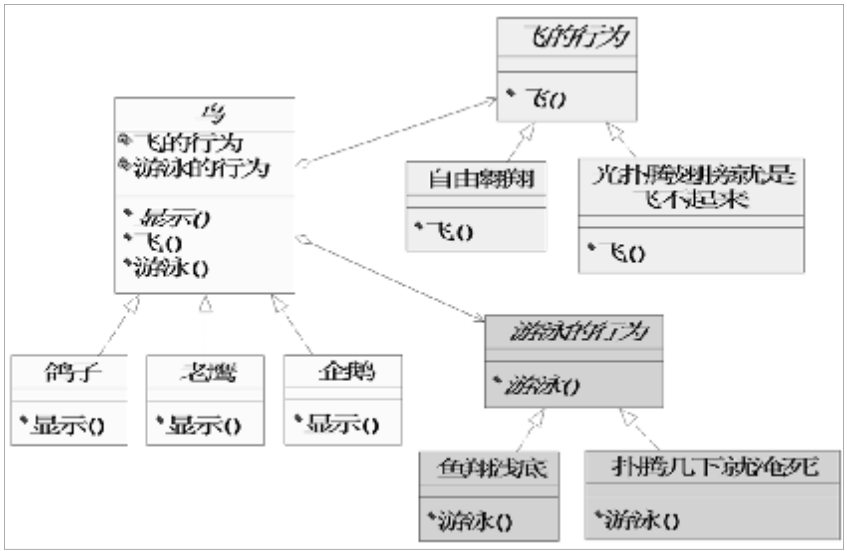
31

Bridge(桥梁)模式

- “使变化点和不变点独立开来”
  - 在这个例子里其实是两个变化点相独立
- “鸟类”和“行为”什么关系？
  - 鸟类拥有行为
  - 鸟类行为的具体实现，委托“行为”类来完成

32

- 鸟儿拥有飞、游泳的行为



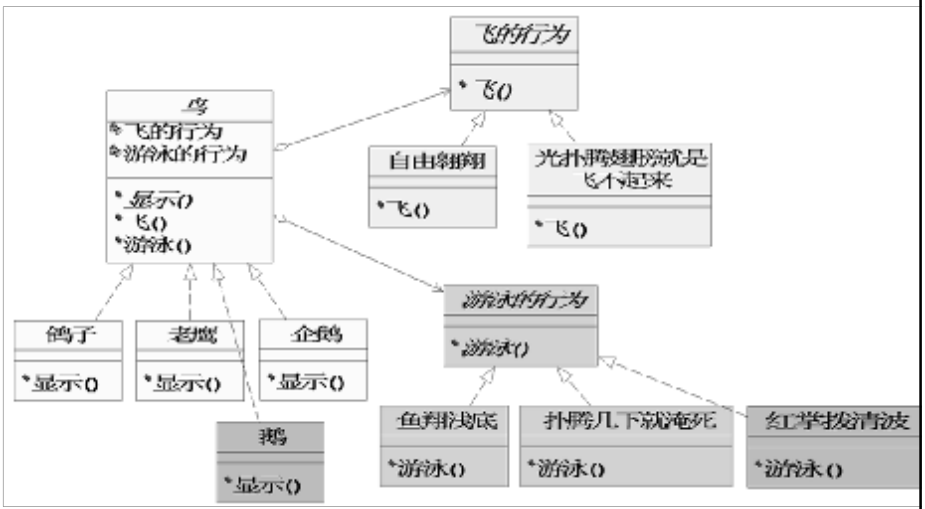
33

Bridge(桥梁)模式

- 使用桥梁模式的效果
  - 比如增加一种鸟类“鹅”，相应的要增加一种游泳的行为“红掌拨清波”
  - 只需要增加一个鸟类的子类“鹅”
  - 增加一个游泳的行为“红掌拨清波”
  - 设置“鹅”的飞翔行为为“飞不起来”
  - 设置“鹅”的游泳行为为“红掌拨清波”
  - 原有代码不需要改动！

34

Bridge(桥梁)模式



35

Bridge(桥梁)模式

- 使用桥梁模式的效果
  - 当需求改变的时候（增加动物或行为），只需要简单添加几个类
  - 对原有代码不需要改动
  - 保证了代码的稳定，提高了可维护性

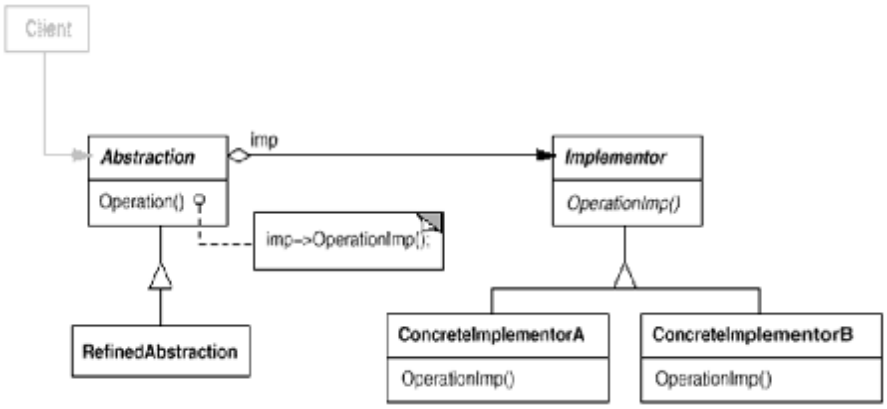
设计原则3：开-闭原则

在设计一个软件的时候，应当使这个软件可以在不被修改的前提下扩展

36

Bridge(桥梁)模式

- 结构



37

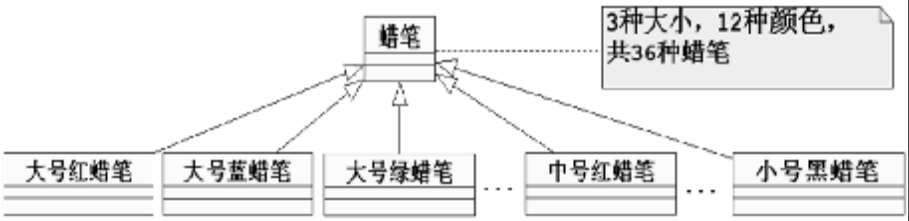
Bridge(桥梁)模式

- 意图
  - 将抽象部分与它的实现部分分离，使它们都可以独立地变化
- 适用性
  - 抽象和它的实现部分可以独立变化
  - 类的抽象以及它的实现都可以通过生成子类的方法加以扩充
  - 实现部分的修改不会对客户产生影响
  - ……

38

Bridge(桥梁)模式

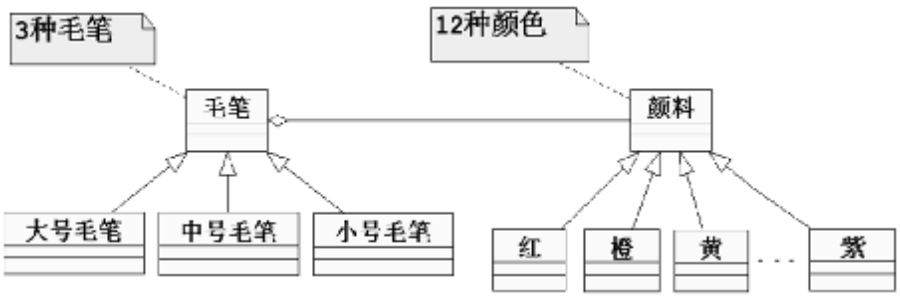
- 应用举例1：“小朋友画画”
  - 使用蜡笔
    - 需要大中小三种型号
    - 每种型号各有12种颜色
    - 共36支



39

Bridge(桥梁)模式

- 使用毛笔：
  - 大、中、小3支毛笔
  - 12种颜料



40

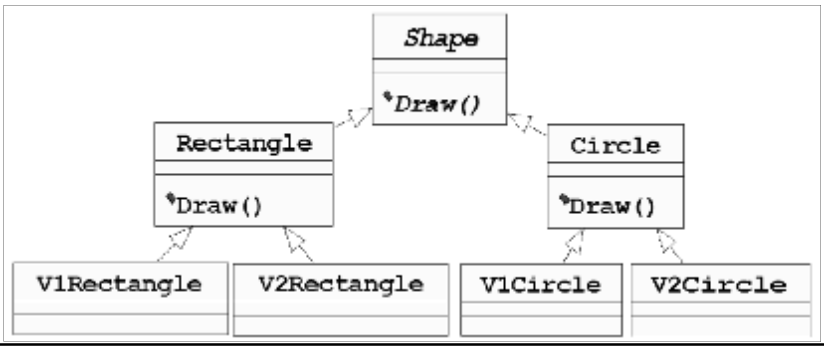
Bridge(桥梁)模式

- 蜡笔和毛笔的差别
  - 蜡笔：笔和颜色无法分离，因此需要36种蜡笔
  - 毛笔：笔和颜色可以独立选择，因此只有3+12=15个子类
- 体现了Bridge模式
  - 将继承关系转换为组合关系，从而降低了系统间的耦合，减少了代码冗余

41

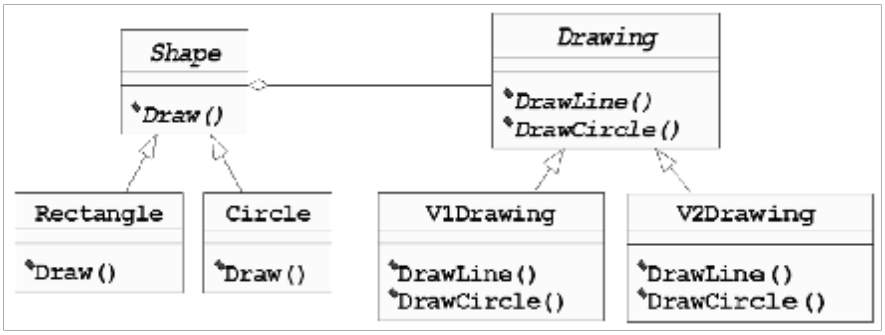
Bridge(桥梁)模式

- 应用举例2
  - 有一个CAD软件，可以画多种图形
  - 同时支持多套绘图算法
- 传统的设计



Bridge(桥梁)模式

- 应用Bridge模式



43

Bridge(桥梁)模式

- 分析
  - 图形Shape是一个抽象概念，它可以有许多具体化(变化点1)
  - 图形的显示Drawing是图形的实现，它也可以有许多套算法(变化点2)
  - Bridge模式使用组合代替继承，避免了复杂的继承体系，使得两个变化点独立变化，互不影响

44

设计原则：开-闭原则、封装可变性

- “开-闭”原则
  - Bertrand Meyer: “Software should be open for extension, but closed for modification”
  - 在设计一个软件的时候，应当使这个软件可以在不被修改的前提下扩展
- 解释
  - 已有模块，尤其是最重要的抽象层模块不能动：保证稳定性和延续性
  - 可以扩展新模块：增加新行为，保证灵活性

45

设计原则：开-闭原则、封装可变性

- **Bertrand Meyer**
  - 对象技术大师
  - 法国工程院院士
  - 苏黎世工学院计算机系教授
  - 发明了Eiffel语言和按契约设计(Design by Contract)的思想
  - 早年参与了Z形式语言的设计
  - 名著《面向对象软件构造》

46

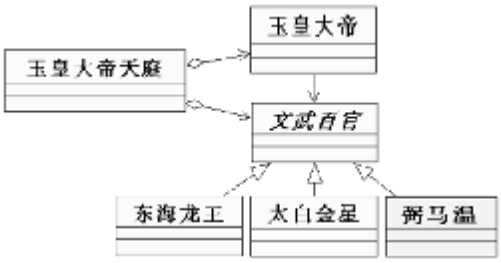
设计原则：开-闭原则、封装可变性

- 玉帝遵照“开-闭”原则维护天庭秩序
  - 当年孙悟空大闹天空，向天庭发出挑战：“皇帝轮流做，明年到我家……只教他搬出去，将天宫让与我！”
  - 太白金星给玉皇大帝建议道：“降一道招安圣旨，把他宣来上界…与他籍名在箓…一则不动众劳师，二则收仙有道也。”

47

设计原则：开-闭原则、封装可变性

- 分析
  - “不动众劳师”、不破坏天规就是“闭”
  - 收仙有道就是“开”
  - 招安，就是玉帝的“开-闭”原则：既让孙悟空满意，又不必更改天庭现有的秩序



48

设计原则：开-闭原则、封装可变性

- 分析
  - 现有的天庭秩序是系统的最高抽象层
  - 弼马温这个职位只是具体的实现层
  - 招安的关键就是不允许更改现有的天庭秩序，但是允许将妖猴纳入到文武百官中，从而扩展了这一秩序的具体实现

49

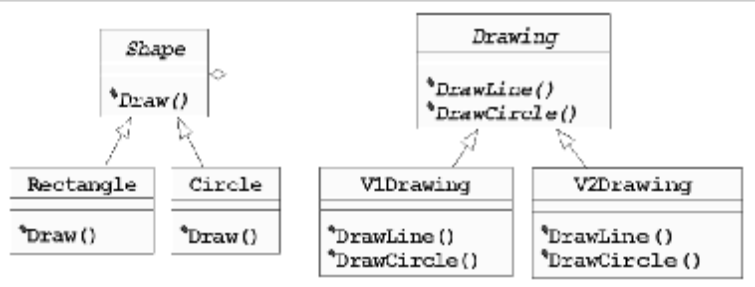
设计原则：开-闭原则、封装可变性

- “封装可变性原则”
  - **gof**：“考虑你的设计中什么可能会发生变化……考虑你允许什么发生变化而不让这一变化导致重新设计”
  - **Shalloway**：“发现变化点，并封装之”
  - 一种可变性不应散落在代码的很多角落
  - 一种可变性不应当与另一种可变性混合在一起

50

设计原则：开-闭原则、封装可变性

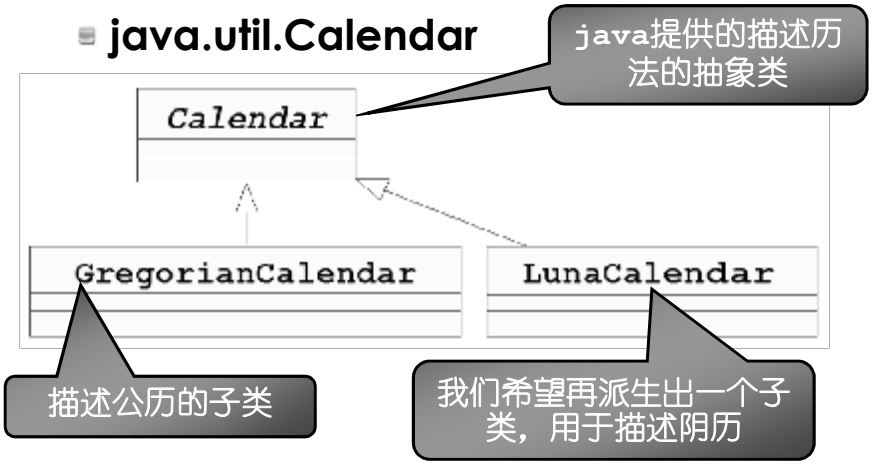
- 设计模式对“开-闭”原则的支持
  - 比如**Bridge**桥梁模式：将抽象部分和实现部分分别封装，可以分别独立变化



51

设计原则：开-闭原则、封装可变性

- 对“开-闭”原则支持的不好的例子
  - **java.util.Calendar**



52

设计原则：开-闭原则、封装可变性

- 问题：
  - Calendar只定义了适用于公历的常量和方法

```
public final static int SUNDAY = 1;
public final static int MONDAY = 2;
...
public final static int JANUARY  = 0;
public final static int FEBRUARY = 1;
...
public void setFistDayOfWeek(int value);
public int getFirstDayOfWeek();
...
```

53

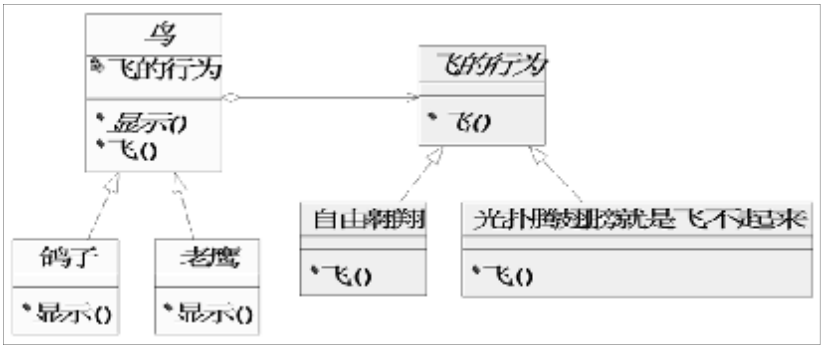
设计原则：开-闭原则、封装可变性

- 问题
  - 英文的星期、月份名称不符合中国阴历的叫法
  - 阴历以10天为一周，公历和阴历每月的天数也不同，所以Calendar关于星期、月份的算法不适合阴历
  - 总之，Calendar无法容纳中国阴历，因此不支持“开-闭”原则

54

Strategy(策略)模式

- 桥梁模式
  - 使得两个变化点的独立



55

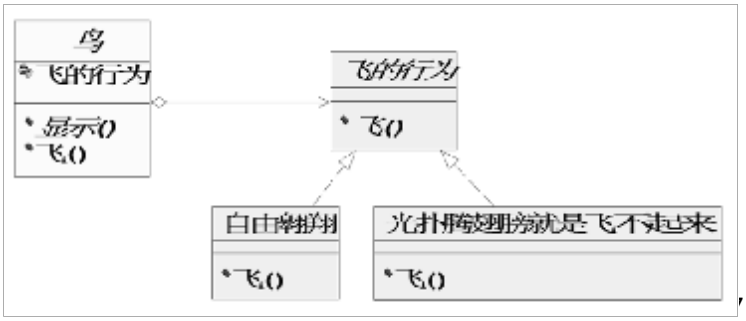
设计原则

- 找出应用中可能需要变化之处
- 把它们独立出来
- 不要和那些不需要变化的代码混在一起

56

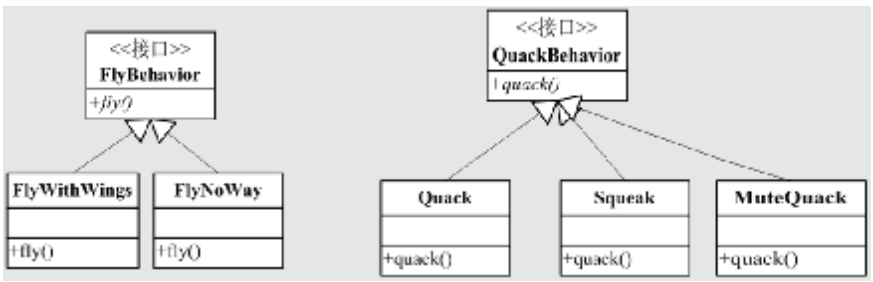
Strategy(策略)模式

- 单独看飞的行为的实现
  - 策略模式：封装了一系列算法，使得它们可以相互替换
  - 效果：算法可以独立变化



设计原则

- 针对接口编程，而不是针对实现编程



58

策略模式-开闭原则 例-计算价格

```
Public class Part{
  private double basePrice;
  public void setPrice(double price){
    basePrice = price;
  }
  public double getPrice(){
    return basePrice;
  }
}
```


某类方法

```
Public double totalprice(Part[] parts){
  double total=0.0;
  for (int i=0;i<parts.length;i++){
     total += parts[i].getPrice();
  }
  return total;
}
```

思考

- 内存折扣？

方法

```
Public double totalprice(Part[] parts){
  double total=0.0;
  for (int i=0;i<parts.length;i++){
     if(parts[I] instanceof Memory)
       total += parts[i].getPrice() * 0.9;
     else
       total += parts[i].getPrice();
  }
  return total;
}
```

思考

- 符合OCP吗？

方法？

```
Public class Memory extends Part{
  public double getPrice(){
    return basePrice * 0.9;
  }
}
```

更好的方法？

- 采用一个PricePolicy类，通过对其进行继承以提供不同的计价策略

方法

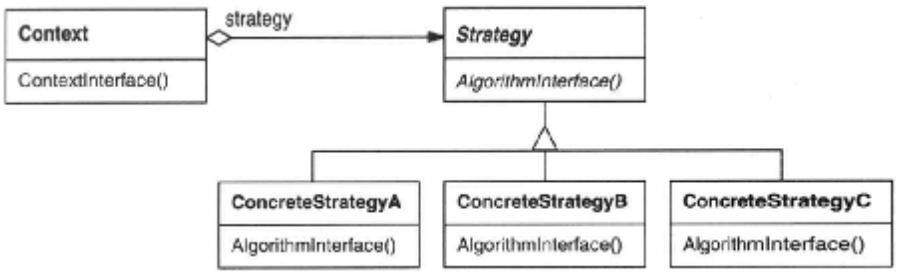
```
Public class Part{
  private PricePolicy pricePolicy;
  public void setPricePolicy(PricePolicy policy){
    pricePolicy = policy;
  }
  public void setPrice(double price){
    pricePolicy.setPrice( price );
  }
  public double getPrice(){
    return pricePolicy.getPrice();
  }
}
```

