第 11 章　面向对象设计

如前所述，分析是提取和整理用户需求，并建立问题域精确模型的过程。设计则是把分析阶段得到的需求转变成符合成本和质量要求的、抽象的系统实现方案的过程。从面向对象分析到面向对象设计(OOD)，是一个逐渐扩充模型的过程。或者说，面向对象设计就是用面向对象观点建立求解域模型的过程。

尽管分析和设计的定义有明显区别，但是在实际的软件开发过程中二者的界限是模糊的。许多分析结果可以直接映射成设计结果，而在设计过程中又往往会加深和补充对系统需求的理解，从而进一步完善分析结果。因此，分析和设计活动是一个多次反复迭代的过程。面向对象方法学在概念和表示方法上的一致性，保证了在各项开发活动之间的平滑(无缝)过渡，领域专家和开发人员能够比较容易地跟踪整个系统开发过程，这是面向对象方法与传统方法比较起来所具有的一大优势。

生命周期方法学把设计进一步划分成总体设计和详细设计两个阶段，类似地，也可以把面向对象设计再细分为系统设计和对象设计。系统设计确定实现系统的策略和目标系统的高层结构。对象设计确定解空间中的类、关联、接口形式及实现服务的算法。系统设计与对象设计之间的界限，比分析与设计之间的界限更模糊，本书不再对它们加以区分。

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

11.1 面向对象设计的准则

所谓优秀设计，就是权衡了各种因素，从而使得系统在其整个生命周期中的总开销最小的设计。对大多数软件系统而言，60%以上的软件费用都用于软件维护，因此，优秀软件设计的二个主要特点就是容易维护。

本书第 5 章曾经讲述了指导软件设计的几条基本原理，这些原理在进行面向对象设计时仍然成立，但是增加了一些与面向对象方法密切相关的新特点，从而具体化为下列的面向对象设计准则。

1.模块化

面向对象软件开发模式，很自然地支持了把系统分解成模块的设计原理：对象就是模块。它是把数据结构和操作这些数据的方法紧密地结合在一起所构成的模块。

2.抽象

面向对象方法不仅支持过程抽象，而且支持数据抽象。类实际上是一种抽象数据类型，它对外开放的公共接口构成了类的规格说明(即协议)，这种接口规定了外界可以使用的合法操作符，利用这些操作符可以对类实例中包含的数据进行操作。使用者无须知道这些操作符的实现算法和类中数据元素的具体表示方法，就可以通过这些操作符使用类中定义的数据。通常把这类抽象称为规格说明抽象。

此外，某些面向对象的程序设计语言还支持参数化抽象。所谓参数化抽象，是指当描述

类的规格说明时并不具体指定所要操作的数据类型，而是把数据类型作为参数。这使得类的抽象程度更高，应用范围更广，可重用性更高。例如，C++语言提供的“模板”机制就是一种参数化抽象机制。

3.信息隐藏

在面向对象方法中，信息隐藏通过对象的封装性实现：类结构分离了接口与实现，从而支持了信息隐藏。对于类的用户来说，属性的表示方法和操作的实现算法都应该是隐藏的。

4.弱耦合

耦合指一个软件结构内不同模块之间互连的紧密程度。在面向对象方法中，对象是最基本的模块，因此，耦合主要指不同对象之间相互关联的紧密程度。弱耦合是优秀设计的一个重要标准，因为这有助于使得系统中某一部分的变化对其他部分的影响降到最低程度。在理想情况下，对某一部分的理解、测试或修改，无须涉及系统的其他部分。

如果一类对象过多地依赖其他类对象来完成自己的工作，则不仅给理解、测试或修改这个类带来很大困难，而且还将大大降低该类的可重用性和可移植性。显然，类之间的这种相互依赖关系是紧耦合的.

当然，对象不可能是完全孤立的，当两个对象必须相互联系相互依赖时，应该通过类的协议(即公共接口)实现耦合，而不应该依赖于类的具体实现细节。

一般说来，对象之间的耦合可分为两大类，下面分别讨论这两类耦合：

如果对象之间的耦合通过消息连接来实现，则这种耦合就是交互耦合。为使交互耦合尽可能松散，应该遵守下述准则：　　J

·尽量降低消息连接的复杂程度。应该尽量减少消息中包含的参数个数，降低参数的复杂程度。

·减少对象发送(或接收)的消息数。

(2)继承耦合

与交互耦合相反，应该提高继承耦合程度。继承是一般化类与特殊类之间耦合的一种形式。从本质上看，通过继承关系结合起来的基类和派生类，构成了系统中粒度更大的模块。因此，它们彼此之间应该结合得越紧密越好。

为获得紧密的继承耦合，特殊类应该确实是对它的一般化类的一种具体化。因此，如果一个派生类摒弃了它基类的许多属性，则它们之间是松耦合的。在设计时应该使特殊类尽量多继承并使用其一般化类的属性和服务，从而更紧密地耦合到其一般化类。

5.强内聚

内聚衡量一个模块内各个元素彼此结合的紧密程度。也可以把内聚定义为：设计中使用的一个构件内的各个元素，对完成一个定义明确的目的所做出的贡献程度。在设计时应该力求做到高内聚。在面向对象设计中存在下述3种内聚。

(1)服务内聚。一个服务应该完成一个且仅完成一个功能。

(2)类内聚。设计类的原则是，一个类应该只有一个用途，它的属性和服务应该是高内聚的。类的属性和服务应该全都是完成该类对象的任务所必需的，其中不包含无用的属性或服务。如果某个类有多个用途，通常应该把它分解成多个专用的类。

(3)一般-特殊内聚。设计出的一般-特殊结构，应该符合多数人的概念，更准确地说，这种结构应该是对相应的领域知识的正确抽取。

例如，虽然表面看来飞机与汽车有相似的地方(都用发动机驱动，都有轮子，……)，但是，如果把飞机和汽车都作为“机动车”类的子类，则明显违背了人们的常识，这样的一般-特殊结构是低内聚的。正确的作法是，设置一个抽象类“交通工具”，把飞机和机动车作为

通工具类的子类，而汽车又是机动车类的子类。

一般说来，紧密的继承耦合与高度的一般一特殊内聚是一致的。

6.可重用

软件重用是提高软件开发生产率和目标系统质量的重要途径。重用基本上从设计阶段开始。重用有两方面的含义：一是尽量使用已有的类(包括开发环境提供的类库，及以往开发类似系统时创建的类)，二是如果确实需要创建新类，则在设计这些新类的协议时，该考虑将来的可重复使用性。关于软件重用问题，将在11.3节进一步讨论。

11.2 启发规则

人们使用面向对象方法学开发软件的历史虽然不长，但也积累了一些经验。总结这经验得出了几条启发规则，它们往往能帮助软件开发人员提高面向对象设计的质量。

1.设计结果应该清晰易懂

使设计结果清晰、易读、易懂，是提高软件可维护性和可重用性的重要措施。显然，人们不会重用那些他们不理解的设计。保证设计结果清晰易懂的主要因素如下。

(1)用词一致。应该使名字与它所代表的事物一致，而且应该尽量使用人们习惯的名字。不同类中相似服务的名字应该相同。

(2)使用已有的协议。如果开发同一软件的其他设计人员已经建立了类的协议，或者在所使用的类库中已有相应的协议，则应该使用这些已有的协议。

(3)减少消息模式的数目。如果已有标准的消息协议，设计人员应该遵守这些协议。如果确需自己建立消息协议，则应该尽量减少消息模式的数目，只要可能，就使消息具有一致的模式，以利于读者理解。

(4)避免模糊的定义。一个类的用途应该是有限的，而且应该从类名可以较容易地推想出它的用途。

2.一般-特殊结构的深度应适当

应该使类等级中包含的层次数适当。一般说来，在一个中等规模(大约包含100个类)的系统中，类等级层次数应保持为7±2。不应该仅仅从方便编码的角度出发随意创建派生类，应该使一般-特殊结构与领域知识或常识保持一致。

3.设计简单的类

应该尽量设计小而简单的类，以便于开发和管理。当类很大的时候，要记住它的所有服务是非常困难的。经验表明，如果一个类的定义不超过一页纸(或两屏)，则使用这个类是比较容易的。为使类保持简单，应该注意以下几点。

(1)避免包含过多的属性。属性过多通常表明这个类过分复杂了，它所完成的功能可能太多了。

(2)有明确的定义。为了使类的定义明确，分配给每个类的任务应该简单，最好能用一两个简单语句描述它的任务。

(3)尽量简化对象之间的合作关系。如果需要多个对象协同配合才能做好一件事，则破坏了类的简明性和清晰性。

(4)不要提供太多服务。一个类提供的服务过多，同样表明这个类过分复杂。典型地，一个类提供的公共服务不超过7个。

在开发大型软件系统时，遵循上述启发规则也会带来另一个问题：设计出大量较小的类，这同样会带来一定复杂性。解决这个问题的办法，是把系统中的类按逻辑分组，也就是划分“主题”。

