面向对象技术

面向对象技术

面向对象的概念

开发模式（Paradigm）

- 开发模式又称为范型、范例、风范或模式(**Pattern**)。开发模式定义了
  - ◆特定问题和应用的开发过程中将遵循的步骤；
  - ◆确定将用于表示问题和解的那些成分的类型；
  - ◆利用这些成分表示与问题解决有关的抽象；
  - ◆直接得到问题的结构。

- 开发模式的选择影响到整个软件开发生存期。就是说，它支配了
  - ◆设计方法
  - ◆编码语言
  - ◆测试和检验技术

  的选择

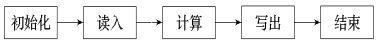
面向过程开发模式

- 面向过程开发模式产生过程的抽象。
- 这些抽象的基础是把软件视为处理流，并定义成由一系列步骤构成的算法。
- 每一步骤都是带有预定输入和特定输出的一个过程，把这些步骤串联在一起可产生合理的稳定的贯通于整个程序的控制流，最终产生一个简单的具有静态结构的体系结构。

面向过程开发模式的特点

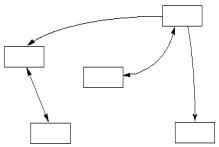
- 过程性开发模式侧重建立构成问题解决的处理流。
- 数据抽象、数据结构根据算法步骤的要求开发，它贯穿于过程，提供过程所要求操作的信息。
- 系统的状态是一组全局变量，这组全局变量保存状态的值，把它们从一个过程传送到另一个过程。

过程性系统

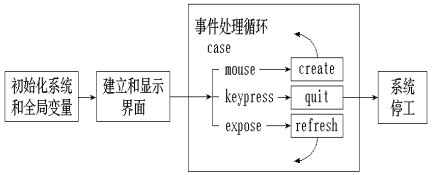


（a） 系统结构基于要执行的任务，改变一个可能需要改变其它所有的

面向对象的系统



（b） 系统结构基于对象间的交互，改变一个通常只具有局部影响



(1) Initialize system;
(2) Create and draw interface;
while QUIT not selected do
case

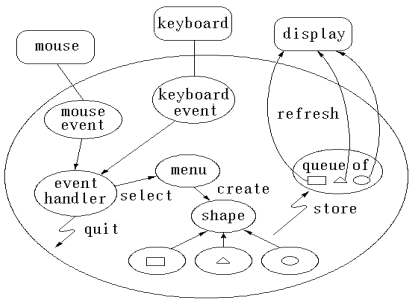

```
Mouse event:
    create shape structure;
    read mouse movements for data;
    store newly created shape on list
        of shape records;
KeyPress event:
    if key = 'q' then exit loop;
    else ignore;
Ecpose event:
    refresh display by drawing each
        shape structure;
(4) Shut down system;
```


面向对象开发模式

- 在面向过程开发模式中优先考虑的是过程抽象，在面向对象开发模式中优先考虑的是实体（问题论域的对象）。
- 在面向对象开发模式中，把标识和模型化问题论域中的主要实体做为系统开发的起点，主要考虑对象的行为而不是必须执行的一系列动作。

面向对象开发模式的特点

- 面向对象系统中的对象是数据抽象与过程抽象的综合。
- 系统的状态保存在各个数据抽象的所定义的数据存储中。
- 控制流包含在各个数据抽象中的操作内。
- 在面向对象体系结构。消息从一个对象传送到另一个对象。算法被分布到各种实体中。



其它流行的开发模式

- 目前流行多种开发模式，它们提供了许多方法，可进行系统分解。
  - 面向过程的；
  - 逻辑的；
  - 面向存取的；
  - 面向进程的；
  - 面向对象的；
  - 函数型的；
  - 说明性的。

- 每个开发模式都有它的支持者和用户；
- 每个开发模式都特别适合于某种类型的问题或子问题；
- 每一个开发模式都用不同的方式考虑问题；
- 每一个开发模式都使用不同的方法来分解问题；
- 每一个开发模式都导致不同种类的块、过程、产生规则。

混合开发模式

- 在大型系统的开发中，很难说哪种开发模式对整个问题的解决最好。
- 系统开发时，通常把大型问题分解成一组子问题。对于每个子问题可以采用适当的软件开发模式。
- 这种设计需要有某种实现语言或一组协同语言的支持。许多流行的功能不断增强的语言可支持不只一种设计开发模式。

- 一个智能数据分析系统的设计，可把它看做是 4 个子系统。系统有
- 一个数据库界面，可以使用面向存取的方法进行设计；
- 智能数据分析用逻辑性的开发模式设计；
- 一组分析算法是过程性的；
- 用户界面是用面向对象开发模式设计出来的。

什么是面向对象

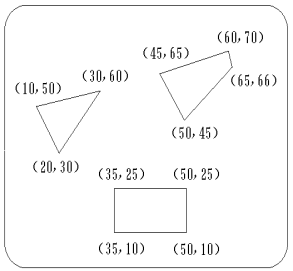
- Coad和Yourdon给出了一个定义：“面向对象=对象+类+继承+通信”。
- 如果一个软件系统是使用这样 4 个概念设计和实现的，则我们认为这个软件系统是面向对象的。
- 一个面向对象的程序的每一成份应是对象，计算是通过新的对象的建立和对象之间的通信来执行的。

对 象 （ object）

- 对象是面向对象开发模式的基本成份。
- 每个对象可用它本身的一组属性和它可以执行的一组操作来定义。
- 属性一般只能通过执行对象的操作来改变。
- 操作又称为方法或服务，它描述了对象执行的功能，若通过消息传递，还可以为其它对象使用。

消 息 （ Message）

- 消息是一个对象与另一个对象的通信单元，是要求某个对象执行类中定义的某个操作的规格说明。发送给一个对象的消息定义了一个方法名和一个参数表（可能是空的），并指定某一个对象。
- 一个对象接收的消息则调用消息中指定的方法，并将形式参数与参数表中相应的值结合起来。



(a) 在计算机屏幕上的三个多边形

triangle	quadrilateral1	quadrilateral2
(10,50) (30,60) (20,30)	(35,10) (50,10) (35,25) (50,25)	(45,65) (50,45) (65,66) (60,70)
draw move(Δx, Δy) contains?(aPoint)	draw move(Δx, Δy) contains?(aPoint)	draw move(Δx, Δy) contains?(aPoint)

(b) 表示多边形的三个对象

类(class)

- 类是一组具有相同数据结构和相同操作的对象的集合。
- 类的定义包括一组数据属性和在数据上的一组合法操作。
- 类定义可以视为一个具有类似特性与共同行为的对象的模板，可用来产生对象。

- 在一个类中，每个对象都是类的实例 **(Instance)**，它们都可使用类中提供的函数。
- 对象的状态则包含在它的实例变量，即实例的属性中。

类 ← 两个四边形对象

Quadrilateral	quadrilateral1	quadrilateral2
point1 point3 point2 point4	(35,10) (50,10) (35,25) (50,25)	(45,65) (50,45) (65,66) (60,70)
draw move(Δx, Δy) contains?(aPoint)	draw move(Δx, Δy) contains?(aPoint)	draw move(Δx, Δy) contains?(aPoint)

表示多边形的三个对象

- **Quadrilateral**类的每个对象有同样的一组实例变量和方法。
- 就这个意义来讲，类**Quadrilateral**给我们提供了一个模板，表示了所有四边形对象。
- 类常常可看做是一个抽象数据类型**(ADT)**的实现。但更合适的是把类看做是某种概念的模型。

- 类的实现常常使用其它类的实例，它们提供了该类所需要的服务。
- 这些实例应当受到保护不被其它对象存取，包括同一个类的其它实例。
- 在四边形的例子中，定义4个**point**类的实例作为**Quadrilateral**类的实例的4个顶点。这些**point**对象不能被其它对象存取。

继承 (Inheritance)

- 继承是使用已存在的定义做为基础建立新定义的技术。
- 新类的定义可以是既存类所声明的数据和新类所增加的声明的组合。新类复用既存的定义，而不要求修改既存类。
- 既存类可当做基类来引用，则新类相应地可当做派生类来引用。

Polygon
referencePoint Vertices
draw move(Δx, Δy) contains?(aPoint)

(a) Polygon类

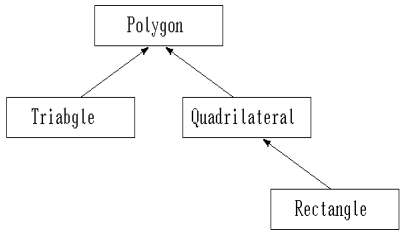
Quadrilateral
referencePoint *Vertices*
draw *move(Δx, Δy)* *contains?(aPoint)*

(b) Polygon类的子类 Quadrilateral

- 使用继承设计一个新类，可以视为描述一个新的对象集，它是既存类所描述对象集的子集合。
- 这个新的子集合可以认为是既存类的一个特殊化。**Quadrilateral**类是**Polygon**类的特殊化。**Quadrilateral**是限制为四条边的多边形。我们还可以进一步地把类**Quadrilateral**特殊化为**Rectangle** 。

- 类**Quadrilateral**的界面可以等同于类**Polygon**的界面， 而**Rectangle**类的界面又与**Quadrilateral**类的界面相同。
- 新类的界面还可以被看做是既存类界面的一个扩充界面。例如，从一个既存的车辆类派生的四轮驱动车类可能不仅是车辆类子集合定义的特殊化，而且还可能在新类的界面中引入新的能力。

类的继承层次

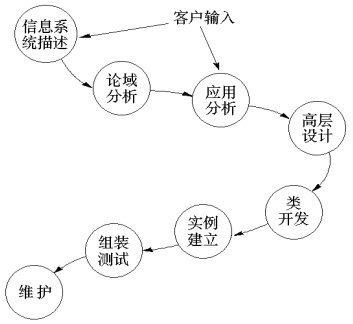


- 在类的继承层次中，**Quadrilateral**的实际参数可以替换**Polygon**的形式参数。
- 类**Quadrilateral**的界面与类**Polygon**的界面是相容的
- **Quadrilateral**的界面可响应**Polygon**界面的所有消息。

