软件工程

选用教材

主教科书《实用软件工程》

清华大学出版社

参考教材

《软件工程 实践者的研究方法》
机械工业出版社

第一部分 软件工程概述

- ❖软件、特点、分类
- ❖软件的发展阶段
- ❖软件工程、软件工程过程、特性
- ❖软件工程的基本目标
- ❖软件生存期
- ❖软件生存期模型

- 软件是计算机系统中与硬件相互依存的另一部分，它是包括程序，数据及其相关文档的完整集合
- 程序是按事先设计的功能和性能要求执行的指令序列
- 数据是使程序能正常操纵信息的数据结构
- 文档是与程序开发，维护和使用有关的图文材料

软件的特点

- 软件是一种逻辑实体，而不是具体的物理实体。因而它具有抽象性
- 软件的生产与硬件不同，在它的开发过程中没有明显的制造过程
- 在软件的运行和使用期间，没有硬件那样的机械磨损，老化问题

- 软件的开发和运行常受到计算机系统的限制，对计算机系统有着不同程度的依赖性
- 软件的开发至今尚未完全摆脱手工艺的开发方式
- 软件本身是复杂的
  - 实际问题的复杂性
  - 程序逻辑结构的复杂性
- 软件成本相当昂贵
- 相当多的软件工作涉及到社会因素

软件的分类

- 按软件的功能进行划分：
  系统软件，支撑软件，应用软件
  - 系统软件
    - 操作系统
    - 数据库管理系统
    - 设备驱动程序
    - 通信处理程序等

- 支撑软件
  - 文本编辑程序
  - 文件格式化程序
  - 磁盘向磁带数据传输的程序
  - 程序库系统
  - 支持需求分析、设计、实现、测试和支持管理的软件

- 应用软件
  - 商业数据处理软件
  - 工程与科学计算软件
  - 计算机辅助设计／制造软件
  - 系统仿真软件
  - 智能产品嵌入软件
  - 医疗、制药软件
  - 事务管理、办公自动化软件
  - 计算机辅助教学软件

- 按软件规模进行划分：

类别	参加人员数	研制期限	源程序行数
微型	1	1～4周	0.5k
小型	1	1～6月	1k～2k
中型	2～5	1～2年	5k～50k
大型	5～20	2～3年	50k～100k
甚大型	100～1000	4～5年	1M(=1000k)
极大型	2000～5000	5～10年	1M～10M

- 按软件工作方式划分：
  - 实时处理软件
  - 分时软件
  - 交互式软件
  - 批处理软件

- 按软件服务对象的范围划分：
  - 项目软件
  - 产品软件

- 按使用的频度进行划分：
  - 一次使用
  - 频繁使用

- 按软件失效的影响进行划分：
  - 高可靠性软件
  - 一般可靠性软件

软件发展阶段

- 程序设计阶段 — 50至60年代
- 程序系统阶段 — 60至80年代

（这一时期主要围绕软件项目，开展了开发模型、支持工具以及开发方法的研究。如：瀑布模型、结构化方法（自顶向下）、结构化语言（Pascal、C、Ada语言）管理方法（费用估算、文档复审）支持工具（ 计划、配置管理工具等）

- 软件工程阶段 — 80年代以后

开展了有关软件生产技术、软件复用技术、软件生产管理的研究和实践；

提出具有广泛应用前景的面向对象方法和相关的语言（Smalltalk、C++）

近年来，软件工程的研究从过程转向产品更加注重程序的开发范型和软件生产。高智能、自动化CASE成为软件工程技术研究的热点

软件工程的定义

- Boehm：运用现代科学技术知识来设计并构造计算机程序及为开发、运行和维护这些程序所必需的相关文件资料
- IEEE：软件工程是开发、运行、维护和修复软件的系统方法
- Fritz Bauer：建立并使用完善的工程化原则，以较经济的手段获得能在实际机器上有效运行的可靠软件的一系列方法

但主要思想是强调在软件开发过程中需要应用工程化原则的重要性

软件工程三要素：方法、工具和过程

- 软件工程方法为软件开发提供了“如何做”的技术
- 软件工具为软件工程方法提供了自动的或半自动的软件支撑环境

- 软件工程过程定义了：
  - 方法使用的顺序
  - 要求交付的文档资料
  - 为保证质量和适应变化所需要的管理
  - 软件开发各个阶段完成的里程碑

软件工程过程

- 软件规格说明：规定软件的功能及其运行的限制
- 软件开发：产生满足规格说明的软件
- 软件确认：确认软件能够完成客户提出的要求
- 软件演进：为满足客户的变更要求，软件必须在使用的过程中演进

软件工程过程的特性

- 易理解性
- 可见性
- 可支持性
- 可接受性
- 可靠性
- 健壮性
- 可维护性
- 速度

软件工程项目的基本目标

- 付出较低的开发成本
- 达到要求的软件功能
- 取得较好的软件性能
- 开发的软件易于移植
- 需要较低的维护费用
- 能按时完成开发工作，及时交付使用

软件生存期 life cycle

- 软件有一个孕育、诞生、成长、成熟、衰亡的生存过程。这个过程即为计算机软件的生存期
- 软件生存期六个步骤：（问题定义和可行性研究）制定计划、需求分析、设计（总体设计和详细设计）、程序编码、测试及运行维护

制定计划

- 确定要开发软件系统的总目标
- 给出功能、性能、可靠性以及接口等方面的要求
- 完成该软件任务的可行性研究
- 估计可利用的资源(计算机硬件，软件，人力等)、成本、效益、开发进度
- 制定出完成开发任务的实施计划,连同可行性研究报告，提交管理部门审查

需求分析和定义

- 对待开发软件提出的需求进行分析并给出详细的定义
- 编写软件需求说明书或系统功能说明书及初步的系统用户手册
- 提交管理机构评审

软件设计

- 概要设计 — 把各项需求转换成软件的体系结构。结构中每一组成部分都是意义明确的模块，每个模块都和某些需求相对应
- 详细设计 — 对每个模块要完成的工作进行具体的描述，为源程序编写打下基础
- 编写设计说明书，提交评审。

程序编写

- 把软件设计转换成计算机可以接受的程序代码，即写成以某一种特定程序设计语言表示的“源程序清单”
- 写出的程序应当是结构良好、清晰易读的，且与设计相一致的

软件测试

- 单元测试，查找各模块在功能和结构上存在的问题并加以纠正
- 组装测试，将已测试过的模块按一定顺序组装起来
- 按规定的各项需求，逐项进行有效性测试，决定已开发的软件是否合格，能否交付用户使用

运行 / 维护

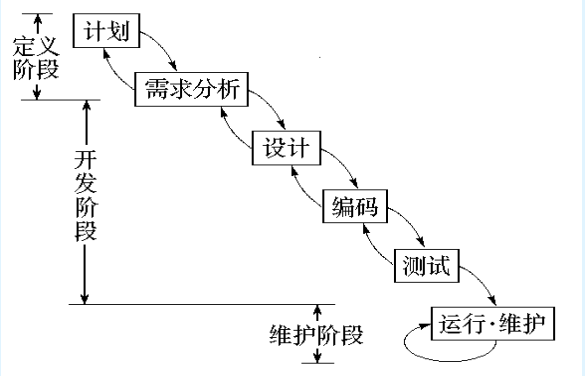
- 改正性维护　运行中发现了软件中的错误需要修正
- 适应性维护　为了适应变化了的软件工作环境，需做适当变更
- 完善性维护　为了增强软件的功能需做变更

软件生存期（开发）模型

软件生存期模型直观地表达软件开发全部过程，明确规定要完成的主要活动和任务

- 瀑布模型
- 演化模型
- 螺旋模型
- 喷泉模型
- 智能模型

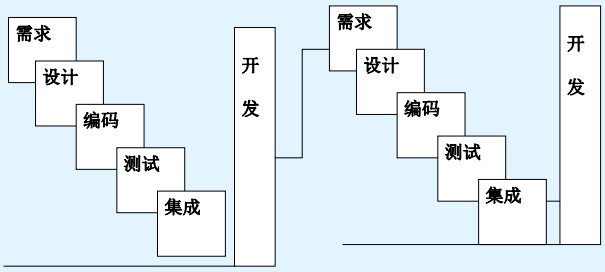
瀑布模型



特征：

- 从上一阶段接受本阶段工作的对象作为输入
- 本阶段的工作成果作为输出传入下一阶段
- 评估各阶段，若本阶段工作得到确认，继续，否则返回前一阶段
- 因而，可以增加反馈线来表示具有反馈回路的瀑布模型

演化模型

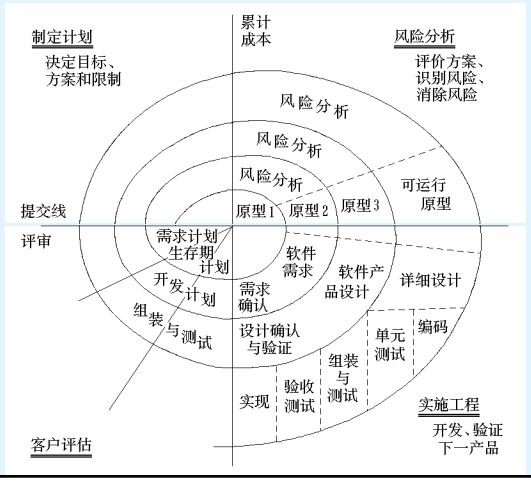


核心系统开发

特点：

- 由于在项目开发的初始阶段人们对软件的需求认识常常不够清晰，用户只能给出系统的核心，并根据实现的核心系统有效地提出反馈，来支持系统的最终设计和实现。
- 第一次只是试验开发核心系统，其目标只是在于探索可行性，弄清软件需求
- 第二次则在此基础上提出精化系统，获得较为满意的软件产品

螺旋模型

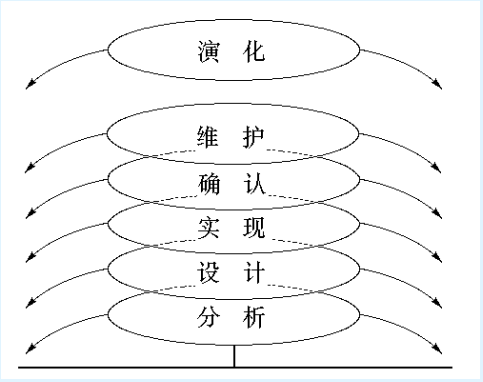


特点：

螺旋模型沿着螺线旋转，在四个象限上分别表达了四个方面的活动

- 制定计划——确定软件目标，选定实施方案，弄清项目开发的限制条件
- 风险分析——分析所选方案，考虑如何识别和消除风险
- 实施工程——实施软件开发
- 客户评估——评价开发工作，提出修正建议

喷泉模型



特点：

- 迭代
  - 重复
  - 演进
- 无间隙
  - 各阶段间无明显界限

智能模型

- 也称为基于知识的软件开发模型它综合上述开发模型，并把专家系统结合在一起。该模型应用基于规则的系统，采用规约和推理机制，帮助软件人员完成开发工作，并使维护在系统规格说明一级进行。
- 建立知识库，将模型、软件工程知识与特定领域的知识分别存入数据库。

第二部分　软件计划

- 软件的范围
- 软件开发中的资源
- 软件项目估算
- 进度安排
- 软件计划文件与复审

软件计划内容

针对不同的目标，软件工程项目需要对各阶段制定相应的工作计划

- 软件开发计划

是软件开发的综合性计划，包括任务、进度、人力、环境、资源和组织。其目的是提供一个框架，项目管理人员对资源、成本以及进度进行合理的估算。在项目开始时完成。

- 质量保证计划

将软件开发的质量要求具体规定为每个开发阶段可以检查的质量保证。

- 软件测试计划

规定测试活动的任务、方法、进度、资源、人员职责。

- 文件编制计划

规定软件开发项目应编写文件的种类、内容、进度、人员职责。

- 用户培训计划

对用户进行技术培训的目标、要求、进度、人员职责。

- 综合支持计划

项目开发过程中所需支持条件

- 软件分发计划

确定软件项目如何提供给用户

软件项目计划的目标

- 软件项目管理人员在开发工作一开始需要进行定量估算。
- 软件项目计划的目标是提供一个能使项目管理人员对资源、成本和进度做出合理估算的框架。
- 这些估算应当在软件项目开始时的一个有限的时间段内做出，并且随着项目的进展定期进行更新。

软件的范围

软件项目计划的第一项活动是确定软件的范围。

- 软件范围包括功能、性能、限制、接口和可靠性。
- 估算开始时，应对软件的功能进行评价，对其进行适当的细化以便提供更详细的细节。由于成本和进度的估算都与功能有关，因此常常采用某种程度的功能分解。

- 性能的考虑包括处理和响应时间的需求。
- 约束条件则标识产品成本、外部硬件、可用存储或其它现有系统对软件的限制。
- 功能、性能和约束必须在一起进行评价。当性能限制不同时，为实现同样的功能，开发工作量可能相差一个数量级。

软件与其它系统元素是相互作用的。要考虑每个接口的性质和复杂性，以确定对开发资源、成本和进度的影响。接口的概念可解释为：

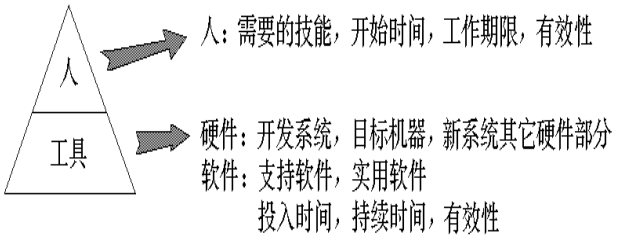
- 运行软件的硬件（如处理机与外设）及间接受软件控制的设备（如机器、显示器）；
- 必须与新软件链接的现有的软件（如数据库存取例程、子程序包、操作系统）；
- 通过终端或其它输入／输出设备使用该软件的人；
- 该软件运行前后的一系列操作过程。

- 对于每一种情况，都必须清楚地了解通过接口的信息转换。

软件开发中的资源

- 软件项目计划的第二个任务是对完成该软件项目所需的资源进行估算。
- 软件开发所需的资源有
- 现成的用以支持软件开发的工具
  ——硬件工具及软件工具
- 最基本的资源
  ——人

软件开发中的资源

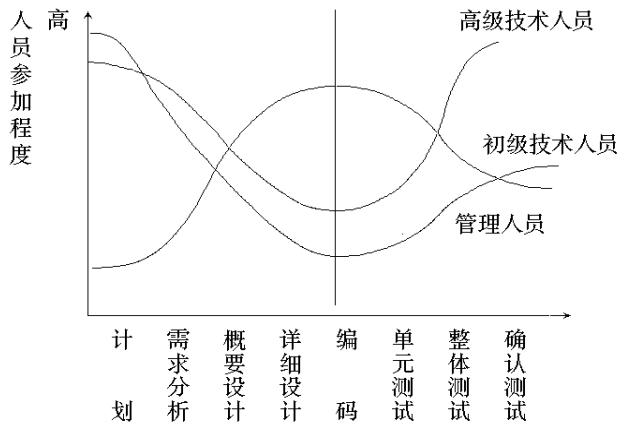


- 通常，对每一种资源，应说明以下四个特性：
  （1）资源的描述；
  （2）资源的有效性说明；
  （3）资源在何时开始需要；
  （4）使用资源的持续时间。
- 最后两个特性统称为时间窗口。对每一个特定的时间窗口，在开始使用它之前就应说明它的有效性。

1. 人力资源

- 在考虑各种软件开发资源时，人是最重要的资源。在安排开发活动时必须考虑人员的技术水平、专业、人数、以及在开发过程各阶段中对各种人员的需要。
- 计划人员首先估算范围并选择为完成开发工作所需要的技能。还要在组织状况（如管理人员、高级软件工程师等）和专业（如通信、数据库、微机等）两方面做出安排。

- 对于一些规模较小的项目（1个人年或者更少），只要向专家做些咨询，也许一个人就可以完成所有的软件工程步骤。
- 对一些规模较大的项目，在整个软件生存期中，各种人员的参与情况是不一样的。下面是各类不同的人员随开发工作的进展在软件工程各个阶段的参与情况的典型曲线。



2. 硬件资源

- 硬件是作为软件开发项目的一种工具而投入的。
  （1）宿主机（Host）— 软件开发时使用的计算机及外围设备；
  （2）目标机（Target）— 运行已开发成功软件的计算机及外围设备；
  （3）其它硬件设备 — 专用软件开发时需要的特殊硬件资源；

- **宿主机**连同**必要的软件工具**构成**软件开发系统**。通常这样的开发系统能够**支持多种用户的需要**，且能**保持大量的由软件开发小组成员共享的信息**。
- 在许多情况下，除了那些很大的系统之外，不一定非要配备专门的开发系统。因此，**所谓硬件资源，可以认为是对现存计算机系统的使用**，而不是去购买一台新的计算机。宿主机与目标机可以是同一种机型。

3. 软件资源

- 软件工程人员在软件开发期间使用了许多软件工具来帮助开发。这种软件工具集叫做**计算机辅助软件工程**（**CASE**）。

（1）**业务系统计划工具集**
（2）**项目管理工具集**
（3）**支援工具**——文档生成工具、网络系统软件、数据库、电子邮件、通报板，以及配置管理工具。

（4）**分析和设计工具**
（5）**编程工具**
（6）**组装和测试工具**
（7）**原型化和模拟工具**
（8）**维护工具**
（9）**框架工具**——这些工具能够提供建立集成项目支撑环境（**IPSE**）的框架。

4. 软件复用性及软件部件库

- 为了促成软件的复用，以提高软件的生产率和软件产品的质量，可建立可复用的软件部件库。

软件项目估算

- **软件成本和工作量的估算中变化的东西太多，人、技术、环境、政治，都会影响软件的最终成本和开发的工作量。**
- **软件项目的估算还是能够通过一系列系统化的步骤，在可接受的风险范围内提供估算结果。**
- **成本估算必须“事前”给出。时间越久，我们了解得越多，估算中出现的严重误差就越少。**

分解技术

- 当一个**待解决的问题过于复杂**时，我们可以把它**进一步分解**，直到**分解后的子问题变得容易解决**为止。然后，**分别解决每一个子问题**，并**将这些子问题的解答综合起来**，从而**得到原问题的解答**。

LOC和FP（功能点）估算

- 在软件项目估算中，在两个方面使用了LOC和FP数据：
  - 把LOC和FP数据当做一个估算变量，用于量度软件每一个元素的规模。
  - LOC和FP数据作为从过去项目中收集到的基线数据，与其它估算变量联合使用，进行成本和工作量的估算。

- LOC和FP是两个不同的估算技术。两者的共性在于：项目计划人员
  - 给出一个有界的软件范围的叙述
  - 由此叙述尝试把软件分解成一些小的可分别独立进行估算的子功能
  - 对每一个子功能估算其LOC或FP
  - 把基线生产率度量（如LOC／PM（人月）或FP／PM）用做特定的估算变量，导出子功能的成本或工作量
  - 将子功能的估算进行综合后就能得到整个项目的总估算。

- LOC或FP估算技术对于分解所需要的详细程度是不同的。
- 用LOC做为估算变量时，必须进行功能分解，且需要达到很详细的程度。而估算FP时需要的数据是宏观的量，当把FP当做估算变量时不需分解得很详细。
- LOC是直接估算的，而FP是通过估计输入、输出、数据文件、查询和外部接口的数目，以及14种复杂性校正值间接地确定的。

- 项目计划人员可对每一个分解的功能提出一个有代表性的估算值范围。
- 利用历史数据或凭实际经验（当其它的方法失效时），对每个功能分别按最佳的、可能的、悲观的三种情况给出LOC或FP估计值。记作a、m、b。
- 接着计算LOC或FP的期望值 E。

$$E = (a + 4m + b) / 6$$

- 所有子功能的总估算变量值除以相应于该估算变量的平均生产率度量得到项目的总工作量。
- 例如，若假定总的FP估算值是310，基于过去项目的平均FP生产率是5.5FP／PM，则项目的总工作量是：

  工作量 ＝ 310／5.5 ＝ 56 PM

  作为LOC和FP估算技术的实例，考察一个为计算机辅助设计（CAD）应用而开发的软件包。

- 系统定义评审指明，软件是在一个工作站上运行，其接口必须使用各种计算机图形设备，包括鼠标器、数字化仪、高分辨率彩色显示器和激光打印机。
- 在这个实例中，使用LOC做为估算变量。
- 根据系统规格说明，软件范围的初步叙述如下

“软件将从操作员那里接收2维或3维几何数据。操作员通过用户界面与CAD系统交互并控制它，这种用户界面将表现出很好的人机接口设计特性。所有的几何数据和其它支持信息保存在一个CAD数据库内。要开发一些设计分析模块以产生在各种图形设备上显示的输出。软件要设计得能控制并与能各种外部设备，包括鼠标器、数字化仪、激光打印机和绘图仪交互。”

经过分解，识别出下列主要软件功能：

- 用户界面和控制功能
- 二维几何分析
- 三维几何分析
- 数据库管理
- 计算机图形显示功能
- 外设控制PC
- 设计分析模块

- 通过分解，可得到如下估算表

估算表

功 能	a 最佳值	m 可能值	b 悲观值	E 期望值	元／行	行／PM	成本（元）	工作量（PM）
用户接口控制	1800	2400	2650	2340	14	315	32760	7.4
二维几何造型	4100	5200	7400	5380	20	220	107600	24.4
三维几何造型	4600	6900	8600	6800	20	220	136000	30.9
数据结构管理	2950	3400	3600	3350	18	240	60300	13.9
计算机图形显示	4050	4900	6200	4950	22	200	108900	24.7
外部设备控制	2000	2100	2450	2140	28	140	59920	15.2
设计分析	6600	8500	9800	8400	18	300	151200	28.0
总 计				33360			656680	144.5

- 从历史的基线数据求出生产率度量，即行／PM和元／行。
- 需要根据复杂性程度的不同，对各功能使用不同的生产率度量值。
- 在表中的成本 = LOC的期望值 E与元／行相乘，工作量 = 用LOC 的期望值 E与行／PM相除。
- 因此可得，该项目总成本的估算值为657,000元，总工作量的估算值为145人月（PM）。

工作量估算

- 工作量估算是估算任何工程开发项目成本的最普遍使用的技术。
- 每一项目任务的解决都需要花费若干工作量 (人日、人月或人年)。
- 每一个工作量单位都对应于一定的货币成本，从而可以由此做出成本估算。

- 工作量估算开始于从软件项目范围抽出软件功能。
- 接着给出为实现每一软件功能所必须执行的一系列软件工程任务，包括需求分析、设计、编码和测试。
- 针对每一软件功能，估算完成各个软件工程任务所需要的工作量（如人月）。同时，把劳动费用率（即成本／单位工作量）加到每个软件工程任务上。

- **对于每个软件工程任务，劳动费用率都可能不同。高级技术人员主要投入到需求分析和早期的设计任务中，而初级技术人员则进行后期设计任务、编码和早期测试工作，他们所需成本比较低。**
- 最后一个步骤就是**计算每一个功能及软件工程任务的工作量和成本**。

- 为了说明工作量估算的使用，考虑上面所介绍的CAD软件。
- **与每个软件工程任务相关的劳动费用率**记入表中**费用率（元）**这一行，这些数据反映了“负担”的劳动成本，即包括公司开销在内的劳动成本。
- 在此例中，**需求分析的劳动成本**为**5,200元／PM**，比编码和单元测试的劳动成本高出22%。

工作量估算表

功能＼任务	需求分析	设计	编码	测试	总计
用户界面控制	1.0	2.0	0.5	3.5	7.0
二维几何分析	2.0	10.0	4.5	9.5	26.0
三维几何分析	2.5	12.0	6.0	11.0	31.5
数据结构管理	2.0	6.0	3.0	4.0	15.0
图形显示功能	1.5	11.0	4.0	10.5	27.0
外设控制功能	1.5	6.0	3.5	5.0	16.0
设计分析模块	4.0	14.0	5.0	7.0	30.0
总计	14.5	61.0	26.5	50.5	152.5
费用率（元）	5200	4800	4250	4500	
成本（元）	75400	292800	112625	227250	708075

总工作量估算

总成本估算

除特别指出的地方之外，都按人月估算工作量。

软件开发成本估算

- 软件开发成本主要是指**软件开发过程中所花费的工作量及相应的代价**。它不包括原材料和能源的消耗，主要是人的劳动的消耗。
- **人的劳动消耗所需代价就是软件产品的开发成本。**
- 软件产品开发成本的计算方法不同于其它物理产品成本的计算。

- 软件的开发成本是**以一次性开发过程所花费的代价**来计算的。
- 软件开发成本的估算，应是从**软件计划、需求分析、设计、编码、单元测试、组装测试**到**确认测试**，整个软件开发全过程所花费的代价作为依据的。

软件开发成本估算方法

- **对于一个大型的软件项目，由于项目的复杂性，开发成本的估算不是一件简单的事，要进行一系列的估算处理。主要靠分解和类推。**
- **基本估算方法分为三类。**
  - **自顶向下的估算方法**
  - **自底向上的估计法**
  - **差别估计法**

自顶向下的估算方法

- **这种方法的主要思想是从项目的整体出发，进行类推。**
- **估算人员根据**以前已完成项目所消耗的总成本（或总工作量），**推算将要开发的软件的总成本（或总工作量），然后按比例将它分配到各开发任务单元中去，再来检验它是否能满足要求。**

软件	库存情况更新	开发者	W.Ward	日期	2 / 8 / 82
阶　段	项目任务	工作量分布(1/53)		小计(1/53)	
计划和需求	软件需求定义	5			
	开发计划	1		6	
产品设计	产品设计	6			
	初步的用户手册	3			
	测试计划	1		10	
详细设计	详细PDL描述	4			
	数据定义	4			
	过程设计	2			
	正式的用户手册	2		12	
编码与单元测试	程序编码	6			
	单元测试结果	10		16	
组装与联合测试	编写文档	4			
	组装与测试	5		9	
总　计				53	

- 这种方法的优点是估算工作量小，速度快。
- 缺点是对项目中的特殊困难估计不足，估算出来的成本盲目性大，有时会遗漏被开发软件的某些部分。

自底向上的估计法

- 这种方法的主要思想是把**待开发的软件细分，直到每一个子任务都已经明确所需要的开发工作量**，然后**把它们加起来，得到软件开发的总工作量**。
- 它的优点是估算各个部分的准确性高。缺点是缺少各项子任务之间相互联系所需要的工作量，还缺少许多与软件开发有关的系统级工作量.