4.使用简单的协议

一般说来，消息中的参数不要超过 3 个。当然，不超过 3 个的限制也不是绝对的，但是，经验表明，通过复杂消息相互关联的对象是紧耦合的，对一个对象的修改往往导致其他对象的修改。

5.使用简单的服务

面向对象设计出来的类中的服务通常都很小，一般只有 3～5 行源程序语句，可以用仅含一个动词和一个宾语的简单句子描述它的的功能。如果一个服务中包含了过多的源程序语句，或者语句嵌套层次太多，或者使用了复杂的 CASE 语句，则应该仔细检查这个服务，设法分解或简化它。一般说来，应该尽量避免使用复杂的服务。如果需要在服务中使用 CASE 语句，通常应该考虑用一般一特殊结构代替这个类的可能性。

6.把设计变动减至最小

通常，设计的质量越高，设计结果保持不变的时间也越长。即使出现必须修改设计的情况，也应该使修改的范围尽可能小。理想的设计变动曲线如图 11.1 所示。

在设计的早期阶段，变动较大，随着时间推移，设计方案日趋成熟，改动也越来越小了。图 11.1 中的峰值与出现设计错误或发生非预期变动的情况相对应。峰值越高，表明设计质量越差，可重用性也越差。

1.重用

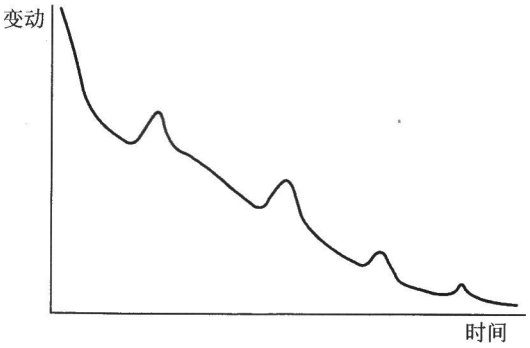


图 11.1 理想的设计变动情况

11.3 软件重用

11.3.1 概述

1.重用

重用也叫再用或复用，是指同一事物不作修改或稍加改动就多次重复使用。广义地说，软件重用可分为以下 3 个层次：

(1)知识重用(例如，软件工程知识的重用)。

(2)方法和标准的重用(例如，面向对象方法或国家制定的软件开发规范的重用)。

(3)软件成分的重用。

前两个重用层次属于知识工程研究的范畴，本节仅讨论软件成分重用问题。

2.软件成分的重用级别

软件成分的重用可以进一步划分成以下3个级别：

(1)代码重用

人们谈论得最多的是代码重用，通常把它理解为调用库中的模块。实际上，代码重用也可以采用下列几种形式中的任何一种：

·源代码剪贴：这是最原始的重用形式。这种重用方式的缺点，是复制或修改原有代码时可能出错，更糟糕的是，存在严重的配置管理问题，人们几乎无法跟踪原始代码块多次修改重用的过程。

·源代码包含：许多程序设计语言都提供包含(include)库中源代码的机制。使用这种重用形式时，配置管理问题有所缓解，因为修改了库中源代码之后，所有包含它的程序自然都必须重新编译。

·继承：利用继承机制重用类库中的类时，无须修改已有的代码，就可以扩充或具体化在库中找出的类，因此，基本上不存在配置管理问题。

(2)设计结果重用

设计结果重用指的是，重用某个软件系统的设计模型(即求解域模型)。这个级别的重用有助于把一个应用系统移植到完全不同的软硬件平台上。

(3)分析结果重用

这是一种更高级别的重用，即重用某个系统的分析模型。这种重用特别适用于用户需求未改变，但系统体系结构发生了根本变化的场合。

3.典型的可重用软件成分

更具体地说，可能被重用的软件成分主要有以下10种：

(1)项目计划。软件项目计划的基本结构和许多内容(例如，软件质量保证计划)都是可以跨项目重用的。这样做减少了用于制定计划的时间，也降低了与建立进度表和进行风险分析等活动相关联的不确定性。

(2)成本估计。因为在不同项目中经常含有类似的功能，所以有可能在只做极少修改或根本不做修改的情况下，重用对该功能的成本估计结果。

(3)体系结构。即使在考虑不同的应用领域时，也很少有截然不同的程序和数据体系结构。因此，有可能创建一组类属的体系结构模板(例如，事务处理体系结构)，并把那些模板作为可重用的设计框架。通常把类属的体系结构模板称为领域体系结构。

(4)需求模型和规格说明。类和对象的模型及规格说明是明显的重用的候选者，此外，用传统软件工程方法开发的分析模型(例如，数据流图)，也是可重用的.

(5)设计。用传统方法开发的体系结构、数据、接口和过程设计结果，是重用的候者，更常见的是，系统和对象设计是可重用的。

(6)源代码。用兼容的程序设计语言书写的、经过验证的程序构件，是重用的候选者。

(7)用户文档和技术文档。即使针对的应用是不同的，也经常有可能重用用户文档和技术文档的大部分。

(8)用户界面。这可能是最广泛被重用的软件成分，GUI(图形用户界面)软件经常被重用。因为它可占到一个应用程序的60 9／6代码量，因此，重用的效果非常显著。

(9)数据。在大多数经常被重用的软件成分中，被重用的数据包括：内部表、列表和记录结构，以及文件和完整的数据库。

(10)测试用例。一旦设计或代码构件将被重用，相关的测试用例应该“附属于”它们也被重用。

11.3.2 类构件

利用面向对象技术，可以更方便更有效地实现软件重用。面向对象技术中的“类”，

是比较理想的可重用软构件，不妨称之为类构件。类构件有3种重用方式，分别是实例重用、继承重用和多态重用。下面进一步讲述与类构件有关的内容。

1.可重用软构件应具备的特点

为使软构件也像硬件集成电路那样，能在构造各种各样的软件系统时方便地重复使用，就必须使它们满足下列要求。

(1)模块独立性强。具有单一、完整的功能，且经过反复测试被确认是正确的。它应该是一个不受或很少受外界干扰的封装体，其内部实现在外面是不可见的。

(2)具有高度可塑性。软构件的应用环境比集成电路更广阔、更复杂。显然，要求一个软构件能满足任何一个系统的设计需求是不现实的。因此，可重用的软构件必须具有高度可裁剪性，也就是说，必须提供为适应特定需求而扩充或修改已有构件的机制，而且所提供的机制必须使用起来非常简单方便。

(3)接口清晰、简明、可靠。软构件应该提供清晰、简明、可靠的对外接口，而且还应该有详尽的文档说明，以方便用户使用。

从本书第9章讲述的面向对象基本概念可以知道，精心设计的“类”基本上能满足上述要求，可以认为它是可重用软构件的雏形。

2.类构件的重用方式

(1)实例重用

由于类的封装性，使用者无须了解实现细节就可以使用适当的构造函数，按照需要创建类的实例。然后向所创建的实例发送适当的消息，启动相应的服务，完成需要完成的工作。这是最基本的重用方式。此外，还可以用几个简单的对象作为类的成员创建出一个更复杂的类，这是实例重用的另一种形式。

虽然实例重用是最基本的重用方式，但是，设计出一个理想的类构件并不是一件容易的事情。例如，决定一个类对外提供多少服务就是一件相当困难的事。提供的服务过多，会增加接口复杂度，也会使类构件变得难于理解；提供的服务过少，则会因为过分一般化而失去重用价值。每个类构件的合理服务数都与具体应用环境密切相关，因此找到一个合理的折衷值是相当困难的。

(2)继承重用

面向对象方法特有的继承性提供了一种对已有的类构件进行裁剪的机制。当已有的类构件不能通过实例重用完全满足当前系统需求时，继承重用提供了一种安全地修改已有类构件，以便在当前系统中重用的手段。

为提高继承重用的效果，关键是设计一个合理的、具有一定深度的类构件继承层次结构。这样做有下述两个好处：

·每个子类在继承父类的属性和服务的基础上，只加入少量新属性和新服务，这就不仅降低了每个类构件的接口复杂度，表现出一个清晰的进化过程，提高了每个子类的可理解性，而且为软件开发人员提供了更多可重用的类构件。因此，在软件开发过程中，应该时刻注意提取这种潜在的可重用构件，必要时应在领域专家帮助下，建立符合领域知识的继承层次。

·为多态重用奠定了良好基础。

(3)多态重用

利用多态性不仅可以使对象的对外接口更加一般化(基类与派生类的许多对外接口是相同的)，从而降低了消息连接的复杂程度，而且还提供了一种简便可靠的软构件组合机制。系统运行时，根据接收消息的对象类型，由多态性机制启动正确的方法，去响应一个一般化的消息，从而简化了消息界面和软构件连接过程。