价格策略

```
Public class PricePolicy{
  private double basePrice;
  public void setPrice(double price){
    basePrice = price;
  }
  public double getPrice(){
    return basePrice;
  }
}
```


销售策略

```
Public class Sale extends PricePolicy{
  private double discount;
  public void setDiscount(double
discount){
    this.discount = discount;
  }
  public double getPrice(){
    return basePrice * discount;
  }
}
```



- **Strategy(策略)**：定义所有支持的算法的公共接口
- **ConcreteStrategy(具体策略)**：实现具体算法
- **Context(上下文)**：
  - 用一个**ConcreteStrategy**对象来配置
  - 维护一个对**Strategy**对象的引用
  - 可定义一个接口来让**Strategy**访问它的数据

69

阶段小结

- 设计原则

组合优先	尽量用组合，而不是继承
开-闭原则	增加新功能，要做到只增加新代码，而不改动老代码
封装可变性	限制变化的影响范围

70

阶段小结

- 策略模式
  - 使得算法可以独立变化
  - 使用组合取代继承，封装了可变性，保证了“开-闭”
- 桥梁模式
  - 使得抽象和实现独立变化
  - 避免了两个变化点的耦合

71

Adapter(适配器)模式

- 例子1：“不合适的插座”
  - 你的电脑的插头是三相的
  - 而墙上的插座只有两相的
  - 插头和插座的“接口”不匹配，怎么办？



72

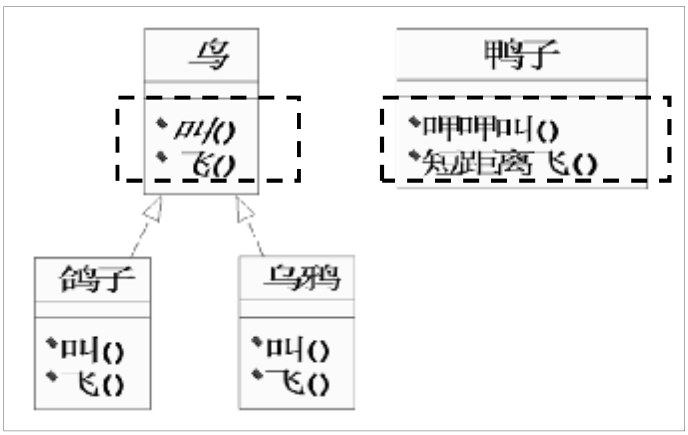
Adapter(适配器)模式

- 例子2：
  - HuntBird游戏中，希望增加一种鸟类“鸭子”
  - 但是发现以前有一个系统中已经有了“鸭子”类，希望重用老代码

73

Adapter(适配器)模式

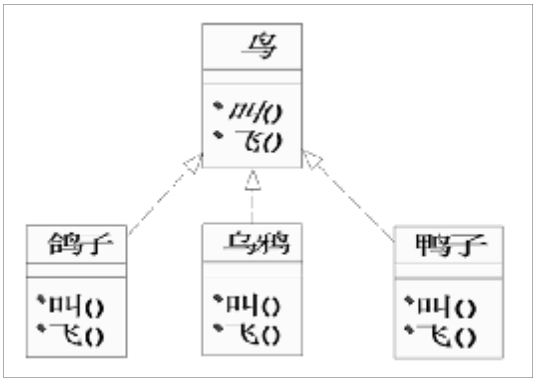
- 新老代码接口不一致



74

Adapter(适配器)模式

- 疑问
  - 把老代码修改一下不就可以了么？如下：



75

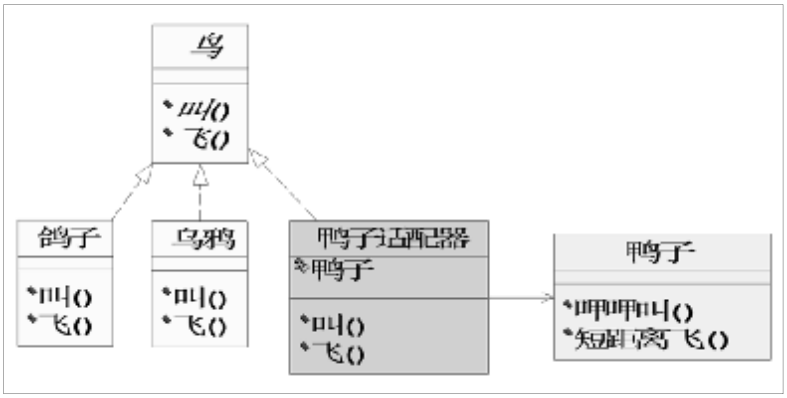
Adapter(适配器)模式

- 否定
  - 首先，老代码不一定允许修改
    - 比如可能根本没有代码，只有链接库
  - 其次，修改代码工作量可能很大
    - 容易出错
    - 还记得“开-闭原则”么

76

Adapter(*适配器*)模式

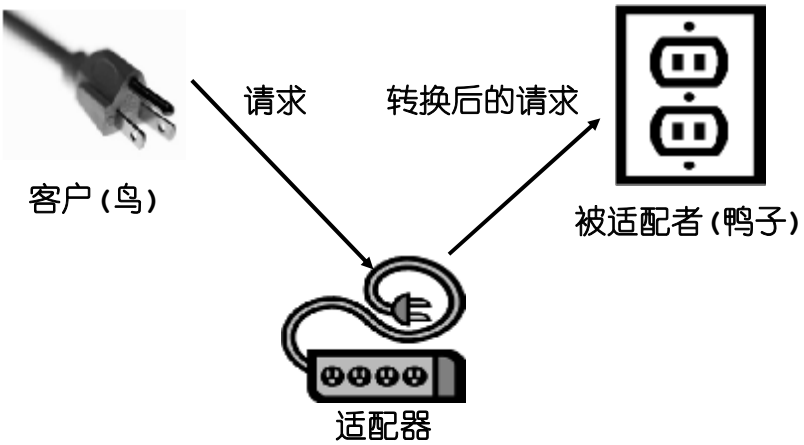
- 应用(对象)适配器模式实现接口转换



77

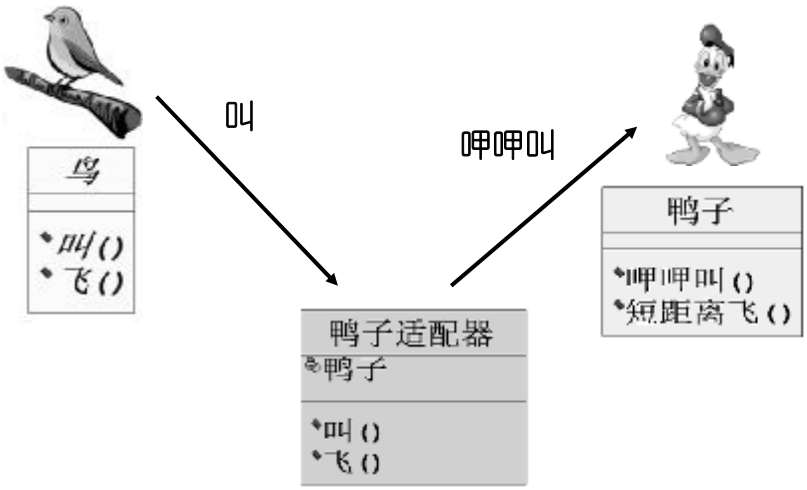
Adapter(*适配器*)模式

- 理解1：接口转换



78

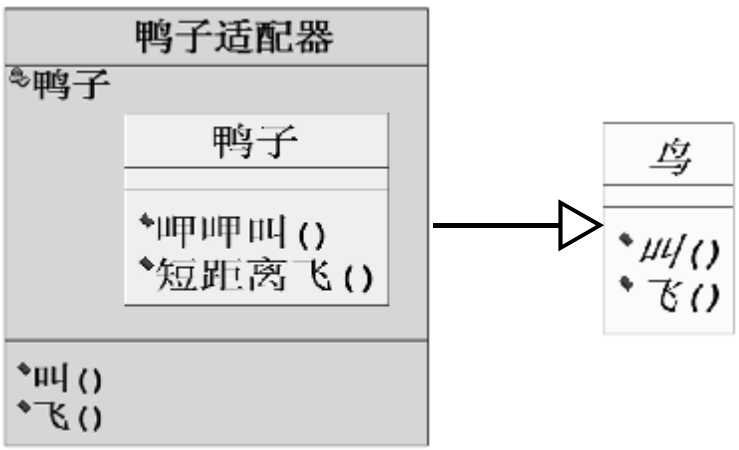
Adapter(适配器)模式



79

Adapter(适配器)模式

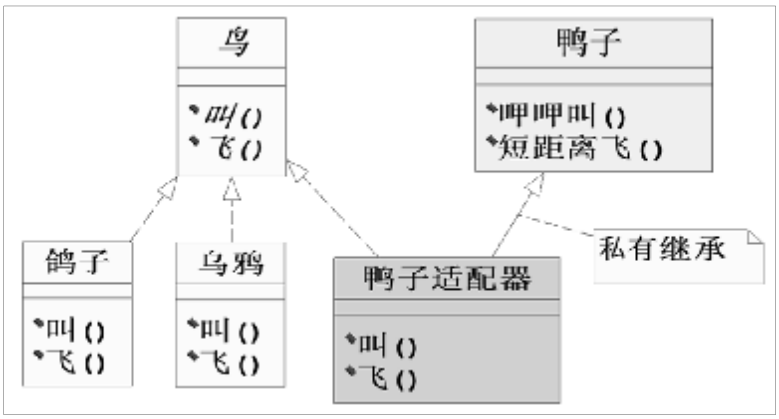
- 理解2：重新包装，改变接口



80

Adapter(适配器)模式

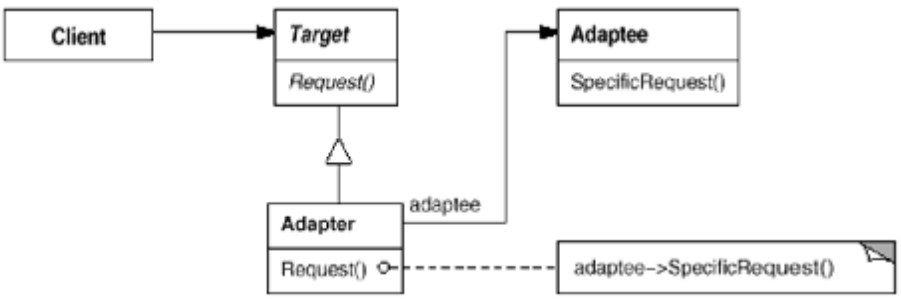
- 类适配器



81

Adapter(适配器)模式

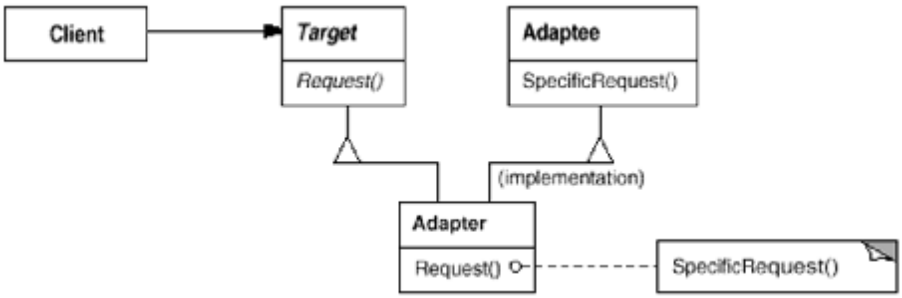
- 结构
  - 对象Adapter



82

Adapter(适配器)模式

- 结构
  - 类Adapter



83

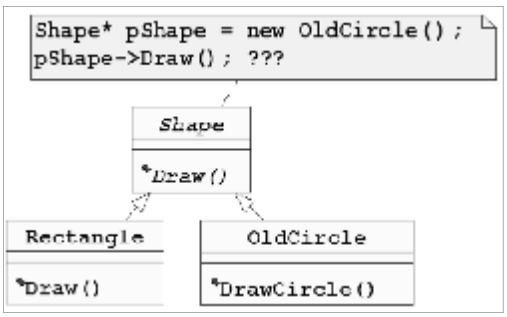
Adapter(适配器)模式

- 意图
  - 将一个类的接口转换成客户希望的另外一个接口
  - Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作

84

Adapter(适配器)模式

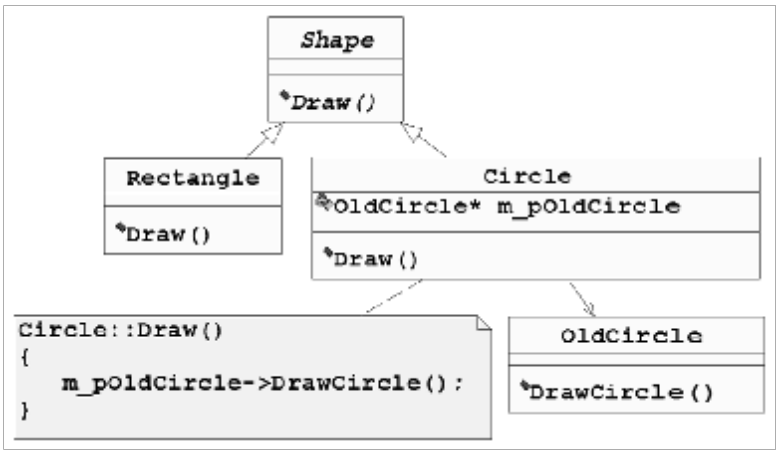
- 应用举例1
  - 我们打算编写一个画图软件
  - 其中画圆形已经有了一个现成的类
  - 但是接口不同，不能直接使用



85

Adapter(适配器)模式

- 使用对象Adapter



86

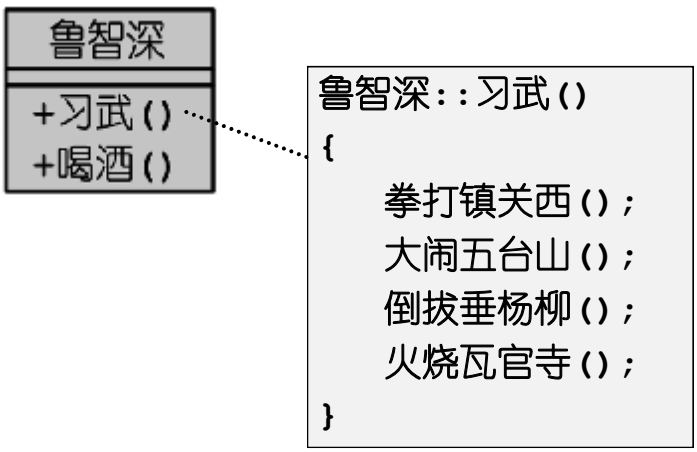
Adapter(适配器)模式

- 应用举例2
  - 缺省适配模式——“鲁达剃度”
  - 凡是和尚都应该如此：

87

Adapter(适配器)模式

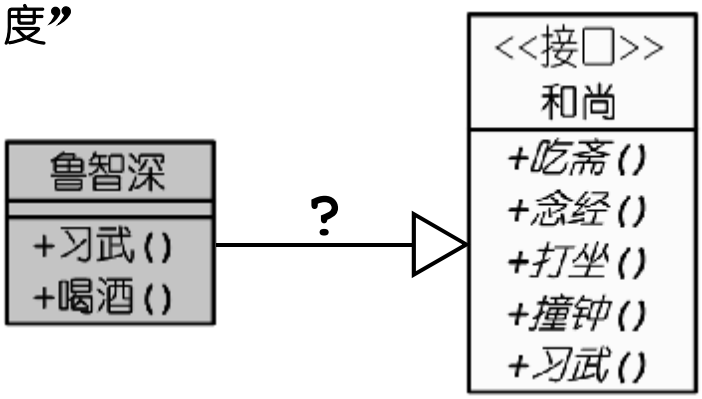
- 但是鲁智深并不是这样



88

Adapter(适配器)模式

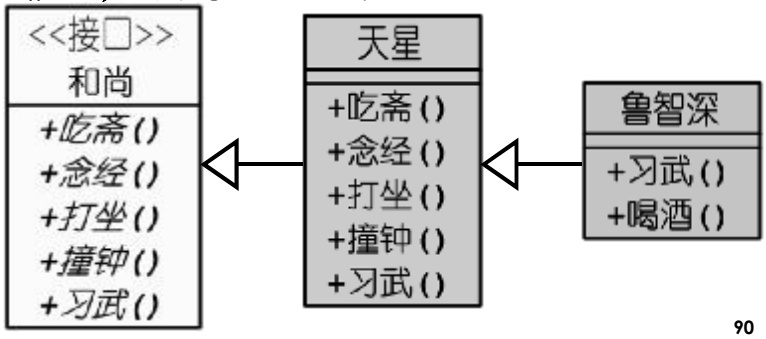
- 所以当初鲁达剃度时，众僧说：
  - “这个人形容丑恶，相貌凶顽，不可剃度”



89

Adapter(适配器)模式

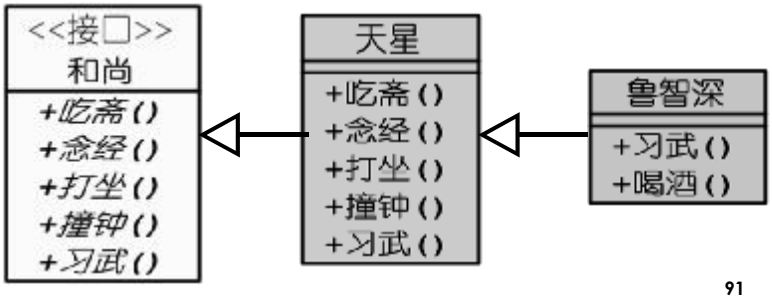
- 但是长老却说：
  - “此人上应天星，心地刚直。虽然时下凶顽，命中驳杂，久后却得清静。证过非凡，汝等皆不及他”



90

Adapter(适配器)模式

- “天星”就是缺省适配器
  - 当你不想/不能实现接口的所有方法时
  - 利用缺省适配器类，提供这些方法的缺省实现
  - 从这个类再派生出的子类就可以不去实现那些不想实现的方法了



91

真实世界中的适配器

- 想一想Java语言中不同版本中有没有需要进行适配的

92

真实世界中的适配器

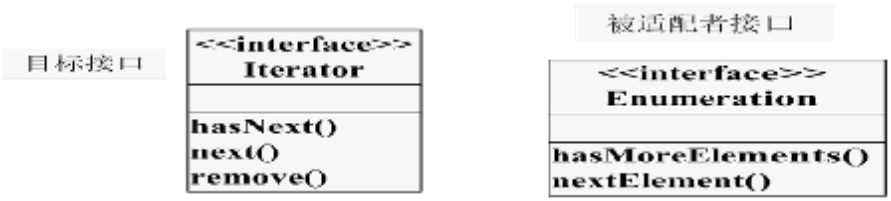
- 早期java版本中集合（Collection）类型（例如：Vector,Stack,Hashtable)都实现了一个elements()方法。该方法返回一个Enumeration（枚举）
- 新版本中开始使用Iterator（迭代器）接口，这个接口和枚举接口很像，但不同的是，迭代器还提供了删除元素的能力。

93

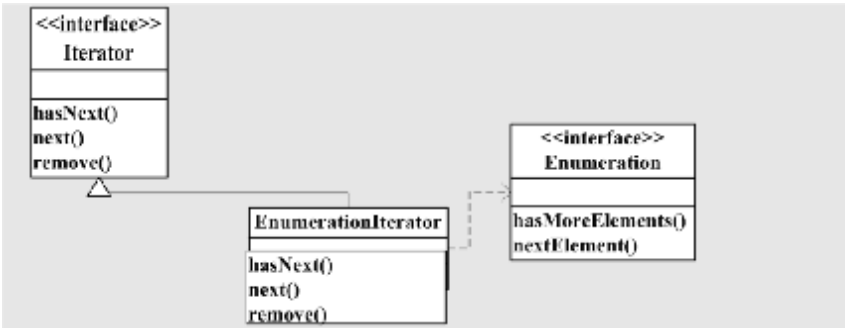
问题

- 面对遗留代码，这些代码会暴露出枚举器接口，但我们又希望在新的代码中只使用迭代器。
- 解决办法
  - 构造一个适配器
  - 将枚举适配到迭代器

94


目标接口
<<interface>>
Iterator
hasNext()
next()
remove()
被适配者接口
<<interface>>
Enumeration
hasMoreElements()
nextElement()

95


<<interface>>
Iterator
hasNext()
next()
remove()
<<interface>>
Enumeration
hasMoreElements()
nextElement()
EnumerationIterator
hasNext()
next()
remove()