差别估计法

- 这种方法综合了上述两种方法的优点，其主要思想是**把待开发的软件项目与过去已完成的软件项目进行类比，从其开发的各个子任务中区分出类似的部分和不同的部分**。
- **类似的部分按实际量进行计算，不同的部分则采用相应方法进行估算**。
- 这种方法的优点是可以提高估算的准确程度，缺点是不容易明确“类似”的界限。

专家判定技术

- **由多位专家进行成本估算**
- 单独一位专家可能会有种种偏见，譬如有乐观的、悲观的、要求在竞争中取胜的、让大家都高兴的种种愿望及政治因素等。
- **最好由多位专家进行估算，取得多个估算值**。
- 有多种方法把这些估算值合成一个估算值。

- 一种方法是**简单地求各估算值的中值或平均值**。其优点是简便。缺点是可能会由于受一、二个极端估算值的影响而产生严重的偏差。
- 一种方法是**召开小组会，使各位专家们统一于或至少同意某一个估算值**。优点是可以摈弃蒙昧无知的估算值，缺点是一些组员可能会受权威或政治因素的影响。

Deiphi技术

- **标准Deiphi技术**

① 组织者发给每位专家一份**软件系统规格说明书**和一张**记录估算值的表格**，请他们进行估算。

② 专家详细研究软件规格说明书的内容，对该软件提出**三个规模**的估算值，即：

a_i（最小） m_i（可能） b_i（最大）

无记名地填写表格

在填表的过程中，专家**互相不进行讨论**但可以向组织者提问。

③ 组织者对专家们填在表格中的答复**进行整理**：

a. **计算各位专家估算的期望值 *Ei*;**

b. **对专家的估算结果分类摘要。**

专家对此估算值另做一次估算。

④ 在综合专家估算结果的基础上，**组织专家再次无记名地填写表格**。比较两次估算的结果。若差异很大，则要通过查询找出差异的原因。

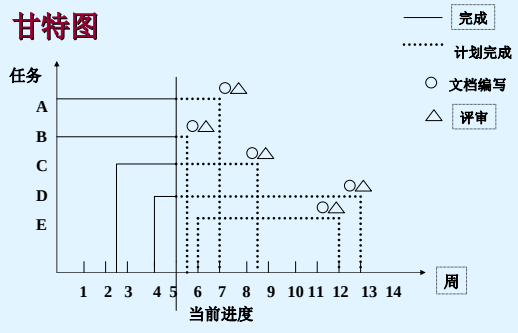
⑤ 上述过程可重复多次。**最终可获得一个得到多数专家共识的软件规模**（源代码行数）。在此过程中不得进行小组讨论。

- **最后，通过与历史资料进行类比，根据过去完成软件项目的规模和成本等信息，推算出该软件每行源代码所需要的成本。然后再乘以该软件源代码行数的估算值，就可得到该软件的成本估算值。**

进度安排

进度安排的图形方法

- **甘特图**



- **PERT技术和CPM方法**

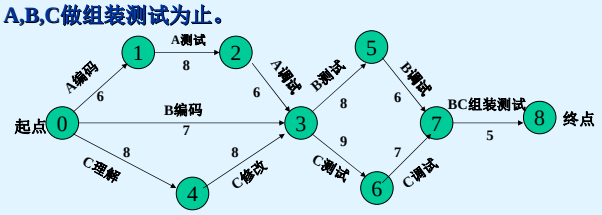
PERT技术叫做计划评审技术

CPM方法叫做关键路径法

它们都是安排开发进度，制定软件开发计划最常用的方法。它们都采用网络图来描述一个项目的任务网络，也就是从一个项目的开始到结束，把应当完成的任务用图的形式表达出来。通常用两张图来表示。一张图给出相见项目的所有任务，另一张图给出应按照什麼次序完成这些任务，给出各任务的衔接。

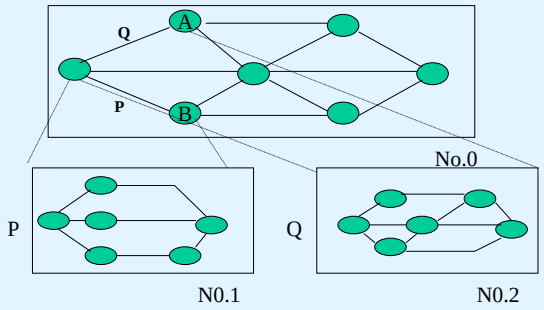
开发模块任务网络图

例：某一项目进入编码阶段考虑如何安排三个模块A,B,C的开发工作，其中A是公用模块，B和C的测试有赖于A的调试C为现成已有的模块，但对它要做理解之后做部分修改。直到A,B,C做组装测试为止。



图中各边表示要完成的任务，数字表示完成该任务的持续时间0为起点，8为终点。

分层任务网络图



在组织较为复杂的项目时或需要对特定的任务进一步做更为详细的计划时,可以使用分层的任务网络图。

软件计划文件与复审

- 软件计划文件
- 软件计划复审

第三部分　软件需求分析

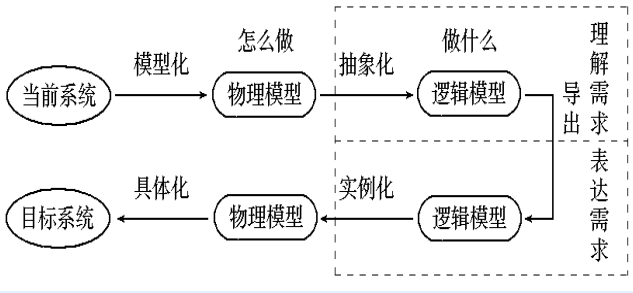
- 软件需求分析的任务
- 需求分析的过程
- 软件需求分析的原则
- 软件需求分析方法
- 结构化分析方法
- 原型化方法
- 动态分析方法
- 数据及数据库需求

软件需求分析的任务

- 深入描述软件的功能和性能
- 确定软件设计的约束和软件同其它系统元素的接口细节
- 定义软件的其它有效性需求

分析员通过需求分析，逐步细化对软件的要求，描述软件要处理的数据域，给软件开发提供一种可转化为数据设计，结构设计和过程设计的数据与功能表示。制定的 软件需求规格说明还要为评价软件质量提供依据。

- 需求分析研究的对象是软件项目的用户要求
- 准确地表达被接受的用户要求
- 确定被开发软件系统的系统元素
- 将功能和数据结构分配到这些系统元素中



- 需求分析的任务就是借助于当前系统的逻辑模型导出目标系统的逻辑模型，解决目标系统的“做什么”的问题。

- 通常软件开发项目是要实现目标系统的物理模型
- 目标系统的具体物理模型是由它的逻辑模型经实例化，即具体到某个业务领域而得到的

需求分析的过程

(1)问题识别

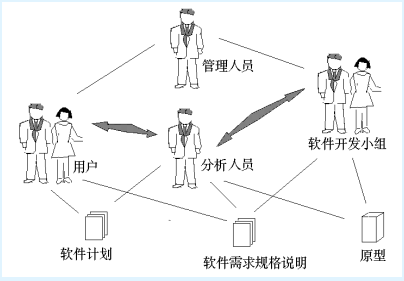
分析人员要研究计划阶段产生的可行性分析报告和软件项目实施计划。

- 从系统的角度来理解软件并评审软件范围是否恰当
- 确定对目标系统的综合要求，即软件的需求
- 提出这些需求实现条件，以及需求应达到的标准

软件需求包括：

- 功能需求
- 性能需求
- 环境需求
- 可靠性需求
- 安全保密要求
- 用户界面需求
- 资源使用需求
- 成本消耗需求
- 开发进度需求
- 预先估计以后系统可能达到的目标

问题识别的另一项工作是建立分析所需要的通信途径，以保证能顺利地对问题进行分析。



（2）分析与综合

需求分析的第二步工作是问题分析和方案的综合。

- 从数据流和数据结构出发，逐步细化所有的软件功能，找出系统各元素之间的联系、接口特性和设计上的约束，分析它们是否满足功能要求，是否合理。剔除其不合理的部分，增加其需要部分。最终综合成系统的解决方案，给出目标系统的详细逻辑模型。

常用的分析方法

- 面向数据流的结构化分析方法（SA）
- 面向数据结构的Jackson方法（JSD）
- 面向对象的分析方法（OOA）等
- 建立动态模型的状态迁移图或Petri网

(3) 编制需求分析阶段的文档

- 软件需求说明书
- 数据要求说明书
- 初步的用户手册
- 修改、完善与确定软件开发实施计划

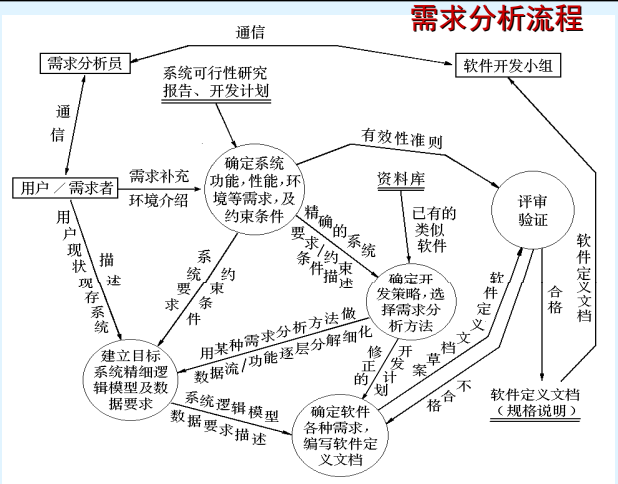
(4) 需求分析评审

- 系统定义的目标是否与用户的要求一致；
- 系统需求分析阶段提供的文档资料是否齐全；
- 文档中的所有描述是否完整、清晰、准确反映用户要求；
- 与所有其它系统成分的重要接口是否都已经描述；

- 被开发项目的数据流与数据结构是否足够，确定；
- 所有图表是否清楚，在不补充说明时能否理解；
- 主要功能是否已包括在规定的软件范围之内，是否都已充分说明；
- 设计的约束条件或限制条件是否符合实际；
- 开发的技术风险是什么；

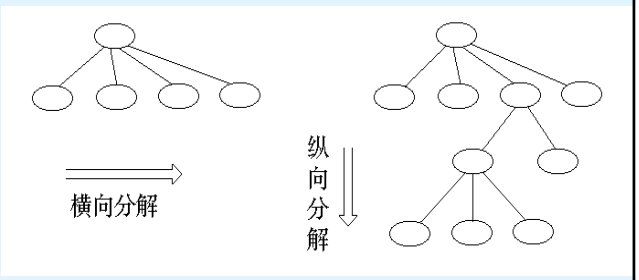
- 是否考虑过软件需求的其它方案；
- 是否考虑过将来可能会提出的软件需求；
- 是否详细制定了检验标准，它们能否对系统定义是否成功进行确认；

需求分析流程



软件需求分析的原则

- 必须能够表达和理解问题的数据域和功能域
- 必须按自顶向下，逐层分解的方式对问题进行分解和不断细化
- 要给出系统的逻辑视图和物理视图



软件需求分析方法

- 需求分析方法由对软件问题的数据域和功能域的系统分析过程及其表示方法组成
- 大多数的需求分析方法是由信息驱动的
- 数据域有三种属性：数据流、数据内容和数据结构。

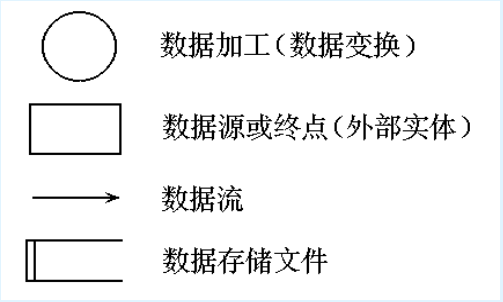
结构化分析方法

- 结构化分析是面向数据流进行需求分析的方法
- 结构化分析方法适合于数据处理类型软件的需求分析

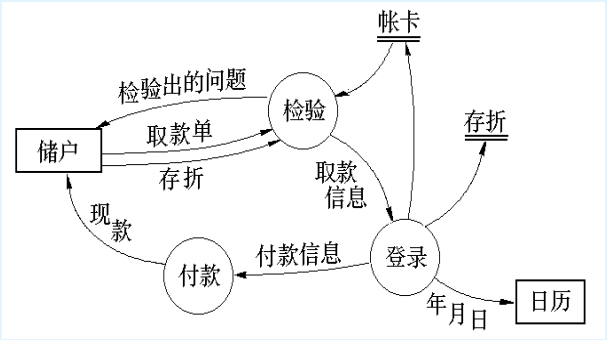
- 具体来说，结构化分析方法就是用抽象模型的概念，按照软件内部数据传递、变换的关系，自顶向下逐层分解，直到找到满足功能要求的所有可实现的软件为止
- 结构化分析方法使用工具：数据流图，数据词典，结构化英语，判定表与判定树

数据流图

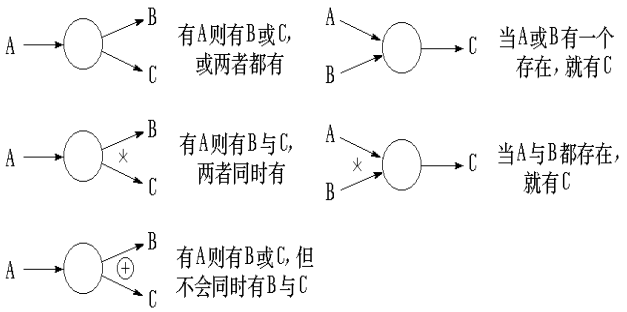
- 数据流图中的主要图形元素



例：办理取款手续的数据流图



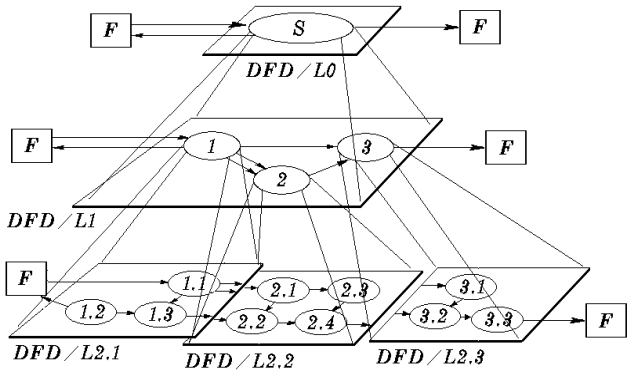
数据流与数据加工之间的关系



分层的数据流图

- 为了表达数据处理过程的数据加工情况，需要采用层次结构的数据流图。按照系统的层次结构进行逐步分解，并以分层的数据流图反映这种结构关系，能清楚地表达和容易理解整个系统

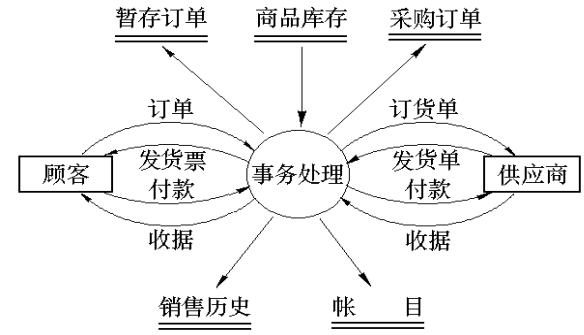
分层数据流图



- 在多层数据流图中，顶层流图仅包含一个加工，它代表被开发系统。它的输入流是该系统的输入数据，输出流是系统所输出数据
- 底层流图是指其加工不需再做分解的数据流图，它处在最底层
- 中间层流图则表示对其上层父图的细化。它的每一加工可能继续细化，形成子图。

结构化分析方法步骤示例

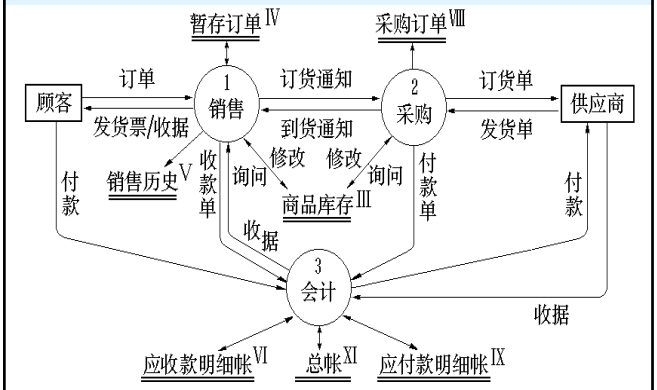
商店业务处理系统



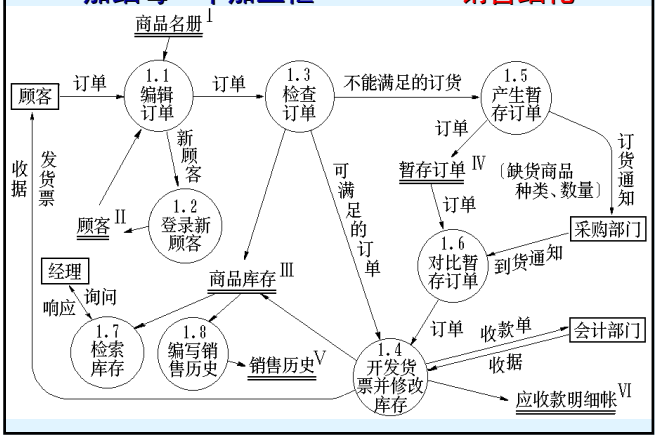
- 这个数据流图只是一个高层的系统逻辑模型，它反映了目标系统要实现的功能
- 数据流图绘制步骤
  - 首先确定系统的输入和输出
  - 根据商店业务，画出顶层数据流图，以反映最主要业务处理流程

- 经过分析，商店业务处理的主要功能应当有销售、采购、会计三大项。主要数据流输入的源点和输出终点是顾客和供应商。
- 然后从输入端开始，根据商店业务工作流程，画出数据流流经的各加工框，逐步画到输出端，得到第一层数据流图

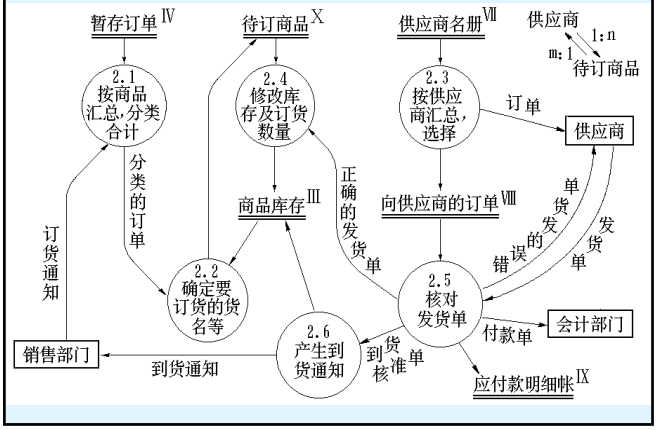
第一层数据流图



加细每一个加工框　　销售细化



采购细化



检查和修改数据流图的原则

- 数据流图上所有图形符号只限于前述四种基本图形元素
- 数据流图的主图必须包括前述四种基本元素，缺一不可
- 数据流图的主图上的数据流必须封闭在外部实体之间
- 每个加工至少有一个输入数据流和一个输出数据流

- 在数据流图中，需按层给加工框编号。编号表明该加工所处层次及上下层的亲子关系
- 规定任何一个数据流子图必须与它上一层的一个加工对应，两者的输入数据流和输出数据流必须一致。此即父图与子图的平衡
- 可以在数据流图中加入物质流，帮助用户理解数据流图

- 图上每个元素都必须有名字
- 数据流图中不可夹带控制流
- 初画时可以忽略琐碎的细节，以集中精力于主要数据流

数据词典

- 数据词典与数据流图配合，能清楚地表达数据处理的要求
- 词条描述 —— 对于在数据流图中每一个被命名的图形元素，均加以定义，其内容有:名字，别名或编号，分类，描述，定义，位置，其它，等

（1）数据流词条描述

- 数据流名：
- 说明：简要介绍作用即它产生的原因和结果
- 数据流来源：来自何方
- 数据流去向：去向何处
- 数据流组成：数据结构
- 数据量流通量：数据量，流通量

（2）数据元素词条描述

- 数据元素名：
- 类型：数字（离散值，连续值），文字（编码类型）
- 长度：
- 取值范围：
- 相关的数据元素及数据结构：

（3）数据文件词条描述

- 数据文件名：
- 简述：存放的是什么数据
- 输入数据：
- 输出数据：
- 数据文件组成：数据结构
- 存储方式：顺序，直接，关键码
- 存取频率：

（4）加工逻辑词条描述

- 加工名：
- 加工编号：反映该加工的层次
- 简要描述：加工逻辑及功能简述
- 输入数据流：
- 输出数据流：
- 加工逻辑：简述加工程序，加工顺序

（5）源点及汇(终)点词条描述

- 名称：外部实体名
- 简要描述：什么外部实体
- 有关数据流：
- 数目：

数据结构的描述

符 号	含 义	举 例
=	被定义为	
+	与	*x* = *a*+*b*
[...,...] 或 [...|...]	或	*x* = [*a*,*b*]，*x* = [*a*|*b*]
{ ... }或 m{...}n	重复	*x* = {*a*}，*x* = 3{*a*}8
(...)	可选	*x* = (*a*)
"..."	基本数据元素	*x* = "*a*"
..	连结符	*x* = 1..9

存折格式

户名　　所号　　帐号

开户日　　性质　　印密

日期 年月日	摘要	支出	存入	余额	操作	复核

- 存折＝户名＋所号＋帐号＋开户日＋性质＋（印密）＋1{存取行}50
- 户名＝2{字母}24
- 所号＝"001".."999"
- 帐号＝"00000001".."99999999"
- 开户日＝年＋月＋日
- 性质＝"1".."6"　注："1"表示普通户，"5"表示工资户等
- 印密＝"0"　注：印密在存折上不显示
- 存取行＝日期＋（摘要）＋支出＋存入＋余额＋操作＋复核