为充分实现多态重用，在设计类构件时，应该把注意力集中在下列一些可能影响重用

性的操作上：

·与表示方法有关的操作。例如，不同实例的比较、显示、擦除等等。

·与数据结构、数据大小等有关的操作。

·与外部设备有关的操作。例如，设备控制。

·实现算法在将来可能会改进(或改变)的核心操作。

如果不预先采取适当措施，上述这些操作会妨碍类构件的重用。因此，必须把它们从类的操作中分离出来，作为“适配接口”。例如，假设类C具有操作M_1，M_2，…，M_n和操作A_1，A_2，…，A_n，其中$A_j(1\leqslant j\leqslant k)$是上面列出的可能影响类C重用的几类操作，$M_i(1\leqslant i\leqslant n)$是其他操作。如果$M_i$通过调用适配接口$A_j$而实现，则实际上M被A参数化了。在不同应用环境下，用户只需在派生类中重新定义$A_j(1\leqslant j\leqslant k)$就可以重用类C。

还可以把适配接口再进一步细分为转换接口和扩充接口。转换接口，是为了克服与表示方法、数据结构或硬件特点相关的操作给重用带来的困难而设计的，这类接口是每个类构件在重用时都必须重新定义的服务的集合。当使用C++语言编程时，应该在根类(或适当的基类)中，把属于转换接口的服务定义为纯虚函数。如果某个服务有多种可能的实现算法，则应该把它当作扩充接口。扩充接口与转换接口不同，并不需要强迫用户在派生类中重新定义它们，相反，如果在派生类中没有给出扩充接口的新算法，则将继承父类中的算法。当用C什语言实现时，在基类中把这类服务定义为普通的虚函数。

11.3.3 软件重用的效益

近几年来软件产业界的实例研究表明，通过积极的软件重用能够获得可观的商业效益，产品质量、开发生产率和整体成本都得到了改善。

1.质量

理想情况下，为了重用而开发的软件构件已被证明是正确的，且没有缺陷。事实上，由于不能定期进行形式化验证，错误可能而且也确实存在。但是，随着每一次重用，都会有一些错误被发现并被清除，构件的质量也会随之改善。随着时间的推移，构件将变成实质上无错误的。

HP公司经研究发现，被重用的代码的错误率是每千行代码中有0.9个错误，而新开发的软件的错误率是每千行代码中有4.1个错误。对于一个包含68%重用代码的应用系统来说，错误率大约是每千行代码中有2.0个错误，与不使用重用的开发相比错误率降低了51%。Henry和Fa11er报告说，使用重用的开发可使软件质量改进35%。虽然不同研究者报告的改善率并不完全相同，但是偏差都在合理的范围内，公正地说，重用确实能给软件产品的质量和可靠性带来实质性的提高。

2.生产率

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

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

3.成本

软件重用带来的净成本节省可以用下式估算：

$C=C_s - C_r - C_d$

其中，C_s是项目从头开发(没有重用)时所需要的成本；C_r是与重用相关联的成本；C_d是
交付给客户的软件的实际成本。

可以使用本书第13章讲述的技术来估算C。，而与重用相关联的成本C，主要包括下述成本：

·领域分析与建模的成本；
·设计领域体系结构的成本；
·为便于重用而增加的文档的成本；
·维护和完善可重用的软件成分的成本；
·为从外部获取构件所付出的版税和许可证费；
·创建(或购买)及运行重用库的费用；
·对设计和实现可重用构件的人员的培训费用。

虽然和领域分析及运行重用库相关联的成本可能相当高，但是它们可以由许多项目分摊。上面列出的很多其他成本所解决的问题，实际上是良好软件工程实践的一部分，不管是否优先考虑重用，这些问题都应该解决。

11.4 系统分解

人类解决复杂问题时普遍采用的策略是，“分而治之，各个击破”。同样，软件工程师在设计比较复杂的应用系统时普遍采用的策略，也是首先把系统分解成若干个比较小的部分，然后再分别设计每个部分。这样做有利于降低设计的难度，有利于分工协作，也有利于维护人员对系统理解和维护。

系统的主要组成部分称为子系统。通常根据所提供的功能来划分子系统，例如，编译系统可划分成词法分析、语法分析、中间代码生成、优化、目标代码生成和出错处理等子系统。一般说来，子系统的数目应该与系统规模基本匹配。

各个子系统之间应该具有尽可能简单、明确的接口。接口确定了交互形式和通过子系统边界的信息流，但是无须规定子系统内部的实现算法。因此，可以相对独立地设计各个子系统。

在划分和设计子系统时，应该尽量减少子系统彼此间的依赖性。

采用面向对象方法设计软件系统时，面向对象设计模型(即求解域的对象模型)，与面向对象分析模型(即问题域的对象模型)一样，也由主题、类与对象、结构、属性、服务等5个层次组成。这5个层次一层比一层表示的细节更多，可以把这5个层次想象为整个模型的水平切片。此外，大多数系统的面向对象设计模型，在逻辑上都由4大部分组成。这4大部分对应于组成目标系统的4个子系统，它们分别是问题域子系统、人机交互子系统、任务管理子系统和数据管理子系统。当然，在不同的软件系统中，这四个子系统的重要程度和规模可能相差很大，规模过大的在设计过程中应该进一步划分成更小的子系统，规模过小的可合并在其他子系统中。某些领域的应用系统在逻辑上可能仅由3个(甚至少于3个)子系统组成。

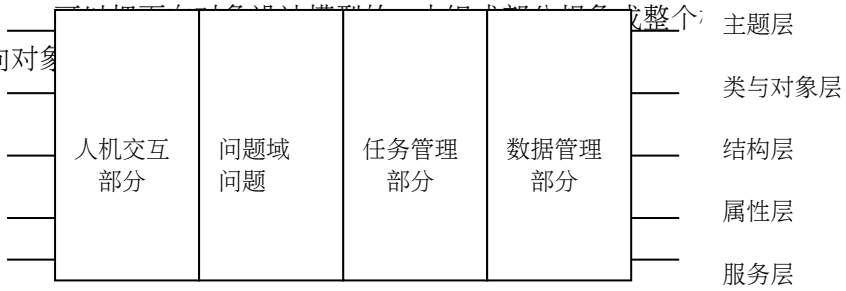


图 11.2 典型的面向对象设计模型

1.子系统之间的两种交互方式

在软件系统中，子系统之间的交互有两种可能的方式，分别是客户一供应商(Client-supplier)关系和平等伙伴(peer-to-peer)关系。

(1)客户-供应商关系

在这种关系中，作为“客户”的子系统调用作为“供应商”的子系统，后者完成某些服务工作并返回结果。使用这种交互方案，作为客户的子系统必须了解作为供应商的子系统的接口，然而后者却无须了解前者的接口，因为任何交互行为都是由前者驱动的。

(2)平等伙伴关系

在这种关系中，每个子系统都可能调用其他子系统，因此，每个子系统都必须了解其他子系统的接口。由于各个子系统需要相互了解对方的接口，因此这种组织系统的方案比起客户-供应商方案来，子系统之间的交互更复杂，而且这种交互方式还可能存在通信环路，从而使系统难于理解，容易发生不易察觉的设计错误。

总的说来，单向交互比双向交互更容易理解，也更容易设计和修改，因此应该尽量使用客户一供应商关系。

2.组织系统的两种方案

把子系统组织成完整的系统时，有水平层次组织和垂直块组织两种方案可供选择。

(1)层次组织

这种组织方案把软件系统组织成一个层次系统，每层是一个子系统。上层在下层的基础上建立，下层为实现上层功能而提供必要的服务。每一层内所包含的对象，彼此间相互独立，而处于不同层次上的对象，彼此间往往有关联。实际上，在上、下层之间存在客户-供应商关系。低层子系统提供服务，相当于供应商，上层子系统使用下层提供的服务，相当于客户。

层次结构又可进一步划分成两种模式：封闭式和开放式。所谓封闭式，就是每层子系统仅仅使用其直接下层提供的服务。由于一个层次的接口只影响与其紧相邻的上一层，因此，这种工作模式降低了各层次之间的相互依赖性，更容易理解和修改。在开放模式中，某层子系统可以使用处于其下面的任何一层子系统所提供的服务。这种工作模式的优点，是减少了需要在每层重新定义的服务数目，使得整个系统更高效更紧凑。但是，开放模式的系统不符合信息隐藏原则，对任何一个子系统的修改都会影响处在更高层次的那些子系统。设计软件系统时到底采用哪种结构模式，需要权衡效率和模块独立性等多种因素，通盘考虑以后再做决定。

通常，在需求陈述中只描述了对系统顶层和底层的需求，顶层就是用户看到的目标系统，底层则是可以使用的资源。这两层往往差异很大，设计者必须设计一些中间层次，以减少不同层次之间的概念差异。

(2)块状组织

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

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

当混合使用层次结构和块状结构时，同一层次可以由若干块组成，而同一块也可以分为若干层。例如，图 11.3 表示一个应用系统的组织结构，这个应用系统采用了层次与块状的混合结构。