96

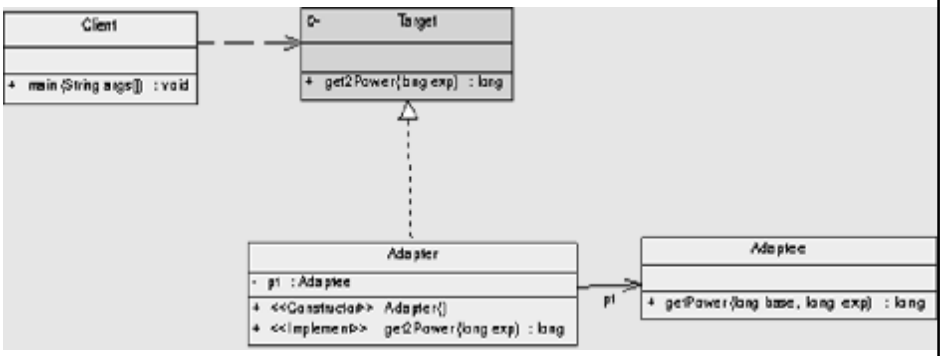
```
public class EnumerationIterator implements Iterator { //适配器看起来就是一个Iterator    //我们使用组合的方式,将枚举结合进适配器中,用实例变量记录
    Enumeration enumeration;
    public EnumerationIterator(Enumeration enumeration) {
        this.enumeration = enumeration;
    }        //迭代器的hasNext其实是委托给enumeration的hasMoreElements方法
    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }        //迭代器的next其实是委托给enumeration的nextElement方法
    public Object next() {
        return enumeration.nextElement();
    }
    //很不幸,我们不能支持迭代器的remove方法,所以必须放弃,这里是抛出一个异常
    public void remove() {
        throw new UnsupportedOperationException();
    }}
```

97

实例

- 有一个类(adaptee)实现了数学中的幂次运算，方法中需要传入两个参数，一个是基数base，另外一个是幂次exp。现在客户端需要一个求得一个数的平方的函数接口(target)，传入一个数，得到它的平方值。为了复用已经存在的类adaptee，使用Adapter来适配adaptee,adapter实现了target接口。

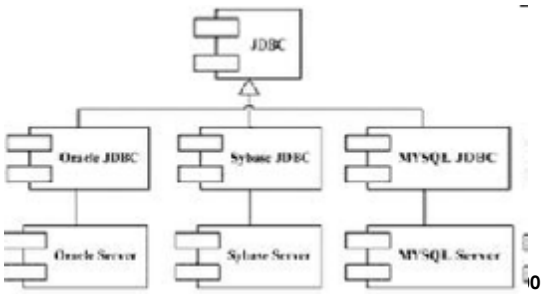
98



99

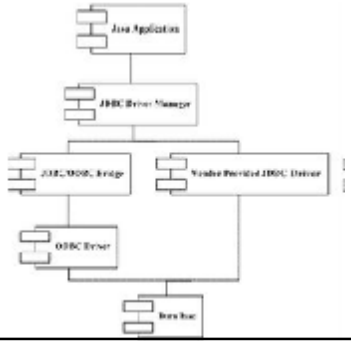
在架构层次上的应用

- JDBC驱动软件与适配器模式
- JDBC给出一个客户端通用的界面。每个数据库引擎的JDBC驱动软件都是一个介于JDBC接口和数据库引擎接口之间的适配器软件
- 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。



40

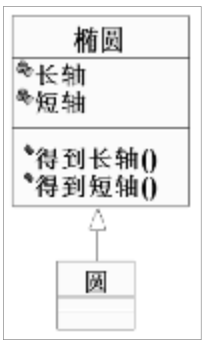
- JDBC/ODBC桥梁
  - 如果没有合适的JDBC驱动软件，用户也可以通过ODBC驱动软件把JDBC通过一个JDBC/ODBC桥梁软件与ODBC驱动软件连接起来，从而达到连接数据库的目的。



101

设计原则：里氏代换原则

- 例子1：“圆是不是椭圆？”
  - 在几何学里，圆是椭圆的一种特殊情况
  - 因此，把椭圆看作父类，把圆作为子类

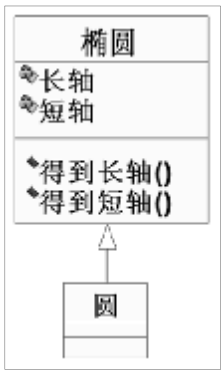


102

设计原则：里氏代换原则

- 问题
  - 椭圆有长轴、短轴
  - 圆会完全继承下来
  - 这些对于圆来说毫无意义
  - 类似的：“正方形不是矩形”

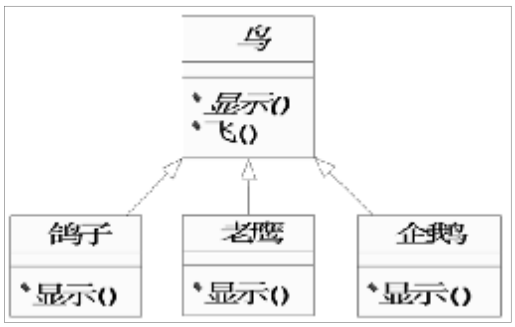
```
Circle circle;
circle.GetMajorAxis();
```



103

设计原则：里氏代换原则

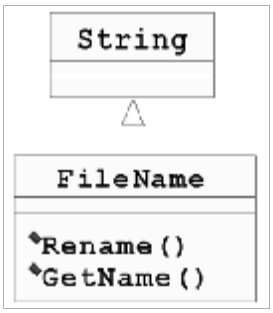
- 例子2：“企鹅不是鸟的子类”
  - 凡是鸟都会飞
  - 但是企鹅不会



104

设计原则：里氏代换原则

- 例子3
  - 我们需要设计一个类FileName来描述文件名，而文件名不就是一个特殊的字符串么？所以我们如此设计：



105

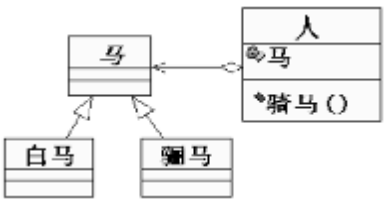
设计原则：里氏代换原则

- 问题
  - 凡是字符串都支持相加操作，也就是说两个字符串相加，结果还是一个字符串
  - 可是两个文件名相加，还是一个合法的文件名么？
    - 比如：**“c:\a.txt” + “d:\b.txt”**
    - 结果是：**“c:\a.txtd:\b.txt”**

106

设计原则：里氏代换原则

- 错在哪里？
- 墨子论“取譬”
  - “白马，马也；乘白马，乘马也。骊马，马也；乘骊马，乘马也。”
  - 解释：白马、骊马(黑马)都是马，既然马可以骑，那么白马和骊马肯定也可以骑



107

设计原则：里氏代换原则

设计原则4：里氏代换原则
凡是父类适用的地方子类应当也适用

- **Liskov Substitution Principle**
- 一个软件如果使用的是一个父类的话，如果把该父类换成子类，它不能察觉出父类对象和子类对象的区别
- 也就是凡是父类适用的地方子类也适用
- 继承只有满足里氏代换原则才是合理的

108

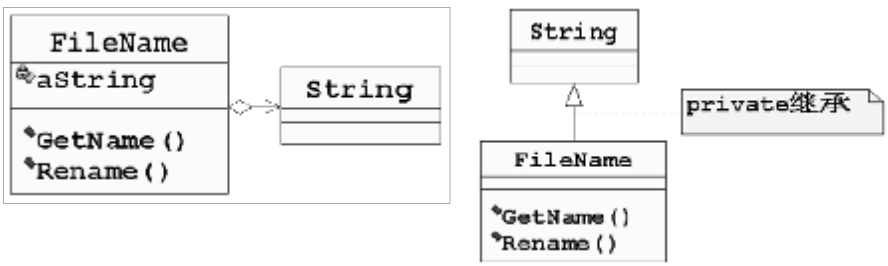
设计原则：里氏代换原则

- 反过来的代换不成立
  - 子类适用的地方不要求父类一定能适用
- 墨子又说
  - “娣，美人也，爱娣，非爱美人也……盗，人也，恶盗，非恶人也”
  - 妹妹是美女，哥哥喜欢妹妹，并不是因为喜欢美女
  - 小偷是人，讨厌小偷，并不讨厌所有人

109

设计原则：里氏代换原则

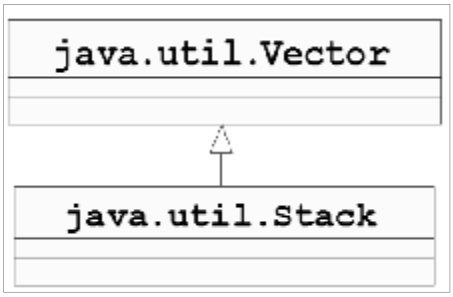
- Java语言对此类问题的防范
  - 它的String类是final的，不能继承
- 正确的方法
  - 使用Adapter模式



110

设计原则：里氏代换原则

- Java中的反例
  - 它的Stack类是从Vector类继承下来的
  - “栈不就是施加了访问限制的数组么？”
  - 所以“Stack Is-A Vector”



111

设计原则：里氏代换原则

- 用里氏代换原则来判断
  - 凡是数组行得通的地方，换成栈也行得通么？
  - Vector可以随机访问，可以任意修改里面的元素...
  - 这些都是Stack所不允许的
  - 因此Stack不能从Vector继承下来，它不能拥有Vector的接口

112

设计原则：里氏代换原则

- 看看C++ STL是怎么办的
  - STL中的stack其实是一个Adapter

```
template <class T, class Cont = deque<T> >
class stack {
public:
    void push(const value_type& x)
    { c.push_back(x); }
    void pop() { c.pop_back(x); }
protected:
    Cont c;
};
```

113

设计模式－工厂模式

- 当看到“new”，就会想到“具体”

115

- 当遇到一群相关的具体类时，通常见到下面的代码

```
Pizza orderPizza(String type){
    Pizza pizza;
    if (type.equals(“cheese”)){
        pizza = new CheeasePizza();
    } else if (type.equals(“greek”)){
        pizza = new GreekPizza();
    } else if (type.equals(“pepperoni”)){
        pizza = new PepperoniPizza();
    }
…
}
```

116

- 如果增加更多的产品类型怎么办

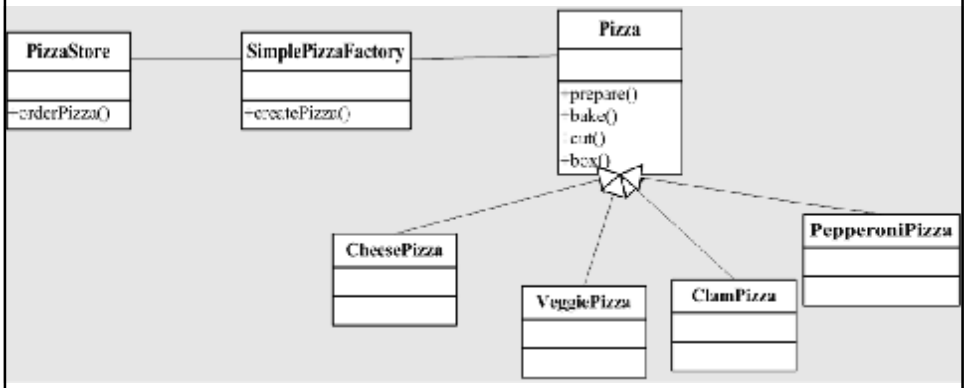
117

- 开始封装创建对象的代码
- 建立一个简单工厂
- 当需要pizza时，就叫工厂做一个

```
Public class SimplePizzaFactory {
    public Pizza createPizza(String type){
        Pizza pizza=null;
        if (type.equals("cheese")){
                pizza = new CheeasePizza();
        } else if (type.equals("greek")){
                pizza = new GreekPizza();
        } else if (type.equals("pepperoni")){
                pizza = new PepperoniPizza();
        }
        return pizza;
    }
}
```

118

定义简单工厂



119

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

120

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}
```

121

- 新的需求
  - 希望每个地方的**pizza**店提供不同风味的**pizza**

122

- 一种做法
  - 利用**SimplePizzaFactory**,写出三种不同的工厂

```
NYPizzaFactory nyFactory =new
   NYPizzaFactory ();
PizzaStore nyStore = new PizzaStore
   (nyFactory);
nyStore.orderPizza("Veggie");
```

123

- 但是制作**pizza**的过程需要规范统一
- 方法
  - 将**createPizza()**方法放回到**PizzaStore**中
  - 将它设置为抽象方法

124

```
Public abstract class PizzaStore {
   public Pizza orderPizza(String type){
          Pizza pizza;
          pizza = createPizaa(type);
          pizza.prepare();
          pizza.bake();
          pizza.cut();
          pizza.box();
          return pizza;
   }
   abstract Pizza createPizza(String type);
}
```

125

```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
          if (item.equals("cheese")) {
                return new NYStyleCheesePizza();
          } else if (item.equals("veggie")) {
                return new NYStyleVeggiePizza();
          } else if (item.equals("clam")) {
                return new NYStyleClamPizza();
          } else if (item.equals("pepperoni")) {
                return new
NYStylePepperoniPizza();
          } else return null;
    }
}
```

126

```
public abstract class Pizza {
        String name;
        String dough;
        String sauce;
        ArrayList toppings = new ArrayList();

        void prepare() {
                System.out.println("Preparing " +
name);
                System.out.println("Tossing
dough...");
                System.out.println("Adding
sauce...");
                System.out.println("Adding toppings:
");
                for (int i = 0; i < toppings.size(); i++) {
                        System.out.println("   " +
toppings.get(i));
                }
        }

        void bake() {
                System.out.println("Bake for 25
minutes at 350");
        }

        void cut() {
                System.out.println("Cutting the pizza
into diagonal slices");
        }

        void box() {
                System.out.println("Place pizza in
official PizzaStore box");
        }

        public String getName() {
                return name;
        }

        public String toString() {
                StringBuffer display =
new StringBuffer();
                display.append("---- "
+ name + " ----\n");

        display.append(dough + "\n");
                display.append(sauce
+ "\n");
                for (int i = 0; i <
toppings.size(); i++) {

        display.append((String )toppings
get(i) + "\n");
                }
                return
display.toString();
        }
}
```

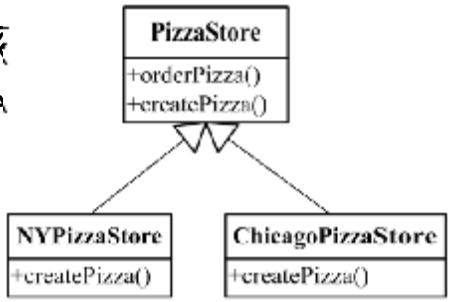
127

```
public class NYStyleCheesePizza extends Pizza {

    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese
Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano
Cheese");
    }
}
```

128

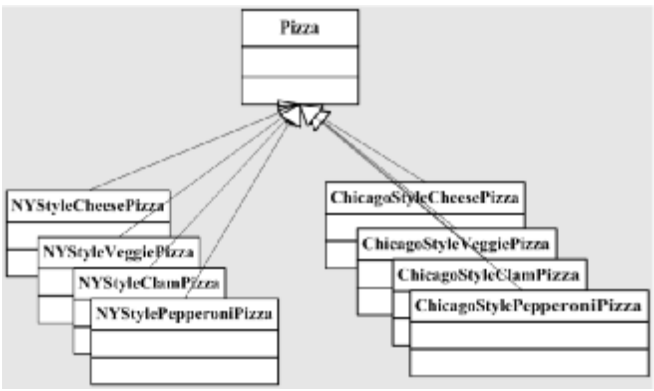
认识工厂模式

- 工厂模式都用来封装对象的创建
- 通过让子类决定该创建的对象是什么
- 涉及到的元素
  - **Creator**类



129

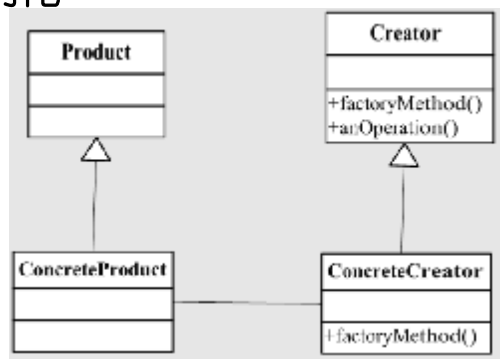
- 产品类



130

定义工厂方法模式

- 工厂方法模式定义了一个创建对象的接口
- 由子类决定实例化的类是哪一个



依赖倒置原则

- 要依赖抽象，不要依赖具体类
- 不使用工厂设计模式

```
Public class PizzaStore {
    public Pizza createPizza(String style, String type) {
        Pizza pizza=null;
        if (style.equals("NY")){
                if (type.equals("cheese")){
                        pizza=new NYStyleCheesePizza();
                }else if
        }else if
        }else {
        }
    }
}
```

- PizzaStore这个高层组件非常依赖具体的低层组件pizza类

132

- 应用依赖倒置原则
- 应用工厂方法
  - 高层组件（PizzaStore）和低层组件都依赖了Pizza抽象

133

遵循倒置依赖原则的指导方针

- 变量不可以持有具体类的引用
  - 如果使用new，就会持有具体类的引用
  - 可以改用工厂来避开这样的做法
- 不要让类派生自具体类
  - 如果派生自具体类，就会依赖具体类
  - 请派生自一个抽象（接口或抽象类）
- 不是随时都要遵循这个原则
  - 如直接实例化字符串对象

134

新需求

- 建造一个原料工厂，为加盟店统一配送原料，保证原料的可靠
- 为了各个加盟店的风味，需要多组不同的原料

135

- 建造原料工厂

```
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}
```

136

实现一个原料工厂

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

public Dough createDough() {
return new ThinCrustDough();
}

public Sauce createSauce() {
return new MarinaraSauce();
}

public Cheese createCheese() {
return new ReggianoCheese();
}

public Veggies[] createVeggies() {
Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };
return veggies;
}

public Pepperoni createPepperoni() {
return new SlicedPepperoni();
}

public Clams createClam() {
return new FreshClams();
}
```

137

```
public abstract class Pizza {
String name;

Dough dough;
Sauce sauce;
Veggies veggies[];
Cheese cheese;
Pepperoni pepperoni;
Clams clam;

abstract void prepare();

void bake() {
System.out.println("Bake for 25 minutes at 350");
}

void cut() {
System.out.println("Cutting the pizza into diagonal
      slices");
}

void box() {
System.out.println("Place pizza in official PizzaStore
      box");
}

void setName(String name) {
this.name = name;
}

String getName() {
return name;
}

public String toString() {
}
}
```

138

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory
    ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

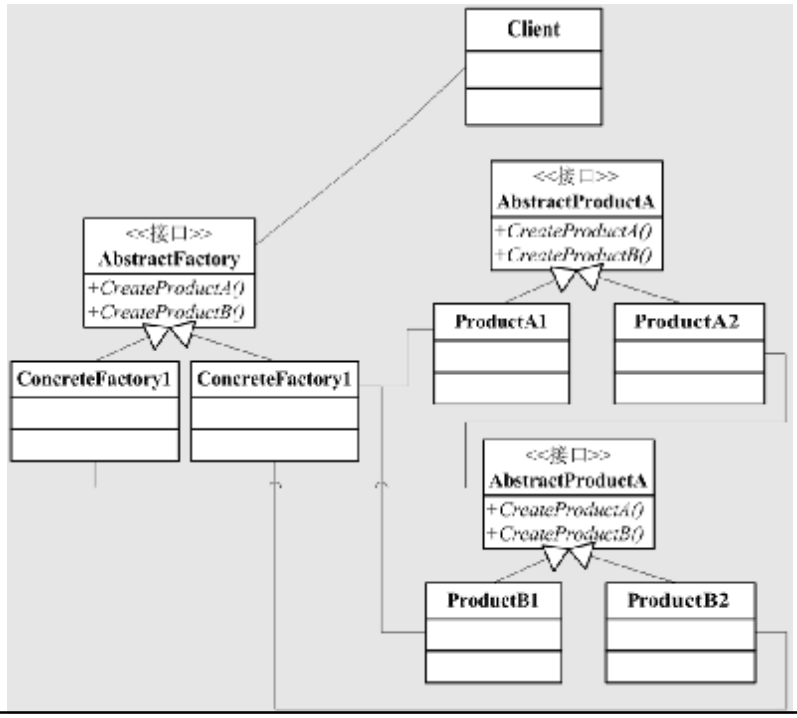
139

```
public abstract class PizzaStore {

protected abstract Pizza createPizza(String item);

public Pizza orderPizza(String type) {
Pizza pizza = createPizza(type);
System.out.println("--- Making a " + pizza.getName()
    + " ---");
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
return pizza;
}
}
```