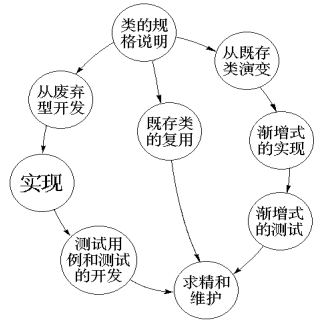
面向对象方法的开发过程

- 面向对象方法改进了在生存期各个阶段之间的接口，因为在生存期各个阶段所开发出来的“部件”都是类。
- 在面向对象生存期的各个阶段对各个类的信息进行细化，类成为分析、设计和实现的基本单元。

用生存期



类生存期



复用（Reusable）

- 在软件开发中，复用扮演了重要角色。软件部件应当独立于当初开发它们的应用而存在。
- 部件的开发瞄准某些局部的设计和实现，它们能够帮助当前问题的解决，但为了在以后的项目中使用，它们还应当足够通用。

- 类就是一个希望能够复用的单元，因此，提出了一个“类生存期”。
- 类生存期是与应用生存期是交叉的。即就是说，类的标识是应用生存期的一个阶段，但类生存期的步骤独立于任一特殊应用的开发。
- 类的开发应能完整地描述一个基本实体。而不仅仅考虑当前正在开发的系统。

类的定义

- 一旦标识了一个类，就给出了它的规格说明，其中包括类的实例可执行的操作和它们的数据表示。
- 对每一个，无论是在哪一个阶段标识的类都是如此。
- 对于那些使应用与数据库交互的类来说，其规格说明应当包括查找数据库和向数据库加入数据的行为。

- 类的规格说明定义了施加于对象的数据存储上的一组操作。
- 这组操作应工作在封装在对象内部的数据存储上，或返回关于对象状态的信息。
- 操作的名字应能反映这个操作本身的含义。

类的设计与实现

- 类的规格说明可指导对存放既存类的软件库进行查找，这些既存类可用来提供为当前应用所需要的功能。
- 三个可能的利用既存类的方向。开发过程可能依赖于这种查找的结果。
  - 既存类的复用
  - 从既存类进行演化
  - 从废弃型进行开发

实现

- 通过变量的声明、操作界面的实现及支持界面操作的函数的实现，可实现一个类的预期行为和状态。
- 实现是与语言有关的。一个好的面向对象语言应当分离共有界面与其内部实现。
- 采取必要措施分别编译界面和内部表示。

测试

- 单个的类为测试提供了自然的单元。
- 如果类的定义提供的界面比较狭窄，那么穷举测试就有可能实现。
- 类的测试在最抽象的层次开始，沿继承关系继续向下进行。
- 已经测试过的部分不需要从新测试。
- 重点放在对新类的测试和组装测试。

求精和维护

- 这是一个在软件生存期中最花费时间的部分。
- 传统的维护活动是针对应用的，而求精过程是针对类，针对把类集成在一起的结构。
- 我们可以标识抽象的抽象，使得继承结构通过一般化增加新的层次，即在既存的根类之上增加新的层次。

概念的封装和实现的隐蔽

- 概念的封装和实现的隐蔽，使得类具有更大的独立性。在任一时刻都可以在类的界面上增加新的操作，并能够修改实现，以改进性能，或引入原来设计中没有的新服务。
- 为便于类的调整，应尽量做到定义与实现分离。对一个类的共有界面的实现所做的多次修改不应影响利用它的那些类。

面向对象分析与模型化

- 面向对象分析是软件开发过程中的问题定义阶段。
- 这一阶段最后得到的是对问题论域的清晰、精确的定义。
- 分析阶段包括两个步骤：论域分析和应用分析。
- 它们都要标识问题论域中的抽象。

- 在分析中，需要
  - 找到特定对象
  - 基于对象的公共特性组合它们
  - 标识出对这个问题的抽象
- 在分析阶段中要标识
  - 抽象之间的关系
- 这些关系在应用系统中常常用对象之间的消息来表示，叫做消息连接。

- 在一个面向对象的应用中的控制流由两部分构成：
  - 每个单独操作内部的控制流
  - 对象之间的消息模式
- 面向对象分析过程分两阶段：
  - 论域分析
  - 应用分析

论域分析

- 论域分析开发问题论域的模型
- 考察问题论域内的一个较宽的范围，分析覆盖的范围应比直接要解决的问题更多。
- 建立大致的系统实现环境

应用分析

- 应用分析则根据特定应用的需求进行论域分析。
- 应用(或系统)分析细化在论域分析阶段所开发出来的信息，把注意力集中于当前要解决的问题。

语义数据模型

- 语义数据模型是一种特别适用的建立构成问题论域模型的技术。
- 它基于实体—关系模型，并对这类模型进行了扩充和一般化。语义数据模型可以表达问题论域的内涵，还可以表示复杂的对象和对象之间的关系。

语义数据模型与面向对象方法

语义数据模型	主要特征	面向对象分析与设计
外部模型	数据的用户视图	与应用有关的类的定义
概念模型	实体及实体之间关系的内涵	类与类之间的应用级关系
物理模型	数据的物理表示	类的实现

- 外部模型层反映应用的外部现实世界的视图，它体现了用户对问题的理解。
- 概念模型层考虑在外部模型层所标识的实体之间的关系。这些关系都是可直接观察到的交互关系。
- 内部模型层考虑实体的物理模型，就是我们生存期中的类设计阶段。

物理模型包括的属性

- 物理模型包括两类属性：
  - 方法：对实体的行为模型化
  - 数据：对实体的状态模型化
- 在模型中方法分为两种：
  - 共有的
  - 私有的
- 在分析阶段所标识的属性是描述性的，

在语义数据模型中的关系

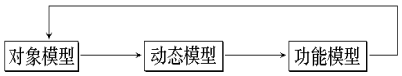
- **一般化和特殊化**关系可用来按层次渐增式地定义抽象(类)。
- 低层抽象是高层抽象的特殊化。
- 这种抽象层次构成论域模型的基础。
- 例如，小汽车，卡车和公共汽车可以归于更一般的概念汽车中。从这个较一般化的概念汽车可以定义其它较特殊的抽象：赛车，面包车和牵引车。

- 聚合关系支持使用几个其它较小和较简单的抽象来开发一个抽象。
- 它相应于一个记录中成份的声明。
- 例如，一个航班可以有6个属性：飞机编号、机组编号、离开和到达地点、起飞和降落时间。因此，航班类有一个聚合关系，它利用了表示飞机、人员、空间的类，并增加了时间窗口。

- **关联关系**指定一个抽象做为其它抽象实例的包容(**container**)。
- 关联和聚合之间的差别在于组合实体的意图。聚合指定一组实体中的某些元素做为一个类的组成，而关联是指群集的相互有关联的实体群。
- 例如，一个部门包含有人，这样一个部门关联了所有被分配给这个部门的人。，这些人在系统其它地方也可能出现。

对象模型化技术OMT

- 对象模型化技术把分析时收集的信息构造在三类模型中，即对象模型、功能模型和动态模型。

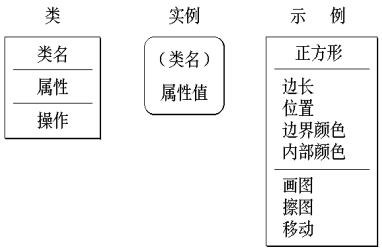


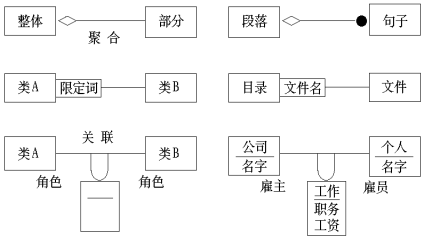
- 这个模型化的过程是一个迭代过程。

对象模型

- 是三个模型中最关键的一个模型，它的作用是描述系统的静态结构，包括构成系统的类和对象，它们的属性和操作，及它们之间的关系。
- 在**OMT**中，类与类之间的关系叫做关联。关联代表一组存在于两个或多个对象之间的、具有相同结构和含义的具体连接。关联可以是物理的，也可以是逻辑的。

- **聚合，代表整体与部分的关系，这是一种特殊形式的关联。**
- **限定，用以对关联的含义做某种约束。**
- **角色，用来说明关联的一端。由于多数关联具有两个端点，因而涉及到两个角色。**
- **附加的说明对象之间的连接的连接属性。**





一般化关系

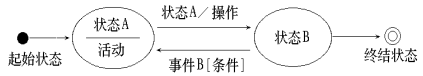
- 也称为继承性。一般化关系包含基类和几个派生类。
- 基类表示了一个较为一般、普遍的概念
- 每个派生类则是它的某个特殊形态
- 派生类除了自然地继承基类所具有的属性和操作外，还具有反映自身特点的属性和操作。

动态模型

- 要想对一个系统了解得比较清楚，还应当考察在任何时刻对象及其关系的改变。
- 系统的这些涉及时序和改变状况用动态模型来描述。
- 动态模型着重于系统的控制逻辑。
- 它包括两个图，一是**状态图**，一是**事件追踪图**。

状态图

- 状态图是一个状态和事件的网络，侧重于描述每一类对象的动态行为。
- 在状态图中，状态是对某一时刻中属性特征的概括。而状态迁移表示这一类对象在何时对系统内外发生的哪些事件做出何种响应。



- 操作是一个伴随状态迁移的瞬时发生的行为，与触发事件一起表示在有关的状态迁移之上。
- 活动则是发生在某个状态中的行为，往往需要一定的时间来完成，因此与状态名一起出现在有关的状态之中。

- 动态模型由多个状态图组成。
- 对于每一个具有重要动态行为的类都有一个状态图，从而表明所有系统活动的模式。
- 各个状态图并发地执行，并可以独立地改变状态。
- 各种类的状态图可以通过共享事件组合到一个动态模型中。

事件

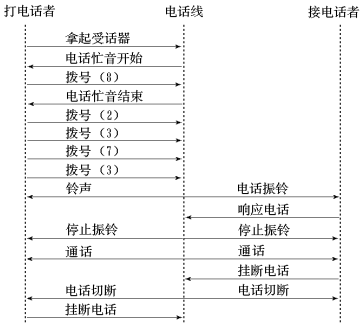
- 一个事件发生在某一时刻
- 每个事件都是单独发生的
- 我们建立事件类，并给每个事件一个名字，以指明共同结构和行为。
- 事件从一个对象向另一个对象传送信息。

- 有些事件类可能传送的是简单的信号“要发生某件事”，而有些事件类则可能传送的是数据值。由事件传送的数据值叫做属性。
  - 列车出发(线路、班次、城市)
  - 撳下鼠标按钮(按钮、位置)
  - 拿起电话受话器
  - 数字拨号(数字)

事件追踪图

- 事件追踪图侧重于说明发生于系统执行过程中的一个特定“场景”。
- 场景也叫做脚本，是完成系统某个功能的一个事件序列。
- 场景通常起始于一个系统外部的输入事件，结束于一个系统外部的输出事件，它可以包括发生在这个期间的系统所有的内部事件。

打电话者拿起电话受话器
电话忙音开始
打电话者拨数字(8)
电话忙音结束
打电话者拨数字(2) .....
打电话者拨数字(3)
接电话者的电话开始振铃
铃声在打电话者的电话上传出
接电话者回答
接电话者的电话停止振铃
铃声在打电话者的电话中消失
通电话 .....