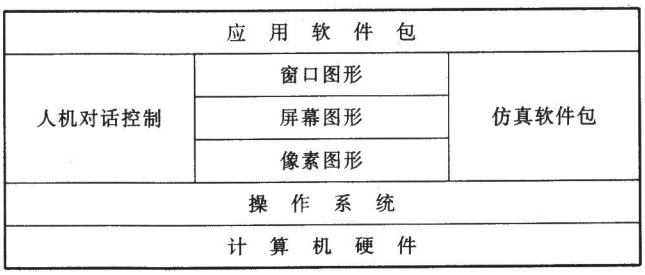


图 11.3 典型应用系统的组织结构

3.设计系统的拓扑结构

由子系统组成完整的系统时，典型的拓扑结构有管道形、树形、星形等。设计者应该采用与问题结构相适应的、尽可能简单的拓扑结构，以减少子系统之间的交互数量。

11.5 设计问题域子系统

使用面向对象方法开发软件时，在分析与设计之间并没有明确的分界线，对于问题域子系统来说，情况更是如此。但是，分析与设计毕竟是性质不同的两类开发工作，分析工作可以而且应该与具体实现无关，设计工作则在很大程度上受具体实现环境的约束。在开始进行设计工作之前(至少在完成设计之前)，设计者应该了解本项目预计要使用的编程语言，可用的软构件库(主要是类库)以及程序员的编程经验。

通过面向对象分析所得出的问题域精确模型，为设计问题域子系统奠定了良好的基础，建立了完整的框架。只要可能，就应该保持面向对象分析所建立的问题域结构。通常，面向对象设计仅需从实现角度对问题域模型做一些补充或修改，主要是增添、合并或分解类与对象、属性及服务，调整继承关系等等。当问题域子系统过分复杂庞大时，应该把它进一步分解成若干个更小的子系统。

使用面向对象方法学开发软件，能够保持问题域组织框架的稳定性，从而便于追踪分析、设计和编程的结果。在设计与实现过程中所做的细节修改(例如，增加具体类，增加属性或服务)，并不影响开发结果的稳定性，因为系统的总体框架是基于问题域的。

对于需求可能随时间变化的系统来说，稳定性是至关重要的。稳定性也是能够在类似系统中重用分析、设计和编程结果的关键因素。为更好地支持系统在其生命期中的扩充，也同样需要稳定性。

下面介绍，在面向对象设计过程中，可能对面向对象分析所得出的问题域模型做的补充或修改。

1.调整需求

有两种情况会导致修改通过面向对象分析所确定的系统需求：一是用户需求或外部环

境发生了变化；二是分析员对问题域理解不透彻或缺乏领域专家帮助，以致面向对象分析模型不能完整、准确地反映用户的真实需求。

无论出现上述哪种情况，通常都只需简单地修改面向对象分析结果，然后再把这些修改反映到问题域子系统中。

2.重用已有的类

代码重用从设计阶段开始，在研究面向对象分析结果时就应该寻找使用已有类的方法。若因为没有合适的类可以重用而确实需要创建新的类，则在设计这些新类的协议时，必须考虑到将来的可重用性。

如果有可能重用已有的类，则重用已有类的典型过程如下：

(1)选择有可能被重用的已有类，标出这些候选类中对本问题无用的属性和服务，尽量重用那些能使无用的属性和服务降到最低程度的类。

(2)在被重用的已有类和问题域类之间添加泛化关系(即从被重用的已有类派生出问题域类)。

(3)标出问题域类中从已有类继承来的属性和服务，现在已经无须在问题域类内定义它们了。

(4)修改与问题域类相关的关联，必要时改为与被重用的已有类相关的关联。

3.把问题域类组合在一起

在面向对象设计过程中，设计者往往通过引入一个根类而把问题域类组合在一起。事实上，这是在没有更先进的组合机制可用时才采用的一种组合方法。此外，这样的根类还可以用来建立协议。

4.增添一般化类以建立协议

在设计过程中常常发现，一些具体类需要有一个公共的协议，也就是说，它们都需要定义一组类似的服务(很可能还需要相应的属性)。在这种情况下可以引入一个附加类(例如，根类)，以便建立这个协议(即命名公共服务集合，这些服务在具体类中仔细定义)。

5.调整继承层次

如果面向对象分析模型中包含了多重继承关系，然而所使用的程序设计语言却并不提供多重继承机制，则必须修改面向对象分析的结果。即使使用支持多重继承的语言，有时也会

出于实现考虑而对面向对象分析结果作一些调整。下面分几种情况讨论：

(1)使用多重继承机制

使用多重继承机制时，应该避免出现属性及服务的命名冲突。下面通过例子说明避免命名冲突的方法。

图11.4是一种多重继承模式的例子，这种模式可以称为窄菱形模式。使用这种模式时出现属性及服务命名冲突的可能性比较大。

图11.5是另一种多重继承模式，称为阔菱形模式。使用这种模式时，属性及服务的名字发生冲突的可能性比较小，但是，它需要用更多的类才能表示同一个设计。

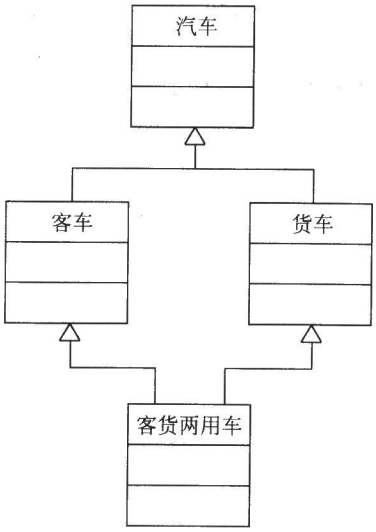


图 11.4 窄菱形模式

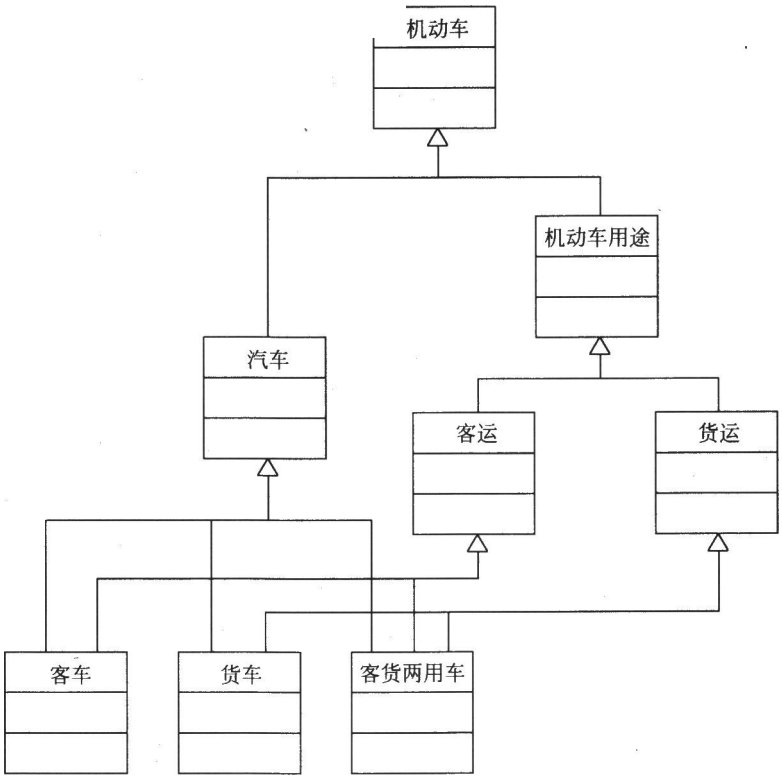


图 11.5 阔菱形模式

(2)使用单继承机制

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

常见的做法是，把多重继承结构简化成单一的单继承层次结构，如图 11.6 所示。显然，在多重继承结构中的某些继承关系，经简化后将不再存在，这表明需要在各个具体类中重复定义某些属性和服务。

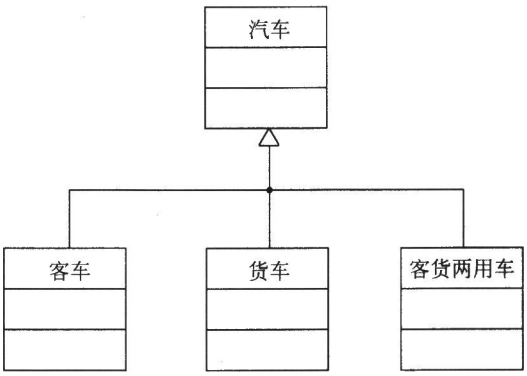


图 11.6　把多重继承简化为单一层次的单继承

6.ATM 系统实例

图 11.7 描绘了上章给出的 ATM 系统的问题域子系统的结构。在面向对象设计过程中，把 ATM 系统的问题域子系统，进一步划分成了 3 个更小的子系统，它们分别是 ATM 站子系统、中央计算机子系统和分行计算机子系统。它们的拓扑结构为星形，以中央计算机为中心向外辐射，同所有 ATM 站及分行计算机通信。物理联结用专用电话线实现。根据 ATM 站号和分行代码，区分由每个 ATM 站和每台分行计算机联向中央计算机的电话线。