基本加工逻辑说明

- 对数据流图的每一个基本加工，必须有一个基本加工逻辑说明
- 基本加工逻辑说明必须描述基本加工如何把输入数据流变换为输出数据流的加工规则
- 加工逻辑说明必须描述实现加工的策略而不是实现加工的细节
- 加工逻辑说明中包含的信息应是充足的，完备的，有用的，没有重复的多余信息

用于写加工逻辑说明的工具

- 结构化英语
- 判定表
- 判定树

（1）结构化英语(PDL)语言

- 结构化英语的词汇表由
  - 英语命令动词
  - 数据词典中定义的名字
  - 有限的自定义词
  - 逻辑关系词 IF_THEN_ELSE、CASE_OF 、 WHILE_DO、REPEAT_UNTIL等组成。

- 是一种介于自然语言和形式化语言之间的语言
- 语言的正文用基本控制结构进行分割，加工中的操作用自然语言短语来表示
- 其基本控制结构有三种：
  - 简单陈述句结构：避免复合语句；
  - 重复结构：WHILE_DO 或 REPEAT_UNTIL结构。
  - 判定结构：IF_THEN_ELSE 或 CASE_OF结构；

商店业务处理系统中“检查发货单”

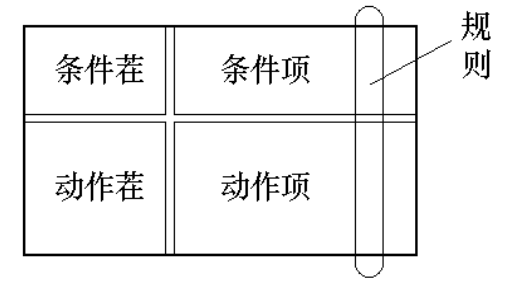
```
IF 发货单金额超过$500 THEN
   IF 欠款超过了60天 THEN
        在偿还欠款前不予批准
   ELSE （欠款未超期）
        发批准书，发货单
   ENDIF
ELSE （发货单金额未超过$500）
   IF 欠款超过60天 THEN
        发批准书，发货单及赊欠报告
ELSE （欠款未超期）
        发批准书，发货单
   ENDIF
ENDIF
```

（2）判定表

- 如果数据流图的加工需要依赖于多个逻辑条件的取值，使用判定表来描述比较合适

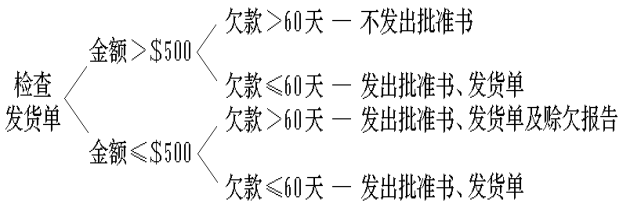
以“检查发货单”为例

		1	2	3	4
条件	发货单金额	＞$500	＞$500	≤$500	≤$500
	赊欠情况	＞60天	≤60天	＞60天	≤60天
操作	不发出批准书	√			
	发出批准书		√	√	√
	发出发货单		√	√	√
	发出除欠报告			√	



（3）判定树

- 判定树也是用来表达加工逻辑的一种工具。有时侯它比判定表更直观。



原型化方法

- 在开发初期，要想得到一个完整准确的规格说明不是一件容易的事。特别是对一些大型的软件项目。
- 用户往往对系统只有一个模糊的想法，很难完全准确地表达对系统的全面要求。

- 软件开发者对于所要解决的应用问题认识更是模糊不清
- 随着开发工作向前推进，用户可能会产生新的要求，或因环境变化，要求系统也能随之变化；开发者又可能在设计与实现的过程中遇到些没有预料到的实际困难，需要以改变需求来解脱困境。

- 因此规格说明难以完善、需求的变更、以及通信中的模糊和误解，都会成为软件开发顺利推进的障碍。
- 为了解决这些问题，逐渐形成了软件系统的快速原型的概念。

软件原型的分类

- 在软件开发中，原型是软件的一个早期可运行的版本，它反映最终系统的部分重要特性。
  - 探索型：目的是要弄清对目标系统的要求，确定所希望的特性，并探讨多种方案的可行性。

– 实验型：这种原型用于大规模开发和实现之前，考核方案是否合适，规格说明是否可靠。

– 进化型：这种原型的目的不在于改进规格说明，而是将系统建造得易于变化，在改进原型的过程中，逐步将原型进化成最终系统。

原型使用策略

– 废弃策略

– 追加策略

建立快速原型，进行系统的分析和构造的好处：

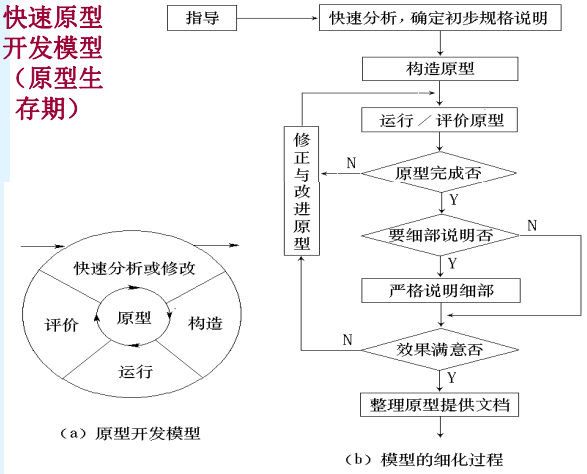
–增进软件人员和用户对系统服务需求的理解，使比较含糊的具有不确定性的软件需求（主要是功能）明确化。

–软件原型化方法提供了一种有力的学习手段。

– 使用原型化方法，可以容易地确定系统的性能，确认各项主要系统服务的可应用性，确认系统设计的可行性，确认系统作为产品的结果。

– 软件原型的最终版本，有的可以原封不动地成为产品，有的略加修改就可以成为最终系统的一个组成部分，这样有利于建成最终系统。

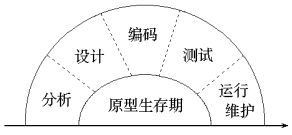
快速原型开发模型（原型生存期）



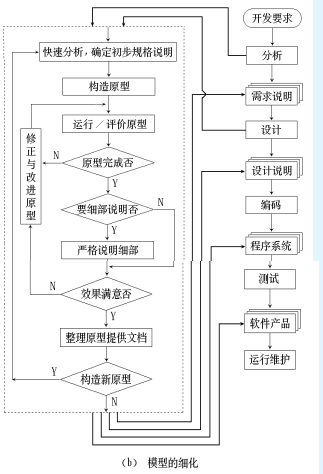
（a）原型开发模型

（b）模型的细化过程

软件开发过程



（a） 软件开发模型



（b） 模型的细化

软件复用技术

- 利用可复用的模块，做出适当的组合，就可得到快速构造的原型系统。
- 为了快速地构造原型，这些模块首先必须有简单而清晰的界面；其次它们应当尽量不依赖其它的模块或数据结构；第三，它们应具有一些通用的功能。

- 软件复用的范围尚无严格定义，它包容了用来开发软件的任何信息的复用。包括现有软件开发方法论的复用；软件要求，规格说明和设计的复用；源代码，模块和操作系统的复用；文档的复用；分析数据的复用；测试信息的复用；维护信息数据库的复用等等。软件工具与支撑环境的复用也属于软件复用工作的范围。

- 软件复用的范围基本上规为五个层次：
  - 复用数据
  - 复用模块
  - 复用结构
  - 复用设计
  - 复用规格说明

软件复用技术

- 合成技术：构件是复用的基础。构件方法以抽象数据类型为基础，将功能与数据结构封装在构件内部，构件可以是对某一个函数，过程，子程序，数据类型，算法等可复用软件成分的抽象。

形成大构件的方法：

连接；

消息传递和继承；

管道机制

生成技术：生成技术利用可复用的模式，通过生成程序产生一个新的程序或程序段，产生的程序可以看成是程序的实例。两种复用模式：

代码模式：将可复用的代码模式存于生成器内，通过特定的参数替换，生成抽象软件模块的具体实现。

规则模式：利用变换规则集合。其变换方法中通常采用高级的规格说明语言，形式化地给出软件的需求规格说明，利用程序变换系统把采用高级的规格说明语言编写的程序转化成某种可执行语言的程序。

系统动态分析

- 系统的需求规格说明通常是用自然语言来叙述的，但是用自然语言描述往往会出现歧义性。
- 为了直观地分析系统的动作，从特定的视点出发描述系统的行为，需要采用动态分析的方法。

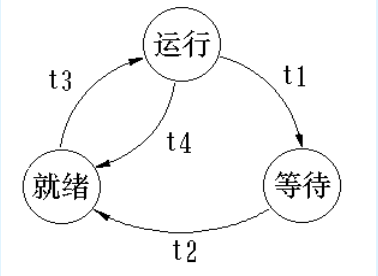
最常用的动态分析方法

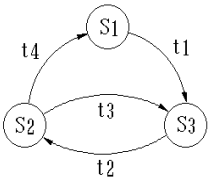
- 状态迁移图
- 时序图
- Petri网

状态迁移图

- 状态迁移图是描述系统的状态如何相应外部的信号进行推移的一种图形表示。
  - 圆圈“○”表示可得到的系统状态
  - 箭头“→”表示从一种状态向另一种状态的迁移。

例如，当有多个申请占用CPU运行的进程时，有关CPU分配的进程的状态迁移。





（a）状态迁移图

状态 / 事件	S1	S2	S3
t1	S3		
t2			S2
t3		S3	
t4		S1	

（b） 状态迁移表

- 可得到的状态＝就绪，运行，等待
- 生成的事件＝t1，t2， t3， t4
  - t1 — 中断事件
  - t2 — 中断已处理
  - t3 — 分配CPU
  - t4 — 用完CPU时间

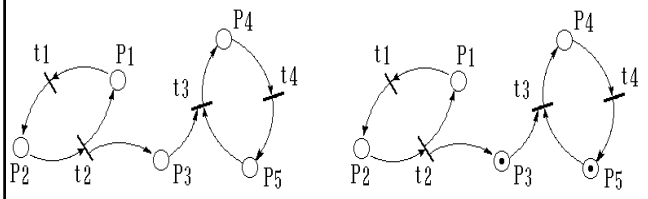
状态迁移图的优点

- 状态之间的关系能够直观地捕捉到
- 由于状态迁移图的单纯性，能够机械地分析许多情况，可很容易地建立分析工具

Petri网

- Petri网已广泛地应用于硬件与软件系统的开发中，它适用于描述与分析相互独立、协同操作的处理系统，也就是并发执行的处理系统。

- Petri网简称PNG (Petri Net Graph)，是一种有项图，它有两种结点：
  - 位置(*place*)：符号为“○”，它用来表示系统的状态。
  - 转移(*transition*)：符号为 “|”或
  - “—”, 它用来表示系统中的事件。
  - 图中的有向边表示对转移的输入，或由转移的输出

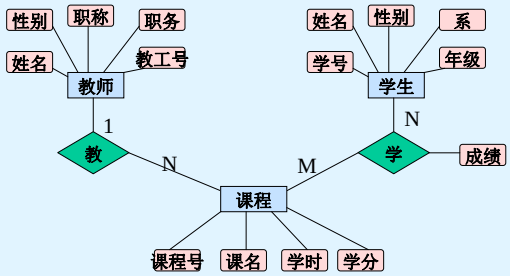


- 标记，或称令牌(*token*)，是表明系统当前处于什么状态的标志

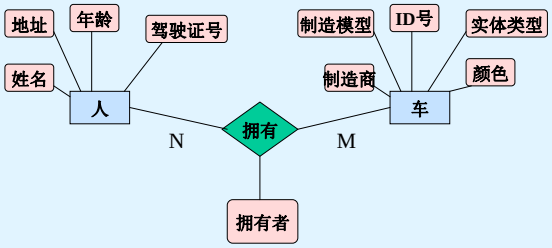
- 图中，表示位置P3与P5的圆圈中间点了一个黑点，称为标记。标记在位置上的出现表明了处理要求的到来。也就是说事件t3激发的两个前提都已具备，转移t3激发。
- 作为执行结果，位置p3与p5上的标记移去，移到了位置p4上。反过来，当激发产生的结果有几个时，将随机地选择一个结果输出，并把作为结果的位置的状态加上标记。

E-R方法（Entity-Relationship Approach）实体关系模型

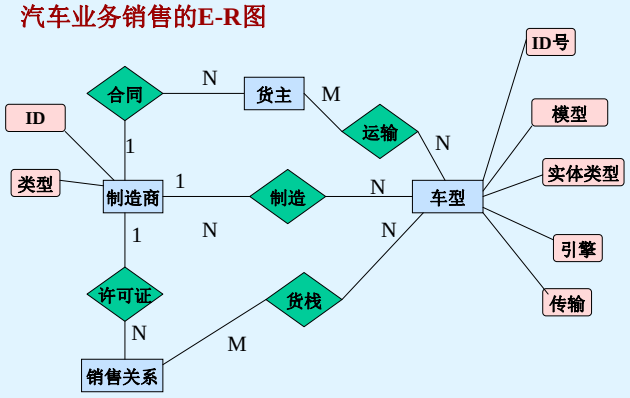
教师-学生-课程E-R 图



人与车关系E-R 图



汽车业务销售的E-R图



数据结构的规范化

三个条件：

- 表格中每个信息项必须是一个不可分割的数据项。
- 表格中每一列中所有信息项必是同一类型，名字各异
- 表格中各行互不相同

不满足以上关系的叫做非规范化关系

例：教学管理

有三个实体型，课程，学生，教师

学生（学号，姓名，性别，年龄，专业，籍贯）

教师（职工号，姓名，年龄，职称，工资级别，工资）

课程（课程号，课程名，学分，学时，课程类别）

为了表示实体型之间的关系，又建立两个关系：

选课（学号，课程号，听课出勤率，作业完成率，分数）

教课（职工号，课程号）

这五个关系组成了数据库模型。在每个关系中，属性加下划线指明关键字。

关系的规范化程度通常按属性间的依赖程度来区分，并以范式来表达。

判断规范化程度的条件：

1) 关系中所有属性都是“单纯域”，即不出现表中有表；
2) 非主属性完全函数依赖于关键字
3) 非主属性相互独立，即任何非主属性间不存在函数依赖

如果满足关系1)则为第一范式（1NF)）

如果满足关系1)和2)为第二范式(2NF)

如果满足关系1)，2)和3)为第三范式（3NF)

第四部分　软件设计

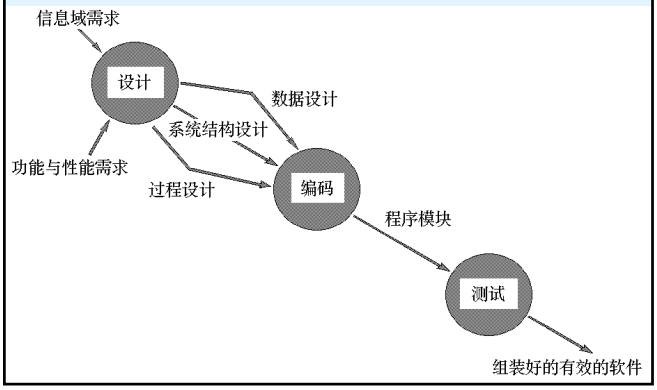
- 软件设计的目标和任务
- 软件设计基础
- 模块独立性
- 结构化设计方法
- 数据设计和文件设计
- 过程设计

软件设计的目标和任务

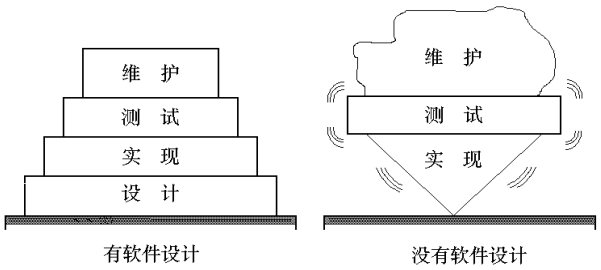
- 根据用信息域表示的软件需求，以及功能和性能需求，进行
  - 数据设计
  - 系统结构设计
  - 过程设计。

- 数据设计侧重于数据结构的定义。
- 系统结构设计定义软件系统各主要成份之间的关系。
- 过程设计则是把结构成分转换成软件的过程性描述。在编码步骤，根据这种过程性描述，生成源程序代码，然后通过测试最终得到完整有效的软件。

开发阶段的信息流



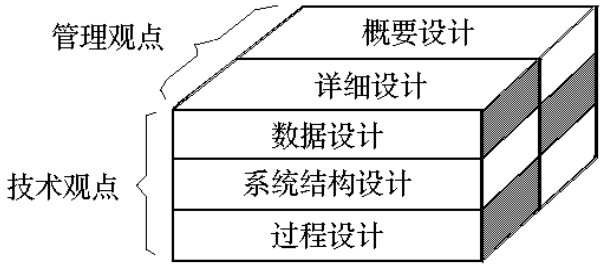
- 软件设计是后续开发步骤及软件维护工作的基础。如果没有设计，只能建立一个不稳定的系统结构



软件设计任务

- 软件设计是一个把软件需求变换成软件表示的过程。
- 从工程管理的角度来看，软件设计分两步完成。
  - 概要设计，将软件需求转化为数据结构和软件的系统结构。
  - 详细设计，即过程设计。通过对结构表示进行细化，得到软件的详细的数据结构和算法。

从管理和技术两个不同角度对设计的认识



软件设计过程（概要设计）

（1）. 制定规范

- 在进入软件开发阶段之初，首先应为软件开发组制定在设计时应该共同遵守的标准，以便协调组内各成员的工作。包括:
  - 阅读和理解软件需求说明书，确认用户要求能否实现，明确实现的条件，从而确定设计的目标，以及它们的优先顺序
  - 根据目标确定最合适的设计方法
  - 规定设计文档的编制标准
  - 规定编码的信息形式，与硬件，操作系统的接口规约，命名规则

(2). 软件系统结构的总体设计

- 基于功能层次结构建立系统。
  - 采用某种设计方法，将系统按功能划分成模块的层次结构
  - 确定每个模块的功能
  - 建立与已确定的软件需求的对应关系
  - 确定模块间的调用关系
  - 确定模块间的接口即模块间传递的信息
  - 评估模块划分的质量及导出模块结构的规则

（3）. 处理方式设计

- 确定为实现系统的功能需求所必需的算法，评估算法的性能
- 确定为满足系统的性能需求所必需的算法和模块间的控制方式。主要性能指标：
  - 周转时间（从输入一处理一输出结果为止整个）
  - 响应时间（用户向计算机发出请求之后，一次输入输出的时间）
  - 吞吐量（单位时间内能够处理的数据量）
  - 精度（运算精确度的要求）
- 确定外部信号的接收发送形式

（4）. 数据结构设计

确定软件涉及的文件系统的结构以及数据库的模式、子模式，进行数据完整性和安全性的设计

- 确定输入，输出文件的详细的数据结构
- 结合算法设计，确定算法所必需的逻辑数据结构及其操作
- 确定对逻辑数据结构所必需的那些操作的程序模块(软件包)

- 若需要与操作系统或调度程序接口所必须的控制表等数据时，确定其详细的数据结构和使用规则
- 数据的保护性设计

防卫性设计：

一致性设计：

冗余性设计：

（5）. 可靠性设计

- 可靠性设计也叫做质量设计
- 在运行过程中，为了适应环境的变化和用户新的要求，需经常对软件进行改造和修正。在软件开发的一开始就要确定软件可靠性和其它质量指标，考虑相应措施，以使得软件易于修改和易于维护

（6）. 编写概要设计阶段的文档

- 概要设计阶段完成时应编写以下文档：
  - 概要设计说明书
  - 数据库设计说明书
  - 用户手册
  - 制定初步的测试计划

（7）. 概要设计评审

- 可追溯性：确认该设计是否复盖了所有已确定的软件需求，软件每一成份是否可追溯到某一项需求
- 接口：确认该软件的内部接口与外部接口是否已经明确定义。模块是否满足高内聚和低耦合的要求。模块作用范围是否在其控制范围之内
- 风险：确认该设计在现有技术条件下和预算范围内是否能按时实现

- 实用性：确认该设计对于需求的解决方案是否实用
- 技术清晰度：确认该设计是否以一种易于翻译成代码的形式表达
- 可维护性：确认该设计是否考虑了方便未来的维护
- 质量：确认该设计是否表现出良好的质量特征

- 各种选择方案：看是否考虑过其它方案，比较各种选择方案的标准是什么
- 限制：评估对该软件的限制是否现实，是否与需求一致
- 其它具体问题：对于文档、可测试性、设计过程..等进行评估

详细设计

- 在详细设计过程中，需要完成的工作是：
  (1). 确定软件各个组成部分内的算法以及各部分的内部数据组织
  (2). 选定某种过程的表达形式来描述各种算法。
  (3). 进行详细设计的评审

软件设计基础

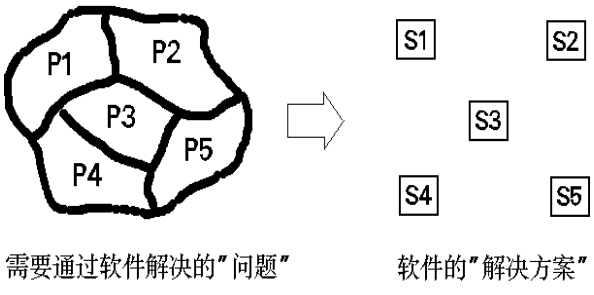
- 自顶向下，逐步细化
- 软件结构
- 程序结构
- 结构图
- 模块化
- 抽象化
- 信息隐蔽

自顶向下，逐步细化

- 将软件的体系结构按自顶向下方式，对各个层次的过程细节和数据细节逐层细化，直到用程序设计语言的语句能够实现为止，从而最后确立整个的体系结构。

软件结构

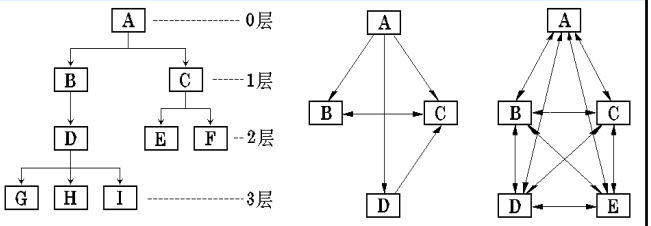
- 软件结构包括两部分。程序的模块结构和数据的结构
- 软件的体系结构通过一个划分过程来完成。该划分过程从需求分析确立的目标系统的模型出发，对整个问题进行分割，使其每个部分用一个或几个软件成份加以解决，整个问题就解决了



需要通过软件解决的"问题"　　软件的"解决方案"

程序结构

- 程序结构表明了程序各个部件(模块)的组织情况，是软件的过程表示。

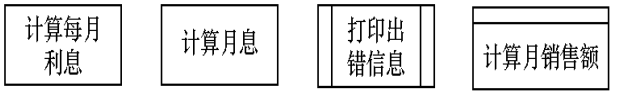


(a) 树状结构　(b) 网状结构　(c) 网状结构

结构图

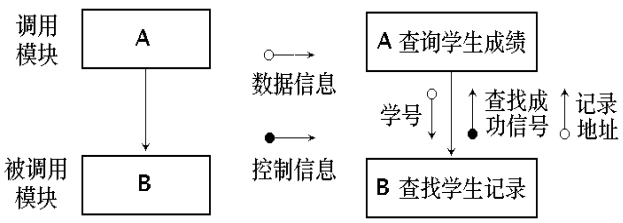
- 结构图反映程序中模块之间的层次调用关系和联系：它以特定的符号表示模块、模块间的调用关系和模块间信息的传递

① 模块：模块用矩形框表示，并用模块的名字标记它。



以功能做模块名　以功能的缩写做模块名　已定义模块　子程序(或过程)

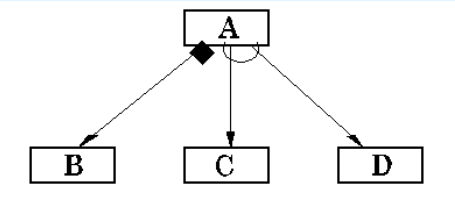
② 模块的调用关系和接口：模块之间用单向箭头联结，箭头从调用模块指向被调用模块，表示调用模块调用了被调用模块。