状态图与事件追踪图的关系

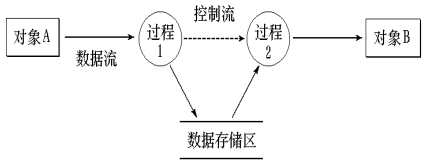
- 状态图叙述一个对象的个体行为，事件追踪图则给出多个对象所表现出来的集体行为。它们从不同侧面来说明同一系统的行为。
- 例如，一个事件追踪图指出某一对象在接受一个事件之后发出另一事件，同一行为在此对象的状态图中也应当有所表示。

功能模型

- 功能模型表明，**通过计算，从输入数据能得到什么样的输出数据，不考虑参加计算的数据按什么时序执行。**
- 功能模型由**多个数据流图组成，**它们指明从外部输入，通过操作和内部存储，直到外部输出，这整个的数据流情况。

- 功能模型中所有的数据流图往往形成一个层次结构。
- 在这个层次结构中，一个数据流图中的过程可以由下一层的数据流图做进一步的说明。
- 一般来讲，高层的过程相应于作用在聚合对象上的操作，而低层的过程则代表作用于一个简单对象上的操作。

- 数据流图中允许加入控制流，但这样做将与动态模型重复，不提倡夹带控制流。



基于三个模型的分析过程

- 功能模型着重于系统内部数据的传送和处理。
  - 功能模型定义“做什么”
  - 动态模型定义“何时做”
  - 对象模型定义“对谁做”。

Coad与Yourdon面向对象分析

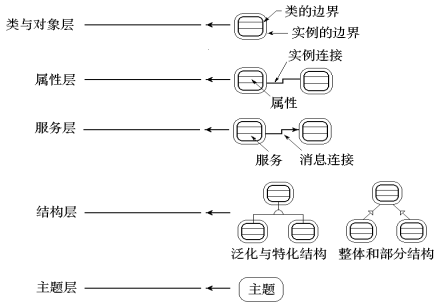
- OOA有两个任务
  - 形式地说明我们所面对的应用问题，最终成为软件系统基本构成的对象，还有系统所必须遵从的，由应用环境所决定的规则和约束。
  - 明确地规定构成系统的对象如何协同合作，完成指定的功能。

OOA概念模型

- 通过**OOA**建立的系统模型是以概念为中心的，因此称为概念模型。
- 这样的模型由一组相关的类组成。
- **软件规格说明就是基于这样的概念模型形成的，**以模型描述为基本部分，再加上接口要求、性能限制等其它方面的要求说明。

构造OOA概念模型的层次

- 构造和评审**OOA**概念模型的顺序和由五个层次组成。
- 这五个层次是分析过程中的层次。
- 每个层次的工作都为系统的规格说明增加了一个组成部分。
- 这五个层次是：**类与对象、属性、服务、结构和主题。**



识别类和对象

- 面向对象分析的第一个层次主要是识别类和对象。
- 类和对象是对与应用有关的概念的抽象。不仅是说明应用问题的重要手段，同时也是构成软件系统的基本元素。
- 这一层工作是整个分析模型的基础。

选择类和对象的原则：

- 目标系统必须记住类和对象的某些事情
- 类和对象应当提供某些服务或处理
- 多属性
- 所有属性对于类中所有实例都应有意义
- 对象类应表示问题论域的需求

基于语言的信息分析

- 在发现对象过程中，可以使用一种十分有用的工具，即LIA(基于语言的信息分析)。
- LIA的目的是**标识出问题论域的所有概念及这些概念之间的关系**。
  - **短语频率分析(PFA)**
  - **矩阵分析(MA)。**

资源库

- **资源库包括相关文件、模型、软件、人员以及包含问题论域或系统知识的其它资源。如果问题论域有参考材料(教材、惯例、操作过程等)，这些材料必须包含在资源库中。**
- **资源库包括其它一些信息：访问记录、形式的或非形式的系统规格说明、 已有的或相关系统的用户手册、日志(如系统变更请求或问题报告)。**

- **LIA**技术通常只应用于资源库的某个子集。这取决于分析员想把什么样的视图用于问题论域或应用系统。
- 通常，根据与问题论域有关的资源建立起来的结果与根据目标系统的规格说明有关的资源建立起来的结果会有所不同。

短语频率分析 PFA

- 短语频率分析搜索选定的问题陈述，标识可以表示问题论域概念的术语。
- PFA清单的建立基本上是一个客观的过程。但可能大多数标识出来的概念是与目标系统无关的。
- PFA的优点就在于能广泛地标识问题论域的概念集合，并对它们进行评估，判定哪些与目标软件无关。

- **PFA**将名词和动词标识为候选实体和属性。但由于名词／动词的标识是非常主观的，可根据什么是名词或动词，以及根据分析员的理解，才能确定哪些名词或动词是要找的。
- PFA是标识概念而不是标识语法单元。
- 所建立的PFA清单并不受建立清单的人的很大影响。

accepted subscription	board of advisors	correspondence address
accompanied payment	brown wrapper, plain	cost, shipping
accounting department	bulk shipment	country
actual expiration date	bureau, subscription service	country, foreign
additional subscription	check payment	credit card order
address, corporate	commision, subscription	credit card payment
address, correspondence	service	current author
address, home	company subscription	customer
address, subscription	complimentary subscription	database
advisors, board of	complimentary subscription	date, actual expiration
agency, subscription service	query	date, expiration
agreement,	complimentary subscription	date, expired
distributor-publisher	review	daleted, complimentary
annual subscription price	complimentary subscription	subscription
article	deleted	department, accounting
associated site	constituent copies	department, corporate
author	continued subscription	direct subscription
author, contributing	contributing author	discount, subscription
		…
		…

- 对于任一有用的应用论域资源，PFA可能会产生一个长长的概念的清单。
- 许多被标识出的概念因与目标软件无关而被丢弃，但其它的则会成为OOA模型的成份，包括对象。
- 将PFA清单转换为OOA／OOD工作表格。列出对各个概念的理解和选择，这将有助于对象的选出。

Small Bytes 订阅系统　OOA／OOD 工作表格

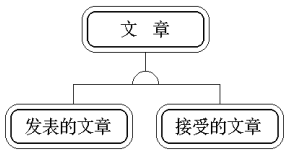
条　目	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	注　释
ACCEPTION SUBSCRI PTION				×						SUBSCRIPTION 的属性
ACCOMPANIED PA YMENT	×									对payment 的不同类型不加以区分
ACCOUNTING DEPAR TMENT	×									已超出SBSS 的应用论域
ACTUAL EXPIRATION DATE				×						SUBSCRIPTION 的属性
ADDITIONAL SU BSCRIPTION			×	×						SUBSCRIPTION 的可能属性，或可能是派生类型-基类型结构
ARTICLE		×								
ASSOCIATED SITE				×						SITE 的属性
AUTHOR		×								
.....										

(0) 不适用，可能无关，超出指定系统的环境
(1) 可能的对象－类
(2) 可能是派生类型－基类型结构的一部分，包括泛化－特化结构和整体－部分结构
(3) 可能描述对象－类的属性或实例关系
(4) 可能描述对象的服务
(5) 与实现相关，可能是属于问题论域部分的条目
(6) 可能是属于人机交互部分的条目
(7) 可能是属于任务管理部分的条目
(8) 可能是属于数据管理部分的条目

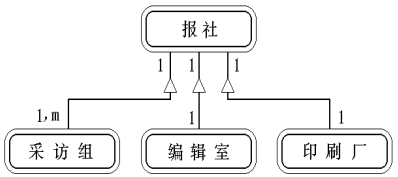
标识结构

- 面向对象分析的下一步工作是标识结构。典型的结构有两种：
  - **一般化-特殊化结构（Gen-Spec结构）**
  - **整体-部分结构（Whole-Part结构）**

一般化-特殊化结构



整体-部分结构



- 以特殊化的视点来看，一个**Gen-Spec**结构可以看作是**“is a”**或**“is a kind of”**结构。例如，

  a Truck Vehicle is a Vehicle

  a Truck Vehicle is a kind of Vehicle
- 在**Gen-Spec**结构中，使用继承将较一般化的属性和服务放在一般化的类和对象中。

- 从整体的视点来看，一个**Whole-Part**结构可看作一个**“has a”**或**“is a part of”**结构。例如，

  Vehicle has a Engine

  Engine is a part of Vehicle
- 其中，Vehicle是整体对象，Engine是局部对象。

标识Gen-Spec结构的方法和策略

- **对于每一个类和对象，将它看作是一个一般化的类，对它的所有特殊情况，考虑以下问题：**
  - **它是否在问题论域中？**
  - **它是否在系统的职责内？**
  - **继承性是否存在？**
  - **它是否能够符合选择类和对象的标准？**