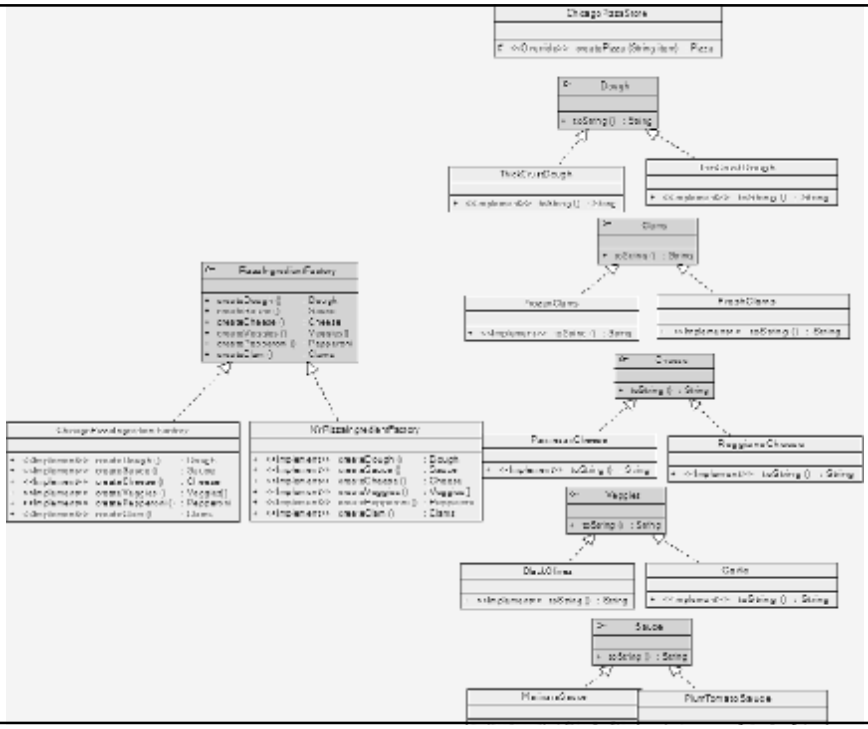
140

```
public class NYPizzaStore extends PizzaStore {

protected Pizza createPizza(String item) {
Pizza pizza = null;
PizzaIngredientFactory ingredientFactory = new NYPizzaIngredientFactory();

if (item.equals("cheese")) {

pizza = new CheesePizza(ingredientFactory);
pizza.setName("New York Style Cheese Pizza");

} else if (item.equals("veggie")) {

pizza = new VeggiePizza(ingredientFactory);
pizza.setName("New York Style Veggie Pizza");

} else if (item.equals("clam")) {

pizza = new ClamPizza(ingredientFactory);
pizza.setName("New York Style Clam Pizza");

} else if (item.equals("pepperoni")) {

pizza = new PepperoniPizza(ingredientFactory);
pizza.setName("New York Style Pepperoni Pizza");

}
return pizza;
}
}
```

141



142



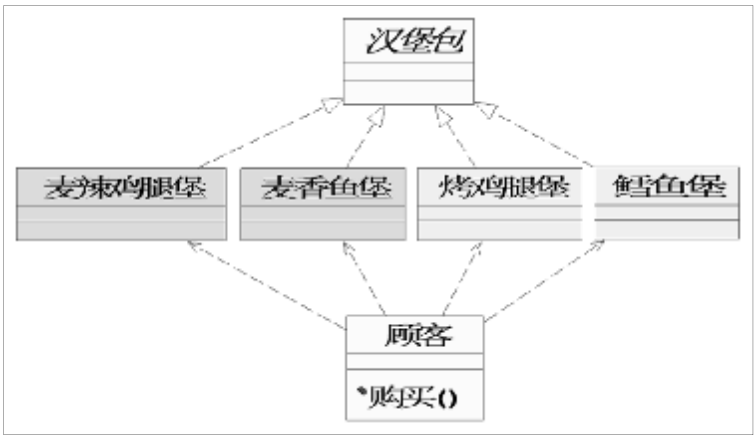
比较工厂方法和抽象方法

- 抽象工厂的需求：创建一个产品家族
- 负责在抽象工厂中创建产品的方法，通常是以“工厂方法”来实现的

144

Factory Method(工厂方法)模式

- 例子：“去快餐厅吃饭”



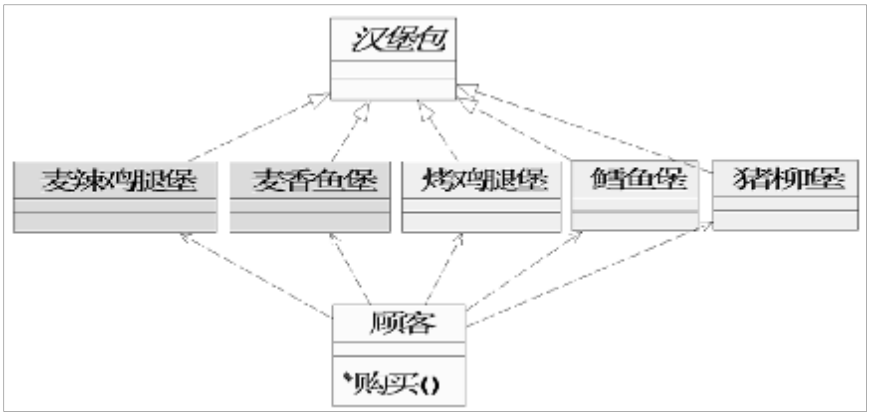
145

```
void BuyFood (string 餐馆, string 食品) {
  if(餐馆 == "KFC") {
     if(食品 == "Chicken")
        hamburger = new KFCChickenHamburger;
     else if(food == "Fish")
        hamburger = new KFCFishHamburger;
     }
  else if(restaurant == "McDonald") {
     if(food == "Chicken")
        hamburger = new McDonaldChickenHamburger;
     else if(food == "Fish")
        hamburger = new McDonaldFishHamburger;
     }
}
```

146

Factory Method(工厂方法)模式

- 增加一种新的食物呢？



147

Factory Method(工厂方法)模式

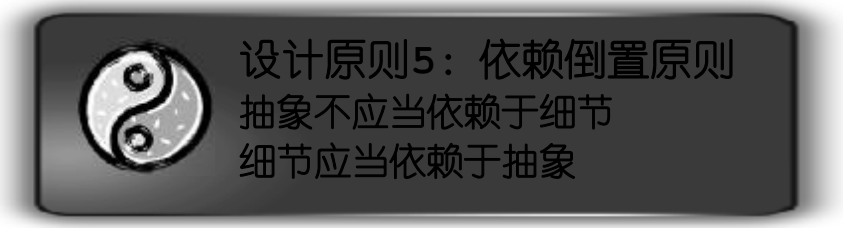
- 传统设计的缺点
  - 依赖具体

```
void BuyFood (string 餐馆, string 食品) {
  if(餐馆 == "KFC") {
     if(食品== "Chicken")
        hamburger = new KFCChickenHamburger;
     else if(food == "Fish")
        hamburger = new KFCFishHamburger;
     }
     ...
```

具体类

148

Factory Method(工厂方法)模式

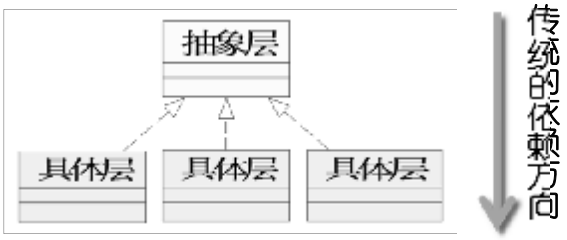


- **“Abstraction should not depend upon details. Details should depend upon abstractions”**

149

设计原则：依赖倒置原则

- 为什么说“倒置”
  - 传统的设计是抽象层依赖具体层
    - 传统的重用，侧重于具体层次的模块，比如算法、数据结构、函数库
    - 因此软件的高层模块依赖低层模块



150

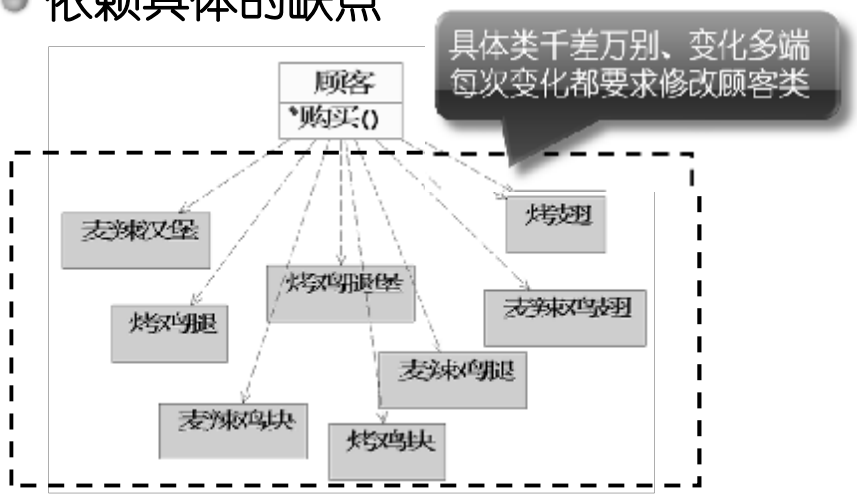
设计原则：依赖倒置原则

- 高层依赖低层的问题
  - 抽象层包含的是系统的业务逻辑和宏观的、战略性的决定，是必然性的体现
  - 具体层则含有与实现相关的算法和逻辑，以及战术性的决定，带有相当大的偶然性选择。具体层经常有变动，难免出现错误
  - 必然依赖偶然，稳定依赖变动？

151

设计原则：依赖倒置原则

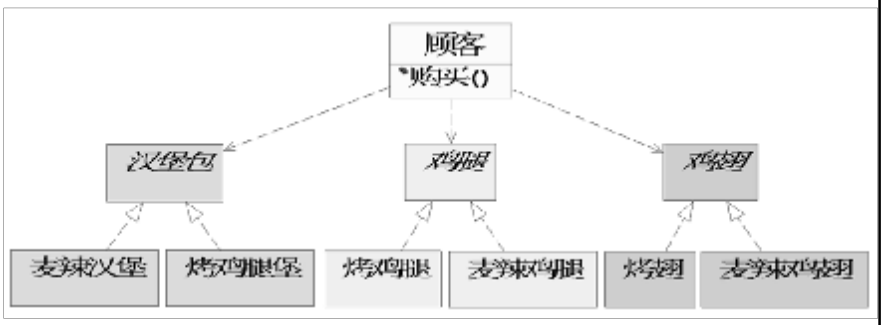
- 依赖具体的缺点



152

设计原则：依赖倒置原则

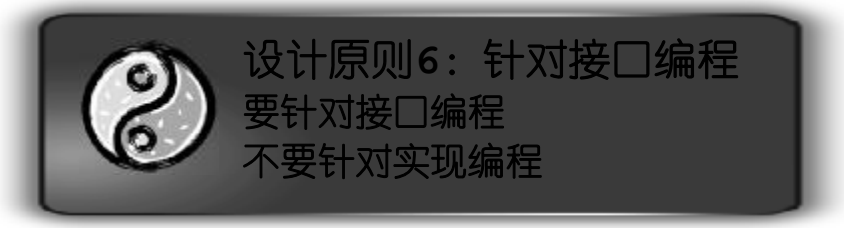
- 依赖抽象
  - 抽象一般不会变动
  - 这样代码不会受易变的具体层影响



153

设计原则：针对接口编程

- 如何做到“依赖倒置”？



  - **“Program to an interface, not an implementation”**

154

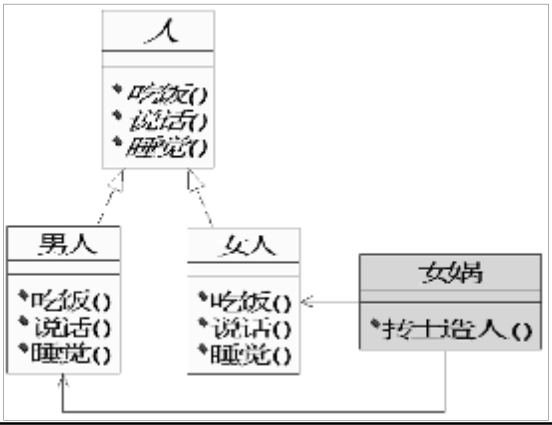
设计原则：针对接口编程

- “针对接口编程”的一些建议
  - 变量、参数、返回值等应声明为抽象类
  - 不要继承非抽象类
  - 不要重载父类的非抽象方法
- 当然这些只是建议
  - 实际情况要权衡利弊

155

Factory Method(工厂方法)模式

- “女娲抟土造人”
  - 《风俗通》：“俗说天开地辟，未有人民。女娲抟黄土为人。”



156

Factory Method(工厂方法)模式

- 简单工厂：根据传入的参数，决定创建哪一个产品类对象

```
Human* NvWa::CreateHuman (string name) {
  if (name == “ZhangSan")
     return new ZhangSan;
  else if (name == “LiSi")
     return new LiSi;
  else if (name == “WangErMaZi")
     return new WangErMaZi;
}
```

157

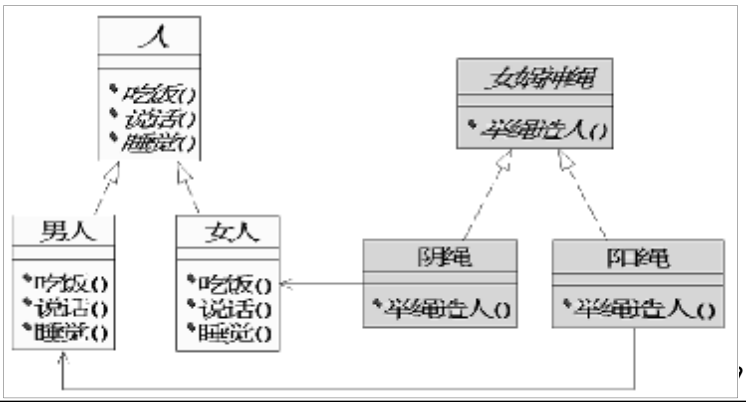
Factory Method(工厂方法)模式

- 简单工厂的优缺点
  - 优点：实现了责任分割
    - 利用判断逻辑，决定实例化哪一个产品类
    - 客户端可以免除直接创建产品类对象的责任，仅仅使用该产品
  - 缺点：没有完全做到“开-闭”
    - 一旦增加新的产品，需要修改工厂的代码
    - 但是客户代码不需要修改
    - “我不入地狱谁入地狱”

158

Factory Method(工厂方法)模式

- “女娲举绳造人”
  - “女娲抟土为人，剧务，力不暇供，乃引绳于桓泥中，举以为人。”



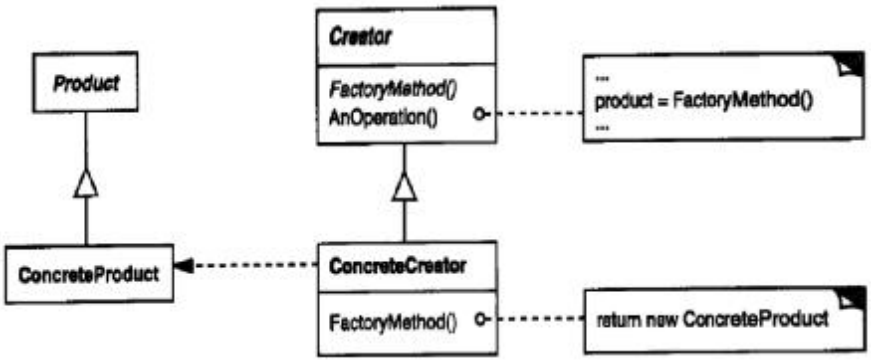
Factory Method(工厂方法)模式

- 简单工厂的问题
  - 所有具体产品对象的创建都放在一个类中，一旦增加新的产品，当然工厂类要被修改
- 工厂方法：使用多态来应对
  - 提供一个抽象工厂的接口
  - 具体工厂分别负责创建具体产品对象
  - 增加新的产品只需要相应增加新的具体工厂类

160

Factory Method(工厂方法)模式

- 结构



161

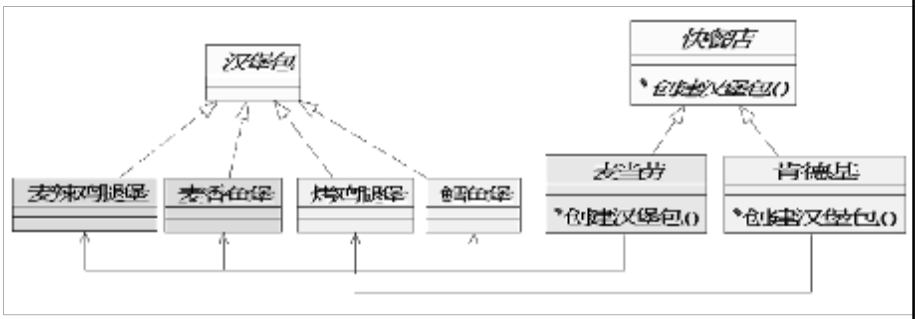
Factory Method(工厂方法)模式

- 意图
  - 定义一个用于创建对象的接口，让子类决定实例化哪一个类
  - 使一个类的实例化延迟到其子类
- 优点
  - 封装了创建具体对象的工作
  - 使得客户代码“针对接口编程”，保持对变化的“关闭”

162

Factory Method(工厂方法)模式

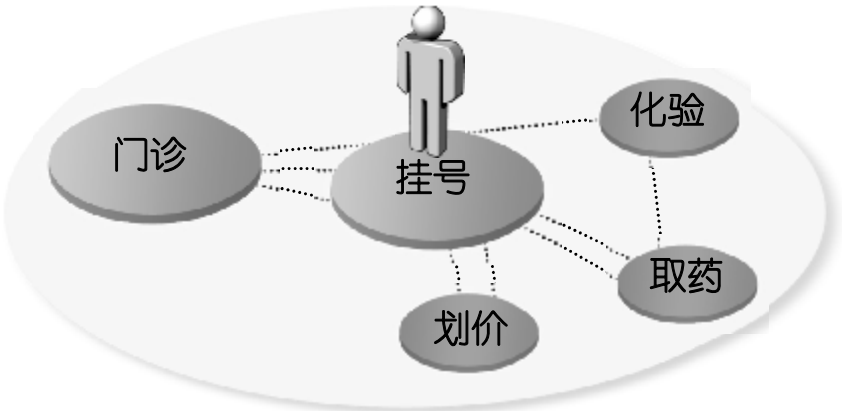
- 工厂方法应用到“快餐店”问题



163

Façade(门面)模式

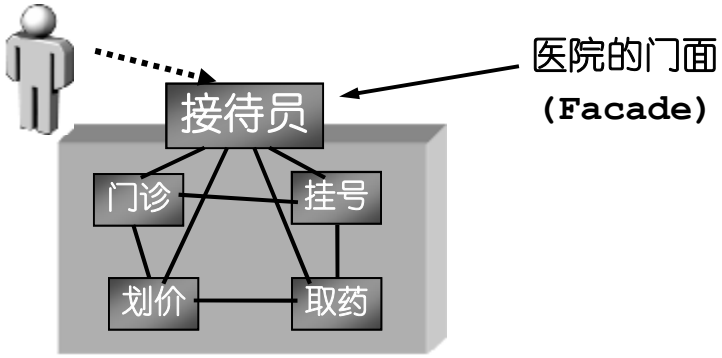
- 例1：传统的医院：
  - 病人需要直接跟各个部门打交道



164

Façade(门面)模式

- 人性化的医院
  - 接待员代替病人进行挂号、划价等
  - 病人只需要和接待员打交道



165

Façade(门面)模式

- 例2：组建家庭影院

166

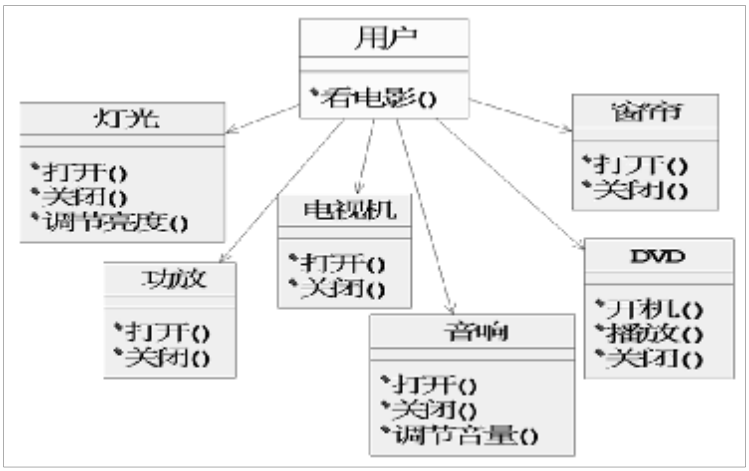
Façade(门面)模式

- 欣赏一部电影的辛苦过程：
  1. 拉上窗帘、调暗灯光
  2. 拿出零食
  3. 打开电视机
  4. 打开功放
  5. 打开音响
  6. 打开DVD机
  7. 找出DVD碟片
  8. 把碟片放入DVD机
  9. 开始播放
  10. ...