(a) 模块调用关系　(b) 模块间接口的表示

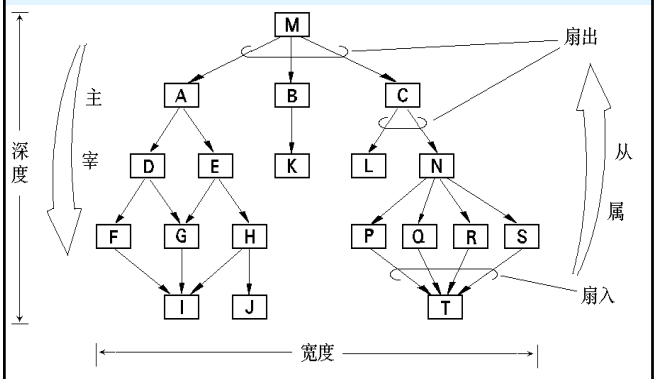
③ 模块间的信息传递：当一个模块调用另一个模块时，调用模块把数据或控制信息传送给被调用模块，以使被调用模块能够运行。而被调用模块在执行过程中又把它产生的数据或控制信息回送给调用模块

④ 在模块A的箭头尾部标以一个菱形符号，表示模块A有条件地调用另一个模块B。当一个在调用箭头尾部标以一个弧形符号，表示模块A反复调用模块C和模块D。



条件调用和循环调用

软件系统的分层模块结构图



模块化

- 软件系统的模块化是指整个软件被划分成若干单独命名和可编址的部分，称之为模块。这些模块可以被组装起来以满足整个问题的需求。
- 把问题／子问题的分解与软件开发中的系统／子系统或系统／模块对应起来，就能够把一个大而复杂的软件系统划分成易于理解的比较单纯的模块结构。

抽象化

- 软件系统进行模块设计时，可有不同的抽象层次。
- 在最高的抽象层次上，可以使用问题所处环境的语言概括地描述问题的解法。
- 在较低的抽象层次上，则采用过程化的方法。

(1) 过程的抽象

在软件工程中，从系统定义到实现，每进展一步都可以看做是对软件解决方法的抽象化过程的一次细化。

- 在软件需求分析阶段，用“问题所处环境的为大家所熟悉的术语”来描述软件的解决方法。
- 在从概要设计到详细设计的过程中，抽象化的层次逐次降低。当产生源程序时到达最低抽象层次。

例:开发一个CAD软件时的三种抽象层次

- 抽象层次Ⅰ. 用问题所处环境的术语来描述这个软件:
  该软件包括一个计算机绘图界面，向绘图员显示图形，以及一个数字化仪界面，用以代替绘图板和丁字尺。所有直线、折线、矩形、圆及曲线的描画、所有的几何计算、所有的剖面图和辅助视图都可以用这个CAD软件实现……。

- 抽象层次Ⅱ. 任务需求的描述。
  CAD SOFTWARE TASKS
  user interaction task;
  2-D drawing creation task;
  graphics display task;
  drawing file management task;
  end.
  在这个抽象层次上，未给出“怎样做”的信息，不能直接实现。

- 抽象层次Ⅲ. 程序过程表示。以2-D(二维)绘图生成任务为例:
  PROCEDURE：2-D drawing creation
  REPEAT UNTIL (drawing creation task terminates)
  DO WHILE (digitizer interaction occurs)
  digitizer interface task;
  DETERMINE drawing request CASE;
  line： line drawing task;
  rectangle：rectangle drawing task;
  circle： circle drawing task;
  ……

(2) 数据抽象
在不同层次上描述数据对象的细节，定义与该数据对象相关的操作。例如，在CAD软件中，定义一个叫做drawing的数据对象。可将drawing规定为一个抽象数据类型，定义它的内部细节为:

- TYPE drawing IS STRUCTURE
  DEFIND
  number IS STRING LENGTH(12);
  geometry DEFIND ……
  notes IS STRING LENGTH(256);
  BOM DEFIND
  END drawing TYPE;

- 数据抽象drawing本身由另外一些数据抽象，如geometry、BOM (bill of materials) 构成
- 定义drawing的抽象数据类型之后，可引用它来定义其它数据对象，而不必涉及drawing的内部细节
- 例如，定义:
  blue-print IS INSTANCE OF drawing;
  或
- schematic IS INSTANCE OF drawing;

信息隐蔽

- 由 parnas 方法提倡的信息隐蔽是指，每个模块的实现细节对于其它模块来说是隐蔽的。也就是说，模块中所包含的信息（包括数据和过程）不允许其它不需要这些信息的模块使用。

模块的独立性

- **模块（Module）**

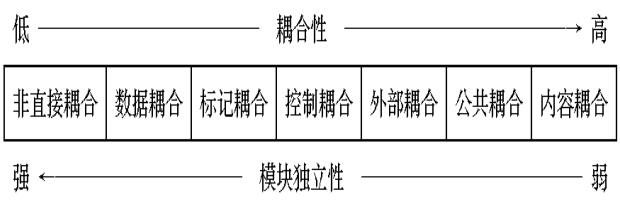
  “模块”，又称“组件”。它一般具有如下三个基本属性：
  - 功能：描述该模块实现什么功能
  - 逻辑：描述模块内部怎么做
  - 状态：该模块使用时的环境和条件

- 在描述一个模块时，还必须按模块的外部特性与内部特性分别描述
- 模块的外部特性
  - 模块的模块名、参数表、其中的输入参数和输出参数，以及给程序以至整个系统造成的影响
- 模块的内部特性
  - 完成其功能的程序代码和仅供该模块内部使用的数据

- **模块独立性**
  - 模块独立性，是指软件系统中每个模块只涉及软件要求的具体的子功能，而和软件系统中其它的模块的接口是简单的
  - 例如，若一个模块只具有单一的功能且与其它模块没有太多的联系，则称此模块具有模块独立性
  - 一般采用两个准则度量模块独立性。即模块间耦合和模块内聚

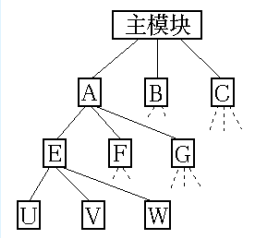
- 耦合是模块之间的互相连接的紧密程度的度量。
- 内聚是模块功能强度(一个模块内部各个元素彼此结合的紧密程度)的度量。
- 模块独立性比较强的模块应是高内聚低耦合的模块。

模块间的耦合



非直接耦合(Nondirect Coupling)

如果两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的，这就是非直接耦合。这种耦合的模块独立性最强。



数据耦合 (Data Coupling)

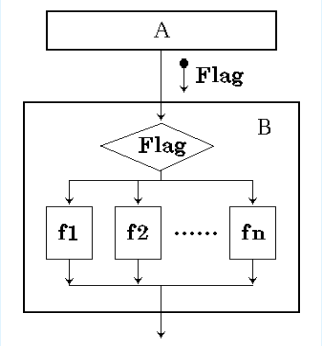
如果一个模块访问另一个模块时，彼此之间是通过**简单数据参数**（不是控制参数、公共数据结构或外部变量）来交换输入、输出信息的，则称这种耦合为数据耦合。

标记耦合 (Stamp Coupling)

如果一组模块通过参数表传递**记录信息**，就是标记耦合。这个记录是某一数据结构的子结构，而不是简单变量。

控制耦合 (Control Coupling)

如果一个模块通过传送开关、标志、名字等控制信息明显地控制选择另一模块的功能，就是控制耦合。



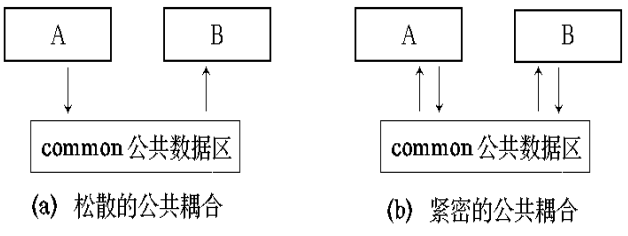
外部耦合（External Coupling）

一组模块都访问**同一全局简单变量**而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

公共耦合（Common Coupling）

若一组模块都访问**同一个公共数据环境**，则它们之间的耦合就称为公共耦合。公共的数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。

- 公共耦合的复杂程度随耦合模块的个数增加而显著增加。若只是**两模块间有公共数据环境**，则公共耦合有两种情况。松散公共耦合和紧密公共耦合。



(a) 松散的公共耦合　　(b) 紧密的公共耦合

内容耦合 (Content Coupling)

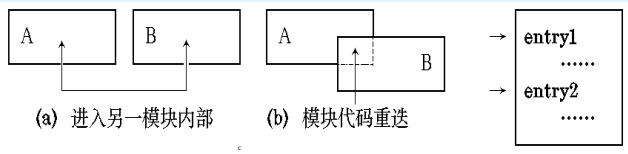
如果发生下列情形，两个模块之间就发生了内容耦合

(1) 一个模块直接访问另一个模块的内部数据;

(2) 一个模块不通过正常入口转到另一模块内部;

(3) 两个模块有一部分程序代码重迭(只可能出现在汇编语言中);

(4) 一个模块有多个入口。



模块内聚

高 ← 内聚性 → 低

功能内聚	信息内聚	通信内聚	过程内聚	时间内聚	逻辑内聚	巧合内聚

强 ← 模块独立性 → 弱

功能单一　　功能分散

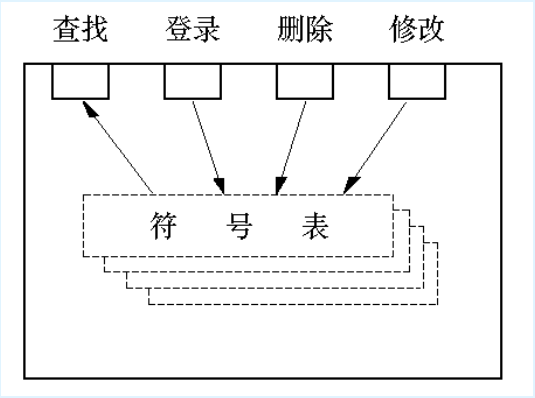
功能内聚 (Functional Cohesion)

一个模块中各个部分都是**完成某一具体功能**必不可少的组成部分，或者说该模块中所有部分都是为了完成一项具体功能而协同工作，紧密联系，不可分割的。则称该模块为功能内聚模块。

函数A-处理1
函数B-处理2
函数C-处理3

信息内聚 (Informational Cohesion)

这种模块**完成多个功能，各个功能都在同一数据结构上操作**，每一项功能有一个唯一的入口点。这个模块将根据不同的要求，确定该执行哪一个功能。由于这个模块的所有功能都是基于同一个数据结构（符号表），因此，它是一个信息内聚的模块。

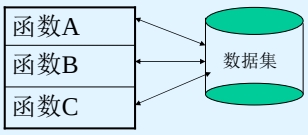


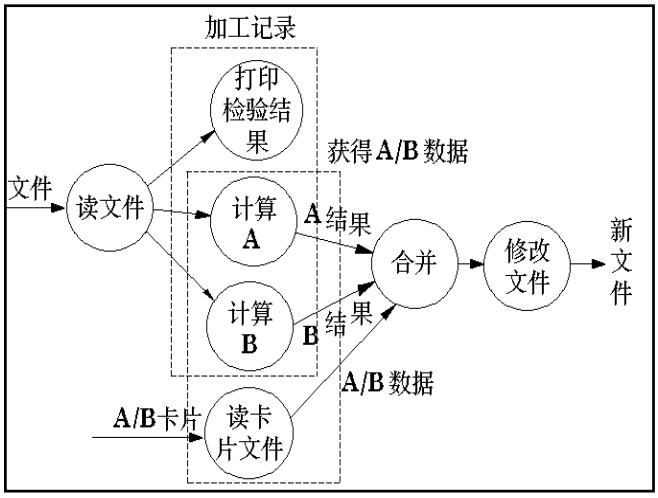
- 信息内聚模块可以看成是多个功能内聚模块的组合，并且达到信息的隐蔽。即把某个数据结构、资源或设备隐蔽在一个模块内，不为别的模块所知晓。

函数A ↓
函数B ↓
函数C ↓

通信内聚 (Communication Cohesion)

如果一个模块内各功能部分都使用了相同的输入数据，或产生了相同的输出数据，则称之为通信内聚模块。通常，通信内聚模块是通过数据流图来定义的。





过程内聚（Procedural Cohesion）

使用流程图做为工具设计程序时，把流程图中的某一部分划出组成模块，就得到过程内聚模块。例如，把流程图中的循环部分、判定部分、计算部分分成三个模块，这三个模块都是过程内聚模块。

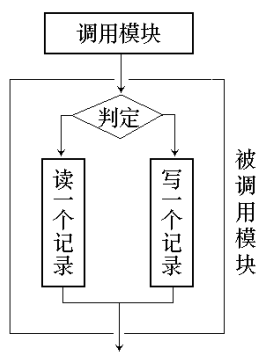
函数A	（循环）
函数B	（判定）
函数C	（计算）

时间内聚（Classical Cohesion）

时间内聚又称为经典内聚。这种模块大多为多功能模块，但模块的各个功能的执行与时间有关，通常要求所有功能必须在同一时间段内执行。例如初始化模块和终止模块。

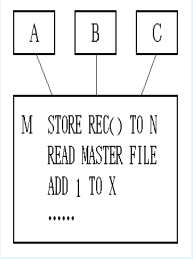
逻辑内聚（Logical Cohesion）

这种模块把几种相关的功能组合在一起，每次被调用时，由传送给模块的判定参数来确定该模块应执行哪一种功能。



巧合内聚（Coincidental Cohesion）

巧合内聚又称为偶然内聚。当模块内各部分之间没有联系，或者即使有联系，这种联系也很松散，则称这种模块为巧合内聚模块，它是内聚程度最低的模块。



函数A	
函数B	函数C
函数D	函数E

结构化设计方法

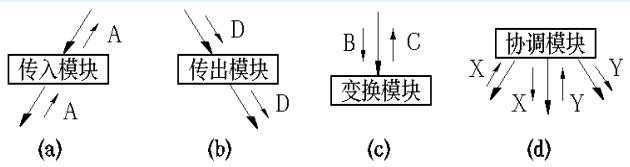
- 首先研究、分析和审查数据流图。从软件的需求规格说明中弄清数据流加工的过程，对于发现的问题及时解决。
- 然后根据数据流图决定问题的类型。数据处理问题典型的类型有两种：**变换型**和**事务型**。针对两种不同的类型分别进行分析处理。

- 由数据流图推导出系统的初始结构图。
- 利用一些启发式原则来改进系统的初始结构图，直到得到符合要求的结构图为止。
- 修改和补充数据词典。
- 制定测试计划。

在系统结构图中的模块

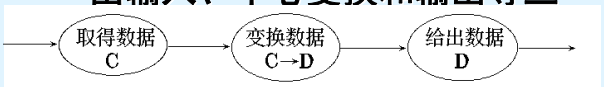
- 传入模块 — 从下属模块取得数据，经过某些处理，再将其传送给上级模块。它传送的数据流叫做逻辑输入数据流。
- 传出模块 — 从上级模块获得数据，进行某些处理，再将其传送给下属模块。它传送的数据流叫做逻辑输出数据流。

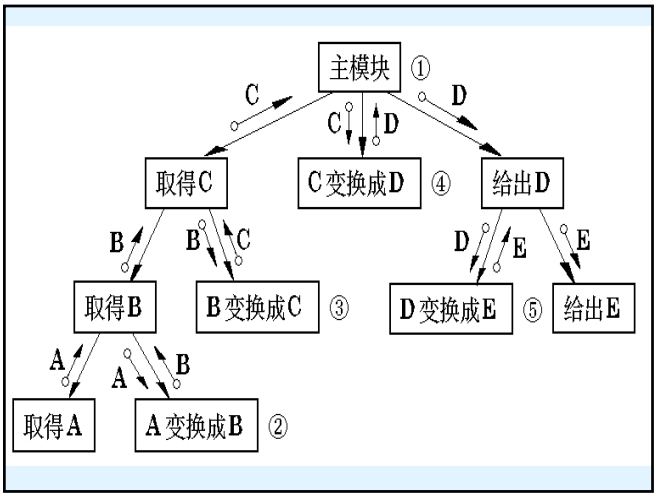
- 变换模块 — 它从上级模块取得数据，进行特定的处理，转换成其它形式，再传送回上级模块。它加工的数据流叫做变换数据流。
- 协调模块 — 对所有下属模块进行协调和管理的模块。



变换型系统结构图

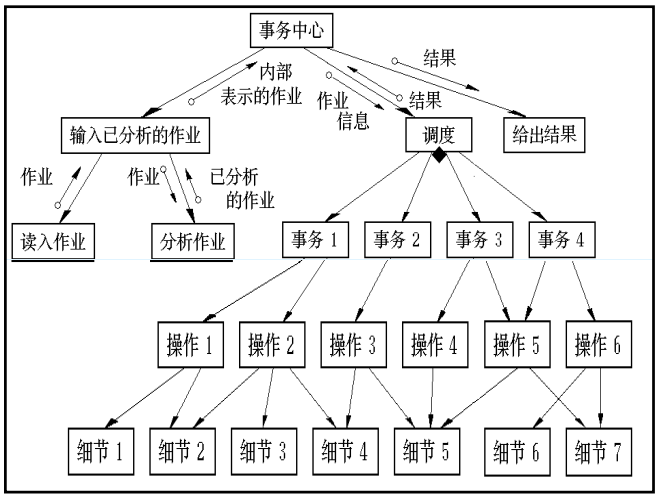
- 变换型数据处理问题的工作过程大致分为三步，即取得数据，变换数据和给出数据。
- 相应于取得数据、变换数据、给出数据，变换型系统结构图由输入、中心变换和输出等三





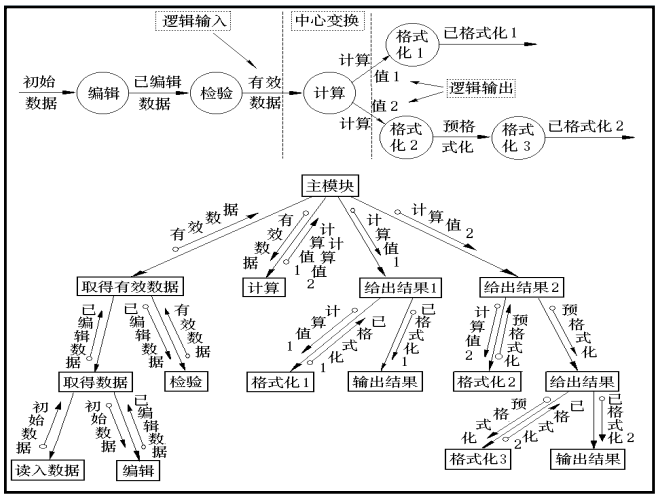
事务型系统结构图

- 它接受一项事务，根据事务处理的特点和性质，选择分派一个适当的处理单元，然后给出结果。
- 在事务型系统结构图中，事务中心模块按所接受的事务的类型，选择某一事务处理模块执行。各事务处理模块并列。每个事务处理模块可能要调用若干个操作模块，而操作模块又可能调用若干个细节模块。

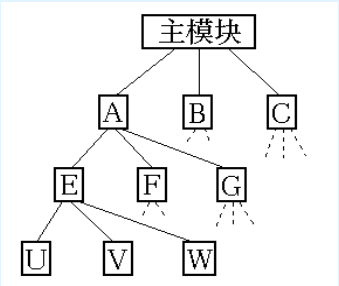


变　换　分　析

- 变换分析方法由以下四步组成：
  - 重画数据流图；
  - 区分有效（逻辑）输入、有效（逻辑）输出和中心变换部分；
  - 进行一级分解，设计上层模块；
  - 进行二级分解，设计输入、输出和中心变换部分的中、下层模块。



① 在选择模块设计的次序时，必须对一个模块的全部直接下属模块都设计完成之后，才能转向另一个模块的下层模块的设计。



② 在设计下层模块时，应考虑模块的耦合和内聚问题，以提高初始结构图的质量。

③ 使用“黑箱”技术：在设计当前模块时，先把这个模块的所有下层模块定义成“黑箱”，在设计中利用它们时，暂时不考虑其内部结构和实现。在这一步定义好的“黑箱”，在下一步就可以对它们进行设计和加工。这样，又会导致更多的“黑箱”。最后，全部“黑箱”的内容和结构应完全被确定。

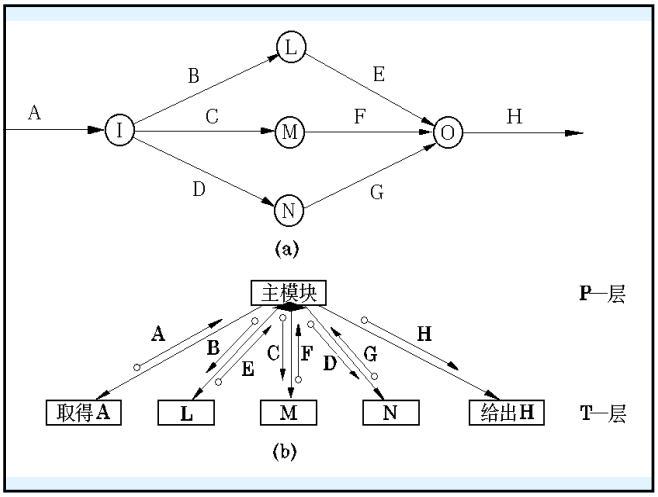
④ 在模块划分时，一个模块的直接下属模块一般在5个左右。如果直接下属模块超过10个，可设立中间层次。

⑤ 如果出现了以下情况，就停止模块的功能分解：

- 当模块不能再细分为明显的子任务时；
- 当分解成用户提供的模块或程序库的子程序时；
- 当模块的界面是输入／输出设备传送的信息时；
- 当模块不宜再分解得过小时。

事　务　分　析

- 在很多软件应用中，存在某种作业数据流，它可以引发一个或多个处理，这些处理能够完成该作业要求的功能。这种数据流就叫做事务。
- 与变换分析一样，事务分析也是从分析数据流图开始，自顶向下，逐步分解，建立系统到结构图。



事务分析过程

① 识别事务源
利用数据流图和数据词典，从问题定义和需求分析的结果中，找出各种需要处理的事务。通常，事务来自物理输入装置。有时，设计人员还必须区别系统的输入、中心加工和输出中产生的事务。

② 规定适当的事务型结构
在确定了该数据流图具有事务型特征之后，根据模块划分理论，建立适当的事务型结构。

③ 识别各种事务和它们定义的操作
从问题定义和需求分析中找出的事务及其操作所必需的全部信息，对于系统内部产生的事务，必须仔细地定义它们的操作。

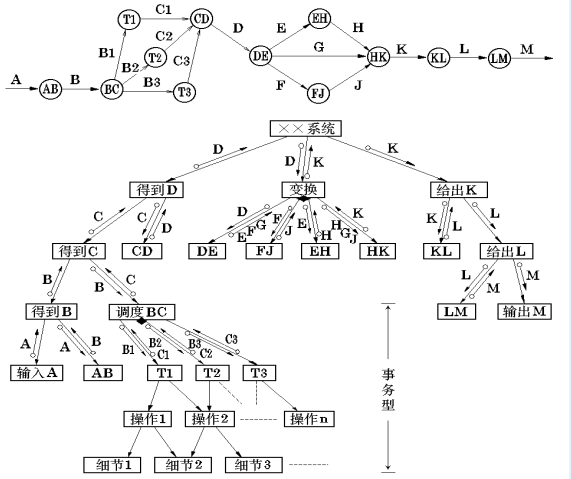
④ 注意利用公用模块
在事务分析的过程中，如果不同事务的一些中间模块可由具有类似的语法和语义的若干个低层模块组成，则可以把这些低层模块构造成公用模块。

⑤ 对每一事务，或对联系密切的一组事务，建立一个事务处理模块；如果发现在系统中有类似的事务，可以把它们组成一个事务处理模块。

⑥ 对事务处理模块规定它们全部的下层操作模块

⑦ 对操作模块规定它们的全部细节模块

变换分析是软件系统结构设计的主要方法。一般，一个大型的软件系统是变换型结构和事务型结构的混合结构。所以，我们通常利用以变换分析为主，事务分析为辅的方式进行软件结构设计。