- **同样地，把每一个类和对象置于特殊化对象的地位，对于它所有的一般化情形，考虑上述4个问题。**
- 检查以前在相同或类似问题论域中面向对象分析的结果，看是否有可直接复用的**Gen-Spec结构。**
- **如果一个一般化对象可能有多个特殊化对象，应当先考虑最简单的特殊化对象和最复杂的特殊化对象，然后再考虑中间其他的特殊化对象。**

标识Whole-Part结构的方法和策略

- 应当寻找什么
  - 总体-部分（**Assembly-Parts**）关联，如**飞机**-发动机之间的关系。
  - 包容-内含（**Container-Content**）关联，如飞机-飞行员之间的关系。
  - 收集-成员（**Collection-Members**）关联，如机构-职员之间的关系。

- **将每一个类看作是一个Whole类，对它的所有可能Parts情况，考虑以下问题：**
  - **它是否在问题论域中？**
  - **它是否在系统的职责内？**
  - **它是否代表一个以上的状态值？**
  - **若不是，是否将它变为Whole中的一个属性？**
  - **它是否提供问题论域中有用的抽象？**

- **同样地，把每一个类置于Part 的地位，对于它所有的Whole情形，考虑上述5个问题。**
- 检查以前在相同或类似问题论域中面向对象分析的结果，看是否有可直接复用的**Whole-Parts结构。**

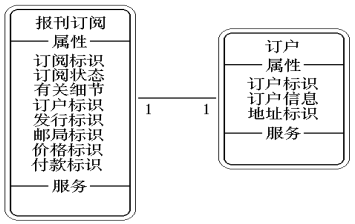
标识属性

- 下一个层次称为属性层，对前面已识别的类和对象做进一步的说明。在这里，对象所保存的信息称为它的属性。
- 类的属性所描述的是状态信息，每个实例的属性值表达了该实例的状态值。

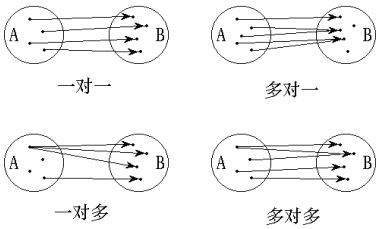
标识属性的方法和策略

- **找出属性**
- **将属性安放到适当的位置**
- **找出实例连接**
- **检查特殊情况**
- **描述属性**
- **考虑取值范围、极限值、缺省值、建立和存取权限、精确度、是否会受到其他属性值等。**

属性层



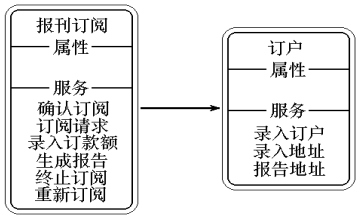
实例连接关系的标识



定义服务

- 对象收到消息后所能执行的操作称为它可提供的服务。
- 对每个对象和结构的增加、修改、删除、选择等服务有时是隐含的，在图中不标出，但在存储类和对象有关信息的对象库中有定义。
- 其它服务则必须显式地在图中画出。

服务层



定义服务的方法和策略

- 找出每一个对象的所有状态，在各种状态需要做的工作。利用状态迁移图；
- 找出必要的操作。
- 建立消息连接。
- 描述服务：利用状态转换图、脚本和事件追踪图，描述服务的功能。

消息连接的标识

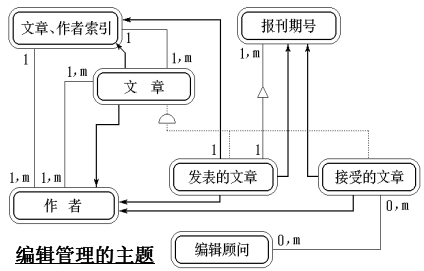
- **两个对象之间可能存在着由于通信需要而形成的关系，这称为消息连接。**
- **消息连接表示从一个对象发送消息到另一个对象，由那个对象完成某些处理。它们在图中用箭头表示，方向从发消息的对象指向收消息的对象。**

找出消息连接的方法及策略

- 对于每一个对象，执行：
  - 查询该对象需要哪些对象的服务，从该对象画一箭头到哪个对象；
  - 查询哪个对象需要该对象的服务，从那个对象画一箭头到该对象；
  - 循消息连接找到下一个对象，重复以上步骤。

识别主题

- 主题可以看成是高层的模块或子系统。
- 对于面向对象分析模型，主题表示此模型的整体框架。可以是一个层次结构。
- 通过对主题的识别，可以让人们能够比较清晰地了解大而复杂的模型。



编辑管理的主题

识别主题

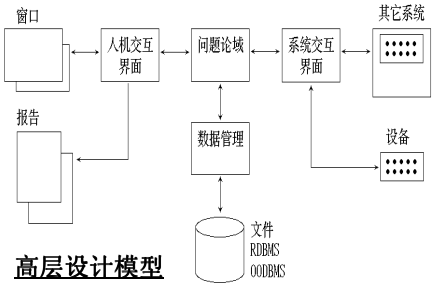
- 将每一种结构（包括整体-部分结构、和一般化-特殊化结构）中最上层的类提升成为主题；
- 将各不属于任何结构的类提升主题；
- 检查在相同或类似的问题论域中以前做面向对象分析的结果，看是否有可直接复用的主题。

面向对象设计（OOD）

- 面向对象设计继续做面向对象分析阶段的工作，建立软件的结构。
- 主要工作分为两个阶段：
  - 高层设计
  - 类设计

高层设计

- 高层设计阶段开发系统的结构，即构造应用软件的总体模型。
- 高层设计阶段标识在计算机环境中进行问题解决工作所需要的概念，并增加了一批需要的类。
- 这些类包括那些可使应用软件与系统的外部世界交互的类。
- 此阶段的输出是适合应用软件要求的类、类间的关系、应用的子系统视图规格说明。



高层设计模型

高层设计的特点

- 高层设计可以表征为标识和定义模块的过程。
- 模块可以是一个单个的类，也可以是由一些类组合成的子系统。
- 定义过程是职责驱动的。
- 类接口的协议如同“合同”：需方提出的请求必须列在协议表中，供方则必须提供所有协议的服务。

高层设计应遵循的原则

- 应使得在子系统的各个高层部件之间的通信量达到最小；
- 子系统应当把那些成组的类打包，形成高度的内聚；
- 逻辑功能分组，提供一个一个单元，识别并定位问题事件；

类设计

- 类与具有概念封装的子系统十分类似。
- 每个子系统都可以被当做一个类来实现，这个类聚集它的部件，提供了一组操作。
- 类和子系统的结构是正交的，一个单个类的实例可能是不止一个子系统的一部分。

- 高层设计和类设计这两个阶段是相对封闭的。
- 应用软件中的每一个事物都是一个对象，包括应用软件自身在内！
- 两个阶段是连接的。
- 应用软件的设计是大类的设计，这种类设计考察应用软件所期望的每一个行为，并利用这些行为形成应用类的界面。

Coad 与 Yourdon 高层设计方法

- **Coad** 与 **Yourdon** 在设计阶段中继续采用分析阶段中提到的五个层次。
- 在设计阶段中，这五个层次用于建立系统的四个组成成份。
  - 问题论域部分
  - 人机交互部分
  - 任务管理部分
  - 数据管理部分

问题论域部分

- 问题论域部分包括与应用问题直接有关的所有类和对象。
- 识别和定义这些类和对象的工作在**OOA**中已经开始，在**OOA**阶段得到的有关应用的概念模型描述了我们要解决的问题。
- 在**OOD**阶段，应当继续**OOA**阶段的工作，对在**OOA**中得到的结果进行改进和增补。

问题论域部分的设计

- **在OOA阶段得到的概念模型描述了要解决的问题**
- **在OOD阶段，继续OOA阶段的工作，对在OOA中得到的结果进行改进和增补。**
- **对OOA模型中的某些类与对象、结构、属性、操作进行组合与分解。**
- **要考虑对时间与空间的折衷、内存管理、开发人员的变更、以及类的调整等。**

1. 复用设计

- **根据问题解决的需要，把从类库或其它来源得到的既存类增加到问题解决方案中去。**
- **标明既存类中不需要的属性和操作，**
- **增加从既存类到应用类之间的一般化-特殊化的关系。**
- **把应用类中因继承既存类而成为多余的属性和操作标出。**
- **修改应用类的结构和连接。**

2. 把问题论域相关的类关联起来

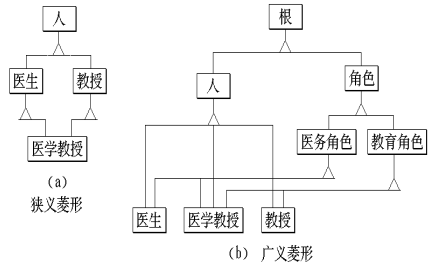
- **在设计时，从类库中引进一个根类，做为包容类，把所有与问题论域有关的类关联到一起，建立类的层次。**
- **把同一问题论域的一些类集合起来，存于类库中。**

3. 加入一般化类以建立类间协议

- **有时，某些特殊类要求一组类似的服务。**
- **此时，应加入一个一般化的类，定义为所有这些特殊类共用的一组服务名，这些服务都是虚函数。**
- **在特殊类中定义其实现。**

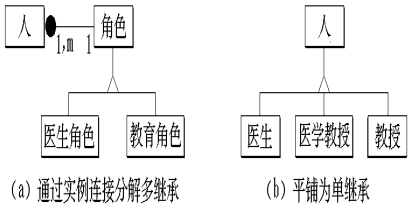
4. 调整继承支持级别

- **在OOA阶段建立的对象模型中可能包括有多继承关系，但实现时使用的程序设计语言可能只有单继承，甚至没有继承机制，这样就需对分析的结果进行修改。**
- **多继承模式有两种：**
  - **狭义的菱形**
  - **广义的菱形**