167

Façade(门面)模式

- 家庭影院系统：



168

Façade(门面)模式

- 用OO语言描述看电影的辛苦过程

```
curtain.close();
lights.setlight(5);
tv.on();
dvd.on();
dvd.insertdisk(disk);
dvd.play();
amplifier.on();
amplifier.setvolumn(5);
```

169

Façade(门面)模式

- 缺点
  - 客户需要和大量的类打交道
  - 操作复杂
  - 有时候操作顺序不能颠倒
  - 关闭整个系统是不是也是一样的复杂
  - 如果家庭影院升级了呢？

170

设计原则：迪米特法则

- 老子论“圣人之治”
  - 使民无知：“是以圣人之治，虚其心，实其腹，弱其志，常使民无知无欲”
  - 老死不相往来：“小国寡民...邻国相望，鸡犬之声相闻，民至老死，不相往来”
  - 软件设计师就是软件系统的统治者
  - 应当使得软件的不同对象彼此之间尽量“老死不相往来”，降低系统维护成本

171

设计原则：迪米特法则

设计原则7：迪米特法则
每个软件单元对其它单元尽可能少了解
而且仅限于那些与自己密切相关的单元

- **Law of Demeter**
- 又叫做最少知识原则
- “只与你直接的朋友们通信”
- 不要跟“陌生人”说话
- 也就是信息隐藏、封装

172

- 不要让太多的类耦合在一起，免得修改系统中的一部分，会影响到其他部分。

173

- 就任何对象而言，在该对象的方法内，只应该调用属于以下范围的方法
  - 该对象本身
  - 被当作方法的参数而传递进来的对象
  - 此方法所创建或实例化的对象
  - 对象的任何组件

如果某对象是调用其他的方法的返回结果，不要调用该对象的方法

组件指被实例变量所引用的任何对象，“有一个“关系

174

- 那么，如果调用从另一个调用中返回的对象的方法，会有什么伤害
  - 如果这么做，相当于向另一个对象的子部分发请求（增加了我们直接认识的对象数目）。
  - 原则要我们改为要求该对象为我们做出请求，这么一来，我们就不需要认识该对象的组件了。

175

比较

```
Public float getTemp () {
    Thermometer thermometer =
    station.getThermometer ();
    return thermometer.getTemperature();
}//不采用这个原则

Public float getTemp () {
    return station.getTemperature();
}//采用这个原则。我们在气象站中加进一个方法，用来向温度计请求温度。这可以减少我们所依赖的类的数目
```

176

遵守最少知识原则的例子

```
public  class Car {
    Engine engine;//这是类的一个组件，我们能够调用它的方法
    public Car () {
    }
    public void start (Key key) //被当作参数传进来的对象，其方法可以被调用。
        Door doors = new Doors();//在这里创建了一个新的对象，它的方法可以被调用。
        boolean authorized = key.turns();
        if (authorized)  {
                engine.start();//可以调用对象组件的方法
                updateDashboardDisplay();//可以调用local method
                doors.lock();//可以调用创建或实例化的对象的方法
        }
    }
    public void updateDashboardDisplay() {
    }
}
```

177

练习

这个类有没有违反最少知识原则

```
public house {
    weatherStation station;
    public float getTemp(){
        return
    station.getThermometer().getTemperature();
    }
}
```

178

练习

这个类没有违反最少知识原则，但是这样做好么?

```
public House {
    weatherStation station;
    public float getTemp(){
        Thermometer thermometer = station.getThermomer();
        return getThempHelper (thermometer );
    }
    public float getThempHelper (Thermometer
    thermometer ){
        return thermometer .getTemperature();
    }
}
```

179

思考

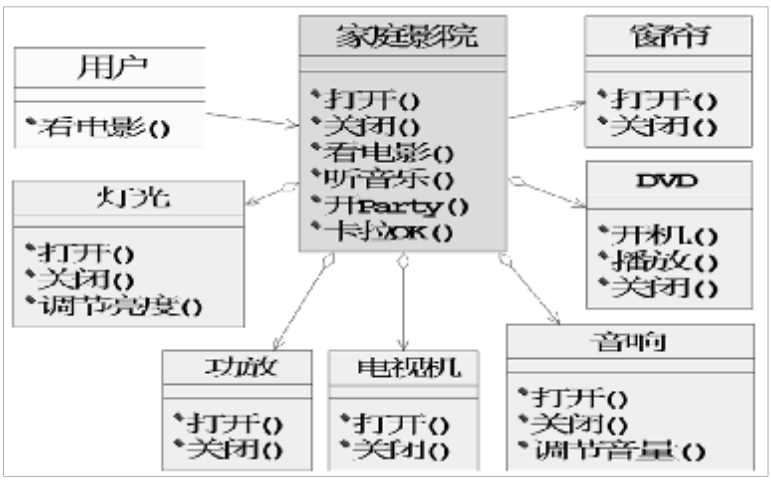
在Java中，有哪些常用的地方违反了最少知识原则？

System.out.println()

180

Façade(门面)模式

- 用一个门面对象简化所有细节



181

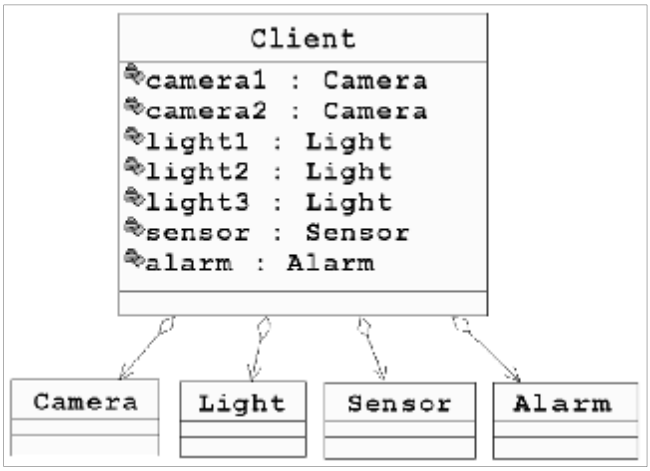
Façade(门面)模式

- 目的
  - 提供一个更简单的系统接口
  - 使得复杂系统和客户代码耦合松散
  - 但是并不屏蔽原有的复杂但是功能强大的接口，用户可以在易用性和通用性之间选择

182

Façade(门面)模式

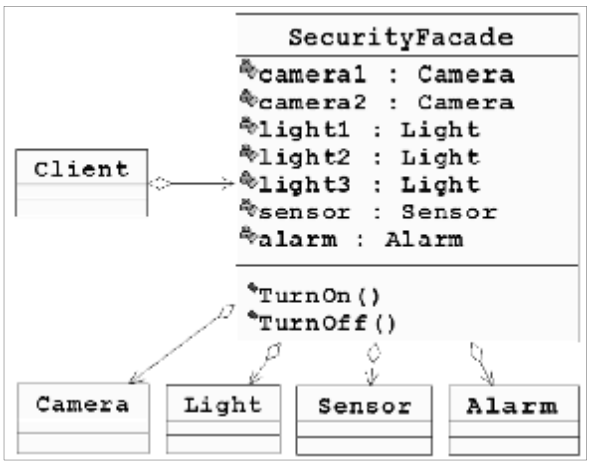
- 举例：保安系统



183

Façade(门面)模式

- 使用门面模式



184

Façade(门面)模式

- **Façade V.S. Adapter**
  - 共同点：改变接口
  - 不同点：目的不同
    - **Façade**目的在于简化接口
    - **Adapter**目的在于适配接口，以便新老系统可以协同工作

185

设计原则：迪米特法则

- 门面模式对迪米特法则的体现
  - 门面模式创造出一个门面对象
  - 将客户端所涉及的属于内部类的数量减到最少
  - 客户只需要和一个门面对象打交道

186

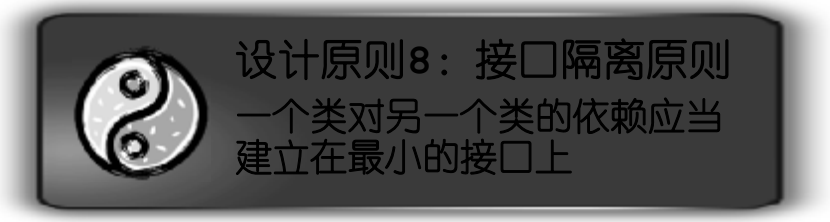
设计原则：迪米特法则

- 迪米特法则在详细设计中的体现
  - 尽量降低成员的访问权限
    - **public -> protected -> private**
  - 限制局部变量的有效范围
    - 变量要用到的时候才去声明它
      - 代码更好懂(一看就知道这个变量是什么类型)
      - 该变量不容易被其它代码误改

```
for (int i=0; i<n; i++)
...
```

187

设计原则：接口隔离原则

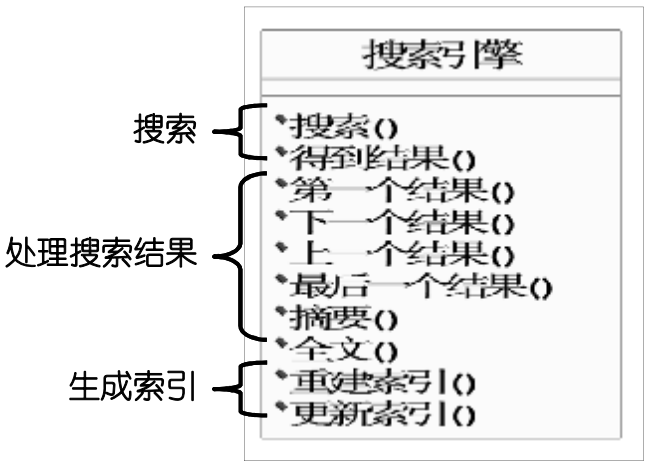


- 角色的合理划分
  - 一个接口应当简单的只代表一个角色
- 接口污染
  - 过于臃肿的接口是对接口的污染
  - 不要把没什么关系的接口合并在一起

188

设计原则：接口隔离原则

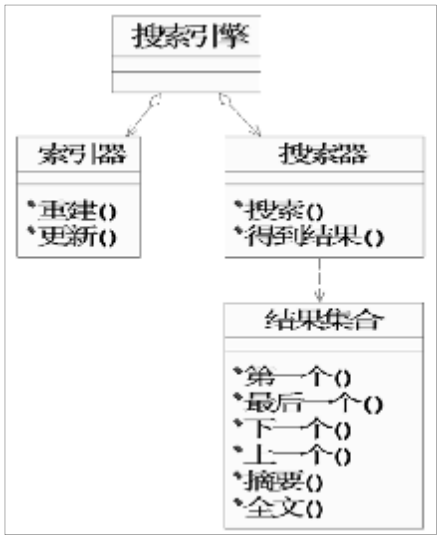
- 例子：全文搜索引擎



189

设计原则：接口隔离原则

- 接口分割：



190

设计原则：接口隔离原则

- 接口隔离原则和迪米特法则的关系
  - 迪米特法则要求尽量限制通信的广度和深度
  - 而对接口进行分割，使其最小化，避免对客户提供不需要的服务，当然是符合迪米特法则的
- 也体现了高内聚、低耦合

接口隔离原则（ISP）

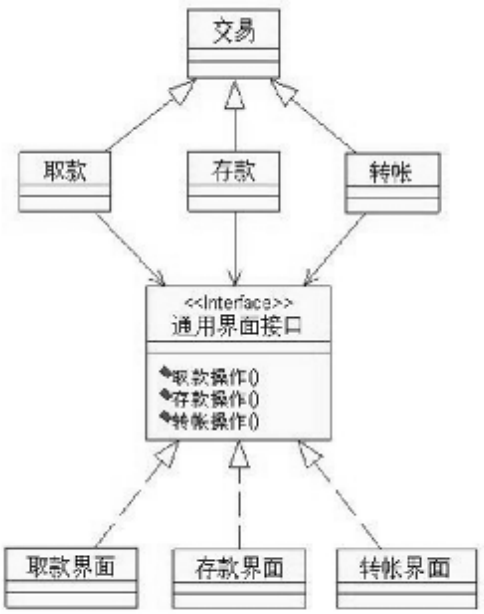
- **使用多个专门的接口比使用单一的总接口好。**
- **一个类对另一个类的依赖性应建立在最小的接口上。**

ISP

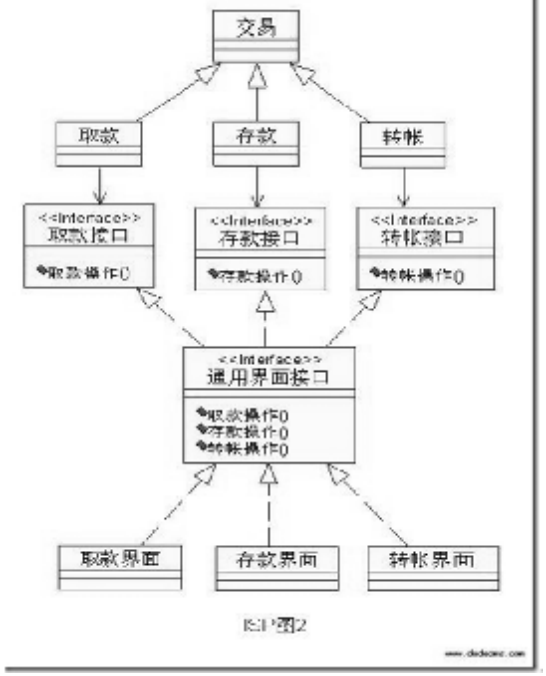
- 定制服务:为同一角色提供宽窄不同的接口。
- （承诺多，维护难）

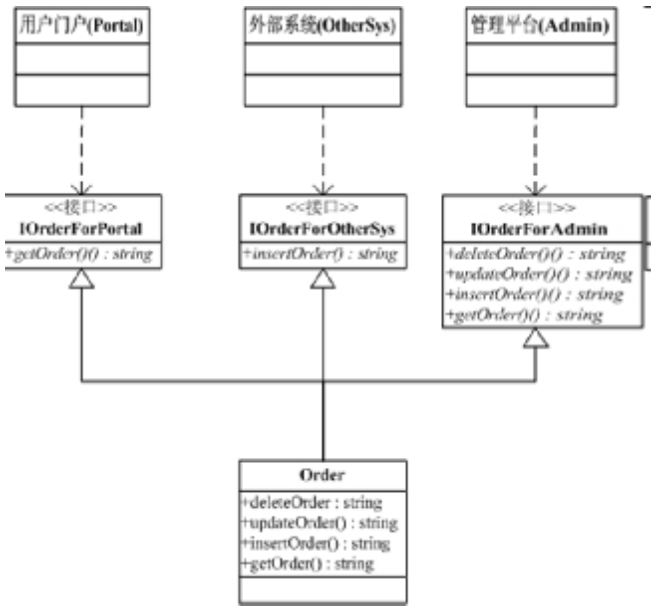
- 如果类的接口不是内聚的，就表示该类具有“胖”的接口。
- ISP建议客户程序不应该看到它们作为单一的类存在。客户程序看到的应该是多个具有内聚接口的抽象基类。

- 现在，考虑自动取款机（**ATM)**
- **ATM**需要一个非常灵活的用户界面。它的输出信息需要被转换成许多不同的语言。输出信息可能被显示在屏幕上，或者通过语音器说出来。显然界面需要创建一个抽象基类
- 同样可以把每个**ATM**可以执行的操作封装为类**Transaction**的派生类。有**DepositTransaction**,**WithdrawTransaction**以及**TransferTransaction**,每个类都调用**UI**的方法。
- 如何设计是违反**LSP**和遵守**LSP** ？



ISP图1


交易
取款
存款
转帐
<<Interface>>
取款接口
<<Interface>>
存款接口
<<Interface>>
转帐接口
<<Interface>>
通用界面接口
取款界面
存款界面
转帐界面


用户门户(Portal)
外部系统(OtherSys)
管理平台(Admin)
<<接口>>
IOrderForPortal
+getOrder()() : string
<<接口>>
IOrderForOtherSys
+insertOrder() : string
<<接口>>
IOrderForAdmin
+deleteOrder()() : string
+updateOrder()() : string
+insertOrder()() : string
+getOrder()() : string
Order
+deleteOrder : string
+updateOrder() : string
+insertOrder() : string
+getOrder() : string

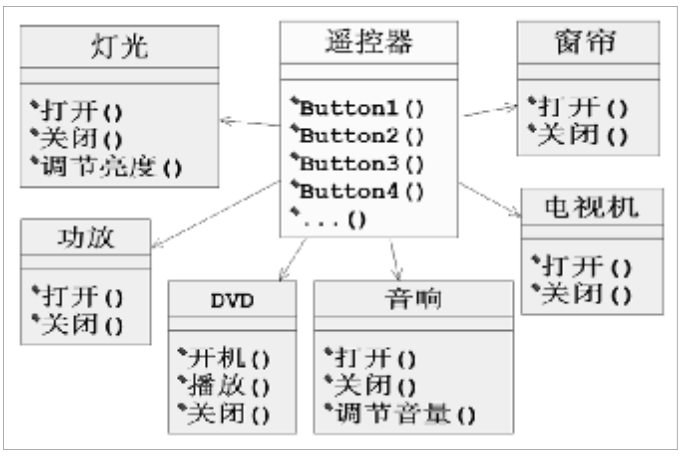
Command(*命令*)*模式*

- 给家庭影院配一个遥控器

199

Command(*命令*)*模式*

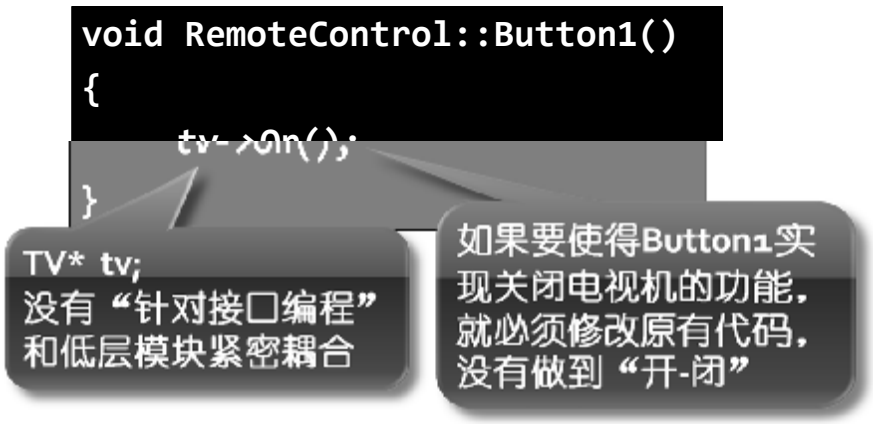
- 传统的想法：



200

Command(命令)模式

- 用户要求：按钮可以自定义



201

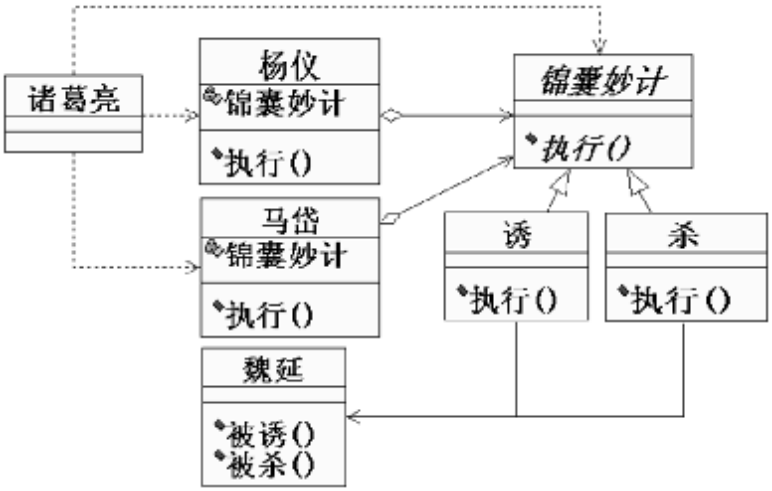
Command(命令)模式

- 例1：“魏延谋反”
  - 诸葛亮在临死前给杨仪一个锦囊，密嘱：“我死，魏延必反，...那时自有斩魏延之人也”。同时授马岱以密计，只待魏延喊叫时，便出其不意斩之
  - 后人有诗曰：“诸葛先机识魏延，已知日后反西川。锦囊遗计人难料，却见成功在马前”
  - 视频：“魏延谋反.rm”

202

Command(命令)模式

- 结构



Command(命令)模式

- 分析
  - 锦囊是对命令的封装
  - 诸葛亮是命令的设计者，他选择魏延作为命令的接收者
  - 杨仪和马岱是命令的调用者，他们依据实际情况决定何时调用、是否调用命令
  - 命令运行的结果就是使得命令的接收者魏延中计，并按照预谋落入陷阱

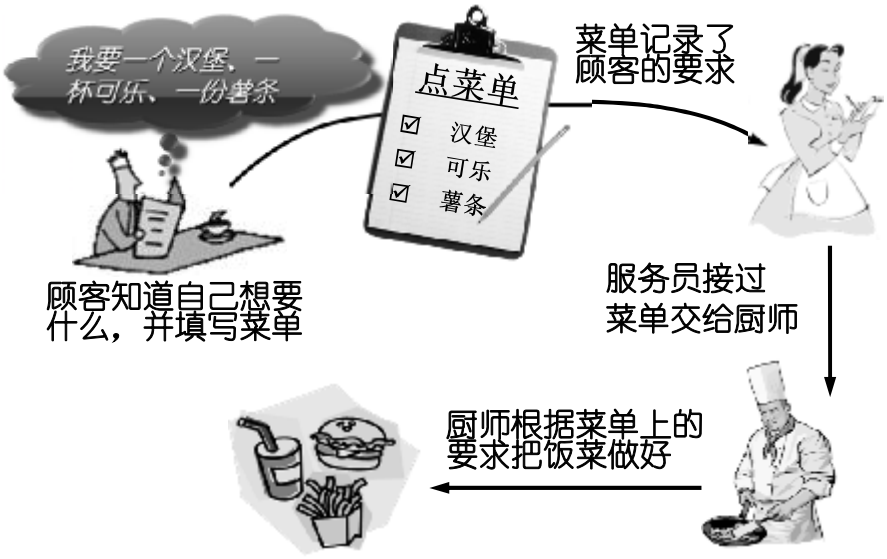
204

Command(命令)模式

- 分析
  - 战场瞬息万变，孔明不能处处临场指挥
  - 在这个故事里，命令模式把命令的设计者诸葛亮、调用者杨仪、马岱和执行者魏延完全分割开，互相可以灵活变化
  - 命令由诸葛亮设计并封装，交由杨仪、马岱等待时机成熟时调用，调用结果就是命令的接收者魏延中计