软件模块结构的改进

- **模块功能的完善化**

  一个完整的模块应当有以下几部分：

  ① 执行规定的功能的部分；

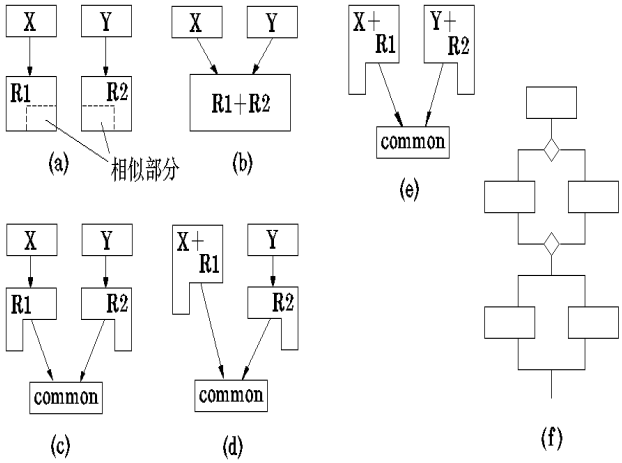
  ② 出错处理的部分。当模块不能完成规定的功能时，必须回送出错标志，出现例外情况的原因。

  ③ 如果需要返回一系列数据给它的调用者，在完成数据加工或结束时， 应当给它的调用者返回一个结束状态标志。

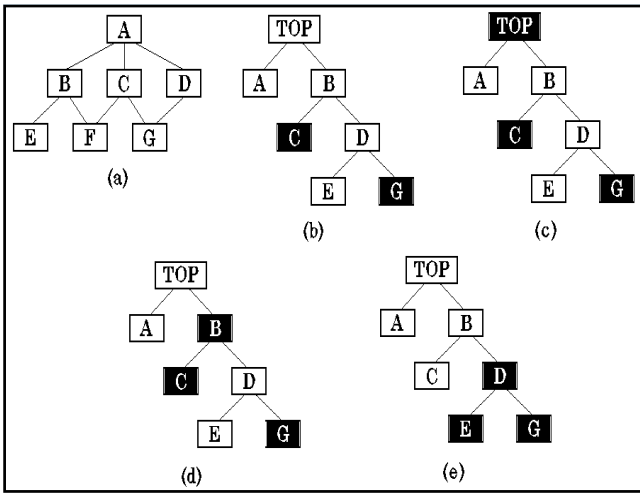
- **消除重复功能，改善软件结构**

  ① 完全相似：在结构上完全相似，可能只是在数据类型上不一致。此时可以采取完全合并的方法。

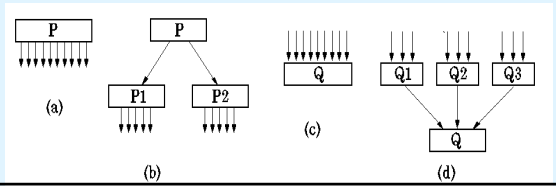
  ② 局部相似：找出其相同部分，分离出去，重新定义成一个独立的下一层模块。还可以与它的上级模块合并。



- **模块的作用范围应在控制范围之内**
  - 模块的控制范围包括它本身及其所有的从属模块。
  - 模块的作用范围是指模块内一个判定的作用范围，凡是受这个判定影响的所有模块都属于这个判定的作用范围。
  - 如果一个判定的作用范围包含在这个判定所在模块的控制范围之内，则这种结构是简单的，否则，它的结构是不简单的。

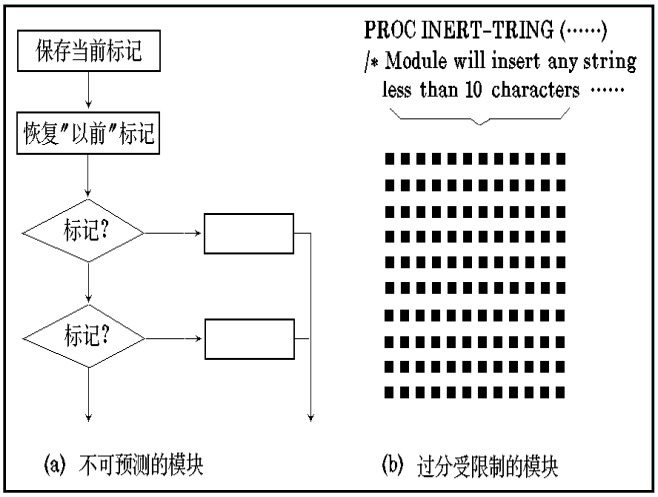


- **尽可能减少高扇出结构，随着深度增大扇入。**
  如果一个模块的扇出数过大，就意味着该模块过分复杂，需要协调和控制过多的下属模块。应当适当增加中间层次的控制模块。



- **模块的大小要适中**
  模块的大小，可以用模块中所含语句的数量的多少来衡量。把模块的大小限制在一定的范围之内。通常规定其语句行数在50～100左右，保持在一页纸之内，最多不超过500行。

- **设计功能可预测的模块，但要避免过分受限制的模块**
  - 一个功能可预测的模块，不论内部处理细节如何，但对相同的输入数据，总能产生同样的结果。但是，如果模块内部蕴藏有一些特殊的鲜为人知的功能时，这个模块就可能是不可预测的。对于这种模块，如果调用者不小心使用，其结果将不可预测。



  - 如果一个模块的局部数据结构的大小、控制流的选择或者与外界(人、硬软件) 的接口模式被限制死了，则很难适应用户新的要求或环境的变更。
  - 为了能够适应将来的变更，软件模块中局部数据结构的大小应当是可控制的，控制流的选择对于调用者来说，应当是可预测的。而与外界的接口应当是灵活的。

- **软件包应满足设计约束和可移植性**
  为了使得软件包可以在某些特定的环境下能够安装和运行，对软件包提出了一些设计约束和可移植的要求。例如，设计约束有时要求一个程序段在存储器中覆盖自身。当这种情况出现时，设计出来的软件程序结构不得不根据重复程度、访问频率、调用间隔等等特性，重新加以组织。

设 计 的 后 处 理

- 为每一个模块写一份处理说明
- 为每一个模块提供一份接口说明
- 确定全局数据结构和局部数据结构
- 指出所有的设计约束和限制
- 进行概要设计的评审
- 进行设计的优化(如果需要和可能的话)

❖数据设计及文件设计

- 数据设计的原则
- 文件设计

❖数 据 设 计 的 原 则

- R. S. Pressman数据设计的过程
  - 为在需求分析阶段所确定的数据对象选择逻辑表示，需要对不同结构进行算法分析，以便选择一个最有效的结构；设计对于这种逻辑数据结构的一组操作，以实现各种所期望的运算。
  - 确定对逻辑数据结构所必需的那些操作的程序模块(软件包)，以便限制或确定各个数据设计决策的影响范围。

Pressman提出了一组原则，用来定义和设计数据。实际上，在进行需求分析时往往就开始了数据设计。

(1). 用于软件的系统化方法也适用于数据。在导出、评审和定义软件的需求和软件系统结构时，必须定义和评审其中所用到的数据流、数据对象及数据结构的表示。应当考虑几种不同的数据组织方案，还应当分析数据设计给软件设计带来的影响。

(2). 确定所有的数据结构和在每种数据结构上施加的操作。设计有效的数据结构，必须考虑到要对该数据结构进行的各种操作。

(3). 应当建立一个数据词典并用它来定义数据和软件的设计。数据词典清楚地说明了各个数据之间的关系和对数据结构内各个数据元素的约束。

(4). 低层数据设计的决策应推迟到设计过程的后期进行。在进行需求分析时确定的总体数据组织，应在概要设计阶段加以细化，在详细设计阶段才规定具体的细节。

(5). 数据结构的表示只限于那些必须直接使用该数据结构内数据的模块才能知道。此原则就是信息隐蔽和与此相关的耦合性原则。

(6). 应当建立一个存放有效数据结构及相关操作的库。数据结构应当设计成为可复用的。建立一个存有各种可复用的数据结构模型的部件库。

(7). 软件设计和程序设计语言应当支持抽象数据类型的定义和实现。

以上原则适用于软件工程的定义阶段和开发阶段。“清晰的信息定义是软件开发成功的关键”。

❖文 件 设 计

文件设计的过程，主要分两个阶段。第一个阶段是文件的逻辑设计，主要在概要设计阶段实施。

(1) 整理必须的数据元素：

在软件设计中所使用的数据，有长期的，有短期的，还有临时的。它们都可以存放在文件中，在需要时对它们进行访问。因此首先必须整理应存储的数据元素，给它们一个易于理解的名字，指明其类型和位数，以及其内容涵义。

(2) 分析数据间的关系：

分析在业务处理中哪些数据元素是同时使用的。把同时使用次数多的数据元素归纳成一个文件进行管理。分析数据元素的内容，研究数据元素与数据元素之间的逻辑关系，根据分析，弄清数据元素的含义及其属性。

(3) 确定文件的逻辑设计：

根据数据关联性分析，明确哪些数据元素应当归于一组进行管理，把应当归于一组的数据元素进行统一布局，产生文件的逻辑设计。应用关系模型设计文件的逻辑结构时，必须使其达到第三范式(3NF)，以减少数据的冗余，提高存取的效率。

顾客文件

数据元素名	属性	长度	备注
顾客号码	X	6	
顾客姓名	K	15	
住址1	K	10	省，市
住址2	K	10	区，街
住址3	K	10	门牌号
电话号码	X	12	
邮政编码	X	6	

商品文件

数据元素名	属性	长度	备注
商品号码	X	8	3英文+
商 品 名	K	20	5数字
单　　价	N	7	
单　　位	X	3	
期首库存量	N	7	
现在库存量	N	7	

X：英文字母＋数字;　K：汉字;　N：数字

第二个阶段是文件的物理设计，主要在软件的详细设计阶段实施

(4) 理解文件的特性：

对于文件的逻辑规格说明，研究从业务处理的观点来看所要求的一些特性，包括文件的使用率、追加率和删除率，以及保护和保密等。考虑需要采用什么文件组织形式。

(5) 确定文件的组织方式；

一般要根据文件的特性，来确定文件的组织方式。

(6)确定文件的存储介质；

(7)确定文件的记录格式；

(8) 估算存取时间和存储容量。

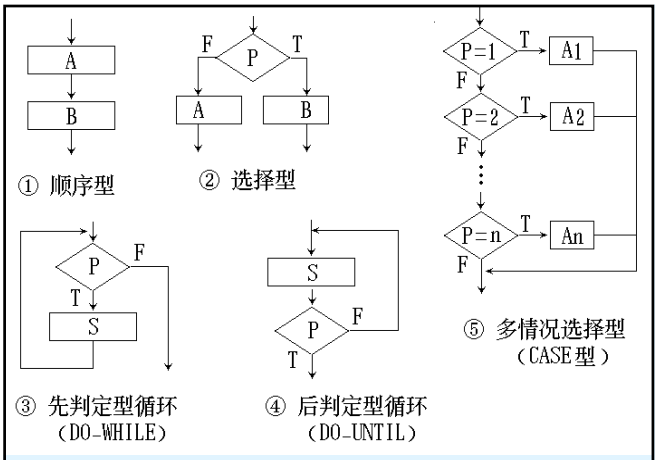
过 程 设 计

- 从软件开发的工程化观点来看，在使用程序设计语言编制程序以前，需要对所采用算法的逻辑关系进行分析，设计出全部必要的过程细节，并给予清晰的表达。这就是过程设计的任务。

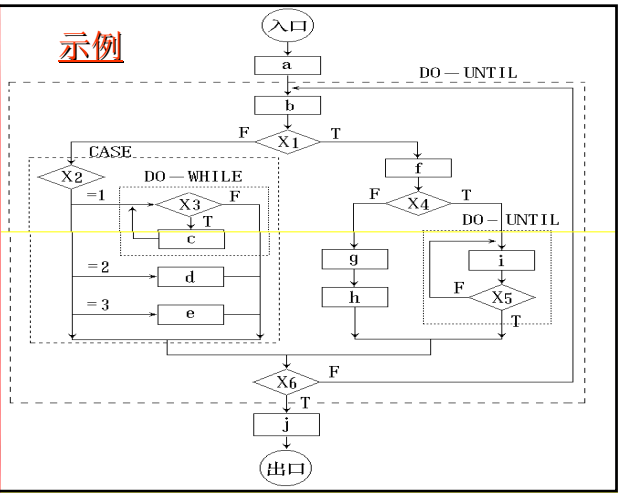
- 在过程设计阶段，要决定各个模块的实现算法，并精确地表达这些算法。表达过程规格说明的工具叫做详细设计工具，它可以分为以下三类：
  - 图形工具
  - 表格工具
  - 语言工具

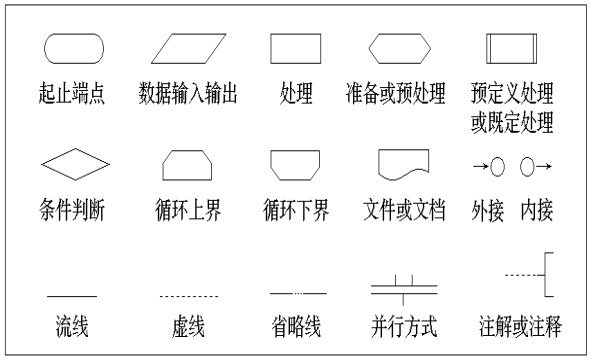
程序流程图

- 程序流程图也称为程序框图，程序流程图使用五种基本控制结构是：



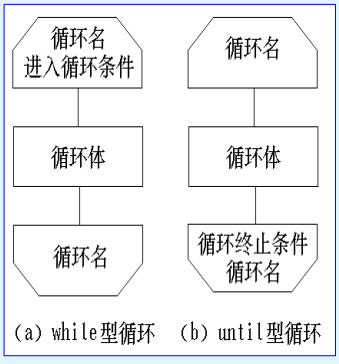
示例



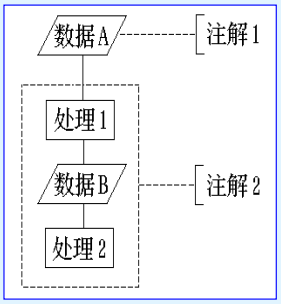


程序流程图的标准符号

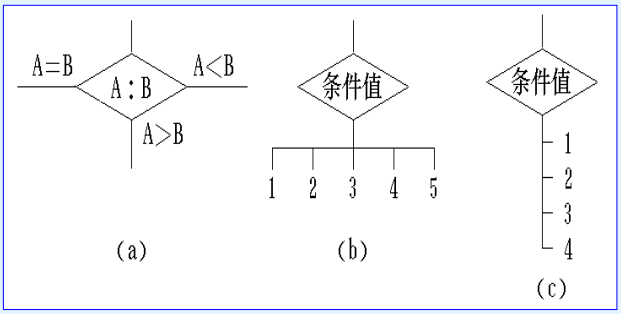
循环的标准符号



注解的使用

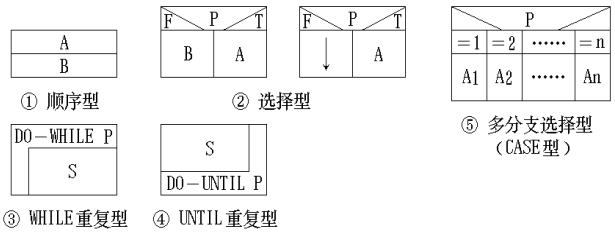


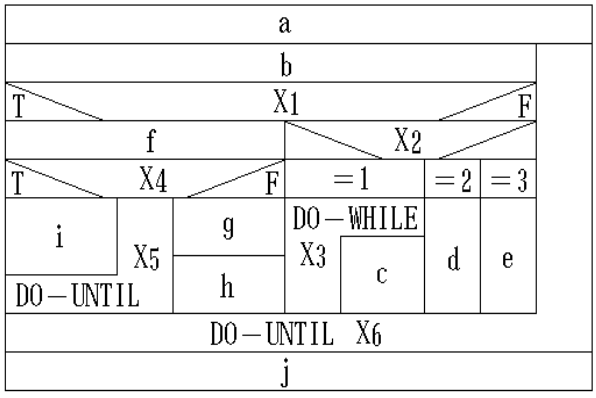
多出口判断



N-S图

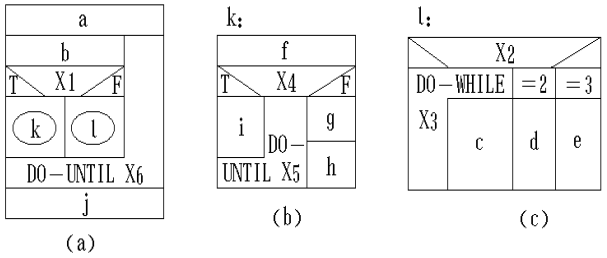
- N-S图也叫做盒图。五种基本控制结构由五种图形构件表示。





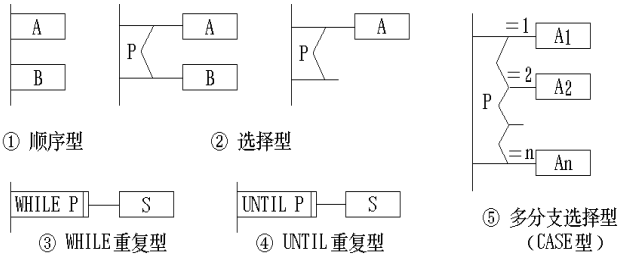
示例

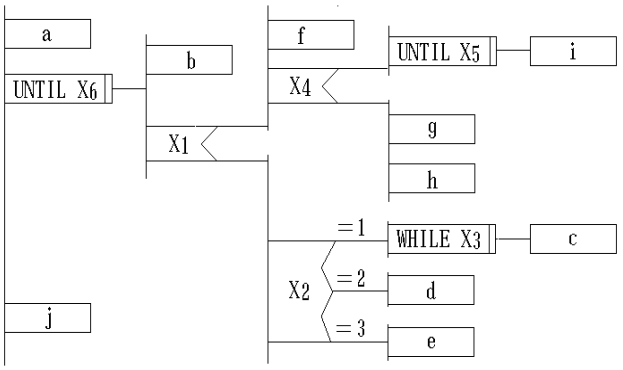
N-S图的嵌套定义形式



问题分析图(PAD)

- PAD也设置了五种基本控制结构的图式，并允许递归使用。



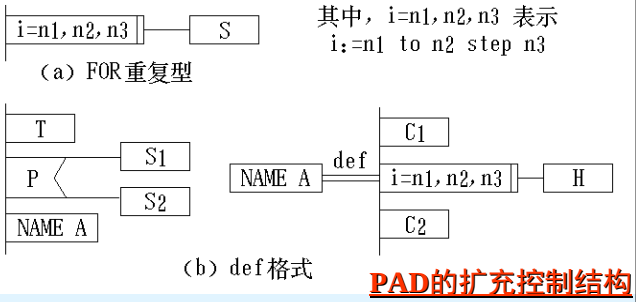


PAD描述的示例

对应于增量型循环结构

for *i* := *n1* **to** *n2* **step** *n3* **do**

在PAD中有相应的循环控制结构

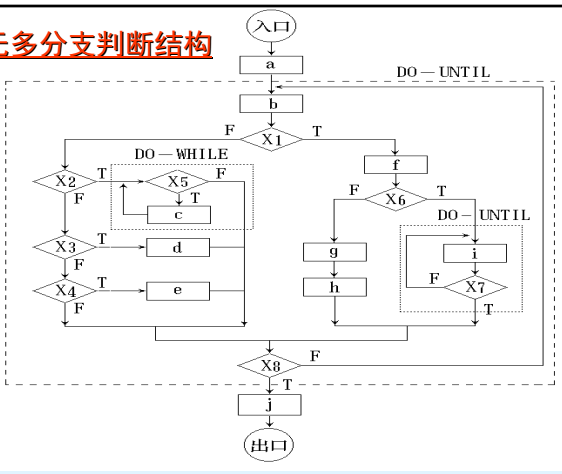


PAD的扩充控制结构

判定表

- 判定表用于表示程序的静态逻辑
- 在判定表中的条件部分给出所有的两分支判断的列表，动作部分给出相应的处理
- 要求将程序流程图中的多分支判断都改成两分支判断

无多分支判断结构



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
X1	T	T	T	T	T	F	F	F	F	F	F	F	F	F
X2	–	–	–	–	–	T	T	T	F	F	F	F	F	F
X3	–	–	–	–	–	–	–	–	F	F	T	T	F	F
X4	–	–	–	–	–	–	–	–	F	F	–	–	T	T
X5	–	–	–	–	–	T	F	F	–	–	–	–	–	–
X6	T	T	T	F	F	–	–	–	–	–	–	–	–	–
X7	T	T	F	–	–	–	–	–	–	–	–	–	–	–
X8	T	F	–	T	F	–	T	F	F	T	T	F	T	F
a	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
b	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
c	–	–	–	–	–	Y	–	–	–	–	–	–	–	–
d	–	–	–	–	–	–	–	–	–	–	Y	Y	–	–
e	–	–	–	–	–	–	–	–	–	–	–	–	Y	Y
f	Y	Y	Y	Y	Y	–	–	–	–	–	–	–	–	–
g	–	–	–	Y	Y	–	–	–	–	–	–	–	–	–
h	–	–	–	Y	Y	–	–	–	–	–	–	–	–	–
i	Y	Y	Y	–	–	–	–	–	–	–	–	–	–	–
j	Y	–	–	Y	–	–	Y	–	–	Y	Y	–	Y	–

建立判定表的步骤

- 列出与一个具体过程(或模块)有关的所有处理。
- 列出过程执行期间的所有条件(或所有判断)。
- 将特定条件取值组合与特定的处理相匹配，消去不可能发生的条件取值组合。
- 将右部每一纵列规定为一个处理规则，即对于某一条件取值组合将有什么动作。

PDL (Program Design Language)

- PDL是一种用于描述功能模块的算法设计和加工细节的语言。称为设计程序用语言。它是一种伪码。
- 伪码的语法规则分为“外语法”和“内语法”。
- PDL具有严格的关键字外语法，用于定义控制结构和数据结构，同时它的表示实际操作和条件的内语法又是灵活自由的，可使用自然语言的词汇。

PDL的特点

- 提供全部结构化控制结构、数据说明和模块特征。能对PDL正文进行结构分割，使之变得易于理解。
- 为了区别关键字，规定关键字一律大写，其它单词一律小写。或者规定关键字加下划线，或者规定它们为黑体字。

- 内语法使用自然语言来描述处理特性。内语法比较灵活，只要写清楚就可以，不必考虑语法错，以利于人们可把主要精力放在描述算法的逻辑上。
- 有数据说明机制，包括简单的(如标量和数组)与复杂的(如链表和层次结构)的数据结构。
- 有子程序定义与调用机制，用以表达各种方式的接口说明。

第五部分　程序编码

- 结构化程序设计
- 程序设计风格
- 程序效率
- 程序复杂性度量

- 做为软件工程过程的一个阶段，**程序编码是设计的继续**。
- 程序设计语言的特性和程序设计风格会深刻地影响软件的质量和可维护性。
- 为了保证程序编码的质量，程序员必须深刻地理解、熟练地掌握并正确地运用程序设计语言的特性。此外，还要求源程序具有良好的结构性和良好的程序设计风格。

结构化程序设计

结构化程序设计主要包括两方面：

(1) 在编写程序时，强调**使用几种基本控制结构**，通过组合嵌套，形成程序的控制结构。尽可能避免使用GOTO语句。

(2) 在程序设计过程中，尽量**采用自顶向下和逐步细化**的原则，由粗到细，一步步展开。

结构化程序设计的主要原则

- 使用语言中的**顺序**、**选择**、**重复**等有限的基本控制结构表示程序逻辑。
- 选用的控制结构只准许有**一个入口**和**一个出口**。
- 程序语句组成**容易识别的块**，每块只有**一个入口**和**一个出口**。
- 复杂结构应该用基本控制结构进行组合嵌套来实现。

- 语言中没有的控制结构，可用一段等价的程序段模拟，但要求该程序段在整个系统中应前后一致。
- **严格控制GOTO语句**，仅在下列情形才可使用：
  ① 用一个非结构化的程序设计语言去实现一个结构化的构造。
  ② 若不使用GOTO语句就会使程序功能模糊。
  ③ 在某种可以改善而不是损害程序可读性的情况下。