针对单继承语言的调整

- 把特殊类的对象看做是一个一般类对象所扮演的角色，通过实例连接把多继承的层次结构转换为单继承的层次结构。
- 把多继承的层次结构平铺，成为单继承的层次结构。在这种情况下，有些属性或操作在同层的特殊类中会重复出现。



(a) 通过实例连接分解多继承　　(b) 平铺为单继承

针对无继承语言的调整

- 当使用无继承的程序设计语言时，必须把具有继承关系的类层次结构平铺开来，成为一组类和对象。
- 一般可利用命名惯例，把这些类或对象关联起来。

5. 改进性能

- 提高执行效率和速度是系统设计的主要指标之一。有时，必须改变问题论域的结构以提高效率。
- 如果类之间经常需要传送大量消息，可合并相关的类以减少消息传递引起的速度损失。
- 增加某些属性到原来的类中，或增加低层的类，以保存暂时结果，避免每次都要重复计算造成速度损失。

6. 加入较低层的构件

- 在做面向对象分析时，分析员往往专注于较高层的类和对象，避免考虑太多较低层的实现细节。
- 在做面向对象设计时，设计师在找出高层的类和对象时，必须考虑到底需要用到哪些较低层的类和对象。

用户界面部分的设计

- 在 OOA 阶段给出了所需的属性和操作，
- 在设计阶段必须根据需求把交互细节加入到用户界面设计中，包括人机交互所必需的实际显示和输入。
- 用户界面部分设计主要由以下几个方面组成。

1. 用户分类

- 按技能层次分类：
  外行 / 初学者 / 熟练者 / 专家
- 按组织层次分类：
  行政人员 / 管理人员 / 专业技术人员 / 其它办事员
- 按职能分类：
  顾客 / 职员

2. 描述人及其任务的脚本

- 对以上定义的每一类用户，列出对以下问题做出的考虑：什么人、目的、特点、成功的关键因素、熟练程度以及任务脚本。
- 在OOATOOL™ 中有一个例子：
  - 什么人——分析员
  - 目的——要求一个工具来辅助分析工作 (摆脱繁重的画图和检查图的工作)。

- 特点——年龄：42岁；教育水平：大学；限制：不要微型打印，小于9个点的打印太小。
- 成功的关键因素——工具应当使分析工作顺利进行；工具不应与分析工作冲突；工具应能捕获假设和思想，能适时做出折衷；应能及时给出模型各个部分的文档，这与给出需求同等重要。
- 熟练程度——专家。

- 任务脚本——
  - 主脚本：
    - 识别“核心的”类和对象；
    - 识别“核心”结构；
    - 在发现了新的属性或操作时随时都可以加进模型中去。
  - 检验模型：
    - 打印模型及其全部文档。

3. 设计命令层

- 研究现行的人机交互活动的内容和准则：这些准则可以是非形式的，如“输入时眼睛不易疲劳”，也可以是正式规定的；
- 建立一个初始的命令层：可以有多种形式，如一系列 Menu Screens、或一个Menu Bar、或一系列Icons.
- 细化命令层：考虑以下几个问题。

- 排列命令层次。把使用最频繁的操作放在前面；按照用户工作步骤排列。
- 通过逐步分解，找到整体－局部模式，以帮助在命令层中对操作分块。
- 根据人们短期记忆的“7±2”或“每次记忆3块 / 每块3项”的特点，把深度尽量限制在三层之内。
- 减少操作步骤：把点取、拖动和键盘操作减到最少。

4. 设计详细的交互

- 用户界面设计有若干原则，包括：
  - 一致性：采用一致的术语、一致的步骤和一致的活动。
  - 操作步骤少：减少敲键和鼠标点取的次数，减少完成某件事所需的下拉菜单的距离。
  - 不要“哑播放”：每当用户等待系统完成一个活动时，要给出一些反馈信息。

- Undo：在操作出现错误时，要恢复或部分恢复原来的状态。
- 减少人脑的记忆负担：不应在一个窗口使用在另一个窗口中记忆或写下的信息；需要人按特定次序记忆的东西应当组织得容易记忆。
- 学习的时间和效果：提供联机的帮助信息。
- 趣味性：尽量采取图形界面，符合人类习惯.

5. 继续做原型

- 用户界面原型是用户界面设计的重要工作。人需要对提交的人机交互活动进行体验、实地操作，并精炼成一致的模式。
- 使用快速原型工具或应用构造器，对各种命令方式，如菜单、弹出、填充以及快捷命令，做出原型让用户使用，通过用户反馈、修改、演示的迭代，使界面越来越有效。

6. 设计 HIC (人机交互) 类

- 窗口需要进一步细化，通常包括：类窗口、条件窗口、检查窗口、文档窗口、画图窗口、过滤器窗口、模型控制窗口、运行策略窗口、模板窗口等。
- 设计HIC类，首先从组织窗口和部件的用户界面界面的设计开始。

- 每个类包括窗口的菜单条、下拉菜单、弹出菜单的定义。还要定义用于创建菜单、加亮选择项、引用相应的响应的操作。
- 每个类负责窗口的实际显示。所有有关物理对话的处理都封装在类的内部。必要时，还要增加在窗口中画图形图符的类、在窗口中选择项目的类、字体控制类、支持剪切和粘贴的类等。与机器有关的操作实现应隐蔽在这些类中。

7. 根据图形用户界面进行设计

- 图形用户界面区分为字型、坐标系统和事件。
  - 字型是字体、字号、样式和颜色的组合。
  - 坐标系统主要因素有原点(基准点)、显示分辨率、显示维数等。
  - 事件则是图形用户界面程序的核心，操作将对事件做出响应。

任务管理部分的设计

- 任务，是进程的别称，是执行一系列活动的一段程序。
- 当系统中有许多并发行为时，需要依照各个行为的协调和通信关系，划分各种任务，以简化并发行为的设计和编码。
- 任务管理主要包括任务的选择和调整，它的工作有以下几种。

- 识别事件驱动任务：一些负责与硬件设备通信的任务是事件驱动的，也就是说，这种任务可由事件来激发。
- 识别时钟驱动任务:以固定的时间间隔激发这种事件，以执行某些处理。某些人机界面、子系统、任务、处理机或与其它系统需要周期性的通信，因此时钟驱动任务应运而生。

- 识别优先任务和关键任务：根据处理的优先级别来安排各个任务。
- 识别协调者：当有三个或更多的任务时，应当增加一个追加任务，起协调者的作用。它的行为可以用状态转换矩阵来描述。
- 评审各个任务：对各任务进行评审，确保它能满足选择任务的工程标准—事件驱动？时钟驱动？优先级/关键任务？协调者？

定义各个任务

- 定义任务的工作主要包括：它是什么任务、如何协调工作及如何通信。
  (1) 它是什么任务——为任务命名，并简要说明这个任务。
  (2) 如何协调工作——定义各个任务如何协调工作。指出它是事件驱动还是时钟驱动。

(3) 如何通信——定义各个任务之间如何通信。任务从哪里取值，结果送往何方。
(4) 一个模版——任务的定义如下：
- Name (任务名)
- Description (描述)
- Priority (优先级)
- Servicesincluded (包含的操作)、
- Communication Via (经由谁通信)。

数据管理部分的设计

- 数据管理部分提供了在数据管理系统中存储和检索对象的基本结构，包括对永久性数据的访问和管理。
- 它分离了数据管理机构所关心的事项，包括文件、关系型DBMS或面向对象DBMS等。

数据管理方法

- 数据管理方法主要有3种：文件管理、关系数据库管理和面向对象库数据管理。
  - 文件管理——提供基本的文件处理能力。
  - 关系数据库管理系统——关系数据库管理系统使用若干表格来管理数据。

- 面向对象数据库管理系统——通常，面向对象的数据库管理系统以两种方法实现：一是扩充的RDBMS，二是扩充的面向对象程序设计语言。
- 扩充的RDBMS主要对RDBMS扩充了抽象数据类型和继承性，再加一些一般用途的操作创建和操纵类与对象。
- 扩充的OOPL在面向对象程序设计语言中嵌入了在数据库中长期管理存储对象的语法和功能。

程序设计语言的影响

- 详细的面向对象设计与语言有关。
- 一般地，所有的语言都可以完成面向对象实现，但某些语言能够提供更丰富的语法，能够显式地描绘在面向对象分析和面向对象设计过程中所使用的表示法。

1. 面向对象设计与过程型语言

- 过程型语言只直接支持过程抽象
- 可以增加数据抽象及封装(如利用结构化设计的信息隐蔽模块)
- 无法明确地表示继承性。也无法明确支持整体与部分、类与成员、对象与属性等关系。
- 具有面向对象特性的过程型语言可以成为一种实用的且可行的语言。

2. 面向对象设计与基于对象的语言

- 基于对象的语言，也叫做面向软件包的语言，如**Ada**等
- 能够直接支持过程抽象、数据抽象、封装和对象与属性关系
- 它无法表示继承性，也无法表示类与成员、整体与部分的关系。
- 基于对象语言的面向对象设计代表一种可行的开发方法。

3. 面向对象设计与面向对象的程序设计语言

- 面向对象的程序设计语言，包括**C++**、**Smalltalk**、**Objective-C**、**Actor**、**Eiffel**等，都直接支持过程抽象、数据抽象、封装、继承、以及对象与属性、类与成员关系。
- 它们不明确地支持整体与部分关系，但可以方便地表示组装对象。

- 因此，从面向对象分析，到面向对象设计，再到面向对象程序设计语言是一种与表示法十分一致的策略。

4. 面向对象设计与面向对象数据库语言(OO-DBL)

- 面向对象数据库管理系统(**OO-DBMS**)及其语言(**OO-DBL**)，是面向对象程序设计语言(**OOPL**)与数据管理能力的组合。**OO-DBMS**有四种不同的体系结构：

- 大属性——扩充关系型**DBMS**，使容纳大属性，如一个文档。例如，Informix公司的面向对象的产品。
- 松散耦合——一个**OOPL**与大量的**DBMS**组合在一起。
- 紧密耦合——一个**OOPL**与某个专用的**DBMS**集成为一个系统。
- 扩充关系型——扩充关系型**DBMS**，可容纳“过程”之类的属性。