205

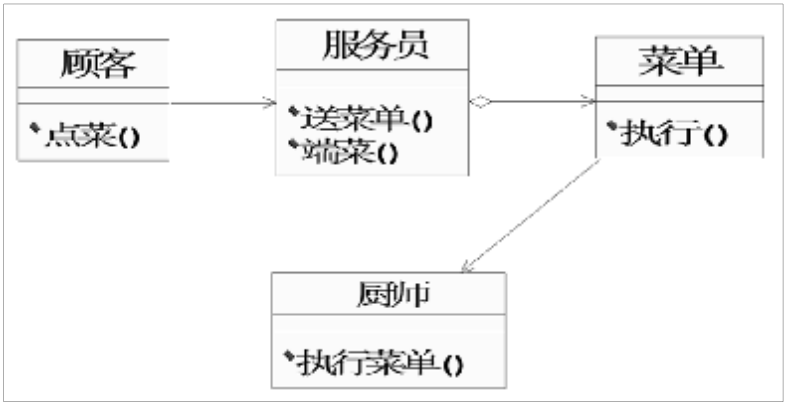
- 例2：餐厅的工作流程



206

Command(命令)模式

- 结构



207

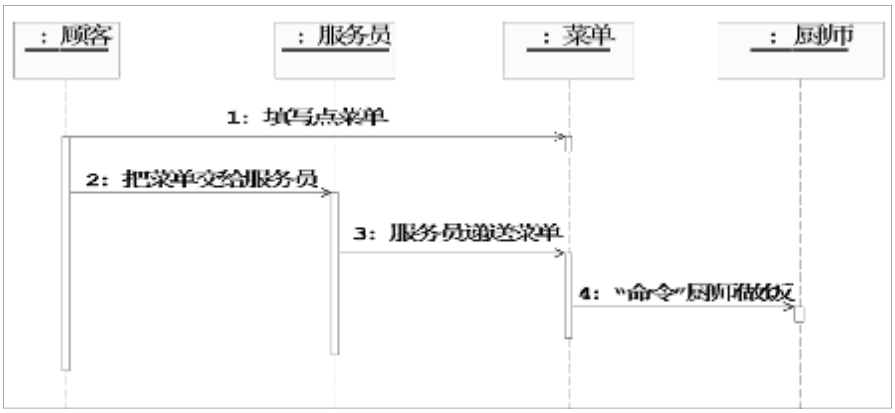
Command(命令)模式

- 人物关系
  - 顾客(Client)：决定要点什么菜
  - 点菜单(Command)：封装了顾客的要求
  - 服务员(Invoker)：决定如何、何时将菜单送达
  - 厨师(Receiver)：顾客要求的接收者，是唯一知道如何做菜的

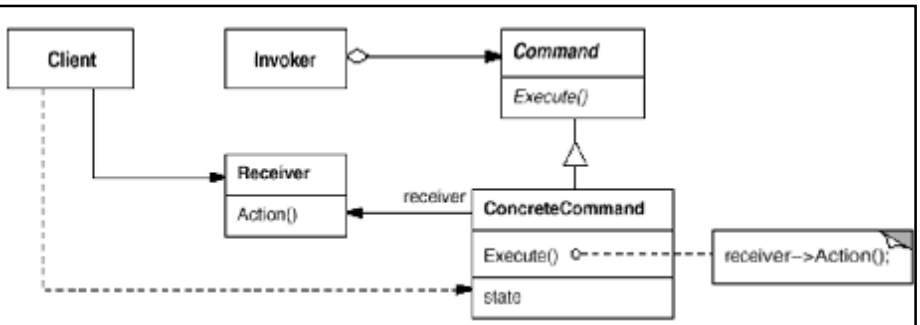
208

Command(命令)模式

- 运作过程：



209



- 命令(**Command**)：声明执行操作的接口
- 具体命令(**Concrete Command**)：将一个接收者对象绑定于一个动作；调用接收者相应的操作，以实现**Execute()**
- 客户(**Client**)：创建具体命令对象并设定它的接收者
- 调用者(**Invoker**)：要求该命令执行这个请求
- 接收者(**Receiver**)：知道如何实施与执行一个请求相关的操作，任何类都可以作为一个接收者

210

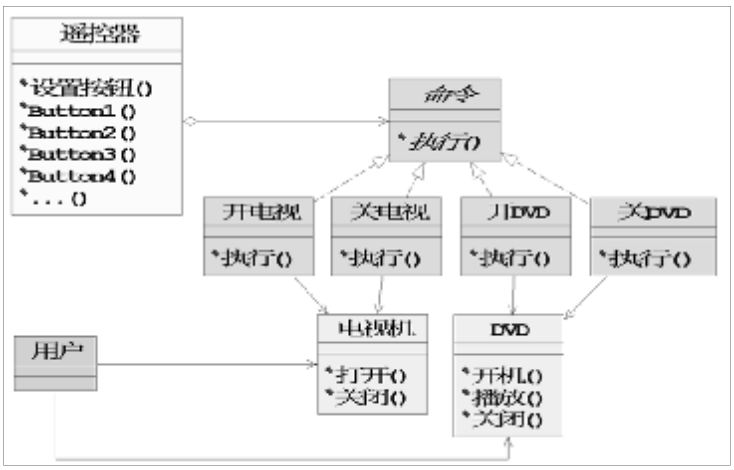
Command(命令)模式

- 效果
  - 可以很方便的增加新的命令
  - 将调用操作的对象(Invoker)与知道如何实现该操作的对象(Receiver)解耦
  - 将命令装配成对象，容易实现Undo/Redo和日志功能
  - 可以实现事务管理：把多个命令合成为一个，相当于宏命令

211

Command(命令)模式

- 命令模式应用在遥控器上：



212

Command(命令)模式

- 效果
  - 命令的调用者遥控器和执行者电视机等设备完全解耦
  - 用户可以动态设置遥控器按钮的功能
  - 这一切都不需要修改遥控器代码
  - 做到了松散耦合，遵循了“开-闭”原则

213

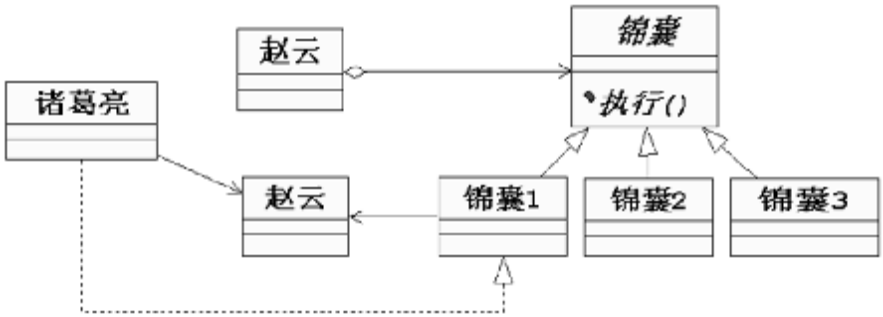
Command(命令)模式

- 应用举例1：“刘备东吴招亲”
  - 周瑜与孙权定下计谋，以孙权之妹为诱饵，骗刘备到东吴招亲，想趁机杀害刘备，索回荆州
  - 诸葛亮早识破了诡计，令赵云护卫刘备前往，并给他三个锦囊，嘱咐他依次执行。结果，赵云按照诸葛亮的锦囊妙计行事，不仅帮助刘备将孙权之妹孙尚香夫人迎娶回来，还得到孙权之母吴国太的欢心，陪同刘备夫妇回了荆州
  - 《三国演义》这段记叙，为后世创造了“锦囊妙计”和“赔了夫人又折兵”两个成语
  - 视频：锦囊妙计.rm

214

Command(命令)模式

- 结构：



215

Command(命令)模式

- 分析
  - 锦囊是对命令的封装
  - 诸葛亮是命令的设计者，他选择赵云为接收者
  - 赵云同时又是命令的调用者，他依据实际情况决定何时调用、是否调用命令
  - 赵云作为命令的接收者，具体执行这个命令
  - 在这个故事里，命令模式把命令的设计者诸葛亮、调用者赵云和执行者赵云完全分割开。命令由诸葛亮设计并封装，交由赵云调用，时机成熟时，赵云调用命令，再由自己去执行之

216

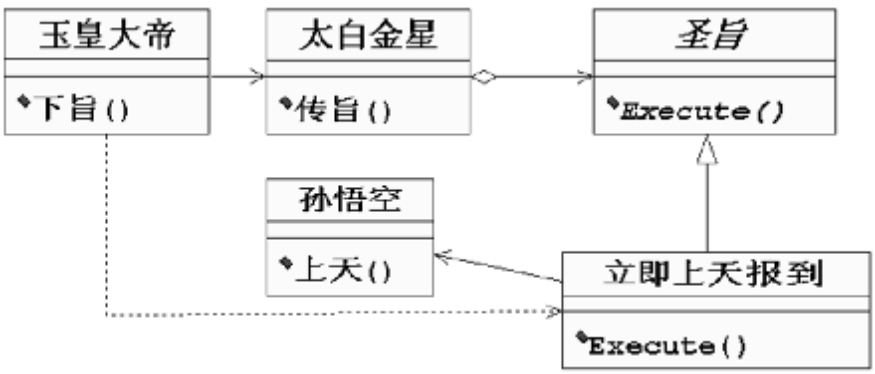
Command(命令)模式

- 应用举例2：“玉帝传美猴王上天”
  - 玉帝命令太白金星召美猴王上天：“金星径入当中，面南立定道：‘我是西方太白金星，奉玉帝招安圣旨，下界请你上天，拜受仙录。’”
  - 玉帝的这一道命令就是要求猴王到上界报到。玉帝只管发出命令，而不管命令是怎样传达到美猴王的。太白金星负责将圣旨传到，可是美猴王怎么执行圣旨、何时执行圣旨是美猴王自己的事。果不然，不久美猴王就大闹了天宫

217

Command(命令)模式

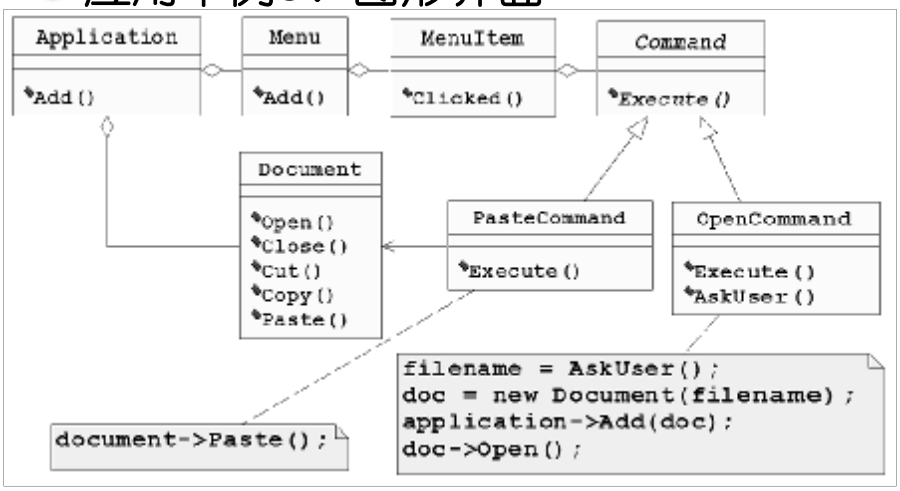
- 结构



218

Command(命令)模式

- 应用举例3：图形界面



219

好莱坞法则

- **“Don’t call us, we’ll call you”**
  - 本来是好莱坞的制片人拒绝演员时说的，意思是“别烦我，我想要你的时候会去找你的”（其实永远不会）
  - 这里是双关，指的是“调用”

220

好莱坞法则

- 回调（Call back）
  - 在框架中，框架知道何时去干一些事情，但不知道具体干什么
  - 客户代码知道干什么，但不知道何时去干
  - 这时候，客户代码传给一个函数指针或者对象，由框架来决定何时调用

221

好莱坞法则

```
class Button
{
public:
     Button() : action(0) {}
     void setA
          acti
     void onClick(     (*action)(); }
private:
     void (*action) ();
};
```

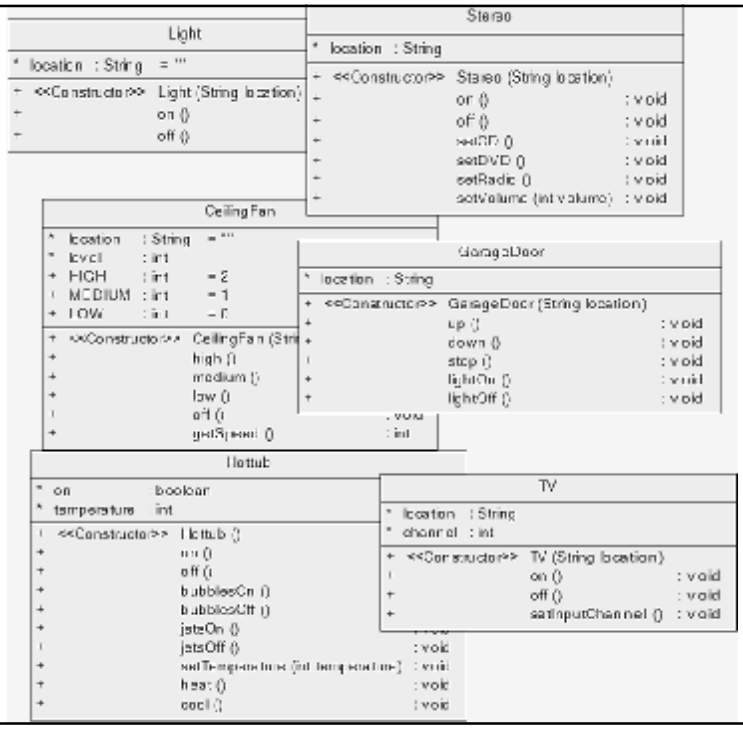
指向回调函数的函数指针

222

遥控器实例

- 一个遥控器有七对开关按钮
- 每对按钮可以控制不同的装置，如灯，电视，影碟机等
- 这些装置来自不同的厂家，接口不同
  - On(),off()
  - Up(),down()
  - …
- 还有一个整体的撤销按钮，会撤销最后一个按钮动作
- 我们不想让遥控器包含一大堆if语句

223



224

命令模式

- 命令模式可将“动作的请求者”从“动作的执行者”对象中解耦合
- 请求者可以是遥控器
- 执行者对象就是厂商类其中之一的实例

225

实现命令接口

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

226

命令对象

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

227

使用命令对象

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

    public void setCommand(Command command)
{
        slot = command;
    }

    public void buttonWasPressed() {
        slot.execute();
    }
}
```

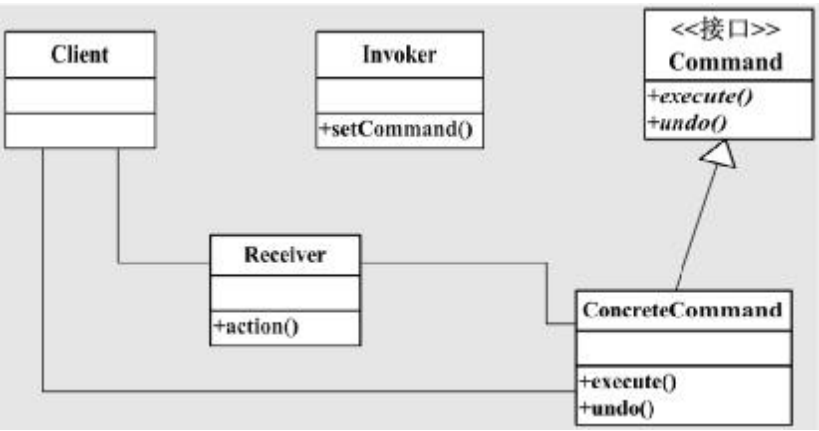
228

简单测试

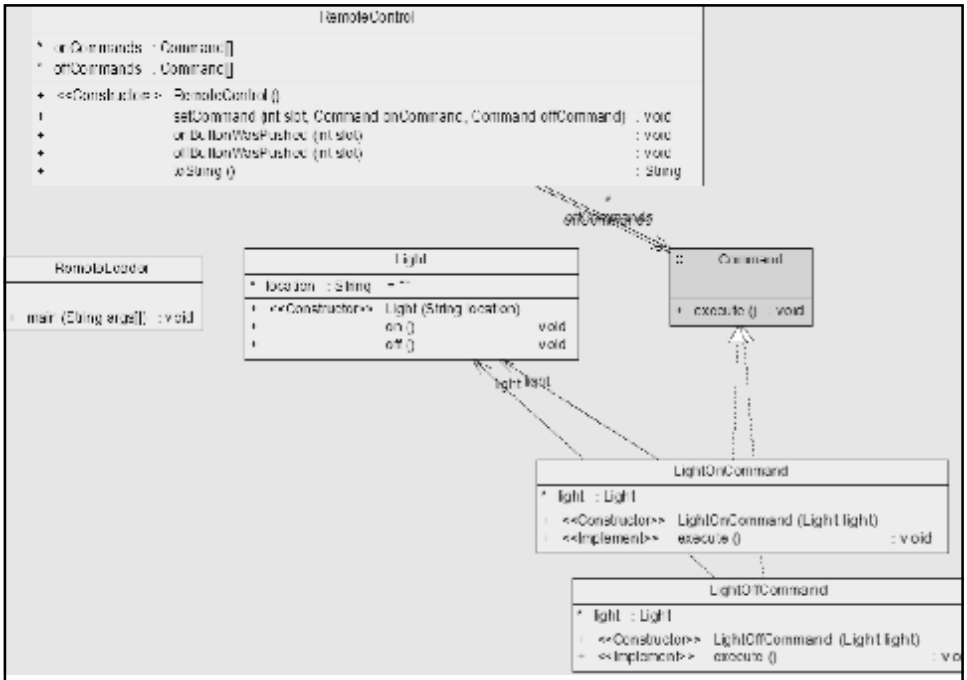
```
public class RemoteControlTest {
    public static void main(String[] args) {
        SimpleRemoteControl remote = new
SimpleRemoteControl();
        Light light = new Light();
        GarageDoor garageDoor = new GarageDoor();
        LightOnCommand lightOn = new
LightOnCommand(light);
        GarageDoorOpenCommand garageOpen =
          new GarageDoorOpenCommand(garageDoor);

        remote.setCommand(lightOn);
        remote.buttonWasPressed();
        remote.setCommand(garageOpen);
        remote.buttonWasPressed();
    }
}
```

229

定义命令模式



230



231

实现遥控器

```
public class RemoteControl {
        Command[] onCommands;
        Command[] offCommands;

        public RemoteControl() {
                onCommands = new Command[7];
                offCommands = new Command[7];

                Command noCommand = new NoCommand();
                for (int i = 0; i < 7; i++) {
                        onCommands[i] = noCommand;
                        offCommands[i] = noCommand;
                }
        }

        public void setCommand(int slot, Command onCommand, Command
    offCommand) {
                onCommands[slot] = onCommand;
                offCommands[slot] = offCommand;
        }

        public void onButtonWasPushed(int slot) {
                onCommands[slot].execute();
        }

        public void offButtonWasPushed(int slot) {
                offCommands[slot].execute();
        }
```

232

实现遥控器

```
public class RemoteLoader {

        public static void main(String[] args) {
                RemoteControl remoteControl = new RemoteControl();

                Light livingRoomLight = new Light("Living Room");
                Light kitchenLight = new Light("Kitchen");
                CeilingFan ceilingFan= new CeilingFan("Living Room");
                GarageDoor garageDoor = new GarageDoor("");
                Stereo stereo = new Stereo("Living Room");

                LightOnCommand livingRoomLightOn =
                                        new LightOnCommand(livingRoomLight);
                LightOffCommand livingRoomLightOff =
                                        new LightOffCommand(livingRoomLight);
                remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
                remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
                remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
                remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

                System.out.println(remoteControl);

                remoteControl.onButtonWasPushed(0);
                remoteControl.offButtonWasPushed(0);
                remoteControl.onButtonWasPushed(1);
                remoteControl.offButtonWasPushed(1);
                remoteControl.onButtonWasPushed(2);
                remoteControl.offButtonWasPushed(2);
                remoteControl.onButtonWasPushed(3);
                remoteControl.offButtonWasPushed(3);
```


支持撤销

```
public interface Command {
    public void execute();
    public void undo();
}
```

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

235

```
public class LightOffCommand implements Command {
    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.off();
    }

    public void undo() {
        light.on();
    }
}
```

236

```
public class RemoteControlWithUndo {
        Command[] onCommands;
        Command[] offCommands;
        Command undoCommand;
        public RemoteControlWithUndo() {
                onCommands = new Command[7];
                offCommands = new Command[7];