自顶向下，逐步求精

- 在详细设计和编码阶段，应当采取自顶向下，逐步求精的方法。
- 把一个模块的功能逐步分解，细化为一系列具体的步骤，进而翻译成一系列用某种程序设计语言写成的程序。

自顶向下，逐步求精方法的优点

- 符合人们解决复杂问题的普遍规律。可提高软件开发的成功率和生产率
- 用先全局后局部，先整体后细节，先抽象后具体的逐步求精的过程开发出来的程序具有清晰的层次结构，程序容易阅读和理解

- 程序自顶向下，逐步细化，分解成一个树形结构。在同一层的节点上的细化工作相互独立。有利于编码、测试和集成
- 程序清晰和模块化，使得在修改和重新设计一个软件时，可复用的代码量最大
- 每一步工作仅在上层节点的基础上做不多的设计扩展，便于检查
- 有利于设计的分工和组织工作。

程序设计风格

- 程序实际上也是一种供人阅读的文章，有一个文章的风格问题。应该使程序具有良好的风格。
  - 源程序文档化
  - 数据说明
  - 语句结构
  - 输入／输出方法

源 程 序 文 档 化

- 标识符的命名
- 安排注释
- 程序的视觉组织

符号名的命名

- 符号名即标识符，包括模块名、变量名、常量名、标号名、子程序名、数据区名以及缓冲区名等。
- 这些名字应能反映它所代表的实际东西，应有一定实际意义。
- 例如，表示次数的量用Times，表示总量的用Total，表示平均值的用Average，表示和的量用Sum等。

- 名字不是越长越好，应当选择精炼的意义明确的名字。必要时可使用缩写名字，但这时要注意缩写规则要一致，并且要给每一个名字加注释。同时，在一个程序中，一个变量只应用于一种用途。

程序的注释

- 夹在程序中的注释是程序员与日后的程序读者之间通信的重要手段。
- 注释决不是可有可无的。
- 一些正规的程序文本中，注释行的数量占到整个源程序的1／3到1／2，甚至更多。
- 注释分为序言性注释和功能性注释。

序言性注释

- 通常置于每个程序模块的开头部分，**它应当给出程序的整体说明**，对于理解程序本身具有引导作用。有些软件开发部门对序言性注释做了明确而严格的规定，要求程序编制者逐项列出

功能性注释

- 功能性注释嵌在源程序体中，用以描述其后的语句或程序段是在做什么工作，或是执行了下面的语句会怎么样。而不要解释下面怎么做。
- 例如，

```
/* ADD AMOUNT TO TOTAL */
TOTAL = AMOUNT＋TOTAL
```

不好。

- 如果注明把月销售额计入年度总额，便使读者理解了下面语句的意图：

```
/* ADD MONTHLY-SALES TO
ANNUAL-TOTAL */
TOTAL = AMOUNT＋TOTAL
```

- 要点
  - 描述一段程序，而不是每一个语句；
  - 用缩进和空行，使程序与注释容易区别；
  - 注释要正确。

视觉组织　空格、空行和移行

- 恰当地利用**空格**，可以**突出运算的优先性**，避免发生运算的错误。
- 例如，将表达式

```
(A＜－17)ANDNOT(B＜＝49)ORC
```

写成

```
(A＜－17) AND  NOT (B＜＝49)
OR  C
```

- 自然的程序段之间可用**空行**隔开；

- **移行**也叫做**向右缩格**。它是指程序中的各行不必都在左端对齐，都从第一格起排列。这样做使程序完全分不清层次关系。
- 对于**选择语句**和**循环语句**，把其中的程序段语句向右做**阶梯式移行**。使程序的逻辑结构更加清晰。
- 例如，两重选择结构嵌套，写成下面的移行形式，层次就清楚得多。

```
IF（…） THEN
    IF（…） THEN
        ……
    ELSE
        ……
    ENDIF
    ……
ELSE
 ……
ENDIF
```

数据说明

- 在设计阶段已经确定了数据结构的组织及其复杂性。在编写程序时，则需要注意数据说明的风格。
- 为了使程序中数据说明更易于理解和维护，必须注意以下几点。
  1. 数据说明的次序应当规范化
  2. 说明语句中变量安排有序化
  3. 使用注释说明复杂数据结构

数据说明的次序应当规范化

- 数据说明次序规范化，使数据属性容易查找，也有利于测试，排错和维护。
- 原则上，数据说明的次序与语法无关，其次序是任意的。但出于阅读、理解和维护的需要，最好使其规范化，使说明的先后次序固定。

说明语句中变量安排有序化

- 当多个变量名在一个说明语句中说明时，应当对这些变量按字母的顺序排列。带标号的全程数据(如FORTRAN的公用块)也应当按字母的顺序排列。
- 例如，把

  integer size, length, width, cost, price

  写成

  integer cost, length, price , size, width

使用注释说明复杂数据结构

- 如果设计了一个复杂的数据结构，应当使用注释来说明在程序实现时这个数据结构的固有特点。
- 例如，对PL/1的链表结构和Pascal中用户自定义的数据类型，都应当在注释中做必要的补充说明。

语句结构

- 在设计阶段确定了软件的逻辑流结构，但构造单个语句则是编码阶段的任务。语句构造力求简单，直接，不能为了片面追求效率而使语句复杂化。

1. 在一行内只写一条语句

- 在一行内只写一条语句，并且采取适当的移行格式，使程序的逻辑和功能变得更加明确。
- 许多程序设计语言允许在一行内写多个语句。但这种方式会使程序可读性变差。因而不可取。

- 例如，有一段排序程序

```
FOR I:=1 TO N－1 DO BEGIN
T:=I；FOR J:=I＋1 TO N DO IF A[J]
＜A[T] THEN T:=J；IF T≠I THEN
BEGIN WORK:=A[T];
A[T]:=A[I]；A[I]:=WORK；END
END;
```

- 由于一行中包括了多个语句，掩盖了程序的循环结构和条件结构，使其可读性变得很差。

```
FOR I:=1 TO N-1 DO      //改进布局
  BEGIN
    T:=I;
    FOR J:=I＋1 TO N DO
      IF A[J]＜A[T] THEN T:=J;
    IF T≠I THEN
      BEGIN
        WORK:=A[T];
        A[T]:=A[I];
        A[I]:=WORK;
      END
  END;
```

2. 程序编写首先应当考虑清晰性

- 程序编写首先应当考虑清晰性，不要刻意追求技巧性，使程序编写得过于紧凑。
- 例如，有一个用Pascal语句写出的程序段：

```
A[I]:=A[I]＋A[T];
A[T]:=A[I]－A[T];
A[I]:=A[I]－A[T];
```

- 此段程序可能不易看懂，有时还需用实际数据试验一下。
- 实际上，这段程序的功能就是交换A[I]和A[T]中的内容。目的是为了节省一个工作单元。如果改一下：

```
WORK:=A[T];
A[T]:=A[I];
A[I]:=WORK;
```

就能让读者一目了然了。

3. 程序要能直截了当地说明程序员的用意。

4. 除非对效率有特殊的要求，程序编写要做到**清晰第一，效率第二**。不要为了追求效率而丧失了清晰性。事实上，**程序效率的提高主要应通过选择高效的算法**来实现。

5. 首先要保证**程序正确**，然后才要求**提高速度**。反过来说，在使程序高速运行时，首先要保证它是正确的。

6. **避免使用临时变量**而使可读性下降。例如，有的程序员为了追求效率，往往喜欢把表达式

```
A[I]＋1 / A[I];
```

写成

```
AI＝A[I];
X＝AI＋1 / AI;
```

这样将一句分成两句写，会产生意想不到的问题。

7. 让编译程序做简单的优化。
8. 尽可能使用**库函数**
9. 避免不必要的转移。同时如果能保持程序可读性，则**不必用 GO TO语句**。
10. 尽量只采用三种基本的控制结构来编写程序。除顺序结构外，使用IF-THEN-ELSE来实现选择结构；使用DO-UNTIL或DO-WHILE来实现循环结构。

11. 避免使用空的ELSE语句和IF… THEN IF…的语句。这种结构容 易使读者产生误解。例如，

```
IF ( CHAR＞＝'A' ) THEN
 IF ( CHAR＜＝'Z' ) THEN
 PRINT “This is a letter。”
 ELSE
 PRINT “This is not a letter。”
```

可能产生二义性问题。

12. **避免采用过于复杂的条件测试**。
13. **尽量减少使用“否定”条件的条件语句。例如，如果在程序中出现**

```
 IF NOT ( ( CHAR＜‘0’ ) OR
( CHAR＞‘9’ ) ) THEN ……
```

改成

```
 IF ( CHAR＞＝'0' ) AND
( CHAR＜＝'9' ) THEN ……
```

不要让读者绕弯子想。

14. 尽可能用通俗易懂的**伪码**来描述程序的流程，然后再翻译成必须使用的语言。
15. 数据结构要有利于程序的简化。
16. 要**模块化**，使模块功能尽可能单一化，模块间的耦合能够清晰可见。
17. 利用**信息隐蔽**，确保每一个模块的独立性。

18. 从**数据**出发去构造程序。
19. 不要修补不好的程序，要重新编写。也不要一味地追求代码的复用，要重新组织。
20. 对太大的程序，要分块编写、测试，然后再集成。
21. 对递归定义的数据结构尽量使用递归过程。

输入和输出

- 输入和输出信息是与用户的使用直接相关的。输入和输出的方式和格式应当尽可能方便用户的使用。一定要避免因设计不当给用户带来的麻烦。
- 因此，在软件需求分析阶段和设计阶段，就应基本确定输入和输出的风格。系统能否被用户接受，有时就取决于输入和输出的风格。

- 不论是**批处理的输入／输出方式**，还是**交互式的输入／输出方式**，在设计和程序编码时都应考虑下列原则：

1. 对所有的输入数据都要进行检验，识别错误的输入，以保证每个数据的有效性；
2. 检查输入项的各种重要组合的合理性，必要时报告输入状态信息；
3. 使得输入的步骤和操作尽可能简单，并保持简单的输入格式；

4. 输入数据时，应允许使用自由格式输入；
5. 应允许缺省值；
6. 输入一批数据时，最好使用输入结束标志，而不要由用户指定输入数据数目；
7. 在交互式输入输入时，要在屏幕上使用提示符明确提示交互输入的请求，指明可使用选择项的种类和取值范围。同时，在数据输入的过程中和输入结束时，也要在屏幕上给出状态信息；

8. 当程序设计语言对输入／输出格式有严格要求时，应保持输入格式与输入语句的要求的一致性；
9. 给所有的输出加注解，并设计输出报表格式。
输入／输出风格还受到许多其它因素的影响。如输入／输出设备（例如终端的类型，图形设备，数字化转换设备等）、用户的熟练程度、以及通信环境等。

程序效率

- **讨论效率的准则**
  程序的效率是指**程序的执行速度**及**程序所需占用的内存的存储空间**。程序编码是最后提高运行速度和节省存储的机会，因此在此阶段不能不考虑程序的效率。让我们首先明确讨论程序效率的几条准则

- 效率是一个性能要求，应当在需求分析阶段给出。**软件效率以需求为准**，不应以人力所及为准。
- 好的设计可以提高效率。
- 程序的**效率与程序的简单性**相关。
- 一般说来，任何对效率无重要改善，且对程序的简单性、可读性和正确性不利的程序设计方法都是不可取的。

算法对效率的影响

- 源程序的**效率与详细设计阶段确定的算法的效率直接有关**。在详细设计翻译转换成源程序代码后，算法效率反映为程序的执行速度和存储容量的要求。
- 设计向程序转换过程中的指导原则：

① 在编程序前，尽可能化简有关的算术表达式和逻辑表达式；

② 仔细检查算法中的嵌套的循环，尽可能将某些语句或表达式移到循环外面；

③ 尽量避免使用多维数组；

④ 尽量避免使用指针和复杂的表；

⑤ 采用“快速”的算术运算；

⑥ 不要混淆数据类型，避免在表达式中出现类型混杂；

⑦ 尽量采用整数算术表达式和布尔表达式；

⑧ 选用等效的高效率算法；

- 许多编译程序具有“优化”功能，可以自动生成高效率的目标代码。

影响存储器效率的因素

- 在大中型计算机系统中，存储限制不再是主要问题。在这种环境下，对**内存采取基于操作系统的分页功能的虚拟存储管理**。**存储效率与操作系统的分页功能直接有关**。

- 采用结构化程序设计，**将程序功能合理分块，使每个模块或一组密切相关模块的程序体积大小与每页的容量相匹配**，可减少页面调度，减少内外存交换，提高存储效率。

- 在微型计算机系统中，存储器的容量对软件设计和编码的制约很大。因此**要选择可生成较短目标代码且存储压缩性能优良的编译程序**，有时需采用汇编程序。
- 提高存储器效率的关键是程序的简单性。

影响输入／输出的因素

- 输入／输出可分为两种类型：
  - 面向人(操作员)的输入／输出
  - 面向设备的输入／输出
- 如果操作员能够十分方便、简单地录入输入数据，或者能够十分直观、一目了然地了解输出信息，则可以说面向人的输入／输出是高效的。

- 关于面向设备的输入/输出，可以提出一些提高输入/输出效率的指导原则：
  - 输入/输出的请求应当最小化；
  - 对于所有的输入/输出操作，**安排适当的缓冲区**，以减少频繁的信息交换。
  - 对辅助存储(例如磁盘)，**选择尽可能简单的，可接受的存取方法**；

  - 对辅助存储的输入/输出，应当**成块传送**；
  - **对终端或打印机的输入/输出，应考虑设备特性**，尽可能改善输入/输出的质量和速度；
  - 任何不易理解的，对改善输入/输出效果关系不大的措施都是不可取的；
  - 任何不易理解的所谓“超高效”的输入/输出是毫无价值的；

程序复杂性度量

- 程序复杂性主要指**模块内程序的复杂性**。它直接关联到软件开发费用的多少，开发周期的长短和软件内部潜伏错误的多少。
- 减少程序复杂性，可提高软件的简单性和可理解性，并使软件开发费用减少，开发周期缩短，软件内部潜藏错误减少。

复杂性度量需要满足的假设

- 为了度量程序复杂性，要求：
  - 它可以用来计算任何一个程序的复杂性；
  - 对于不合理的程序，例如对于长度动态增长的程序，或者对于原则上无法排错的程序，不应当使用它进行复杂性计算；
  - 如果程序中指令条数、附加存储量、计算时间增多，不会减少程序的复杂性。

代码行度量法

- 源代码行数度量法基于两个前提：
  - **程序复杂性随着程序规模的增加不均衡地增长；**
  - **控制程序规模的方法最好是采用分而治之的办法。将一个大程序分解成若干个简单的可理解的程序段。**

- 方法的基本考虑是统计一个程序模块的源代码行数目，并以源代码行数做为程序复杂性的度量。
- 设每行代码的出错率为每100行源程序中可能有的错误数目。

McCabe度量法

- McCabe度量法，又称环路复杂性度量，是一种基于程序控制流的复杂性度量方法。
- 它基于一个程序模块的程序图中环路的个数，因此计算它先要画出程序图。
- 程序图是退化的程序流程图。流程图中每个处理都退化成一个结点，流线变成连接不同结点的有向弧。

- 程序图仅描述程序内部的控制流程，完全不表现对数据的具体操作，以及分支和循环的具体条件。
- **计算环路复杂性的方法：**根据图论，在一个强连通的有向图G中，环的个数由以下公式给出：

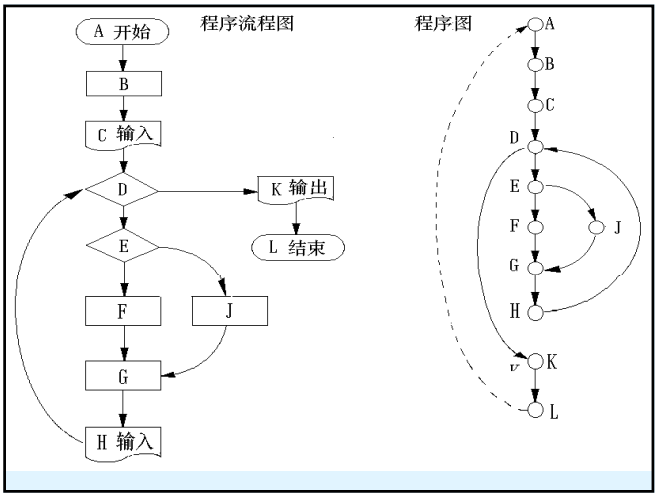
$$V(G)=m-n+p$$

其中，$V(G)$是有向图G中环路个数，m是图G中弧数，n是图G中结点数，p是图G中的强连通分量个数。

- 为使图成为强连通图，从图的入口点到出口点加一条用虚线表示的有向边，使图成为强连通图。这样就可以使用上式计算环路复杂性。
- 在例示中，结点数$n=11$，弧数$m=13$，$p=1$，则有

$$V(G)=m-n+p=13-11+1=3.$$

- 等于程序图中弧所封闭的区域数。



- 这种度量的缺点是：
  - 对于不同种类的控制流的复杂性不能区分
  - 简单IF语句与循环语句的复杂性同等看待
  - 嵌套IF语句与简单CASE语句的复杂性是一样的
  - 模块间接口当成一个简单分支一样处理
  - 一个具有1000行的顺序程序与一行语句的复杂性相同

第六部分　软件测试

- 软件测试的定义
- 软件测试基础
- 软件测试用例设计
- 软件测试策略
- 软件测试种类
- 程序调试

软件测试的定义

软件测试是在软件投入运行前，对软件需求分析，设计规格说明和编码的最终复审，是软件质量保证的关键步骤。
如果下定义：软件测试是为了发现错误而执行程序的过程。或者说软件测试是根据软件开发各阶段的规格说明和程序的内部结构而精心设计一批测试用例，并利用这些测试用例去运行程序，以发现程序错误的过程。

软件测试的基础

- 软件测试的目的
- 软件测试的原则
- 软件测试的对象
- 测试信息流
- 测试与软件开发各阶段的关系

软件测试的目的

- 基于不同的立场，存在着两种完全不同的测试目的。
- 从用户的角度出发，普遍希望通过软件测试暴露软件中隐藏的错误和缺陷，以考虑是否可接受该产品。
- 从软件开发者的角度出发，则希望测试成为表明软件产品中不存在错误的过程，验证该软件已正确地实现了用户的要求，确立人们对软件质量的信心。

Myers软件测试目的

(1) 测试是程序的执行过程，目的在于发现错误；
(2) 一个好的测试用例在于能发现至今未发现的错误；
(3) 一个成功的测试是发现了至今未发现的错误的测试。

- 换言之，测试的目的是
  - 系统地找出软件中潜在的各种错误和缺陷。
  - 能够证明软件的功能和性能与需求说明相符合。
  - 测试不能表明软件中不存在错误，它只能说明软件中存在错误。

软件测试的原则

1. 应当把“尽早地和不断地进行软件测试”作为软件开发者的座右铭。
2. 测试用例应由测试输入数据和对应的预期输出结果这两部分组成。
3. 程序员应避免检查自己的程序。
4. 在设计测试用例时，应当包括合理的输入条件和不合理的输入条件。

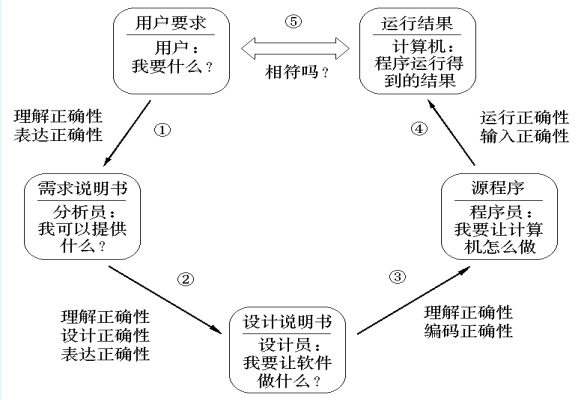
5. **充分注意测试中的群集现象。**
经验表明，测试后**程序中残存的错误数目与该程序中已发现的错误数目成正比**。
6. **严格执行测试计划，排除测试的随意性。**
7. 应当对每一个测试结果做全面检查。
8. 妥善保存测试计划，测试用例，出错统计和最终分析报告，为维护提供方便。

软件测试的对象

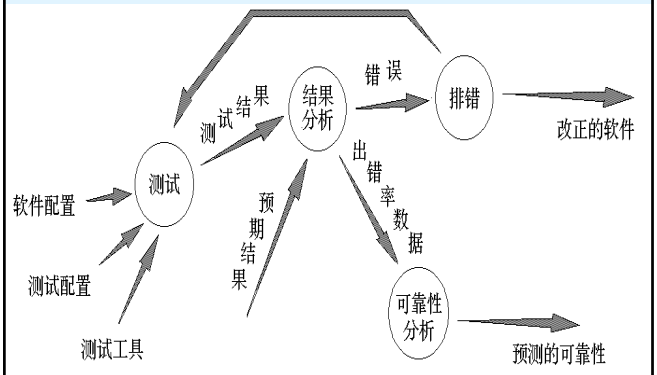
- 软件测试并不等于程序测试。**软件测试应贯穿于软件定义与开发的整个期间**。
- **需求分析、概要设计、详细设计以及程序编码**等各阶段所得到的**文档**，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，**都应成为软件测试的对象**。

- 为把握软件开发各个环节的正确性，需要进行各种**确认**和**验证**工作。
- **确认(Validation)**，是一系列的活动和过程，目的是想证实在一个给定的外部环境中软件的逻辑正确性。
  - 需求规格说明的确认
  - 程序的确认
- **验证(Verification)**，试图证明在软件生存期各个阶段，以及阶段间的逻辑协调性、完备性和正确性。

软件生存期各阶段之间需要保持的正确性



测试信息流



测试信息流

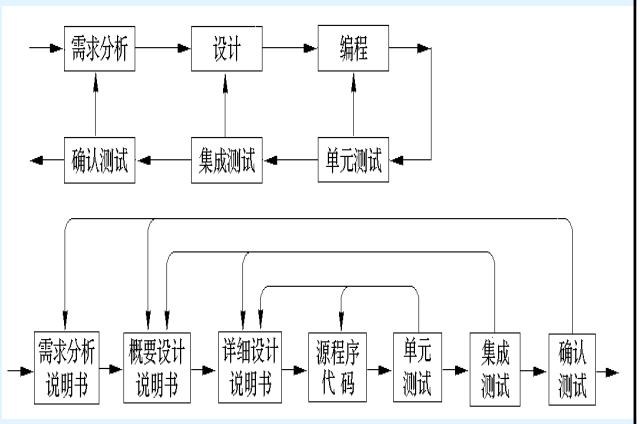
- **软件配置**：软件需求规格说明、软件设计规格说明、源代码等；
- **测试配置**：测试计划、测试用例、测试程序等；
- **测试工具**：测试数据自动生成程序、静态分析程序、动态分析程序、测试结果分析程序、以及驱动测试的测试数据库等等。

- **测试结果分析**：比较实测结果与预期结果，评价错误是否发生。
- **排错(调试)**：对已经发现的错误进行错误定位和确定出错性质，并改正这些错误，同时修改相关的文档。
- **修正后的文档再测试**：直到通过测试为止。

- 通过收集和分析测试结果数据，对软件建立可靠性模型
- 利用可靠性分析，评价软件质量：
  - **软件的质量和可靠性达到可以接受的程度；**
  - **所做的测试不足以发现严重的错误；**
- 如果测试发现不了错误，可以肯定，测试配置考虑得不够细致充分，错误仍然潜伏在软件中。

测试与软件开发各阶段的关系

- 软件开发过程是一个自顶向下，逐步细化的过程
- 软件计划阶段定义软件作用域
- 软件需求分析建立软件信息域、功能和性能需求、约束等
- 软件设计把设计用某种程序设计语言转换成程序代码
- 测试过程是依相反顺序安排的自底向上，逐步集成的过程。



软件测试用例设计

- 两种常用的测试方法
  - 黑盒测试
  - 白盒测试

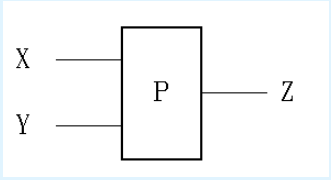
黑盒测试

- 这种方法是把**测试对象**看做**一个黑盒子**，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。
- 黑盒测试又叫做**功能测试**或**数据驱动测试**。

- 黑盒测试方法是在程序接口上进行测试，主要是为了发现以下错误：
  - 是否有不正确或遗漏了的功能?
  - 在接口上，输入能否正确地接受?能否输出正确的结果?
  - 是否有数据结构错误或外部信息(例如数据文件)访问错误?
  - 性能上是否能够满足要求?
  - 是否有初始化或终止性错误?