类的设计

- 应用分析过程包括了对问题论域所需的类的模型化
- 但在最终实现应用时不只有这些类，还需要追加一些类
- 在类设计的过程中应当做这些工作。

类设计的目标

- 单一概念的模型
  - 使用多个类来表示一个“概念”。
  - 常常把一个概念进行分解，用一组类来表示这个概念。
  - 也可以只用一个单个类来表示一个概念。
  - 在类的文档中应对类的用途做出清楚的标识和精确的陈述，类的共有界面应当使用操作的特征、先决条件和后置条件加以定义。

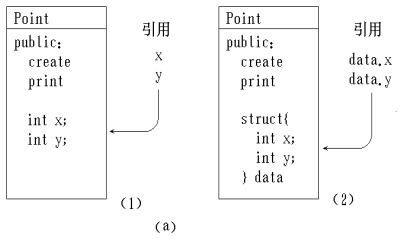
- 可复用的“插接相容性”部件
  - 部件可以在未来的应用中使用。
  - 界面的标准化
  - 类的“插接相容性”
- 可靠的部件
  - 可靠的(健壮的和正确定义的)部件。
  - 每个部件必须经过充分的测试。
  - 每个操作尽可能小和作用单一。

- 可集成的部件
  - 类的界面应当尽可能小
  - 一个类所需要的数据和操作都定义在类定义中
  - 避免命名冲突
  - 封装特性保证了把一个概念的所有细节都组合在一个界面下
  - 信息隐蔽保证了实现级的名字将不会与其它类的名字互相干扰。

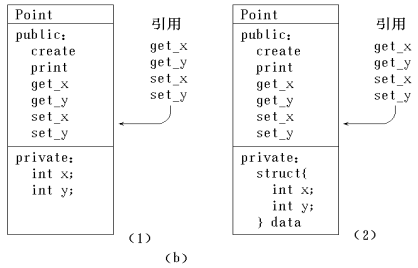
类设计的方针

- 信息隐蔽
  - 保护抽象数据类型的存储表示不被抽象数据类型实例的用户直接存取。
  - 对其表示的唯一存取途径只能是界面。

直接引用类中的数据

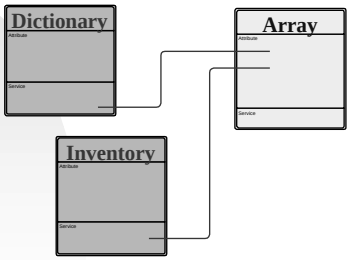


通过界面引用类中的数据



- 消息限制
  - 避开直接引用另一个类的数据
  - 类A的数据表示中包括了类C的实例，类B的数据表示则直接使用了类C。如果类A的实例发送一个消息给类B的一个实例，则类A必须知道类B的实现是如何使用类C的实例的，并把这种知识包括到它自己的实现中去。当类B需要改变自己的实现，改动类C的数据表示时，类A的实现也必须随之改变。

类间的相互影响



- 狭窄界面
  - 不是所有的操作都是公共的。
  - 对于一个**HashTable**类，界面应包括**插入**和**检索**表的操作，而不应包括**使用一个表项的关键码计算散列值**的操作。散列函数不应由类的实例的用户来访问。它应是一个单独的操作，以便容易调整或改变散列函数，它应是隐蔽实现的部分。

- 强内聚
  - 模块内部各个部分之间应有较强的关系，它们不能分别标识。
- 弱耦合
  - 一个单独模块应尽量不依赖于其它模块。如果➢在类**A**的实例中建立了类**B**的实例，➢类**A**的操作需要类**B**的实例做为参数，➢如果类**A**是类**B**的一个派生类，➔则称类**A**“依赖于”类**B**。一个类应当尽可能少地依赖于其它类。

- 显式信息传递
  - 在类之间全局变量的共享隐含了信息的传递，并且是一种依赖形式。因此，两个类之间的交互应当仅涉及显式信息传递。
  - 显式信息传递是通过参数表来完成的。借助于显式地列出将要通过参数表传递给一个操作的值，可以循特定的路径来跟踪错误。
  - 显式信息传递要最小化

- 派生类当做派生类型
  - 在继承结构中，每个派生类应该当做基类的特殊化来开发，而基类所具有的公共界面成为派生类的共有界面的一个子集。
  - C++允许设计者选择类的基类是共有的或私有的。
  - 如果基类是共有的，则其共有界面将成为新的派生类的共有界面部分，这类似于类型与派生类型之间的关系。

- 如果基类是私有的，它的行为将不是派生类的公共行为部分而是实现部分。它的提出是为了提供实现新类的服务。
- 在实现一个新类时通过声明一个类的实例，就可以使得该类的服务有效。
- **Dictionary**类的实现可采用**Array**类的实例，这样可以把存储提供给**Dictionary**项，而不给**Dictionary**类的界面增加不适当的操作。

- 抽象类
  - 某些语言提供了一个类，用它做为继承结构的开始点，所有用户定义的类都直接或间接以这个类为基类。
  - **C++**支持多重继承结构。每一种结构都包含了一组类，它们是某种概念的特殊化。这个概念应抽象地由结构的根类来表示。因此，每个继承结构的根类应当是目标概念的一个抽象模型。

- 这个抽象模型起始于一个根类，它不产生实例。它定义了一个最小的共有界面，许多派生类可以加到这个界面上以给出概念的一个特定视图。
- 考虑一组涉及“List”概念的类，根类应提供一组操作做为界面而不考虑是什么表。这个抽象类可以提供某些操作的缺省实现，但在派生类中将根据特殊化要求给出特定实现。

通过复用设计类

- 利用既存类来设计类，有4种方式：选择，分解，配置和演变。
- 选择
  - 设计一个类最简单的服务是从既存的部件中简单地选择合乎需要的软件部件。

部件库

- 一个面向对象开发环境应提供一个常用部件库。
- 大多数语言环境都带有一个初始部件库，如整数、实数和字符，它是提供其它所有功能的基础层。
- 任一基本部件库(如“基本数据结构”部件)都应建立在这些原始层上。
- 这个层还包括一组提供其它应用论域方法的一般类，如窗口系统和图形图元。

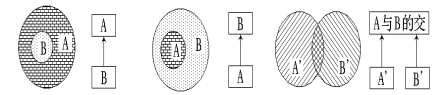
一个面向对象部件库的层次

- 特定组的部件（一个小组为他们自己组内所有成员使用而开发）
- 特定项目的部件（一个小组为某一个项目而开发）
- 特定问题论域的部件（购自某一个特定论域的软件销售商）
- 一般部件（购自专门提供部件的销售商）
- 特定语言原操作（购自一个编译器的销售商）

- 分解
  - 最初标识的“类”常常是几个概念的组合。在着手设计时，必须把一个类分成几个类，希望新标识的类容易实现，或它们已经存在。
- 配置
  - 在设计类时，我们可能会要求由既存类的实例提供类的某些特性。通过把相应类的实例声明为新类的属性来配置新类。

- 一种仿真服务器可能要求使用一个计时器来跟踪服务时间。设计者应当找到计时器类，并在服务器类的定义中声明它。
- 这个服务器还要求有一个队列类的实例来作客户排队工作。
- 对每一个客户的服务时间由一个已知的概率分布来确定，因此，可能使用一个具有泊松分布或具有均匀分布的随机变量的类的实例。

- 演化
- 要求开发的新类可能与一个既存类非常类似，但不完全相同。此时，可以利用继承机制。一般化-特殊化处理有三种可能的方式。



(a) 新类B中的对象是类A中对象的特殊化　(b) 新类B中的对象是类A中对象的一般化　(c) 新类B中对象与类A中对象共享某些公共属性

面向对象软件的实现与测试

- 在开发过程中，类的实现是核心问题。在只用面向对象风格所写的系统中，所有的数据都被封装在类的实例中而整个应用则被封装在一个更高级的类中。这种封装和类提供的标准界面很容易把类所表达的特性嵌入到应用中去。

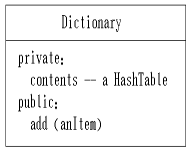
类级关系

- 当我们实现类的时候就会遇到类级的关系。
- 一个类的实现常常在某些方面依赖于其它类的实例。类级关系可以是应用级关系的实现，也可以是类内属性的实现。
  - 消息
  - 组装
  - 继承

消息(messaging)

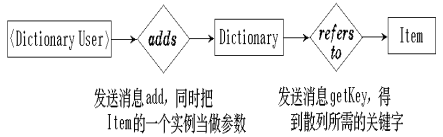
- 在应用程序中，应用级关系大多是以类的实例之间的消息连接方式实现通信的。
- 在消息的参数表中指定消息的接受者(一个类的实例)。还可以通过参数表向接收者提供信息。
- 消息指定一个属于接收者的服务，这个服务必须对应到该类共有界面规定的行为。

- Dictionary类设计的例子
- 一个Dictionary是包含一些可按关键码的值排序和检索对象的部件。
- 对于要存储在Dictionary内的一个实例来说，类必须提供一个操作来取得关键码。



词典
私有域
散列表
共有域
方法add

- 关系 *refers to* 表示了“一个类引用另一个类”，后者的实例可当作参数由前者在消息中使用。



- 由消息构成的流图形成了面向对象系统结构的核心。

- 例如，Dictionary类有一个操作*add*，该操作将把一个属于Item类的对象*item*当作参数，把这个对象加入到Dictionary中。具体地，*add*操作首先发送一个消息给做为参数的对象*item*，再利用它的关键码，到该对象所在的Item类中引用(*refers to*)相应的实例，把它加入到词典中去。
- **在设计阶段，在这样两个类之间消息关系的建立要求协调这些类的共有界面的定义。**

组装(Composition)