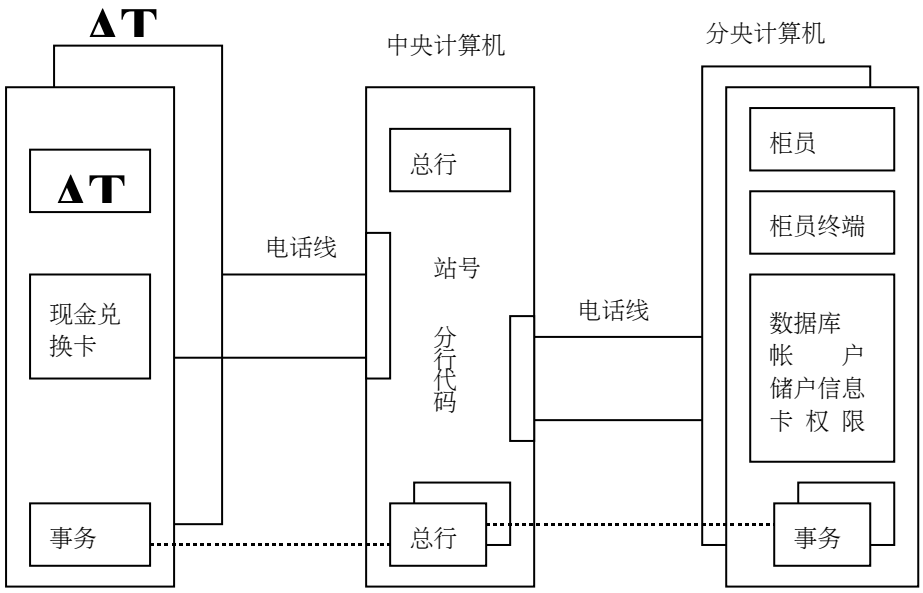


图　11.7 ATM 系统问题域子系统的结构

由于在面向对象分析过程中已经对 ATM 系统做了相当仔细的分析，而且假设所使用的实现环境能完全支持面向对象分析模型的实现，因此，在面向对象设计阶段无须对已有的问题域模型作实质性的修改或扩充。

11.6 设计人机交互子系统

在面向对象分析过程中，已经对用户界面需求做了初步分析，在面向对象设计过程中，则应该对系统的人机交互子系统进行详细设计，以确定人机交互的细节，其中包括指定窗口和报表的形式、设计命令层次等项内容。

人机交互部分的设计结果，将对用户情绪和工作效率产生重要影响。人机界面设计得好，则会使系统对用户产生吸引力，用户在使用系统的过程中会感到兴奋，能够激发用户的创造力，提高工作效率；相反，人机界面设计得不好，用户在使用过程中就会感到不方便、不习惯，甚至会产生厌烦和恼怒的情绪。

由于对人机界面的评价，在很大程度上由人的主观因素决定，因此，使用由原型支持的系统化的设计策略，是成功地设计人机交互子系统的关键。

本书6.2节已经全面系统地讲述了人机界面设计的问题、过程和设计指南，此处不再赘述。本节仅从面向对象设计的角度补充讲述一下设计人机交互子系统的策略。

1.分类用户

人机交互界面是给用户使用的，显然，为设计好人机交互子系统，设计者应该认真研究使用它的用户。应该深入到用户的工作现场，仔细观察用户是怎样做他们的工作的，这对设计好人机交互界面是非常必要的。

在深入现场的过程中，设计者应该认真思考下述问题：用户必须完成哪些工作?设计者能够提供什么工具来支持这些工作的完成?怎样使得这些工具使用起来更方便更有效?

为了更好地了解用户的需要与爱好，以便设计出符合用户需要的界面，设计者首先应该把将来可能与系统交互的用户分类。通常从下列几个不同角度进行分类：

·按技能水平分类(新手、初级、中级、高级)。

·按职务分类(总经理、经理、职员)。

·按所属集团分类(职员、顾客)。

2.描述用户

应该仔细了解将来使用系统的每类用户的情况，把获得的下列各项信息记录下来：

·用户类型。

·使用系统欲达到的目的。

·特征(年龄、性别、受教育程度、限制因素等)。

·关键的成功因素(需求、爱好、习惯等)。

·技能水平。

·完成本职工作的脚本。

3.设计命令层次

设计命令层次的工作通常包含以下几项内容。

(1)研究现有的人机交互含义和准则

现在，Windows已经成了微机上图形用户界面事实上的工业标准。所有Windows应用程序的基本外观及给用户的感受都是相同的(例如，每个程序至少有一个窗口，它由标题栏标识；程序中大多数功能可通过菜单选用；选中某些菜单项会弹出对话框，用户可通过它输入附加信息;……)。Windows程序通常还遵守广大用户习以为常的许多约定(例如，File菜单的最后一个菜单项是Exit；在文件列表框中用鼠标单击某个表项，则相应的文件名变亮，若用鼠标双击则会打开该文件;……)。

设计图形用户界面时，应该保持与普通Windows应用程序界面相一致，并遵守广大用

户习惯的约定，这样才会被用户接受和喜爱。

(2)确定初始的命令层次

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

(3)精化命令层次

为进一步修改完善初始的命令层次，应该考虑下列一些因素：

·次序：仔细选择每个服务的名字，并在命令层的每一部分内把服务排好次序。排序时或者把最常用的服务放在最前面，或者按照用户习惯的工作步骤排序。

·整体一部分关系：寻找在这些服务中存在的整体一部分模式，这样做有助于在命令层中分组组织服务。

·宽度和深度：由于人的短期记忆能力有限，命令层次的宽度和深度都不应该过大。

·操作步骤：应该用尽量少的单击、拖动和击键组合来表达命令，而且应该为高级用户提供简捷的操作方法。

4.设计人机交互类

人机交互类与所使用的操作系统及编程语言密切相关。例如，在Windows环境下运行的Visual C++语言提供了MFC类库，设计人机交互类时，往往仅需从MFC类库中选出一些适用的类，然后从这些类派生出符合自己需要的类就可以了。

11.7 设计任务管理子系统

虽然从概念上说，不同对象可以并发地工作，但是，在实际系统中，许多对象之间往往存在相互依赖关系。此外，在实际使用的硬件中，可能仅由一个处理器支持多个对象。因此，设计工作的一项重要内容就是，确定哪些是必须同时动作的对象，哪些是相互排斥的对象。然后进一步设计任务管理子系统。

1.分析并发性

通过面向对象分析建立起来的动态模型，是分析并发性的主要依据。如果两个对象彼此间不存在交互，或者它们同时接受事件，则这两个对象在本质上是并发的。通过检查各个对象的状态图及它们之间交换的事件，能够把若干个非并发的对象归并到一条控制线中。所谓控制线，是一条遍及状态图集合的路径，在这条路径上每次只有一个对象是活动的。在计算机系统中用任务(task)实现控制线，一般认为任务是进程(process)的别名。通常把多个任务的并发执行称为多任务。

对于某些应用系统来说，通过划分任务，可以简化系统的设计及编码工作。不同的任务标识了必须同时发生的不同行为。这种并发行为既可以在不同的处理器上实现，也可以在单个处理器上利用多任务操作系统仿真实现(通常采用时间分片策略仿真多处理器环境)。

2.设计任务管理子系统

常见的任务有事件驱动型任务、时钟驱动型任务、优先任务、关键任务和协调任务等。设计任务管理子系统，包括确定各类任务并把任务分配给适当的硬件或软件去执行。

(1)确定事件驱动型任务

某些任务是由事件驱动的，这类任务可能主要完成通信工作。例如，与设备、屏幕窗口、其他任务、子系统、另一个处理器或其他系统通信。事件通常是表明某些数据到达的信号。

在系统运行时，这类任务的工作过程如下：任务处于睡眠状态(不消耗处理器时间)，

等待来自数据线或其他数据源的中断；一旦接收到中断就唤醒了该任务，接收数据并把数据放入内存缓冲区或其他目的地，通知需要知道这件事的对象，然后该任务又回到睡眠状态。

(2)确定时钟驱动型任务

某些任务每隔一定时间间隔就被触发以执行某些处理，例如，某些设备需要周期性地获得数据；某些人机接口、子系统、任务、处理器或其他系统也可能需要周期性地通信。在这些场合往往需要使用时钟驱动型任务。

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

(3)确定优先任务

优先任务可以满足高优先级或低优先级的处理需求：

·高优先级：某些服务具有很高的优先级，为了在严格限定的时间内完成这种服务，可能需要把这类服务分离成独立的任务。

·低优先级：与高优先级相反，有些服务是低优先级的，属于低优先级处理(通常指那些背景处理)。设计时可能用额外的任务把这样的处理分离出来。

(4)确定关键任务

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

(5)确定协调任务

当系统中存在3个以上任务时，就应该增加一个任务，用它作为协调任务。

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

(6)尽量减少任务数

必须仔细分析和选择每个确实需要的任务。应该使系统中包含的任务数尽量少。

设计多任务系统的主要问题是，设计者常常为了自己处理时的方便而轻率地定义过多的任务。这样做加大了设计工作的技术复杂度，并使系统变得不易理解，从而也加大了系统维护的难度。