- 用黑盒测试发现程序中的错误，必须在**所有可能的输入条件和输出条件**中确定测试数据，来检查程序是否都能产生正确的输出。
- 但这是**不可能**的。

- 假设一个**程序P**有**输入量X**和**Y**及**输出量Z**。在字长为32位的计算机上运行。若**X**、**Y**取整数，按黑盒方法进行穷举测试：
- 可能采用的测试数据组：
  $2^{32}\times 2^{32}=2^{64}$

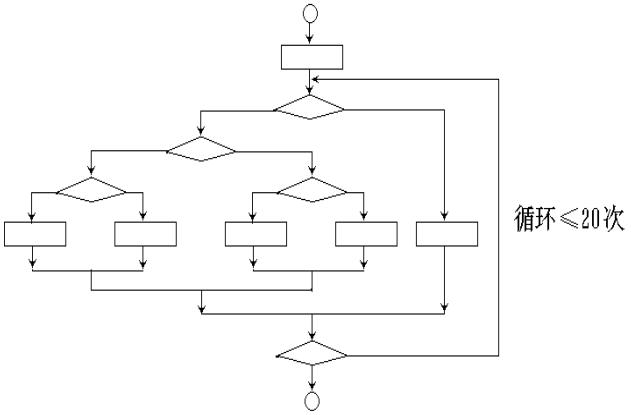


- 如果测试一组数据需要1毫秒，一年工作365×24小时，完成所有测试需5亿年。

白盒测试

- 此方法**把测试对象看做一个透明的盒子**，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。
- 通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。因此白盒测试又称为结构测试或逻辑驱动测试。

- 软件人员使用白盒测试方法，主要想对程序模块进行如下的检查：
  - 对程序模块的所有独立的执行路径至少测试一次；
  - 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次；
  - 在循环的边界和运行界限内执行循环体；
  - 测试内部数据结构的有效性，等。



- 对一个具有**多重选择和循环嵌套**的程序，**不同的路径数目可能是天文数字**。给出一个小程序的流程图，它包括了一个执行**20**次的循环。
- 包含的不同执行路径数达5^{20}条，对每一条路径进行测试需要**1**毫秒，假定一年工作**365 × 24**小时，要想把所有路径测试完，需**3170**年。

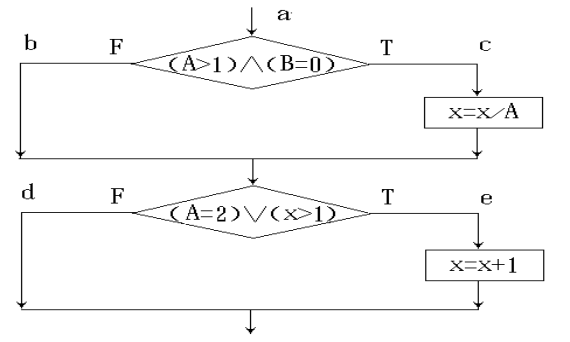
白盒测试的测试用例设计

逻辑覆盖

逻辑覆盖是以**程序内部的逻辑结构为基础**的设计测试用例的技术。它属白盒测试。

- **语句覆盖**
- **判定覆盖**
- **条件覆盖**
- **判定—条件覆盖**
- **条件组合覆盖**
- **路径覆盖。**

举例：所有路径为：L1(a->c->e) ,L2(a->b->d), L3(a->b->e), L4(a->c->d)



$$L1(a \to c \to e)$$
$$= \{(A>1)\ and\ (B=0)\}\ and\ \{(A=2)\ or\ (X/A>1)\}$$
$$= (A>1)\ and\ (B=0)\ and\ (A=2)\ or$$
$$(A>1)\ and\ (B=0)\ and\ (X/A>1)$$
$$= (A=2)\ and\ (B=0)\ or$$
$$(A>1)\ and\ (B=0)\ and\ (X/A>1)$$

$$L2(a \to b \to d)$$
$$= \overline{\{(A>1)\ and\ (B=0)\}}\ and\ \overline{\{(A=2)\ or\ (X>1)\}}$$
$$= \{\overline{(A>1)}\ or\ \overline{(B=0)}\}\ and\ \{\overline{(A=2)}\ and\ \overline{(X>1)}\}$$
$$= \overline{(A>1)}\ and\ \overline{(A=2)}\ and\ \overline{(X>1)}\ or$$
$$\overline{(B=0)}\ and\ \overline{(A=2)}\ and\ \overline{(X>1)}$$
$$= (A\le 1)\ and\ (X\le 1)\ or$$
$$(B\ne 0)\ and\ (A\ne 2)\ and\ (X\le 1)$$

$$L3(a \to b \to c)$$
$$= \overline{\{(A>1)\ and\ (B=0)\}}\ and\ \{(A=2)\ or\ (X>1)\}$$
$$= \{\overline{(A>1)}\ or\ \overline{(B=0)}\}\ and\ \{(A=2)\ or\ (X>1)\}$$
$$= \overline{(A>1)}\ \text{and}\ (X>1)\ \text{or}$$
$$\overline{(B=0)}\ \text{and}\ (A=2)\ \text{or}\ \overline{(B=0)}\ \text{and}\ (X>1)$$
$$= (A\le 1)\ and\ (X>1)\ or$$
$$(B\ne 0)\ and\ (A=2)\ or\ (B\ne 0)\ and\ (X>1)$$

L4 $(a \to c \to d)$

$= \{(A>1)\ and\ (B=0)\}\ and\ \overline{\{(A=2)\ or\ (X/A>1)\}}$

$= (A>1)\ and\ (B=0)\ and\ (A\neq 2)\ and\ (X/A\leq 1)$

依据以上推导出来的结果就可以设计满足要求的测试用例。

语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得**每一可执行语句至少执行一次**。
- 在图例中，正好所有的可执行语句都在**路径L1**上，所以选择**路径 L1**设计测试用例，就可以覆盖所有的可执行语句。

- 测试用例的设计格式如下
  【输入的(A, B, X)，输出的(A, B, X)】
- 为图例设计满足**语句覆盖**的测试用例是:
  【(2, 0, 4)，(2, 0, 3)】
  覆盖 ace【L1】

$$(A=2)\ and\ (B=0)\ or\ (A>1)\ and\ (B=0)\ and\ (X/A>1)$$

判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得**程序中每个判断的取真分支和取假分支至少经历一次**。
- 判定覆盖又称为**分支覆盖**。
- 对于图例，如果选择**路径L1**和**L2**，就可得满足要求的测试用例:

- 【(2, 0, 4)，(2, 0, 3)】覆盖 ace【L1】
  【(1, 1, 1)，(1, 1, 1)】覆盖 abd【L2】

$$(A=2)\ and\ (B=0)\ or\ (A>1)\ and\ (B=0)\ and\ (X/A>1)$$

$$(A\leq 1)\ and\ (X\leq 1)\ or\ (B\neq 0)\ and\ (A\neq 2)\ and\ (X\leq 1)$$

- 如果选择路径L3和L4，还可得另一组可用的测试用例:
  【(2, 1, 1)，(2, 1, 2)】覆盖 abe【L3】
  【(3, 0, 3)，(3, 1, 1)】覆盖 acd【L4】

$$(A\leq 1)\ and\ (X>1)\ or\ (B\neq 0)\ and\ (A=2)\ or\ (B\neq 0)\ and\ (X>1)$$

$$(A>1)\ and\ (B=0)\ and\ (A\neq 2)\ and\ (X/A\leq 1)$$

条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中**每个判断的每个条件的可能取值至少执行一次**。
- 在图例中，我们事先可对所有条件的取值加以标记。例如，
- 对于第一个判断：
  - 条件 A＞1 取真为 T_1，取假为 $\overline{T_1}$
    条件 B＝0 取真为 T_2 取假为 $\overline{T_2}$

- 对于第二个判断：
  - 条件A＝2 取真为T_3，取假为 $\overline{T_3}$
    条件X＞1 取真为T_4，取假为 $\overline{T_4}$

测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 0, 1)，(1, 0, 1)】	L2(b, d)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$

或

测 试 用 例	覆盖分支	条件取值
【(1, 0, 3)，(1, 0, 4)】	L3(b, e)	$T_1\overline{T_2}\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1T_2T_3T_4$

判定—条件覆盖

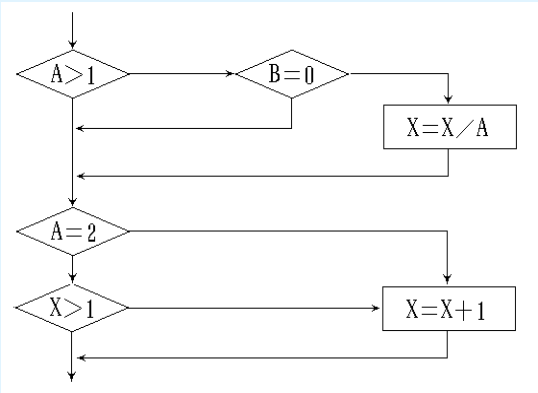
- 判定—条件覆盖就是设计足够的测试用例，使得**判断中每个条件的所有可能取值至少执行一次**，同时**每个判断中的每个条件的可能取值至少执行一次**。

测 试 用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】	L2(b, d)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$

$$(A=2)\ and\ (B=0)\ or\ (A>1)\ and\ (B=0)\ and\ (X/A>1)$$

$$(A\le 1)\ and\ (X\le 1)\ or\ (B\ne 0)\ and\ (A\ne 2)\ and\ (X\le 1)$$

由多个基本判断组成的流程图



条件组合覆盖

- 条件组合覆盖就是设计足够的测试用例，运行被测程序，使得**每个判断的所有可能的条件取值组合至少执行一次**。

  记 ① A＞1, B＝0 作 T_1T_2
  ② A＞1, B≠0 作 $T_1\overline{T_2}$
  ③ A≯1, B＝0 作 $\overline{T_1}T_2$
  ④ A≯1, B≠0 作 $\overline{T_1}\overline{T_2}$

⑤ A＝2, X＞1 作$T_3\overline{T_4}$
⑥ A＝2, X≯1 作$T_3\overline{T_4}$
⑦ A≠2, X＞1 作$\overline{T_3}T_4$
⑧ A≠2, X≯1 作$\overline{T_3}\overline{T_4}$

测 试 用 例	覆盖条件	覆盖组合
【(2, 0, 4), (2, 0, 3)】	(L1) $T_1T_2T_3T_4$	①, ⑤
【(2, 1, 1), (2, 1, 2)】	(L3) $T_1T_2T_3\overline{T_4}$	②, ⑥
【(1, 0, 3), (1, 0, 4)】	(L3) $\overline{T_1}T_2\overline{T_3}T_4$	③, ⑦
【(1, 1, 1), (1, 1, 1)】	(L2) $\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$	④, ⑧

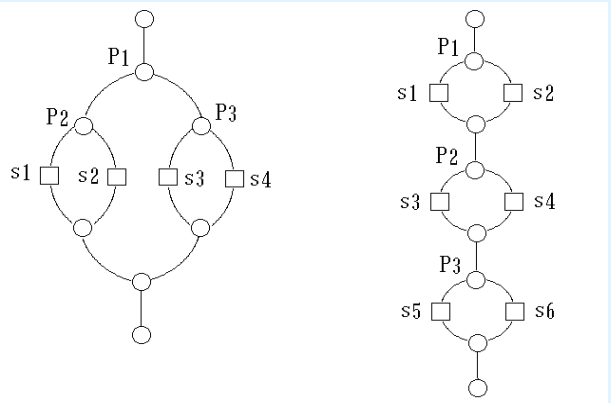
路径测试

- 路径测试就是设计足够的测试用例，**覆盖程序中所有可能的路径**。

测 试 用 例	通过路径	覆盖条件
【(2, 0, 4), (2, 0, 3)】	ace (L1)	$T_1T_2T_3T_4$
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$\overline{T_1}\overline{T_2}\overline{T_3}T_4$
【(3, 0, 3), (3, 0, 1)】	acd (L3)	$T_1T_2T_3\overline{T_4}$

条 件 测 试 路 径 选 择

- 当程序中判定多于一个时，形成的分支结构可以分为两类：**嵌套型分支结构**和**连锁型分支结构**。
- 对于嵌套型分支结构，若有n个判定语句，需要n+1个测试用例；
- 对于连锁型分支结构， 若有n个判定语句，需要有2^n个测试用例，覆盖它的2^n条路径。当n较大时将无法测试。



（a）嵌套型分支结构　　（b）连锁型分支结构

循环测试路径选择

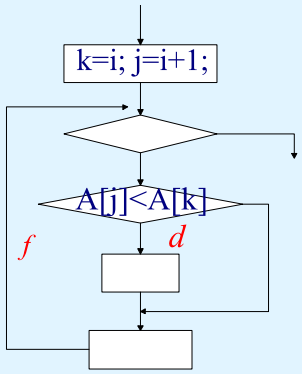
- 循环分为4种不同类型：**简单循环**、**连锁循环**、**嵌套循环**和**非结构循环**。

(1) 简单循环

① 零次循环：从循环入口到出口
② 一次循环：检查循环初始值
③ 二次循环：检查多次循环
④ m次循环： 检查在多次循环
⑤ 最大次数循环、比最大次数多一次、少一次的循环。

例：求最小值

```
k = i;
  ❶
for ( j = i+1;  j <= n;  j++ )
      ❷        ❸      ❻
  if ( A[j] < A[k] ) then  k = j;
        ❹              ❺
```



测试用例选择

循环	i	n	A[i]	A[i+1]	A[i+2]	k	路 径
0	1	1				i	ac
1	1	2	1	2		i	abefc
			2	1		i+1	abdfc
2	1	3	1	2	3	i	abefefc
			2	3	1	i+2	abefdfc
			3	2	1	i+2	abdfdfc
			3	1	2	i+1	abdfefc

d 改 k 的值，e 不改 k 的值

(2) 嵌套循环

① 对最内层循环做简单循环的全部测试。所有其它层的循环变量置为最小值；

② 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。。

③ 反复进行，直到所有各层循环测试完毕。

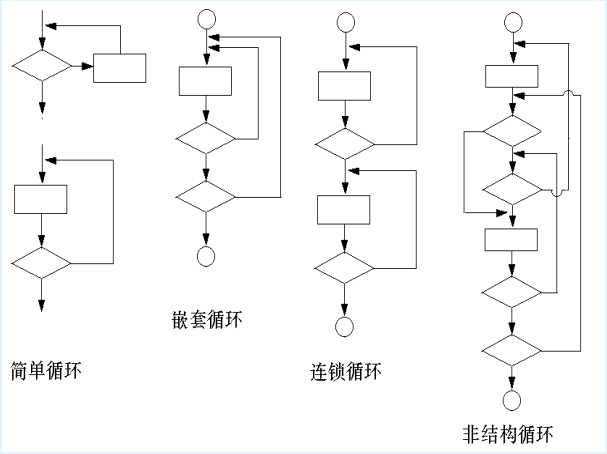
④ 对全部各层循环同时取最小循环次数，或者同时取最大循环次数

(3) 连锁循环

如果各个循环互相独立，则可以用与简单循环相同的方法进行测试。但如果几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。

(4) 非结构循环

这一类循环应该使用结构化程序设计方法重新设计测试用例。



黑盒测试的测试用例设计

- 等价类划分
- 边界值分析
- 错误推测法
- 因果图

等价类划分

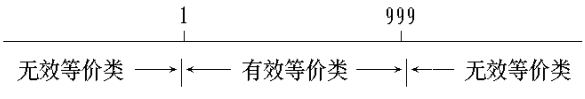
- 等价类划分是一种典型的黑盒测试方法，使用这一方法时，**完全不考虑程序的内部结构，只依据程序的规格说明来设计测试用例**。
- 等价类划分方法**把所有可能的输入数据**，即程序的输入域**划分成若干部分**，然后**从每一部分中选取少数有代表性的数据做为测试用例**。

- 使用这一方法设计测试用例要经历**划分等价类**（列出等价类表）和**选取测试用例**两步。
- **划分等价类**
  等价类是指某个输入域的子集合。在该子集合中，**各个输入数据对于揭露程序中的错误都是等效的**。测试某等价类的代表值就等价于对这一类其它值的测试。

- 等价类的划分有两种不同的情况：
  ① **有效等价类**：是指对于程序的规格说明来说，是合理的，有意义的输入数据构成的集合。
  ② **无效等价类**：是指对于程序的规格说明来说，是不合理的，无意义的输入数据构成的集合。
- 在设计测试用例时，要同时考虑有效等价类和无效等价类的设计。

- 划分等价类的原则。
  (1) 如果输入条件规定了取值范围，或值的个数，则可以确立一个有效等价类和两个无效等价类。

- 例如，在程序的规格说明中，对输入条件有一句话：
  “…… 项数可以从1到999 ……”
  则有效等价类是**“1≤项数≤999”**
  两个无效等价类是**“项数＜1”**或**“项数＞999”**。在数轴上表示成：



(2) 如果输入条件规定了输入值的集合，或者是规定了“必须如何”的条件，这时可确立一个有效等价类和一个无效等价类。

- 例如，在**Pascal**语言中对变量标识符规定为**“以字母打头的……串”**。那么所有以字母打头的构成有效等价类，而不在此集合内（不以字母打头）的归于无效等价类。

(3) 如果输入条件是一个布尔量，则可以确定一个有效等价类和一个无效等价类。

(4) 如果规定了输入数据的一组值，而且程序要对每个输入值分别进行处理。这时可为 每一个输入值确立一个有效等价类，此外针对这组值确立一个无效等价类，它是所有不允许的输入值的集合。

- 例如，在教师上岗方案中规定对教授、副教授、讲师和助教分别计算分数，做相应的处理。因此可以确定4个有效等价类为教授、副教授、讲师和助教，一个无效等价类，它是所有不符合以上身分的人员的输入值的集合。

(5) 如果规定了输入数据必须遵守的规则，则可以确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

- 例如，**Pascal**语言规定“一个语句必须以分号‘;’结束”。这时，可以确定一个有效等价类“以‘;’结束”，若干个无效等价类“以‘:’结束”、“以‘,’结束”、“以‘ ’结束”、“以**LF**结束”等。
- 确立测试用例
  在确立了等价类之后，建立等价类表，列出所有划分出的等价类。

输入条件	有效等价类	无效等价类
……	……	……
……	……	……

- 再从划分出的等价类中按以下原则选择测试用例：

(1) 为每一个等价类规定一个唯一编号；

(2) 设计一个新的测试用例，使其**尽可能多地覆盖尚未被覆盖的有效等价类**，重复这一步，直到所有的有效等价类都被覆盖为止；

（3）设计一个新的测试用例，使其**仅覆盖一个尚未被覆盖的无效等价类**，重复这一步，直到所有的无效等价类都被覆盖为止。

- 用等价类划分法设计测试用例的实例
  在某一**PASCAL**语言版本中规定：“标识符是由字母开头，后跟字母或数字的任意组合构成。有效字符数为8个，最大字符数为80个。”
  并且规定：“标识符必须先说明，再使用。”“在同一说明语句中，标识符至少必须有一个。”

用等价类划分的方法，建立输入等价类表：

输入条件	有效等价类	无效等价类
标识符个数	1个（1），多个（2）	0个（3）
标识符字符数	1~8个（4）	0个（5），>8个（6），>80个（7）
标识符组成	字母（8），数字（9）	非字母数字字符（10），保留字（11）
第一个字符	字母（12）	非字母（13）
标识符使用	先说明后使用（14）	未说明已使用（15）

- 下面选取了9个测试用例，它们覆盖了所有的等价类。

① VAR x，T1234567：REAL；
BEGIN x := 3.414；
T1234567 := 2.732；
……
(1), (2), (4), (8), (9), (12), (14)

② VAR ： REAL； (3)

③ VAR x，： REAL； (5)

④ VAR T12345678：REAL； (6)

⑤ VAR T12345……：REAL； (7)
多于80个字符

⑥ VAR T$：CHAR； (10)

⑦ VAR GOTO：INTEGER； (11)

⑧ VAR 2T：REAL； (13)

⑨ VAR PAR：REAL； (15)
BEGIN ……
PAP := SIN (3.14 * 0.8) / 6；

边界值分析

- 边界值分析也是一种黑盒测试方法，是对等价类划分方法的补充。
- 人们从长期的测试工作经验得知，**大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部**。因此针对各种边界情况设计测试用例，可以查出更多的错误。

- 比如，在做三角形计算时，要输入三角形的三个边长：A、B和C。 我们应注意到这三个数值应当满足
A＞0、B＞0、C＞0、
A＋B＞C、A＋C＞B、B＋C＞A，
才能构成三角形。但如果把六个不等式中的任何一个大于号“＞”错写成大于等于号“≥”，那就不能构成三角形。问题恰出现在容易被疏忽的边界附近。

- 这里所说的边界是指，相当于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。
- 使用边界值分析方法设计测试用例，首先应确定边界情况。**应当选取正好等于，刚刚大于，或刚刚小于边界的值做为测试数据**，而不是选取等价类中的典型值或任意值做为测试数据。

错误推测法

- 人们也可以靠经验和直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的例子。这就是错误推测法。
- 错误推测法的基本想法是：**列举出程序中所有可能有的错误和容易发生错误的特殊情况，根据它们选择测试用例**。

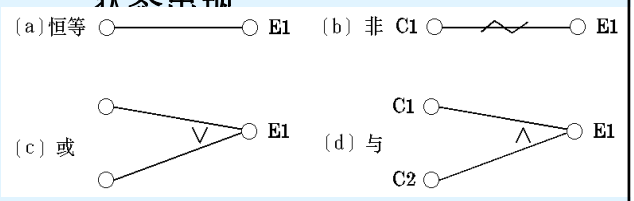
因果图

- 因果图的适用范围
  如果在测试时必须考虑**输入条件的各种组合**，可使用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来设计测试用例，这就需要利用因果图。
  因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

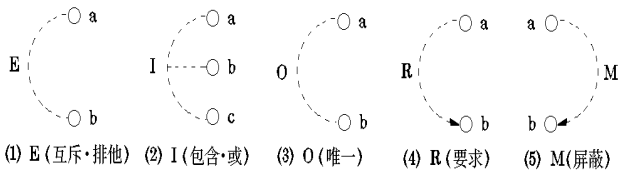
- 用因果图生成测试用例的基本步骤
  (1) 分析软件规格说明描述中，哪些是原因 (即输入条件或输入条件的等价类)，哪些是结果 (即输出条件)，并给每个原因和结果赋予一个标识符。
  (2) 分析软件规格说明描述中的语义，找出原因与结果之间，原因与原因之间对应的是什么关系? 根据这些关系，画出因果图。

(3) 由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出现。为表明这些特殊情况，在因果图上用一些记号标明约束或限制条件。
(4) 把因果图转换成判定表。
(5) 把判定表的每一列拿出来作为依据，设计测试用例。

- 在因果图中出现的基本符号
  通常在因果图中用Ci表示原因，用Ei表示结果，各结点表示状态，可取值“0”或“1”。“0”表示某状态不出现，“1”表示某状态出现



- 表示约束条件的符号
  为了表示原因与原因之间，结果与结果之间可能存在的约束条件，在因果图中可以附加一些表示约束条件的符号。



- 例如，有一个处理单价为5角钱的饮料的自动售货机软件测试用例的设计。其规格说明如下：
  若投入5角钱或1元钱的硬币，押下〖橙汁〗或〖啤酒〗的按钮，则相应的饮料就送出来。若售货机没有零钱找，则一个显示〖零钱找完〗的红灯亮，这时在投入1元硬币并押下按钮后，饮料不送出来而且1元硬币也退出来；若有零钱找，则显示〖零钱找完〗的红灯灭，在送出饮料的同时退还5角硬币。”