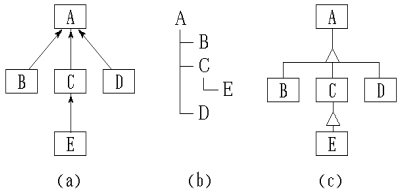
- 组装关系是一个实现级关系，它对应于应用级的聚合关系。
- 它也叫做*component*（部件）或叫做 ***is part of***（是…的一部分）。
- 组装与消息两者都是类间的关系，在这种关系中，一个类的实例将是另一个类的实现的一部分。

- 考虑Dictionary类的实现。
  - 在Dictionary中存储*item*的一种数据表示是使用散列表(HashTable)。
  - 进行Dictionary类的低层设计时，要指明在Dictionary类和HashTable类之间的一个 ***is part of*** 关系。
  - 在实现时，应当在Dictionary类的定义中声明这个Hash Table的实例。

继承(Inheritance)

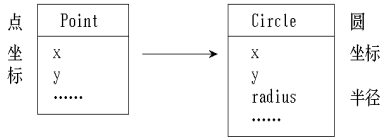
- **继承允许在既存类的基础上定义新的类。**
- 一个新类B继承了既存类A，则B包括了A定义的某些行为，以及它自定义的某些附加行为。
- 有多少种面向对象程序设计语言，就有多少种不同的继承实现方式。

继承图



① 针对实现的继承

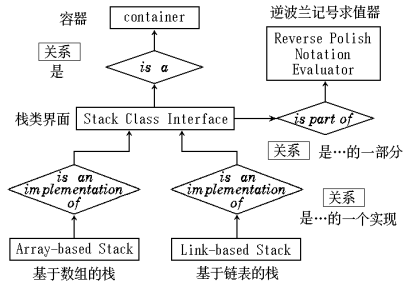
- 两个类之间“针对实现”的继承关系的建立指的是使用既存类的内部表示来做为新类的内部表示的一部分。我们不推荐这种继承方式。
- 考虑使用继承来实现一个Circle类，为了定义一个圆，需要定义一个点和一个值，做为圆的圆心和半径。因此，Point类可支持Circle类的一部分实现。把Point当做派生类。



- 如果Circle类直接使用Point的数据成员x和y，将失去抽象。而且失去做为一个点的圆心的标识。
- 针对实现的继承一般在原型开发中使用。

② 针对特殊化的继承

- 这种继承的使用适合于大多数面向对象程序设计语言所提供的关系，是针对一般化-特殊化关系的。
- 这种继承使用*is a*关系。类B的一个实例是(*is a*)类A的一个实例。
- 在使用中，继承将使得既存类的界面成为新类的界面。这表明新类具有它的基类的所有行为。



- 为了定义Dictionary类，应当首先查找既存的抽象，看Dictionary类会是哪个既存抽象的特殊情况。
- Dictionary应是一个有序表，但具有它自己特有的操作，如使用关键码进行搜索等。既存的Ordered List类可以提供Dictionary类的某些行为，但不是全部。还要确认，在Ordered List中是否有的行为在Dictionary中是不需要的。如果有，可能需要重新组织层次或者开发某些附加的抽象。

is kind of (是一种…)继承

- 这种继承允许有选择地包含既存类的属性，从而建立新的定义。
- 一个鸟类可能有一个关于飞行的属性。一个鸵鸟派生类在模型化时可能就不选择这个属性，因为鸵鸟不会飞。鸵鸟是一种(*is kind of*)鸟，但具有的属性与鸟不完全相同。
- *is kind of* 继承是不严格继承。

类的实现

- 一种方案是先开发一个比较小的比较简单的类，做为开发比较大的比较复杂的类的基础。即从简单到复杂的开发方案。
- 在这种方案中，类的开发是分层的。一个类建立在一些既存的类的基础上，而这些既存的类又是建立在其它既存的类的基础上。通过诸如“*is a*”或“*is part of*”之类的关系，利用既存代码就能着手建立新的类。

(1) 软件库(Software Base)

- 建立软件库的目的是为了引用既存的部件。
- 存储在软件库中的类以多种途径发生关联，同时，库可以追踪这些关联。
- 软件库工具利用这些关联可以有效地进行开发。

(2) 复用(Reuse)

- 伴随着类的设计，应当从复用开始着手类的实现。
- 类的设计可以使用各种抽象的类。
- 在类设计期间，我们必须开发这些类中的“具体的”对象。
- 一旦一个数据对象被确认是应用所需求的，则必须把它组织成类，以便有效地提交所需要的模型。

产生所需功能的次序

- 寻找“原封不动(As_is)”使用的既存类，提供所需要的特性；
- 寻找可以用做开发新类的基础的既存的类；
- 不用任何复用，开发一个新类。

一“原封不动”复用

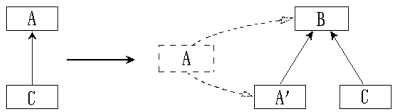
- 所需要的类已经存在，我们建立它的一个实例，用以提供所需要的特性。
- 这个实例可直接被应用利用，或者它可以用来做另一个类的实现部分。
- 通过复用一个既存类，我们可得到不加修改就能工作的已测试的代码。

一进化性复用

- 一个能够完全符合要求特性的类可能并不存在。但具有类似功能的类存在，则可以通过继承，由既存的类渐增地设计新类。
- 如果新类将要成为一个既存类的派生类，它应当继承这个既存类的所有特性。然后新类可以对需要追加的数据及必需的功能做局部定义。

- 还可以将几个既存类的特性混合起来开发出新的类。每个既存类是某些概念的模型。混合起来则产生了一个为特定目标软件所用的具有多重概念的类。
- 有时，一个既存的类可能会提供某些在我们的新类中需要的特性以及某些新类中不需要的特性。因此，我们先建立一个新的更抽象的类，使之成为我们要设计的类的基类，然后，修改既存类以继承新的基类。

- 既存类A的某些特性成为新类B的一个部分，同时被类A′和类C继承。类A的某些特性保留在类A′中，它不被类C继承。



一“废弃性”开发

- 在新类的实现时，通过说明一些既存类的实例，可以加快一个类的实现。像表格、硬件接口，或其它某些能力都可以用来作为一个新类的局部。

(3) 断言(Asserttions)

- 实现类的一个主动方法是把来自类的设计信息直接纳入代码。特别要求把参数约束、循环执行等编入到代码中。这可以通过某些表示断言的语言机制来实现。
- 一个断言就是一个语句，它表达了对一个过程、一个值，甚至一段代码的约束。

- 在栈的描述中，可以使用断言来控制进栈和退栈功能的操作：

```
procedure push (var S : Stack_Type;
        New_Item : Item_Type);
  assert:  The stack S is not full
  ……
  assert:  The top of stack S contains
           New_Item
end;
```



```
procedure pop (var S : Stack_Type)
  return Item_Type;
  assert:  The stack S is not empty
        ……
  assert:  The stack S has one fewer
  items that it did on entry
end;
```

- 先决条件
- 后置条件

- 在C与C++中有一种头文件，叫做“assert.h”，它支持断言的格式。
- 例如，实现者可以针对*pop*操作，作出断言如下：

  assert (*TOP*＞0)
- 这样，宏就会检查在试图从栈中退出一项之前栈是否空。如果条件测试失败，则会打印出一条消息，报告源文件名及在文件中发生失效的行号。

(4) 调试(Debugging)

- 数据封装限定了许多用以修改数据值的手段，也限定了对错误的数据值进行调查以找出真正原因的功能。
- 某些面向对象的程序设计环境支持使用交互工具进行调试。
- 工具包括断点的设置、访问源代码、检查对象(包括修改数据值和表达式求值)及编辑源代码。
- 标准**UNIX**调试工具**DBX**已经做了扩充，可用于调试C++程序。

(5) 错误处理(Error Handling)

- 我们期望一个类能够自负错误处理的责任。类的实例负责定位和报告错误。
- C在错误处理中使用状态码方法。各种不同的状态码的值能够指明任务的执行是成功还是失败，若是失败又是哪种程度的失败。
- 例如,C中函数“fopen”返回的状态码。如果打开失败，则返回零值；如果打开成功，则返回文件的标志。

- 使用状态码方法的难点在于：各层程序必须知道该层所调用函数的状态码，并且检验这些状态码及采用行动。
- 问题在比它发生的那一层更高的一层进行处理，这将产生比预想更高程度的耦合。
- 问题尽可能在它发生的那一层进行处理。例如，在*fopen*打开文件失败时，如果当前的文件名不存在，软件可以要求用户键入另一个文件名。

(6) 内建错误处理(Built_In_Error Handling)

- Ada程序员可以利用语言所提供的例外处理机制帮助做错误处理。
- 一个“例外”所要做的事情是与众不同的处理。“例外处理器”是一段代码，一个特定的例外出现时调用。它可以是终止软件的执行，可以是发信号给一个更高层的例外处理器，还可以是对问题进行定位处理。


```
package SIMPLE is
   EQUAL : exception;
   function max ( a : in INTEGER;  b : in
        INTEGER) return INTEGER;
   --返回 a 与 b 中的最大值
   --如果 a = b，则出现例外EQUAL.
end SIMPLE;
package body SIMPLE is
function max (a : in INTEGER;  b : in
        INTEGER) return INTEGER is
```



```
   begin
      if a = b then raise EQUAL;
      else if a < b then return b;
           else return a;
   end max;
end SIMPLE;

with SIMPLE;
   procedure MAIN is
   x : INTEGER;
   begin
```



```
   begin
      x := SIMPLE.max(7,7);
         --将会出现例外
   exception
      when SIMPLE.EQUAL => x := 7;
         --处理例外
   end;
      --处理例外并给x赋值?
end MAIN;
```


(7) 用户定义的错误处理(User_Defined Error Handling)

- 有两种相对简单的错误处理技术，它们提供了打印出错信息和终止软件执行的能力。它们都不允许嵌套的错误处理。
- 第一种技术使用了一个全局错误处理器对象。每一个类都能对这个全局对象进行存取。

- 当在一个用户对象中检测出一个错误的时候，就把一个消息发送给这个全局对象。这个消息运载了一个字符串，它就是要被打印的出错信息，消息中还有一个整数，它指出错误的严重程度。消息格式为：