                Command noCommand = new NoCommand();
                for(int i=0;i<7;i++) {
                        onCommands[i] = noCommand;
                        offCommands[i] = noCommand;
                }
                undoCommand = noCommand;
        }
        public void onButtonWasPushed(int slot) {
                onCommands[slot].execute();
                undoCommand = onCommands[slot];
        }
        public void offButtonWasPushed(int slot) {
                offCommands[slot].execute();
                undoCommand = offCommands[slot];
        }
        public void undoButtonWasPushed() {
                undoCommand.undo();
        }
```

237

```
public class RemoteLoader {

        public static void main(String[] args) {
                RemoteControlWithUndo remoteControl = new
RemoteControlWithUndo();

                Light livingRoomLight = new Light("Living Room");

                LightOnCommand livingRoomLightOn =
                                new LightOnCommand(livingRoomLight);
                LightOffCommand livingRoomLightOff =
                                new LightOffCommand(livingRoomLight);

                remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

                remoteControl.onButtonWasPushed(0);
                remoteControl.offButtonWasPushed(0);
                System.out.println(remoteControl);
                remoteControl.undoButtonWasPushed();
                remoteControl.offButtonWasPushed(0);
                remoteControl.onButtonWasPushed(0);
                System.out.println(remoteControl);
                remoteControl.undoButtonWasPushed();
```

238

- 新要求
  - 按下一个按钮，就同时能弄暗灯光、打开音响和电视，设置好dvd

239

使用宏命令

```
public class MacroCommand implements Command {
    Command[] commands;

    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }

    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }

    public void undo() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].undo();
        }
    }
}
```

240

- 首先，创建想要进入宏的命令集合
  - Light light = new Light("Living Room");
  - TV tv = new TV("Living Room");
  - Stereo stereo = new Stereo("Living Room");
  - Hottub hottub = new Hottub();
  -
  - LightOnCommand lightOn = new LightOnCommand(light);
  - StereoOnCommand stereoOn = new StereoOnCommand(stereo);
  - TVOnCommand tvOn = new TVOnCommand(tv);
  - HottubOnCommand hottubOn = new HottubOnCommand(hottub);
  - LightOffCommand lightOff = new LightOffCommand(light);
  - StereoOffCommand stereoOff = new StereoOffCommand(stereo);
  - TVOffCommand tvOff = new TVOffCommand(tv);
  - HottubOffCommand hottubOff = new HottubOffCommand(hottub);

241

- 创建两个命令数组
  - Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};
  - Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};
  -
  - MacroCommand partyOnMacro = new MacroCommand(partyOn);
  - MacroCommand partyOffMacro = new MacroCommand(partyOff);

242

- 将宏命令指定给我们所希望的按钮

```
remoteControl.setCommand(0,
partyOnMacro, partyOffMacro);
```

243

命令模式的其他用途

- 工作队列
- 日志请求

244

设计模式 – 模版方法模式

封装

- 目前，已经封装了
  - 对象的创建
  - 方法的调用
  - 复杂接口
- 现在
  - 要封装算法模块

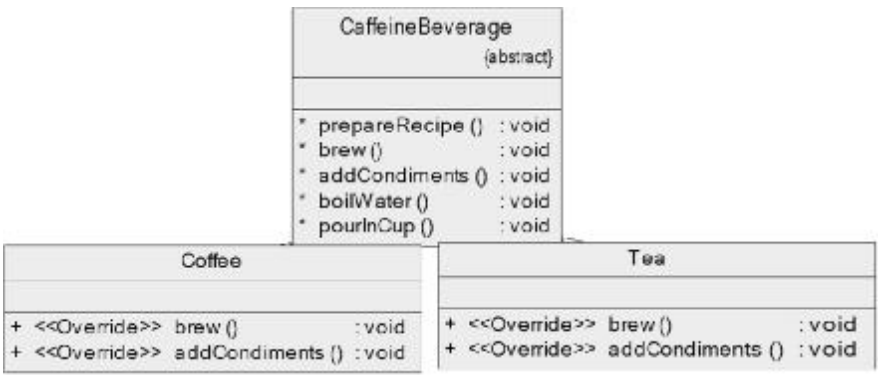
246

背景

- 茶和咖啡的冲泡方法
- 步骤有些是重复的
  - 烧水
  - 冲咖啡（茶）
  - 将茶（咖啡）倒入杯中
  - 加糖、奶（柠檬）

247

- 从以上两个类中抽象出相同算法



248

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

249

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

250

```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

251

认识模版方法

- perpareRecipe()是模版方法
  - 它是一个方法
  - 用作一个算法的模版
  - 算法的每一个步骤都被一个方法代表了
  - 某些方法是由超类处理的
  - 某些方法是由子类处理的
  - 需要由子类提供的方法，必须在超类中声明为抽象

252

对模版方法进行挂钩

- 钩子是一种被声明在抽象类中的方法，只有空的或者默认的实现
- 钩子可以让子类有能力对算法的不同点进行挂钩
- 要不要挂钩，由子类决定

253

- **if (customerWantsCondiments()) {**
- ** addCondiments();**
- **}**
- **boolean customerWantsCondiments() {**
- ** return true;**
- **}**

254

使用钩子

```
public class CoffeeWithHook extends CaffeineBeverageWithHook {
        public boolean customerWantsCondiments() {

                String answer = getUserInput();

                if (answer.toLowerCase().startsWith("y")) {
                        return true;
                } else {
                        return false;
                }
        }

        private String getUserInput() {
                String answer = null;

                System.out.print("Would you like milk and sugar with your coffee (y/n)?
");

                BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
                try {
                        answer = in.readLine();
                } catch (IOException ioe) {
                        System.err.println("IO error trying to read your answer");
                }
                if (answer == null) {
                        return "no";
                }
                return answer;
        }
```

255

用模版方法排序

- 参见章节 框架

256

Applet

- 任何applet都必须继承自Applet
- Applet类中提供了好些钩子

257

```
public class MyApplet extends Applet {
   String message;

   public void init() {
      message = "Hello World, I'm alive!";
      repaint();
   }

   public void start() {
      message = "Now I'm starting up...";
      repaint();
   }

   public void stop() {
      message = "Oh, now I'm being stopped...";
      repaint();
   }

   public void destroy() {
      message = "Goodbye, cruel world";
      repaint();
   }

   public void paint(Graphics g) {
      g.drawString(message, 5, 15);
   }
}
```

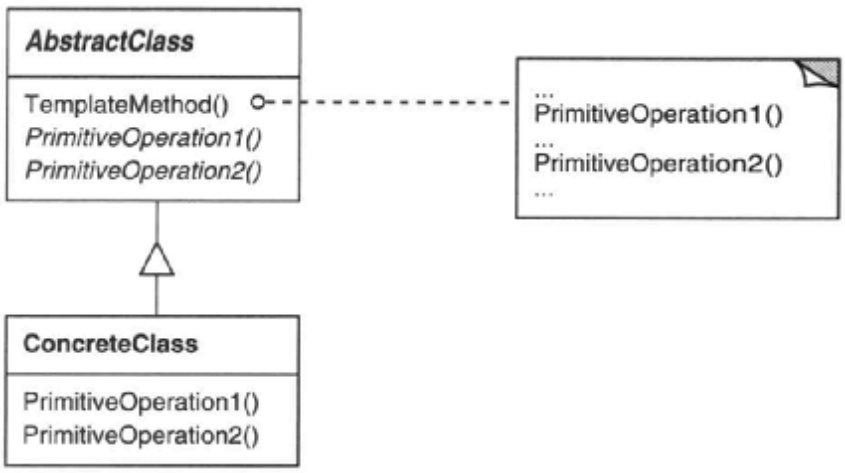
258

Template Method 模板方法

- 意图
  - 定义一个操作中的算法的框架，而将一些步骤延迟到子类中。
  - 子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤

259

Template Method 模板方法



260

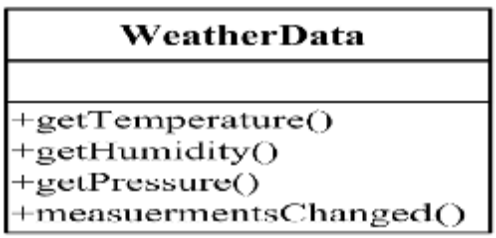
- 应用举例：**Java Applet**

```
public class MyApplet extends Applet{
      String message;
      public void init() {
            message = "Hello"; repaint();
      }
      public void start() {
            message = "Now I'm starting up";
            repaint();
      }
      public void stop() {
            message = "I'm being stopped";
            repaint();
      }
      public void destroy() {}
      public void paint(Graphics g) {
            g.drawString(message, 5, 15);
      }
};
```

设计模式－观察者模式

- 一个气象站应用
  - 现在需要建立一个应用
  - 利用气象站已有的WeatherData对象取得数据
  - 并及时更新三个布告板：
    - 目前状况布告板（显示湿度、温度、气压）
    - 气象统计布告板
    - 天气预报布告板

263



264

- 现在来实现measurementsChanged

```
Public void measurementsChanged(){
    float temp=getTemperature();
    float humidity=getHumidity();
    float pressure=getPressure();

    currentConditionDisplay.update(temp,humidity,pressure);
    statisticsDisplay.update(temp,humidity,pressure);
    forecastDisplay.update(temp,humidity,pressure);
}
```

265

- 在这个实现中
  - 完全针对具体实现编程，而非针对接口
  - 对于每个新的布告板，我们都得修改代码
  - 无法在运行时动态地增加（和删除）布告板
  - 尚未封装改变的部分

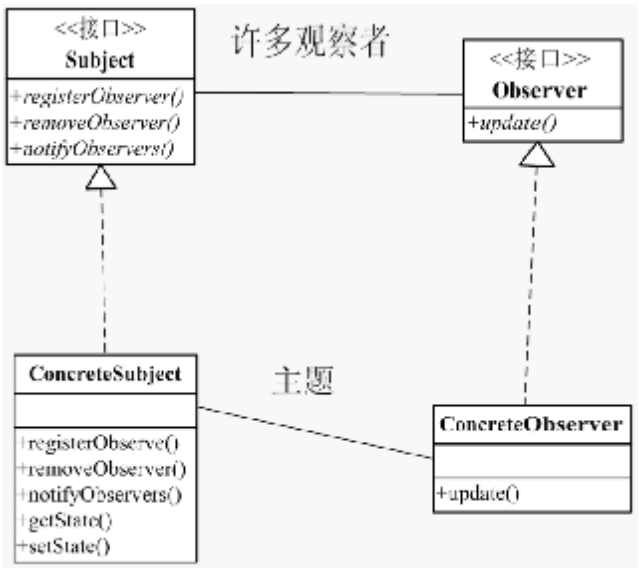
266

认识观察者模式

- 报纸和杂志的订阅
  - 报社的业务就是出版报纸
  - 向某家报社订阅报纸，只要他们有新报纸出版，就会给你送来。只要你是他们的订户，你就会一直收到新报纸
  - 当你不想再看报纸时，取消订阅

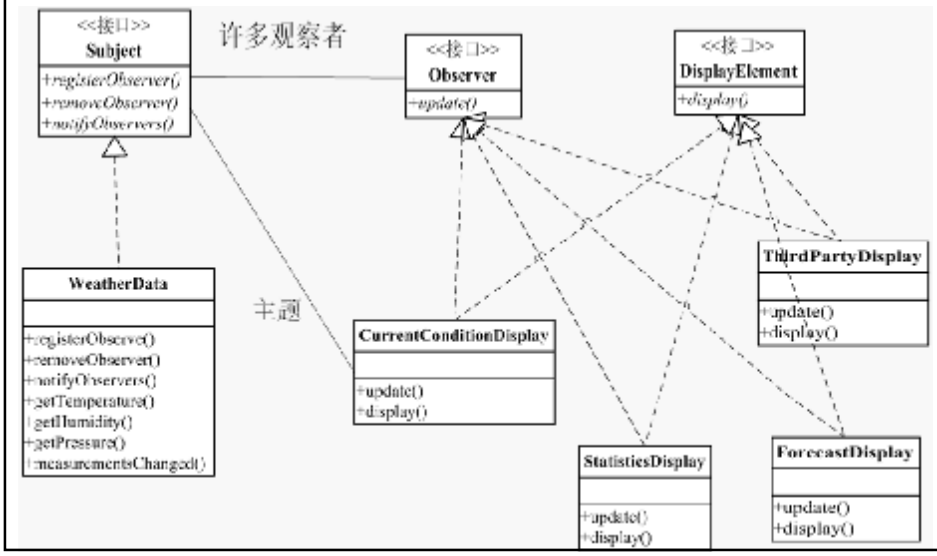
267

定义观察者模式



268

设计气象站

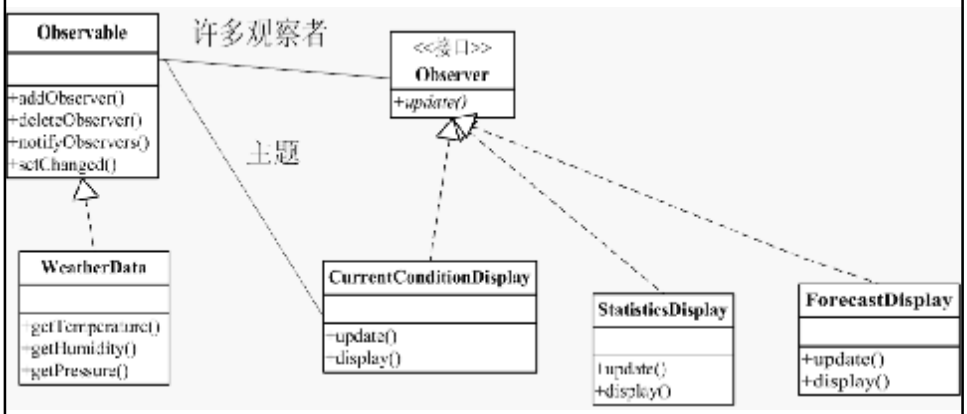


Java内置的观察者模式

- Java.util包内包含最基本的Observer接口与Observable类
- 和Observer接口与subject接口很相似

270

重新设计气象站



271

在JDK中，还有哪些观察者模式

- Swing API:JButton
- 超类AbstractButton有许多增加和删除倾听者（listener)的方法
- Button.addActionListerner(Action Listerner actionListerner)

272

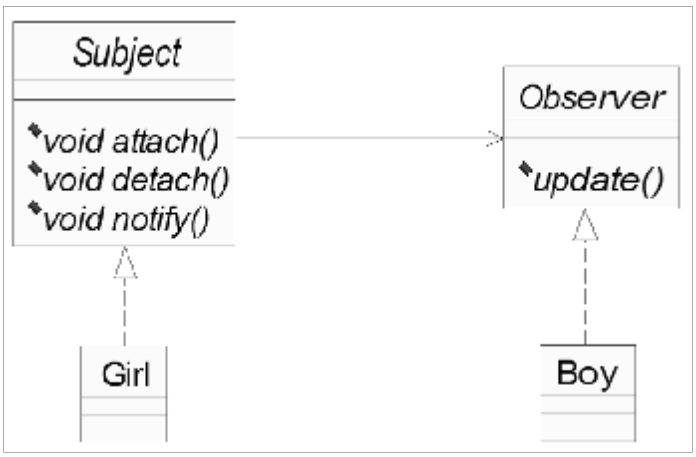
Observer(观察者)模式

- 例子：“网络聊天室”
  - 只要是女孩，一进入聊天室，所有的男孩都会立刻知晓，并做出反应(比如打招呼)
- 代码
  - 详见：**“ChatRoom”**

273

Observer(观察者)模式

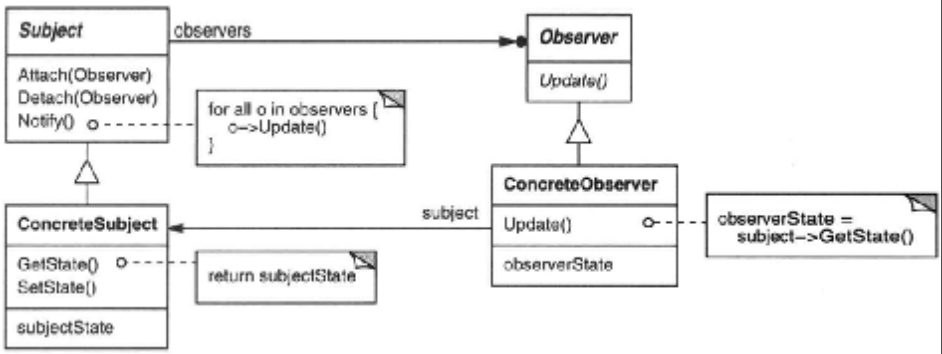
- 原理



274

Observer(*观察者*)模式

- 结构

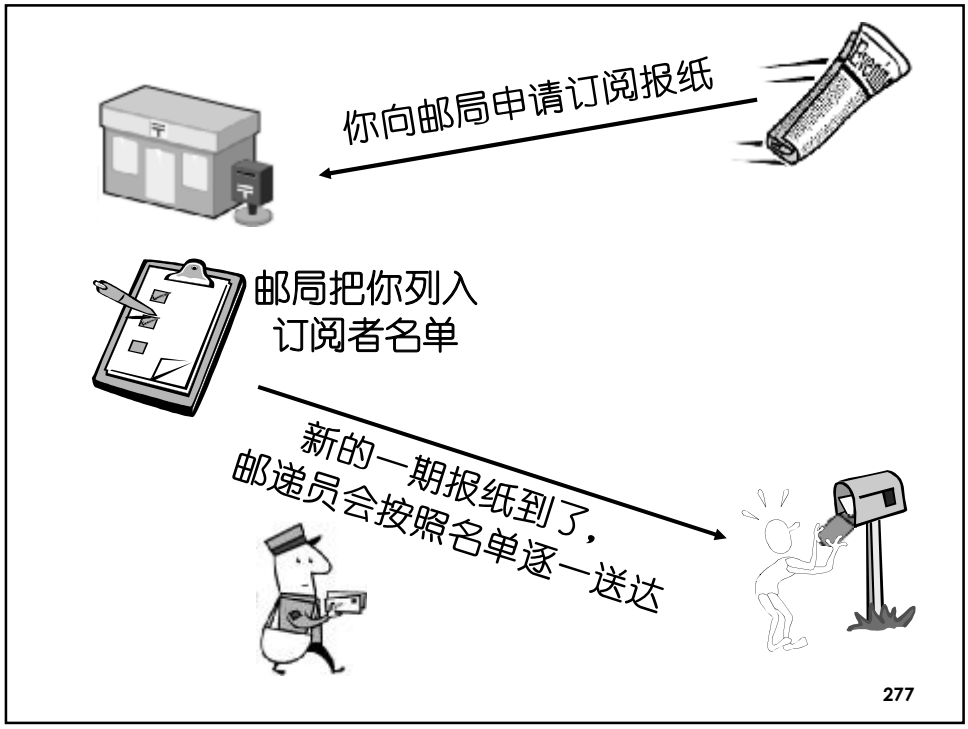


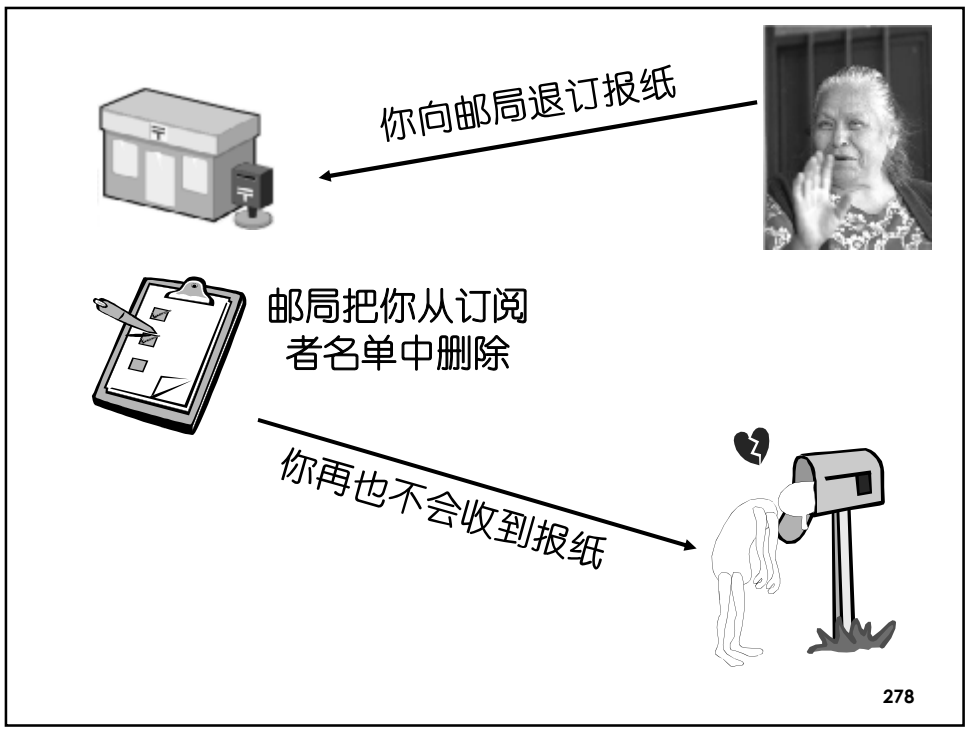
275

Observer(*观察者*)模式

- 观察者模式 = 订阅 + 广播
- 类比：邮局报刊订阅
  - 你向邮局订阅报刊
  - 邮局把你加入订阅人列表
  - 每当有新的报纸，邮局会及时送到每个订阅人手中
  - 当你不需要该报纸时，你向邮局退订，从此邮局不再给你送报纸