(7)确定资源需求

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

设计者应该综合考虑各种因素，以决定哪些子系统用硬件实现，哪些子系统用软件实现。下述两个因素可能是使用硬件实现某些子系统的主要原因：

·现有的硬件完全能满足某些方面的需求，例如，买一块浮点运算卡比用软件实现浮点运算要容易得多。

·专用硬件比通用的CPU性能更高。例如，目前在信号处理系统中广泛使用固件实现快速傅里叶变换。

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

11.8 设计数据管理子系统

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

11.8.1 选择数据存储管理模式

不同的数据存储管理模式有不同的特点，适用范围也不相同，设计者应该根据应用系统的特点选择适用的模式。

1.文件管理系统

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

2.关系数据库管理系统

关系数据库管理系统的理论基础是关系代数，它不仅理论基础坚实而且有下列一些主要优点：

(1)提供了各种最基本的数据管理功能(例如，中断恢复，多用户共享，多应用共享，完整性，事务支持等)。

(2)为多种应用提供了一致的接口。

(3)标准化的语言(大多数商品化关系数据库管理系统都使用 SQL 语言)。

但是，为了做到通用与一致，关系数据库管理系统通常都相当复杂，且有下述一些具体缺点，以致限制了这种系统的普遍使用：

(1)运行开销大：即使只完成简单的事务(例如，只修改表中的一行)，也需要较长的时间。

(2)不能满足高级应用的需求：关系数据库管理系统是为商务应用服务的，商务应用中数据量虽大但数据结构却比较简单。事实上，关系数据库管理系统很难用在数据类型丰富或操作不标准的应用中。

(3)与程序设计语言的连接不自然：SQL 语言支持面向集合的操作，是一种非过程性语言；然而大多数程序设计语言本质上却是过程性的，每次只能处理一个记录。

3.面向对象数据库管理系统

面向对象数据库管理系统是一种新技术，主要有两种设计途径：扩展的关系数据库管理系统和扩展的面向对象程序设计语言。

(1)扩展的关系数据库管理系统是在关系数据库的基础上，增加了抽象数据类型和继承机制，此外还增加了创建及管理类和对象的通用服务。

(2)扩展的面向对象程序设计语言扩充了面向对象程序设计语言的语法和功能，增加了在数据库中存储和管理对象的机制。开发人员可以用统一的面向对象观点进行设计，不再需要区分存储数据结构和程序数据结构(即生命期短暂的数据)。

目前，大多数“对象”数据管理模式都采用“复制对象”的方法：先保留对象值，然后，在需要时创建该对象的一个副本。扩展的面向对象程序设计语言则扩充了这种机制，它支持“永久对象”方法：准确存储对象(包括对象的内部标识在内)，而不是仅仅存储对象

值。使用这种方法，当从存储器中检索出一个对象的时候，它就完全等同于原先存在的那个对象。“永久对象”方法，为在多用户环境中从对象服务器中共享对象奠定了基础。

11.8.2 设计数据管理子系统

设计数据管理子系统，既需要设计数据格式又需要设计相应的服务。

1.设计数据格式

设计数据格式的方法与所使用的数据存储管理模式密切相关，下面分别介绍适用于每种数据存储管理模式的设计方法：

(1)文件系统

·定义第一范式表：列出每个类的属性表；把属性表规范成第一范式，从而得到第一范式表的定义。

·为每个第一范式表定义一个文件。

·测量性能和需要的存储容量。

·修改原设计的第一范式，以满足性能和存储需求。

必要时把泛化结构的属性压缩在单个文件中，以减少文件数量。

必要时把某些属性组合在一起，并用某种编码值表示这些属性，而不再分别使用独立的域表示每个属性。这样做可以减少所需要的存储空间，但是增加了处理时间。

(2)关系数据库管理系统

·定义第三范式表：列出每个类的属性表；把属性表规范成第三范式，从而得出第三范式表的定义。

·为每个第三范式表定义一个数据库表。

·测量性能和需要的存储容量。

·修改先前设计的第三范式，以满足性能和存储需求。

(3)面向对象数据库管理系统

·扩展的关系数据库途径：使用与关系数据库管理系统相同的方法。

·扩展的面向对象程序设计语言途径：不需要规范化属性的步骤，因为数据库管理系统本身具有把对象值映射成存储值的功能。

2.设计相应的服务

如果某个类的对象需要存储起来，则在这个类中增加一个属性和服务，用于完成存储对象自身的工作。应该把为此目的增加的属性和服务作为“隐含”的属性和服务，即无须在面向对象设计模型的属性和服务层中显式地表示它们，仅需在关于类与对象的文档中描述它们。

这样设计之后，对象将知道怎样存储自己。用于“存储自己”的属性和服务，在问题域子系统和数据管理子系统之间构成一座必要的桥梁。利用多重继承机制，可以在某个适当的基类中定义这样的属性和服务，然后，如果某个类的对象需要长期存储，该类就从基类中继承这样的属性和服务。

下面介绍使用不同数据存储管理模式时的设计要点。

(1)文件系统

被存储的对象需要知道打开哪个(些)文件，怎样把文件定位到正确的记录上，怎样检索出旧值(如果有的话)，以及怎样用现有值更新它们。

此外，还应该定义一个Objectserver(对象服务器)类，并创建它的实例。该类提供下

列服务：

·通知对象保存自身；

·检索已存储的对象(查找，读值，创建并初始化对象)，以便把这些对象提供给其他子系统使用。

注意，为提高性能应该批量处理访问文件的要求。

(2)关系数据库管理系统

被存储的对象，应该知道访问哪些数据库表，怎样访问所需要的行，怎样检索出旧值(如果有的话)，以及怎样用现有值更新它们。

此外，还应该定义一个 ObjectServer 类，并声明它的对象。该类提供下列服务：

·通知对象保存自身；

·检索已存储的对象(查找，读值，创建并初始化对象)，以便由其他子系统使用这些对象。

(3)面向对象数据库管理系统

·扩展的关系数据库途径：与使用关系数据库管理系统时方法相同。

·扩展的面向对象程序设计语言途径：无须增加服务，这种数据库管理系统已经给每个对象提供了“存储自己”的行为。只需给需要长期保存的对象加个标记，然后

由面向对象数据库管理系统负责存储和恢复这类对象。

11.8.3 例子

为具体说明数据管理子系统的设计方法，让我们再看看图 11.7 所示的 ATM 系统。

从图中可以看出，惟一的永久性数据存储放在分行计算机中。因为必须保持数据的一致性和完整性，而且常常有多个并发事务同时访问这些数据，因此，采用成熟的商品化关系数据库管理系统存储数据。应该把每个事务作为一个不可分割的批操作来处理，由事务封锁账户直到该事务结束为止。

在这个例子中，需要存储的对象主要是账户类的对象。为了支持数据管理子系统的实现，账户类对象必须知道自己是怎样存储的，有两种方法可以达到这个目的。

(1)每个对象自己保存自己

账户类对象在接到“存储自己”的通知后，知道怎样把自身存储起来(需要增加一个属性和一个服务来定义上述行为)。

(2)由数据管理子系统负责存储对象

账户类对象在接到“存储自己”的通知后，知道应该向数据管理子系统发送什么消息，以便由数据管理子系统把它的状态保存起来，为此也需要增加属性和服务来定义上述行为。使用这种方法的优点，是无须修改问题域子系统。

如上一小节所述，应该定义一个数据管理类 ObjectServer，并声明它的对象。这个类提供下列服务：

·通知对象保存自身或保存需长期存储的对象的状态；

·检索已存储的对象并使之“复活”。

11.9 设计类中的服务

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

11.9.1 确定类中应有的服务

需要综合考虑对象模型、动态模型和功能模型，才能正确确定类中应有的服务。对象模型是进行对象设计的基本框架。但是，面向对象分析得出的对象模型，通常只在每个类中列出很少几个最核心的服务。设计者必须把动态模型中对象的行为以及功能模型中的数据处理，转换成由适当的类所提供的服务。

一张状态图描绘了一类对象的生命周期，图中的状态转换是执行对象服务的结果。对象的许多服务都与对象接收到的事件密切相关，事实上，事件就表现为消息，接收消息的对象必然有由消息选择符指定的服务，该服务改变对象状态(修改相应的属性值)，并完成对象应做的动作。对象的动作既与事件有关，也与对象的状态有关。因此，完成服务的算法自然也和对象的状态有关。如果一个对象在不同状态可以接受同样事件，而且在不同状态接收到同样事件时其行为不同，则实现服务的算法中需要有一个依赖于状态的 DO_CASE 型控制结构。