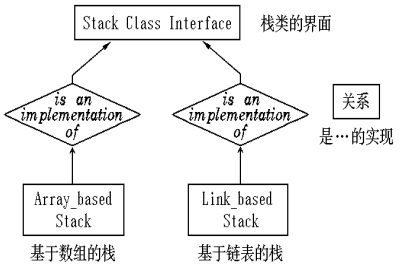
```
ERROR_HANDLER.handle
        ("Message to be printed", 1);
```

- ***ERROR_HANDLER***将打印消息并终止应用的执行。

- 第二种用户定义错误处理的技术要求每个类都定义或再定义一个命名为***error***的操作。这个操作不应是类的共有界面部分，它应是一个隐蔽的实现部分，可以被一些公共操作调用以检测错误。这种***error***操作可以打印消息，在适当时候请求一些额外输入，在必要时终止软件的执行。

(8) 多重实现 (Multiple Implementation)

- 同一个类可以多种方式实现。为此，软件库必须对库中的每一部分都能保留充足的信息，使得定义能同时关联到不止一个实现。
- 为了定义连接到几个实现所使用的关系。程序员应能指出要求的实例所在的类，并确定所期待的特定实现。



应用的实现

- 应用的实现是在所有的类都被实现之后的事情。
- 实际上，当把类开发出来时就已经实现了应用。
- 每个类提供了完成应用所需要的某种功能。
- 在C++和C中有一个***main*( )**函数。可以使用这个过程来说明构成应用的主要对象的那些类的实例。

- C++系统中主过程的两个主要职责就是建立实例和通过指针建立对象之间的通信。
- 以图形系统为例，首先建立一个用户界面的单一实例。一旦它建立起来，就发送一个消息，启动绘图程序的命令循环。
- 然后，这个对象担负起在系统寿命的其余时期协调通信关系和对象建立的责任。

- 对于纯面向对象的语言，在系统中的每个“事物”都是对象。
- 在这些语言中没有“主过程”。
- 用户建立起一个类的实例，然后，通过实例接受控制和执行服务，产生实例输出的结果或接收由用户发送来的消息。
- 由那些原始消息而产生的消息序列就成为目标软件的功能。

测试一个面向对象的应用

- 传统软件测试经历单元测试、组装测试、确认测试和系统测试等4个阶段。
- 单元测试主要针对最小的程序单元——程序模块进行测试。
- 一旦这些程序模块分别测试完成后，就将它们组装起来形成程序结构。
- 对整个系统进行一系列的测试，查找和排除在需求方面的问题。

面向对象环境下的测试策略

- 单元测试 （类测试）
  - ◆ 在面向对象环境下，最小的可测试的单元是封装了的类或对象，而不是程序模块。
  - ◆ 面向对象软件的类测试等价于传统软件开发方法中的单元测试。但它是由类中封装的操作和和类的状态行为驱动的。
  - ◆ 完全孤立地测试类的各个操作是不行的。

- ◆ 考虑一个类的层次。在基类中我们定义了一个操作**X**。
- ◆ 每一个派生类都使用操作**X**，它是在各个类所定义的私有属性和操作的环境中使用的。因使用操作**X**的环境变化太大，所以必须在每一个派生类的环境下都测试操作**X**。
- ◆ 在面向对象开发环境下，把操作完全孤立起来进行测试，其收效是很小的。

- 组装测试
  - ◆ 因为面向对象软件没有一个层次的控制结构，所以传统的自顶向下和自底向上的组装策略意义不大。
  - ◆ 每次将一个操作组装到类中（像传统的增殖式组装那样）常常行不通，因为在构成类的各个部件之间存在各种**直接的和非直接的交互**。
  - ◆ 对于面向对象系统的组装测试，存在两种不同的测试策略。

- 基于线索测试 **(Thread-based Test)**
  - ◆ 它把为响应某一系统输入或事件所需的一组类组装在一起。每一条线索将分别测试和组装。
- 基于应用的测试 **(Use-based Test)**
  - ◆ 它着眼于系统结构，首先测试独立类，这些类只使用很少的服务器类。再测试那些使用了独立类的相关类。一系列测试各层相关类的活动继续下去，直到整个系统构造完成。

- 确认测试
- 在进行确认测试和系统测试时，不关心类之间连接的细节。着眼于用户的要求和用户能够认可的系统输出。
- 为了帮助确认测试的执行，测试者需要回到分析模型，根据那里提供的事件序列（脚本）进行测试。
- 可以利用黑盒测试的方法来驱动确认测试。

- 测试方法学检测软件中的故障并确定软件是否执行了预定要开发的功能。
- 测试过程包括了一组测试用例的开发，每一个测试用例要求能检验应用的一个特定的元素。还需要分析用各个测试用例执行测试的结果来收集有关软件的信息。

按不同层次进行测试

- 测试类中各个操作，主要测试类
- 这种测试是某些单元测试与组装测试的组合
- 假定测试一个软件与测试一个类一样。这个测试者常常就是一个特定类的开发者。
- 下面讨论测试，主要集中于测试类和它们的各个操作，而不考虑确认测试或其它系统测试。

类的测试用例组

- 一个类的测试用例组由满足测试需求的用例组成。
- 每个测试用例是一系列输入值，它们将在要求的处理中执行，以满足测试需求。
- 每个测试用例应当包括送给构造函数的参数，以把对象在测试之前置于一个初始化的状态中。

类测试

- 类，作为在语法上独立的部件，应当允许用在许多不同的应用中。
- 每个类都应是可靠的，并且不需了解任何实现的细节就能复用。
- 因此，类应尽可能孤立地进行测试。

测试类操作的测试用例组

- 首先定义测试类的各个操作的测试用例组。
- 然后再把测试用例组扩充，针对被测操作调用类中其它操作的情况，进行组装测试。
- 如果一个类中的所有操作的先决条件和后置条件都已定下来，就为各个独立操作的测试用例的开发提供了指导。

类测试的种类

- 基于定义的测试
  - ◆ 把类当做一个黑盒对待，确认类的实现是否遵照它的定义。例如，若类是一个“**Stack**”，则测试应当确保 **LIFO** 原则得以实施。
- 基于程序的测试
  - ◆ 考虑类的实现，确定代码编写得是否正确。例如，在**stack**类中，确认所有语句至少应被运行一次，同时正确地执行了操作。

基于定义的测试

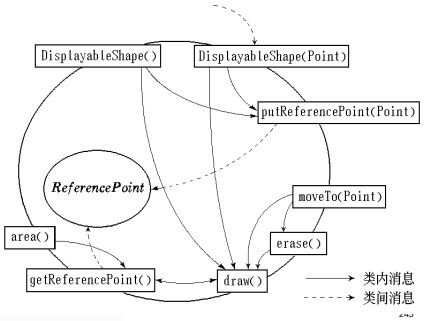
- 基于定义的测试包括两个级别：类定义和服务定义。
- 类定义
  - 一个类的定义由各个服务的定义和一些表示类的概念的语句组合而成。
  - 例，一个**stack**类包括了服务***push***和***pop***的定义。还表达了**LIFO** 的思想。

- **C++**中类的定义是多层次的。
- 对于大多数的类，检验类的定义主要检验在类定义的**public**域中所包含的那些服务。
- 对于派生类，要检查包括**public**和**protected**这两个域在内的扩充界面。
- 如果完全地检查类中定义的服务，则需要检查包括所有三个访问级别**public**，**protected**以及**private**的界面。

- 服务定义
  - 对于一个单独的服务，可通过该服务的先决条件和后置条件，以及它的名字加以定义。
  - 根据先决条件选择测试用例，产生输出，以便让测试者能够判断后置条件是否能够得到满足。
  - 各个服务的定义的测试与对于各个过程定义的测试基本相同。

基于程序的测试

- 基于程序的类的测试将测试类的各个服务，并把类当做一个单元进行测试。
- 首先，测试计划考虑测试属于该类的各个单个服务中的代码
- 然后考虑测试各个服务之间的相互作用：类内通信 / 类间通信。



- 测试可以覆盖每个服务的整个输入域。但这是不够的，还必须测试这些服务的相互作用，才能认为测试是充分的。
- 完全的单元应当保证类的执行必须覆盖它的一个有代表性的状态集合。
- 构造函数和消息序列（线索）的参数值的选择应当满足这个规则。

处于隔离的服务

- 基于程序的测试考虑测试每一个单独的服务，可以使用那些与过程性测试相同的方式对它们进行测试。
- 在测试一个服务与测试一个过程之间最明显的不同就是服务可能会改变它所在的实例的状态。
- 在测试一个服务时，该服务发送给其它实例的消息都将被隔离，由桩(stub)代替其它实例返回合适的值。

处于组装的服务

- 基于程序的测试需要考虑
  - **在同一个类内部一个服务调用另一个服务时的相互作用**(类内消息)
  - **从一个类到另一个类的消息**(类间消息)。
- 加入**检查相互作用的测试用例**到测试用例组中，确定这种交互影响是否处理得当。
- 类内测试需要执行类的所有主要的状态。

组装测试

- 类组装
  - 测试一个新类时，需要先测试在定义中所涉及的类，再考虑这些类的组装。
  - 关系“***is a***”“***is part of***”和“***refers to***”建立了测试几个类时的次序之间的关联。一旦基本类测试完成，使用这些类的那些类可以接着测试，然后按层次继续测试下去。

- 总体组装
  - 把所有组成完整软件的各个部分集合在一起。
  - 在**C++****的主过程中，仅建立几个高层的和全局的类的实例**，这些实例之间必须经常互相通信。
  - 这种测试所选择的测试用例应当瞄准待开发软件的目标，并且应当提供数据给测试者，以确定软件开发是否与它的目标相吻合。

测试一个派生类

- 对基类和继承关系进行完全测试。
- 从基类的测试用例组复用已存在的测试用例到派生类的测试用例组中。这种技术基于类的带有祖先的层次关系，渐增地开发类的测试用例组，因此叫做分层增殖式测试。
- 我们首先安排一个针对单独的类的测试计划，然后考虑分层增殖式测试计划和算法。