276


你向邮局申请订阅报纸
邮局把你列入
订阅者名单
新的一期报纸到了，
邮递员会按照名单逐一送达
277


你向邮局退订报纸
邮局把你从订阅
者名单中删除
你再也不会收到报纸
278

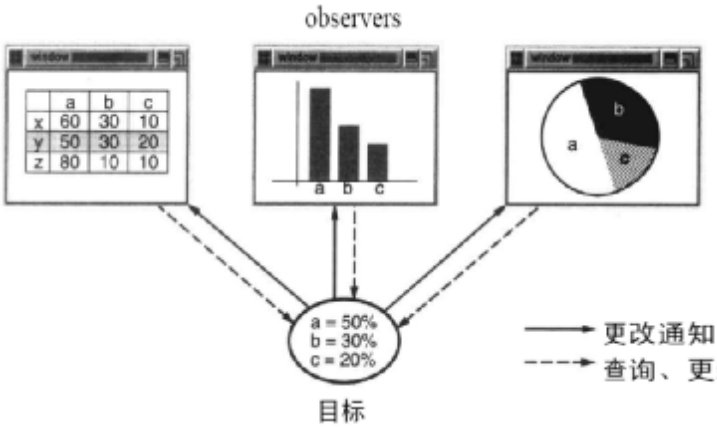
Observer(观察者)模式

- 意图
  - 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新
- 适用性
  - 当一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用
  - 当对一个对象的改变需要同时改变其它对象，而不知道具体有多少对象有待改变
  - 当一个对象必须通知其它对象，而它又不能假定其它对象是谁

279

Observer(观察者)模式

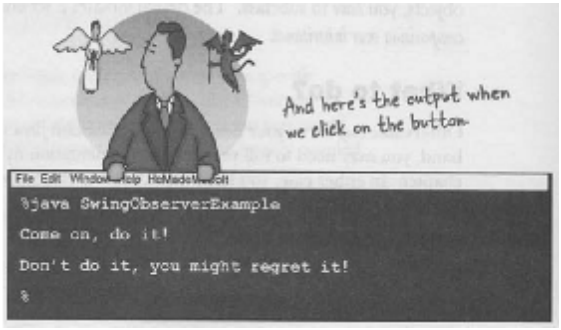
- 应用举例1
  - **MVC(Model/View/Control)模式**



280

Observer(观察者)模式

- 应用举例2：Java中的ActionListener



281

设计模式－单件模式

- 如何实例化一个唯一的对象
- 这样做的用途
  - 有些对象我们只需要一个
    - 注册表
    - 日志
    - 线性池
    - 缓存

283

经典单件模式实现

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

284

- 分析一下
- 这个经典模式多个线程处理时不能获得同一个实例

285

处理多线程

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

286

急切创建实例

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

287

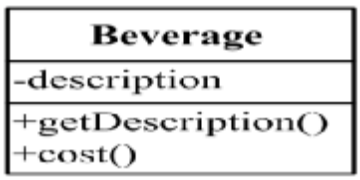
双重检查加锁

```
public class Singleton {
    private volatile static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

288

设计模式 – 装饰者模式

- 假设一个饮料店
  - 抽象一个饮料类



290

- 但是，就某种饮料来说
  - 可以要求在其中加入各种调料
  - 例如：奶，巧克力，糖等
  - 加入不同的调料，会形成不同名称的饮料
  - 这样饮料的数量将会很多，即继承于饮料父类的子类将会很多
  - 每个子类的cost（）方法将计算出特定饮料加上调料的价钱

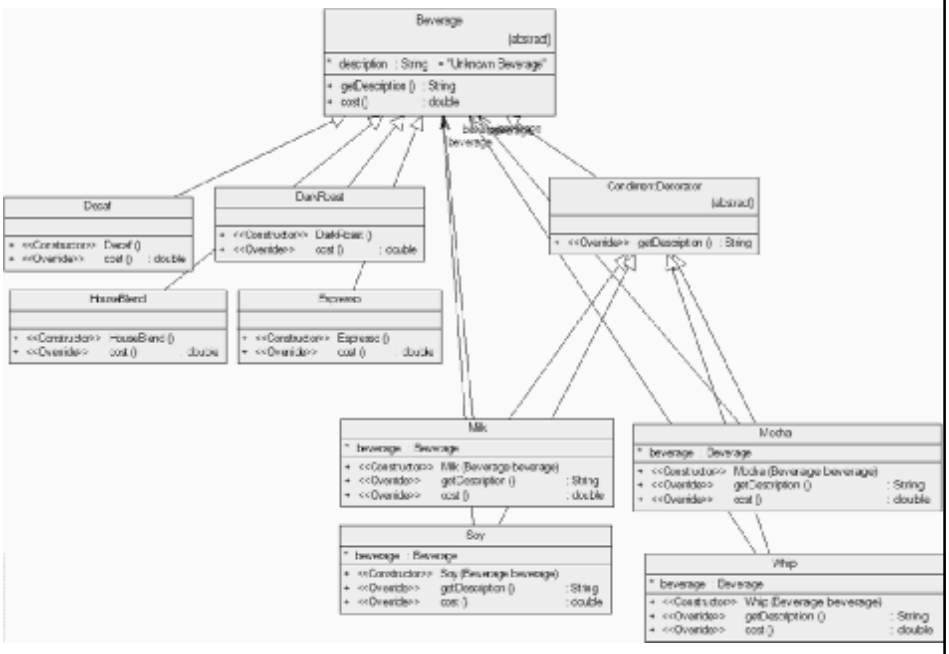
291

- 这会造成维护的噩梦
  - 如果奶的价钱上涨
  - 如果新增一种调料

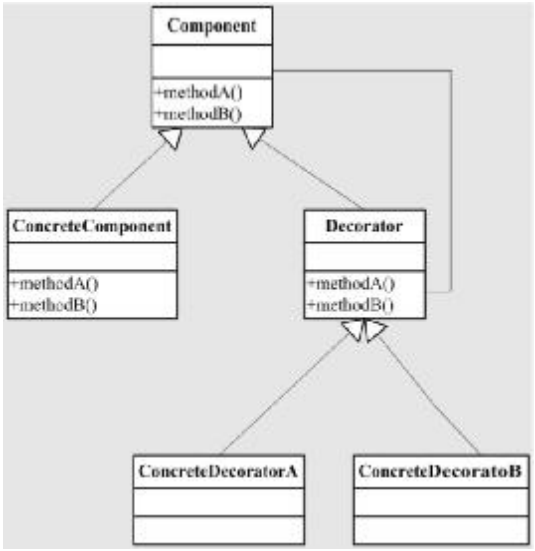
292

- 现在要以饮料为主体
- 运行时以调料来“装饰”（decorate）饮料

293



定义装饰者模式



295

- public class DarkRoast extends Beverage {
- public DarkRoast() {
- description = "Dark Roast Coffee";
- }
-
- public double cost() {
- return .99;
- }
- }

296

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

297

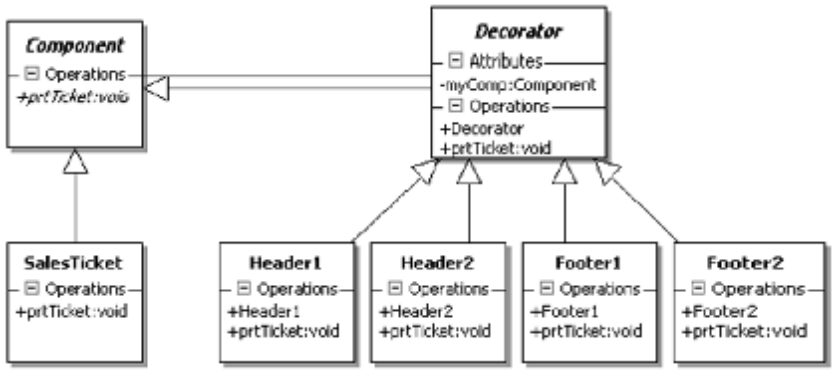
```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

298

实例

- 假设你需要打印发票 **sales ticket**，发票有抬头、正文和脚注，发票抬头可以是企事业单位，发票号等等，脚注也是一样，可能有很多不同种类的脚注需要打印。如果发票格式固定那也就没必要继续讨 论了，现在的问题是，不同的客户需要的发票或者收据的抬头或脚注，他们需要的条目是不一样的，有的需要著明单位，有的只需要发票号，但是脚注需要开票人， 等等，对你来说跟现在的 **Web** 系统一样，客户的要求是动态；不过发票的正文是不会变化的，是固定的

299



300

- 如果你的发票格式为：
- **Sales Ticket Header1**
  Sales Ticket Body
  Sales Ticket Footer1
- 那么你可以这样去创建对象：
- **new Header1(new Footer1(new SalesTicket()));**
- 如果你的发票格式为：
- **Sales Ticket Header1**
  Sales Ticket Header2
  Sales Ticket Body
  Sales Ticket Footer1
- 那么你可以这样去创建对象：
- **new Header1(new Header2(new Footer1(new SalesTicket())));**

301

设计模式－状态模式

背景

- 自动售货机
  - 所有的状态
    - 有25分钱
    - 没有25分钱
    - 糖果售罄
    - 售出糖果

303

背景

```
public class GumballMachine {

    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER =
    2;
    final static int SOLD = 3;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int
    count) {
        this.count = count;
        if (count > 0) {
                state =
    NO_QUARTER;
        }
    }

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
                System.out.println("You
    can't insert another quarter");
        } else if (state == NO_QUARTER) {
                state = HAS_QUARTER;
                System.out.println("You
    inserted a quarter");
        } else if (state == SOLD_OUT) {
                System.out.println("You
    can't insert a quarter, the machine is sold
    out");
        } else if (state == SOLD) {
    System.out.println("Please wait, we're
    already giving you a gumball");
        }
    }
```

304

背景

```
public void ejectQuarter() {
    if (state == HAS_QUARTER)
{

    System.out.println("Quarte
r returned");
            state =
NO_QUARTER;
    } else if (state ==
NO_QUARTER) {

    System.out.println("You
haven't inserted a quarter");
    } else if (state == SOLD) {

    System.out.println("Sorry,
you already turned the crank");
    } else if (state ==
SOLD_OUT) {
    System.out.println("You
can't eject, you haven't
inserted a quarter yet");
    }
```

```
public void turnCrank() {
    if (state == SOLD) {

    System.out.println("Turning twice
doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
            System.out.println("You
turned but there's no quarter");
    } else if (state == SOLD_OUT) {
            System.out.println("You
turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
            System.out.println("You
turned...");
            state = SOLD;
            dispense();
    }
}
```

305

背景

```
public void ejectQuarter() {
    if (state == HAS_QUARTER)
{

    System.out.println("Quarte
r returned");
            state =
NO_QUARTER;
    } else if (state ==
NO_QUARTER) {

    System.out.println("You
haven't inserted a quarter");
    } else if (state == SOLD) {

    System.out.println("Sorry,
you already turned the crank");
    } else if (state ==
SOLD_OUT) {
    System.out.println("You
can't eject, you haven't
inserted a quarter yet");
    }
}
```

```
public void turnCrank() {
    if (state == SOLD) {

    System.out.println("Turning twice
doesn't get you another gumball!");
    } else if (state == NO_QUARTER) {
            System.out.println("You
turned but there's no quarter");
    } else if (state == SOLD_OUT) {
            System.out.println("You
turned, but there are no gumballs");
    } else if (state == HAS_QUARTER) {
            System.out.println("You
turned...");
            state = SOLD;
            dispense();
    }
}
```

306

背景

```
public void dispense() {
        if (state == SOLD) {
                System.out.println("A gumball comes rolling out
    the slot");
                count = count - 1;
                if (count == 0) {
                        System.out.println("Oops, out of
    gumballs!");
                        state = SOLD_OUT;
                } else {
                        state = NO_QUARTER;
                }
        } else if (state == NO_QUARTER) {
                System.out.println("You need to pay first");
        } else if (state == SOLD_OUT) {
                System.out.println("No gumball dispensed");
        } else if (state == HAS_QUARTER) {
                System.out.println("No gumball dispensed");
        }
    }
```

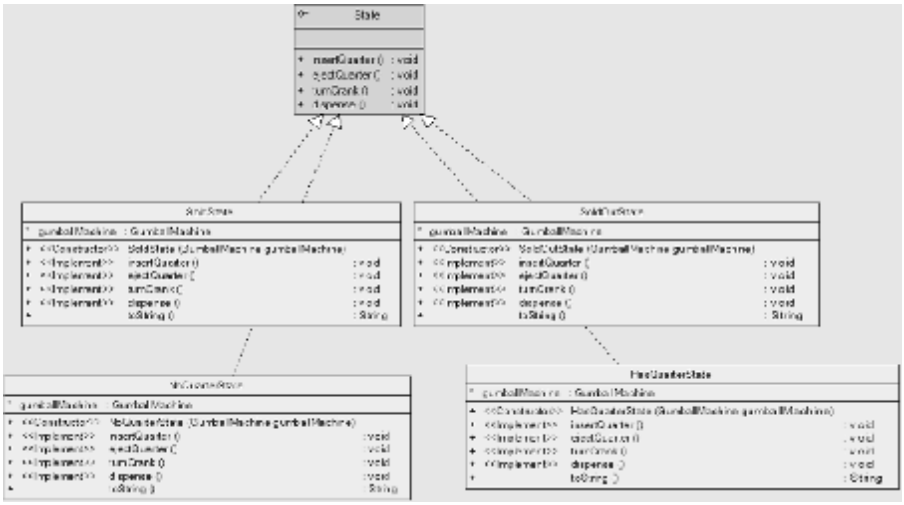
307

变更请求

- 当曲柄被转动时，有10%的几率掉下来的是两个糖果

308

定义状态类和接口



309

```
public interface State {

    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

310

```
public class NoQuarterState implements State {
   GumballMachine gumballMachine;

   public NoQuarterState(GumballMachine gumballMachine) {
      this.gumballMachine = gumballMachine;
   }

     public void insertQuarter() {
           System.out.println("You inserted a quarter");
           gumballMachine.setState(gumballMachine.getHasQuarterState());
     }

     public void ejectQuarter() {
           System.out.println("You haven't inserted a quarter");
     }

     public void turnCrank() {
           System.out.println("You turned, but there's no quarter");
      }

     public void dispense() {
           System.out.println("You need to pay first");
     }

     public String toString() {
           return "waiting for quarter";
```

311

```
public class HasQuarterState implements State {
      GumballMachine gumballMachine;

      public HasQuarterState(GumballMachine gumballMachine) {
            this.gumballMachine = gumballMachine;
      }

      public void insertQuarter() {
            System.out.println("You can't insert another quarter");
      }

      public void ejectQuarter() {
            System.out.println("Quarter returned");
            gumballMachine.setState(gumballMachine.getNoQuarterState());
      }

      public void turnCrank() {
            System.out.println("You turned...");
            gumballMachine.setState(gumballMachine.getSoldState());
      }

   public void dispense() {
      System.out.println("No gumball dispensed");
   }

      public String toString() {
            return "waiting for turn of crank";
      }
}
```

312

```
public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert a quarter, the machine is sold out");
    }

    public void ejectQuarter() {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }

    public void turnCrank() {
        System.out.println("You turned, but there are no gumballs");
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }

    public String toString() {
        return "sold out";
    }
}
```

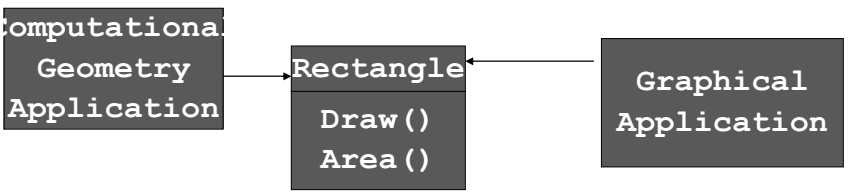
313

```
public class SoldState implements State {

    GumballMachine gumballMachine;

    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }

    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }

    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }

    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }

    public String toString() {
```

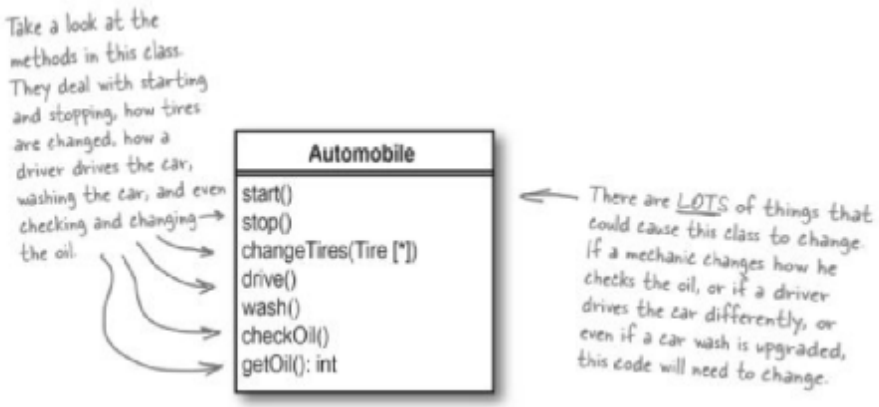
314

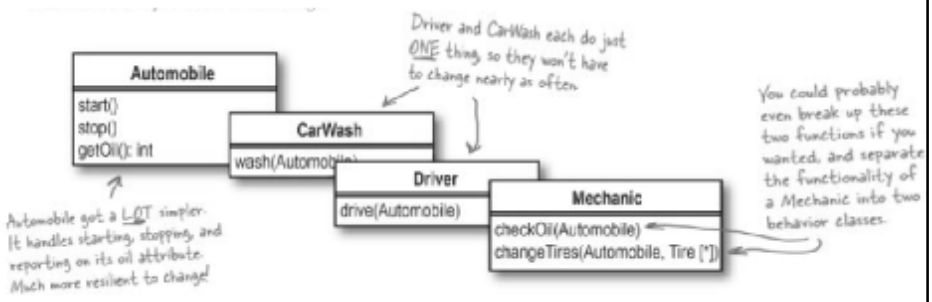
```
public class GumballMachine {

        State soldOutState;
        State noQuarterState;
        State hasQuarterState;
        State soldState;

        State state = soldOutState;
        int count = 0;

        public GumballMachine(int numberGumballs) {
                soldOutState = new SoldOutState(this);
                noQuarterState = new NoQuarterState(this);
                hasQuarterState = new HasQuarterState(this);
                soldState = new SoldState(this);

                this.count = numberGumballs;
                if (numberGumballs > 0) {
                        state = noQuarterState;
                }
        }

        public void insertQuarter() {
                state.insertQuarter();
        }

        public void ejectQuarter() {
                state.ejectQuarter();
        }
        public void turnCrank() {
                state.turnCrank();
                state.dispense();
```


设计原则-单一职责原则（SRP）



- **Modem示例**
  - **Dial()**
  - **Hangup()**
  - **Send()**
  - **Receive()**
- 却显示两个职责
  - 连接管理
  - 数据通信





设计模式和设计原则

- 设计原则
  - 开-闭：增加程序的功能时，不应该改动原有代码，只要增加新的代码即可
  - 封装可变性：限制变化的影响范围
  - 组合优先：优先使用组合，而不是继承
  - 依赖倒置：依赖抽象，而不是具体
  - 针对接口编程：用抽象类来声明变量等
  - 迪米特法则：耦合尽量松散
  - 接口隔离：一个类只做一件事情
  - 单一职责原则（**SRP**）

设计模式和设计原则

- 设计模式
  - 设计模式是在实践中具体如何贯彻设计原则的通用方案
    - 桥梁**(Bridge)**
    - 策略**(Strategy)**
    - 工厂方法**(Factory Method)**
    - 适配器**(Adapter)**
    - 命令**(Command)**
    - 观察者**(Observer)**
    - 多个模式的组合：**MVC**
    - 装饰者
    - 单件

321

设计模式：参考资料

- 《设计模式：可复用面向对象软件的基础》
  - **gof**著
  - 最权威
  - 但是比较难懂

322

设计模式：参考资料

- 《设计模式解析》
  - Alan Shalloway著
  - 比较通俗
  - 适合初学者

323

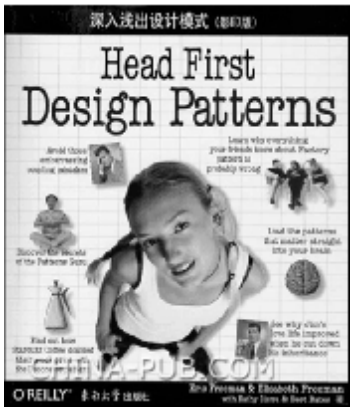
设计模式：参考资料

- 《Java与模式》
  - 阎宏著
  - 国产的精品
  - 包含了丰富的道家思想
  - 形象生动

324

设计模式：参考资料

- 《深入浅出设计模式》
  - **Elisabeth Freeman**等
  - 东南大学出版社
  - 风格就如书名



325

TDD + Design Pattern + Refactoring

- 目前OOD流行技术的大综合
  - **TDD + Design Pattern + Refactoring**

326