功能模型指明了系统必须提供的服务。状态图中状态转换所触发的动作，在功能模型中有时可能扩展成一张数据流图。数据流图中的某些处理可能与对象提供的服务相对应，下列规则有助于确定操作的目标对象(即应该在该对象所属的类中定义这个服务)：

(1)如果某个处理的功能是从输人流中抽取一个值，则该输入流就是目标对象。

(2)如果某个处理具有类型相同的输入流和输出流，而且输出流实质上是输入流的另一种形式，则该输入输出流就是目标对象。

(3)如果某个处理从多个输入流得出输出值，则该处理是输出类中定义的一个服务。

(4)如果某个处理把对输入流处理的结果输出给数据存储或动作对象，则该数据存储或动作对象就是目标对象。

当一个处理涉及多个对象时，为确定把它作为哪个对象的服务，设计者必须判断哪个对象在这个处理中起主要作用。通常在起主要作用的对象类中定义这个服务。下面两条规则有助于确定处理的归属：

(1)如果处理影响或修改了一个对象，则最好把该处理与处理的目标(而不是触发者)联系在一起。

(2)考察处理涉及的对象类及这些类之间的关联，从中找出处于中心地位的类。如果其他类和关联围绕这个中心类构成星形，则这个中心类就是处理的目标。

11.9.2 设计实现服务的方法

在面向对象设计过程中还应该进一步设计实现服务的方法，主要应该完成以下几项工作。

1.设计实现服务的算法

设计实现服务的算法时，应该考虑下列几个因素：

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

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

(3)易修改。应该尽可能预测将来可能做的修改，并在设计时预先做些准备。

2.选择数据结构

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

3.定义内部类和内部操作　　。

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

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

11.10 设计关联

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

为了更好地设计实现关联的途径，首先应该分析使用关联的方式。

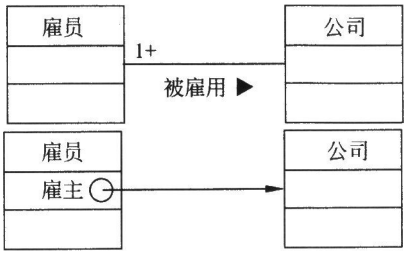
1.关联的遍历

在应用系统中，使用关联有两种可能的方式：单向遍历和双向遍历。在应用系统中，某些关联只需要单向遍历，这种单向关联实现起来比较简单，另外一些关联可能需要双向遍历，双向关联实现起来稍微麻烦一些。

在使用原型法开发软件的时候，原型中所有关联都应该是双向的，以便于增加新的行为，快速地扩充和修改原型。

2.实现单向关联　　·

用指针可以方便地实现单向关联。如果关联的重数是一元的(如图11.8所示)，则实现关联的指针是一个简单指针；如果重数是多元的，则需要用一个指针集合实现关联(参见图11.9)。



(a)关[illegible]

图11.8用指针实现单向关联

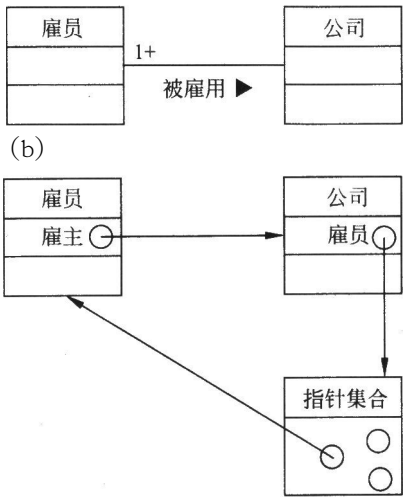


图 11.9 用指针实现双向关联
(a)关联；(b)实现

3.实现双向关联

许多关联都需要双向遍历，当然，两个方向遍历的频度往往并不相同 ◦实现双向关联有下列 3 种方法:

(1)只用属性实现一个方向的关联，当需要反向遍历时就执行一次正向查找。如果

两个方向遍历的频度相差很大，而且需要尽量减少存储开销和修改时的开销，则这是一种很有效的实现双向关联的方法。

(2)两个方向的关联都用属性实现。具体实现方法已在前面讲过，如图 11.9 所示。这种方法能实现快速访问，但是，如果修改了一个属性，则相关的属性也必须随之修改，才能保持该关联链的一致性。当访问次数远远多于修改次数时，这种实现方法很有效。

(3)用独立的关联对象实现双向关联。关联对象不属于相互关联的任何一个类，它是独立的关联类的实例，如图 11.10 所示。

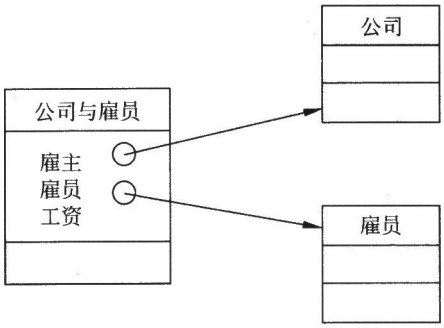


图 11.10 用对象实现关联

4.关联对象的实现

本书 9.4.2 节曾经讲过，可以引入_个关联类来保存描述关联性质的信息，关联中的每个连接对应着关联类的一个对象。实现关联对象的方法取决于关联的重数。对于一对一关联来说，关联对象可以与参与关联的任一个对象合并 ◦对于一对多关联来说，关联对象可以与“多”端对象合并 ◦如果是多对多关联，则关联链的性质不可能只与一个参与关联的对象有关，通常用一个独立的关联类来保存描述关联性质的信息，这个类的每个实例表示一条具体的关联链及该链的属性(参见图 11.10)。

11.11 设计优化

11.11.1 确定优先级

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

系统的整体质量与设计人员所制定的折衷方案密切相关。最终产品成功与否，在很大程度上取决于是否选择好了系统目标。最糟糕的情况是，没有站在全局高度正确确定各项质量指标的优先级，以致系统中各个子系统按照相互对立的目标做了优化，将导致系统资源的严重浪费。

在折衷方案中设置的优先级应该是模糊的。事实上，不可能指定精确的优先级数值(例如，速度48%，内存25%，费用8%，可修改性19%)。

最常见的情况，是在效率和清晰性之间寻求适当的折衷方案。下面两小节分别讲述在优化设计时提高效率的技术，以及建立良好的继承结构的方法。

11.11.2 提高效率的几项技术

1.增加冗余关联以提高访问效率

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

图11.11是从面向对象分析模型中摘取的一部分。公司类中的服务find_skill返回具有指定技能的雇员集合。例如，用户可能询问公司中会讲日语的雇员有哪些人。

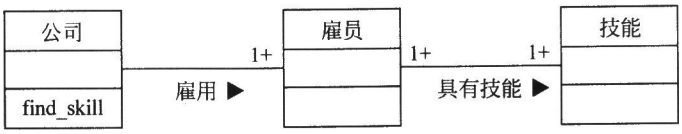


图11.11 公司、雇员及技能之间的关联链

假设某公司共有2 000名雇员，平均每名雇员会10种技能，则简单的嵌套查询将遍历雇员对象2 600次，针对每名雇员平均再遍历技能对象10次。如果全公司仅有5名雇员精通日语，则查询命中率仅有1／4 000。

提高访问效率的一种方法是使用哈希(Hash)表：“具有技能”这个关联不再利用无序表实现，而是改用哈希表实现。只要“会讲日语”是用惟一一个技能对象表示，这样改进后就会使查询次数由20 000次减少到2 000次。

但是，当仅有极少数对象满足查询条件时，查询命中率仍然很低。在这种情况下，更有效的提高查询效率的改进方法是，给那些需要经常查询的对象建立索引。例如，针对上述例子，可以增加一个额外的限定关联“精通语言”，用来联系公司与雇员这两类对象，如图

11.12 所示。利用适当的冗余关联，可以立即查到精通某种具体语言的雇员，而无须多余的访问。当然，索引也必然带来开销：占用内存空间，而且每当修改基关联时也必须相应地修改索引。因此，应该只给那些经常执行并且开销大、命中率低的查询建立索引。

2.调整查询次序

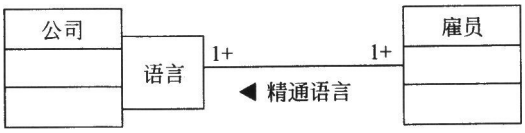


图 11.12 为雇员技能数据库建立索引

改进了对象模型的结构，从而优化了常用的遍历之后，接下来就应该优化算法了。优化算法的一个途径是尽量缩小查找范围。例如，假设用户在使用上述的雇员技能数据库的过程中，希望找出既会讲日语又会讲法语的所有雇员。如果某公司只有 5 位雇员会讲日语，会讲法语的雇员却有 200 人，则应该先查找会讲日语的雇员，然后再从这些会讲日语的雇员中查找同时又会讲法语的人。

3.保留派生属性

通过某种运算而从其他数据派生出来的数据，是一种冗余数据。通常把这类数据“存储”(或称为“隐藏”)在计算它的表达式中。如果希望避免重复计算复杂表达式所带来的开销，可以把这类冗余数据作为派生属性保存起来。

派生属性既可以在原有类中定义，也可以定义新类，并用新类的对象保存它们。每当修改了基本对象之后，所有依赖于它的、保存派生属性的对象也必须相应地修改。

11.11.3 调整继承关系

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

下面讨论与建立类继承有关的问题。

1.抽象与具体

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

对于某些类继承来说，这是一个持续不断的演化过程。

图 11.13 用一个人们在日常生活中熟悉的例子，说明上述从具体到抽象，再到具体的过程。

2.为提高继承程度而修改类定义

如果在一组相似的类中存在公共的属性和公共的行为，则可以把这些公共的属性和行为抽取出来放在一个共同的祖先类中，供其子类继承，如图 11.13(a)和(b)所示。在对现有类进行归纳的时候，要注意下述两点：(1)不能违背领域知识和常识；(2)应该确保现有类的协议(即同外部世界的接口)不变。

更常见的情况是，各个现有类中的属性和行为(操作)，虽然相似却并不完全相同，在这种情况下需要对类的定义稍加修改，才能定义一个基类供其子类从中继承需要的属性或行为。

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

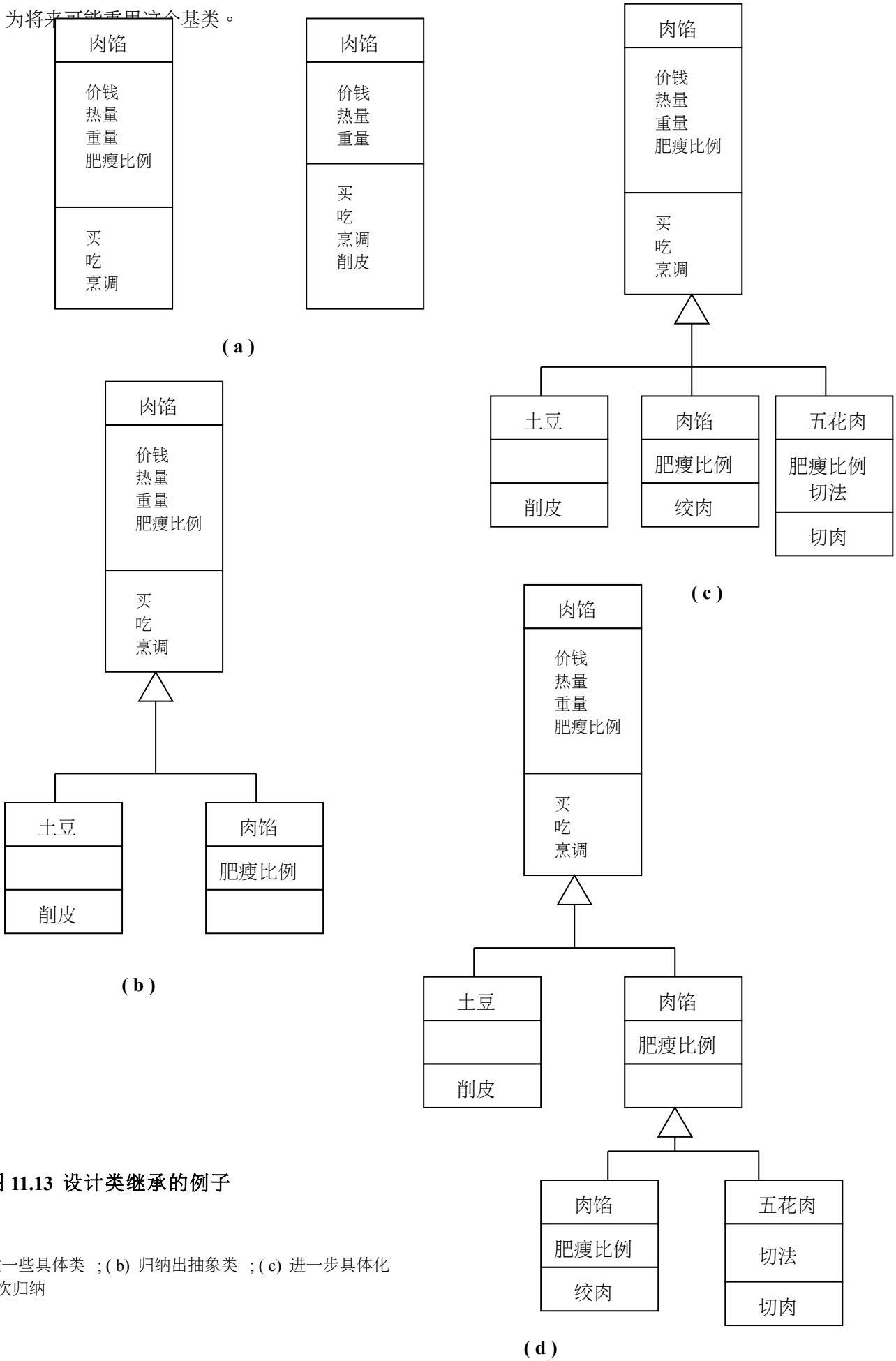


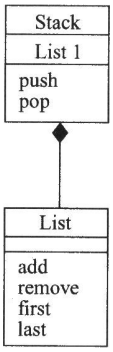
图 11.13 设计类继承的例子

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

3.利用委托实现行为共享

仅当存在真实的一般一特殊关系(即子类确实是父类的一种特殊形式)时，利用继承机制实现行为共享才是合理的。

有时程序员只想用继承作为实现操作共享的一种手段，并不打算确保基类和派生类具有相同的行为。在这种情况下，如果从基类继承的操作中包含了子类不应有的行为，则可能引起麻烦。例如，假设程序员正在实现一个 Stack(后进先出栈)类，类库中已经有一个 List(表)类。如果程序员从 List 类派生出 Stack 类，则如图 11.14(a)所示：把一个元素压入栈，等价于在表尾加入一个元素；把一个元素弹出栈，相当于从表尾移走一个元素。但是，与此同时，也继承了一些不需要的表操作。例如，从表头移走一个元素或在表头增加一个元素。万一用户错误地使用了这类操作，Stack 类将不能正常工作。



(a)　　(b)

图 11.14 用表实现栈的两种方法
(a)用继承实现；(b)用委托实现

如果你只想把继承作为实现操作共享的一种手段，则利用委托(即把一类对象作为另一类对象的属性，从而在两类对象间建立组合关系)也可以达到同样目的，而且这种方法更安全。使用委托机制时，只有有意义的操作才委托另一类对象实现，因此，不会发生不慎继承了无意义(甚至有害)操作的问题。

图 11.14(b)描绘了委托 List 类实现 Stack 类操作的方法。Stack 类的每个实例都包含一个私有的 List 类实例(或指向 List 类实例的指针)。Stack 对象的操作 push(压栈)，委托 List 类对象通过调用 last(定位到表尾)和 add(加入一个元素)操作实现，而 pop(出栈)操作则通过 List 的 last 和 remove(移走一个元素)操作实现。

11.12 小　结

面向对象设计就是用面向对象观点建立求解空间模型的过程。通过面向对象分析得出的

问题域模型为建立求解空间模型奠定了坚实基础。分析与设计本质上是一个多次反复迭代的过程，而面向对象分析与面向对象设计的界限尤其模糊。

优秀设计是使得目标系统在其整个生命周期中总开销最小的设计，为获得优秀的设计结果，应该遵循一些基本准则。本章结合面向对象方法学固有的特点讲述了面向对象设计准则，并介绍了一些有助于提高设计质量的启发式规则。

重用是提高软件生产率和目标系统质量的重要途径，它基本上始于设计。本章结合面向对象方法学的特点，对软件重用做了较全面的介绍，其中着重讲述了类构件重用技术。

用面向对象方法设计软件，原则上也是先进行总体设计(即系统设计)，然后再进行详细设计(对象设计)，当然，它们之间的界限非常模糊，事实上是一个多次反复迭代的过程。

大多数求解空间模型，在逻辑上由4大部分组成。本章分别讲述了问题域子系统、人机交互子系统、任务管理子系统和数据管理子系统的设计方法。此外还讲述了设计类中服务的方法及实现关联的策略。

通常应该在设计工作开始之前，对系统的各项质量指标的相对重要性做认真分析和仔细权衡，制定出恰当的系统目标。在设计过程中根据既定的系统目标，做必要的优化工作。

习 题 11

1.面向对象设计应该遵循哪些准则?简述每条准则的内容，并说明遵循这条准则的必要性。

2.简述有助于提高面向对象设计质量的每条主要启发规则的内容和必要性。

3.为什么说类构件是目前比较理想的可重用软构件?它有哪些重用方式?

4.试用面向对象方法，设计本书第2章中给出的定货系统的例子。

5.试用面向对象方法，设计本书习题2第2题中描述的储蓄系统。

6.试用面向对象方法，设计本书习题2第3题中描述的机票预订系统。

7.试用面向对象方法，设计本书习题2第4题中描述的患者监护系统。