(1) 分析这一段说明，列出原因和结果

原因: 1. 售货机有零钱找

2. 投入1元硬币

3. 投入5角硬币

4. 押下橙汁按钮

5. 押下啤酒按钮

建立中间结点，表示处理中间状态

11. 投入1元硬币且押下饮料按钮

12. 押下〖橙汁〗或〖啤酒〗的按钮

13. 应当找5角零钱并且售货机有零钱找

14. 钱已付清

结果： 21. 售货机〖零钱找完〗灯亮

22. 退还1元硬币

23. 退还5角硬币

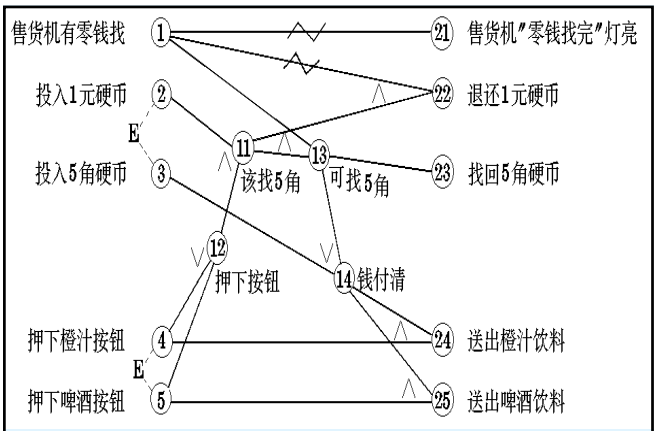
24. 送出橙汁饮料

25. 送出啤酒饮料

(2) 画出因果图。所有原因结点列在左边，所有结果结点列在右边。

(3) 由于 2 与 3，4 与 5 不能同时发生，分别加上约束条件E。

(4) 因果图

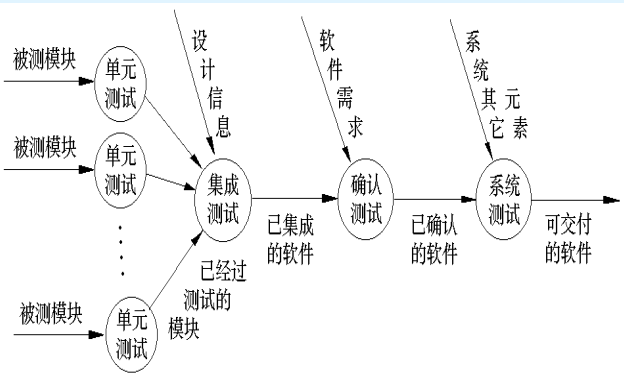


(5) 转换成判定表

序号		1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1	2
条件	①	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	②	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	③	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0
	④	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	⑤	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
中间结果	⑪						1	1	0		0	0	0		0	0	0						1	1	0		0	0	0		0	0	0
	⑫						1	1	0		1	1	0		1	1	0						1	1	0		1	1	0		1	1	0
	⑬						1	1	0		0	0	0		0	0	0						0	0	0		0	0	0		0	0	0
	⑭						1	1	0		1	1	1		0	0	0						0	0	0		1	1	1		0	0	0
结果	㉑						0	0	0		0	0	0		0	0	0						1	1	1		1	1	1		1	1	1
	㉒						0	0	0		0	0	0		0	0	0						1	1	0		0	0	0		0	0	0
	㉓						1	1	0		0	0	0		0	0	0						0	0	0		0	0	0		0	0	0
	㉔						1	0	0		1	0	0		0	0	0						0	0	0		1	0	0		0	0	0
	㉕						0	1	0		0	1	0		0	0	0						0	0	0		0	1	0		0	0	0
测试用例							Y	Y	Y		Y	Y	Y		Y	Y							Y	Y	Y		Y	Y	Y		Y	Y	

软件测试的策略

- 测试过程按4个步骤进行，即**单元测试**、**组装测试**、**确认测试**和**系统测试**。
- 开始是**单元测试**，集中对用源代码实现的每一个程序单元进行测试，检查各个程序模块是否正确地实现了规定的功能。



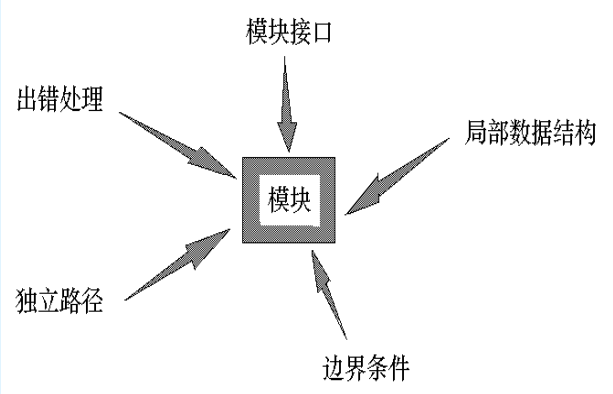
- **组装测试**把已测试过的模块组装起来，主要对与设计相关的软件体系结构的构造进行测试。
- **确认测试**则是要检查已实现的软件是否满足了需求规格说明中确定了的各种需求，以及软件配置是否完全、正确。
- **系统测试**把已经经过确认的软件纳入实际运行环境中，与其它系统成份组合在一起进行测试。

单元测试 (Unit Testing)

- 单元测试又称模块测试，**是针对软件设计的最小单位 — 程序模块，进行正确性检验**的测试工作。其目的在于发现各模块内部可能存在的各种差错。
- 单元测试需要**从程序的内部结构出发设计测试用例**。多个模块可以平行地独立进行单元测试。

1. 单元测试的内容

- 在单元测试时，测试者需要依据详细设计说明书和源程序清单，了解该模块的I/O条件和模块的逻辑结构，主要采用白盒测试的测试用例，辅之以黑盒测试的测试用例，使之对任何合理的输入和不合理的输入，都能鉴别和响应。



(1) 模块接口测试

- 在单元测试的开始，应对**通过被测模块的数据流**进行测试。测试项目包括：
  - 调用本模块的输入参数是否正确；
  - 本模块调用子模块时输入给子模块的参数是否正确；
  - 全局量的定义在各模块中是否一致；

- 在做**内外存交换**时要考虑：
  - 文件属性是否正确；
  - OPEN与CLOSE语句是否正确；
  - 缓冲区容量与记录长度是否匹配；
  - 在进行读写操作之前是否打开了文件；
  - 在结束文件处理时是否关闭了文件；
  - 正文书写 / 输入错误，
  - I / O错误是否检查并做了处理。

(2) 局部数据结构测试

- 不正确或不一致的数据类型说明
- 使用尚未赋值或尚未初始化的变量
- 错误的初始值或错误的缺省值
- 变量名拼写错或书写错
- 不一致的数据类型
- 全局数据对模块的影响

(3) 路径测试

- 选择适当的测试用例，对模块中**重要的执行路径**进行测试。
- 应当设计测试用例查找由于**错误的计算**、**不正确的比较**或**不正常的控制流**而导致的错误。
- 对基本执行路径和循环进行测试可以发现大量的路径错误。

(4) 错误处理测试

- 出错的描述是否难以理解
- 出错的描述是否能够对错误定位
- 显示的错误与实际的错误是否相符
- 对错误条件的处理正确与否
- 在对错误进行处理之前，错误条件是否已经引起系统的干预等

(5) 边界测试

- 注意数据流、控制流中刚好等于、大于或小于确定的比较值时出错的可能性。对这些地方要仔细地选择测试用例，认真加以测试。
- 如果对模块运行时间有要求的话，还要专门进行关键路径测试，以确定最坏情况下和平均意义下影响模块运行时间的因素。

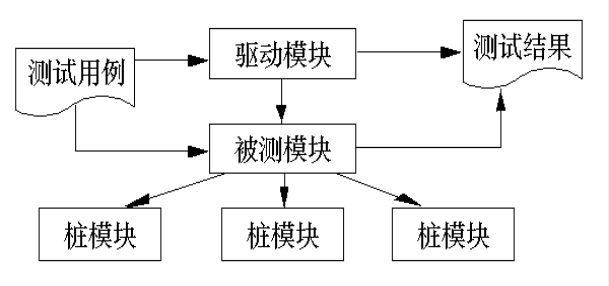
2. 单元测试的步骤

- 模块并不是一个独立的程序，在考虑测试模块时，同时要考虑它和外界的联系，用一些辅助模块去模拟与被测模块相联系的其它模块。
  - **驱动模块 (driver)**
  - **桩模块 (stub) —— 存根模块**

驱动模块 (driver) —— 相当于所测模块的主程序。它接收测试数据，把这些数据传送给所测模块，最后再输出实测结果。

桩模块 (stub) —— 存根模块。用以代替所测模块调用的子模块。

单元测试的测试环境



组装测试（Integrated Testing）

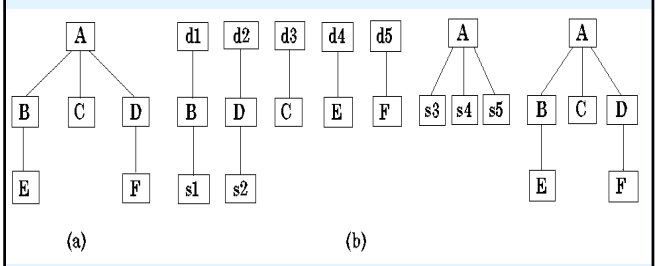
- 组装测试 (集成测试、联合测试）
- 通常，在单元测试的基础上，需要将所有模块按照设计要求组装成为系统。这时需要考虑的问题是：
  - 在把各个模块连接起来的时候，**穿越模块接口的数据**是否会丢失；
  - **一个模块的功能是否会对另一个模块的功能产生不利的影响**；
  - **各个子功能组合起来，能否达到预期要求的父功能**；
  - **全局数据结构是否有问题**；
  - **单个模块的误差累积起来，是否会放大**，从而达到不能接受的程度。

在单元测试的同时可进行组装测试，发现并排除在模块连接中可能出现的问题，最终构成要求的软件系统。

- 子系统的组装测试特别称为**部件测试**，它所做的工作是要找出组装后的**子系统与系统需求规格说明之间**的不一致。
- 通常，把模块组装成为系统的方式有两种
  - 一次性组装方式
  - 增殖式组装方式

1. 一次性组装方式 (big bang)

- 它是一种非增殖式组装方式。也叫做整体拼装。
- 使用这种方式，首先对每个模块分别进行模块测试，然后再把所有模块组装在一起进行测试，最终得到要求的软件系统。

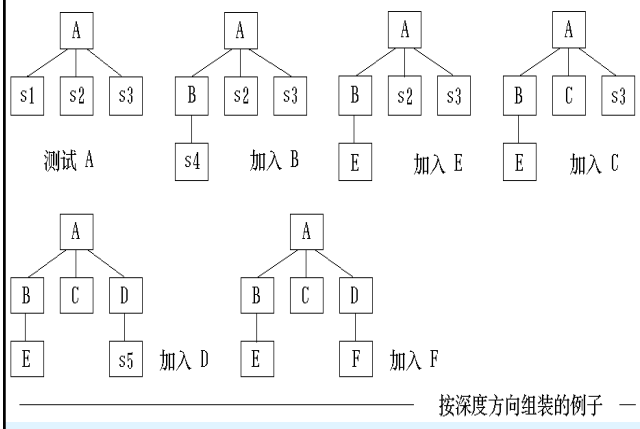


2. 增殖式组装方式

- 这种组装方式又称**渐增式组装**
- 首先对一个个模块进行模块测试，然后将这些模块逐步组装成较大的系统
- 在组装的过程中边连接边测试，以发现连接过程中产生的问题
- 通过增殖逐步组装成为要求的软件系统。

(1) 自顶向下的增殖方式

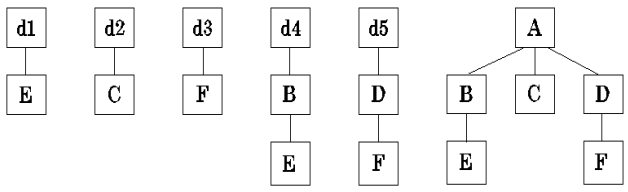
- 这种组装方式将模块**按系统程序结构，沿控制层次自顶向下进行组装**。
- 自顶向下的增殖方式在测试过程中较早地验证了主要的控制和判断点。
- 选用按深度方向组装的方式，可以首先实现和验证一个完整的软件功能。



按深度方向组装的例子

(2) 自底向上的增殖方式

- 这种组装的方式是从**程序模块结构的最底层的模块开始组装和测试**。
- 因为模块是自底向上进行组装，对于一个给定层次的模块，它的子模块（包括子模块的所有下属模块）已经组装并测试完成，所以**不再需要桩模块**。在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到。



- 自顶向下增殖的方式和自底向上增殖的方式各有优缺点。
- 一般来讲，一种方式的优点是另一种方式的缺点。

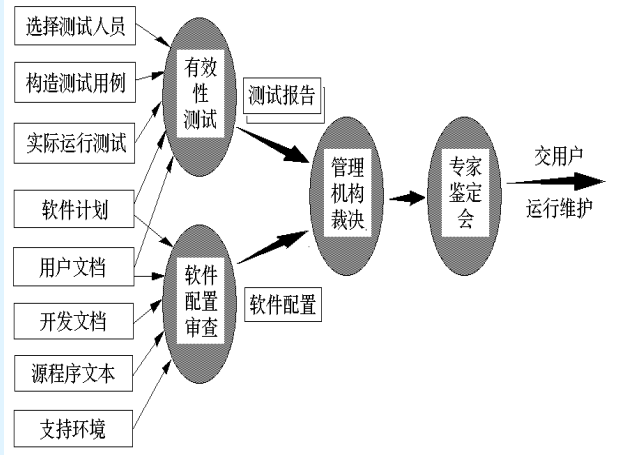
(3) 混合增殖式测试

- **衍变的自顶向下的增殖测试**
  - 首先对输入 / 输出模块和引入新算法模块进行测试;
  - 再自底向上组装成为功能相当完整且相对独立的子系统;
  - 然后由主模块开始自顶向下进行增殖测试。

- **自底向上–自顶向下的增殖测试**
  - 首先对含读操作的子系统自底向上直至根结点模块进行组装和测试;
  - 然后对含写操作的子系统做自顶向下的组装与测试。
- **回归测试**
  - 这种方式采取自顶向下的方式测试被修改的模块及其子模块;
  - 然后将这一部分视为子系统，再自底向上测试。

确认测试（Validation Testing）

- 确认测试又称**有效性测试**。任务是验证软件的功能和性能及其它特性是否与用户的要求一致。
- 对软件的功能和性能要求在软件需求规格说明书中已经明确规定。它包含的信息就是软件确认测试的基础。



1. 进行有效性测试（黑盒测试）

- 有效性测试是在模拟的环境 (可能就是开发的环境) 下，运用黑盒测试的方法，验证被测软件是否满足需求规格说明书列出的需求。
- 首先制定测试计划，规定要做测试的种类。还需要制定一组测试步骤，描述具体的测试用例。

- 通过实施预定的测试计划和测试步骤，确定
  - 软件的特性是否与需求相符;
  - 所有的文档都是正确且便于使用;
  - 同时，对其它软件需求，例如可移植性、兼容性、出错自动恢复、可维护性等，也都要进行测试

- 在全部软件测试的测试用例运行完后，所有的测试结果可以分为两类:
  - **测试结果与预期的结果相符**。这说明软件的这部分功能或性能特征与需求规格说明书相符合，从而这部分程序被接受。
  - **测试结果与预期的结果不符**。这说明软件的这部分功能或性能特征与需求规格说明不一致，因此要为它提交一份问题报告。

2. 软件配置复查

- 软件配置复查的目的是保证
  - 软件配置的所有成分都齐全；
  - 各方面的质量都符合要求；
  - 具有维护阶段所必需的细节；
  - 而且已经编排好分类的目录。
- 应当严格遵守用户手册和操作手册中规定的使用步骤，以便检查这些文档资料的完整性和正确性。

3. α 测试和 β 测试

- 在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。
- **α 测试**是由一个**用户在开发环境下进行的测试**，也可以是**公司内部的用户在模拟实际操作环境下进行的测试**。

- **α 测试**的目的是评价软件产品的FLURPS（即功能、局域化、可使用性、可靠性、性能和支持）。尤其注重产品的界面和特色。
- **α 测试**可以从软件产品编码结束之时开始，或在模块（子系统）测试完成之后开始，也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。

- **β 测试**是由软件的**多个用户在实际使用环境下进行的测试**。这些用户返回有关错误信息给开发者。
- 测试时，开发者通常不在测试现场。因而，**β 测试**是在开发者无法控制的环境下进行的软件现场应用。
- 在 β 测试中，由用户记下遇到的所有问题，包括真实的以及主观认定的，定期向开发者报告。

- **β 测试**主要衡量产品的FLURPS。着重于产品的支持性，包括文档、客户培训和支持产品生产能力。
- 只有当**α 测试**达到一定的可靠程度时，才能开始**β 测试**。它处在整个测试的最后阶段。同时，产品的所有手册文本也应该在此阶段完全定稿。

4.验收测试（Acceptance Testing）

- **在通过了系统的有效性测试及软件配置审查之后，就应开始系统的验收测试。**
- **验收测试是以用户为主的测试。软件开发人员和QA（质量保证）人员也应参加。**
- **由用户参加设计测试用例，使用生产中的实际数据进行测试。**

- 在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。
- 确认测试应交付的文档有：
  - 确认测试分析报告
  - 最终的用户手册和操作手册
  - 项目开发总结报告。

系统测试（System Testing）

- 系统测试，是将通过确认测试的软件，作为整个基于计算机系统的一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其它系统元素结合在一起，在实际运行环境下，对计算机系统进行一系列的组装测试和确认测试。
- 系统测试的目的在于通过与系统的需求定义作比较，发现软件与系统的定义不符合或与之矛盾的地方。

测试种类

- 软件测试是由一系列不同的测试组成。主要目的是对以计算机为基础的系统进行充分的测试。

功能测试

功能测试是在规定的一段时间内运行软件系统的所有功能，以验证这个软件系统有无严重错误。

可靠性测试

如果系统需求说明书中有对可靠性的要求，则需进行可靠性测试。

① 平均失效间隔时间 MTBF (Mean Time Between Failures) 是否超过规定时限?

② 因故障而停机的时间 MTTR (Mean Time To Repairs) 在一年中应不超过多少时间。

强度测试

强度测试是要检查在系统运行环境不正常乃至发生故障的情况下，系统可以运行到何种程度的测试。

- 强度测试的一个变种就是敏感性测试。在程序有效数据界限内一个小范围内的一组数据可能引起极端的或不平稳的错误处理出现，或者导致极度的性能下降的情况发生。此测试用以发现可能引起这种不稳定性或不正常处理的某些数据组合。

性能测试

性能测试是要检查系统是否满足在需求说明书中规定的性能。特别是对于实时系统或嵌入式系统。

性能测试常常需要与强度测试结合起来进行，并常常要求同时进行硬件和软件检测。

- 通常，对软件性能的检测表现在以下几个方面：响应时间、吞吐量、辅助存储区，例如缓冲区，工作区的大小等、处理精度，等等。

恢复测试

恢复测试是要证实在**克服硬件故障**(包括掉电、硬件或网络出错等)**后，系统能否正常地继续进行工作**，并不对系统造成任何损害。

- 为此，可采用各种人工干预的手段，模拟硬件故障，故意造成软件出错。并由此检查：
  - **错误探测功能**——系统能否发现硬件失效与故障；
  - 能否**切换或启动备用的硬件**；
  - 在故障发生时能否**保护正在运行的作业和系统状态**；
  - 在系统恢复后能否**从最后记录下来的无错误状态开始继续执行作业**，等等。
  - **掉电测试**：其目的是测试软件系统在发生电源中断时能否**保护当时的状态且不毁坏数据，然后在电源恢复时从保留的断点处重新进行操作。**

启动／停止测试

这类测试的目的是验证**在机器启动及关机阶段，软件系统正确处理的能力。**

这类测试包括

- **反复启动软件系统** (例如，操作系统自举、网络的启动、应用程序的调用等)
- **在尽可能多的情况下关机**。

配置测试

- 这类测试是要检查**计算机系统内各个设备或各种资源之间的相互联结和功能分配中的错误。**
- 它主要包括以下几种：
  - **配置命令测试**：验证全部配置命令的可操作性（有效性）；特别对最大配置和最小配置要进行测试。软件配置和硬件配置都要测试。
  - **循环配置测试**：证明对每个设备物理与逻辑的，逻辑与功能的每次循环置换配置都能正常工作。
  - **修复测试**：检查每种配置状态及哪个设备是坏的。并用自动的或手工的方式进行配置状态间的转换。

- **安全性测试**

安全性测试是要检验**在系统中已经存在的系统安全性、保密性措施是否发挥作用，有无漏洞**。

- 力图破坏系统的保护机构以进入系统的主要方法有以下几种：
  - 正面攻击或从侧面、背面攻击系统中易受损坏的那些部分；
  - 以系统输入为突破口，利用输入的容错性进行正面攻击；

- 申请和占用过多的资源压垮系统，以破坏安全措施，从而进入系统；
- 故意使系统出错，利用系统恢复的过程，窃取用户口令及其它有用的信息；
- 通过浏览残留在计算机各种资源中的垃圾（无用信息），以获取如口令，安全码，译码关键字等信息；
- 浏览全局数据，期望从中找到进入系统的关键字；
- 浏览那些逻辑上不存在，但物理上还存在的各种记录和资料等。

可使用性测试

- 可使用性测试主要从使用的**合理性**和**方便性**等角度对软件系统进行检查，发现人为因素或使用上的问题。
- 要保证在足够详细的程度下，**用户界面便于使用**；**对输入量可容错**、**响应时间和响应方式合理可行**、**输出信息有意义**、**正确并前后一致**；**出错信息能够引导用户去解决问题**；**软件文档全面**、**正规**、**确切**。

可支持性测试

这类测试是要验证**系统的支持策略对于公司与用户方面是否切实可行**。

- 它所采用的方法是
  - **试运行支持过程**(如对有错部分打补丁的过程，热线界面等)；
  - 对其结果进行**质量分析**；
  - **评审诊断工具**；
  - **维护过程**、**内部维护文档**；
  - **修复一个错误所需平均最少时间**。

安装测试

安装测试的目的**不是找软件错误，而是找安装错误。**

- 在安装软件系统时，会有多种选择。
  - 要分配和装入文件与程序库
  - 布置适用的硬件配置
  - 进行程序的联结。
- 而安装测试就是要找出在这些安装过程中出现的错误。

- 安装测试是在系统安装之后进行测试。它要检验：
  - 用户选择的一套任选方案是否相容；
  - 系统的每一部分是否都齐全；
  - 所有文件是否都已产生并确有所需要的内容；
  - 硬件的配置是否合理，等等。

过程测试

- 在一些大型的系统中，部分工作由软件自动完成，其它工作则需由各种人员，包括操作员，数据库管理员，终端用户等，按一定规程同计算机配合，靠人工来完成。
- **指定由人工完成的过程也需经过仔细的检查**，这就是所谓的过程测试。

互连测试

- 互连测试是要验证**两个或多个不同的系统之间的互连性**。

兼容性测试

- 这类测试主要想验证**软件产品在不同版本之间的兼容性**。有两类基本的兼容性测试：
  - **向下兼容**
  - **交错兼容**

容量测试

- 容量测试是要检验**系统的能力最高能达到什么程度**。例如，
  - 对于编译程序，让它处理特别长的源程序；
  - 对于操作系统，让它的作业队列“满员”；
  - 对于信息检索系统，让它使用频率达到最大。

  在使系统的**全部资源达到“满负荷”**的情形下，**测试系统的承受能力**。

文档测试

这种测试是检查**用户文档(如用户手册)的清晰性和精确性。**

- 用户文档中所使用的例子必须在测试中一一试过，确保叙述正确无误。

调试（Debug）

- 软件调试是在进行了成功的测试之后才开始的工作。它与软件测试不同，调试的任务是**进一步诊断和改正程序中潜在的错误。**
- 调试活动由两部分组成：
  - **确定程序中可疑错误的确切性质和位置。**
  - **对程序(设计,编码)进行修改，排除这个错误。**

- 调试工作是一个具有很强技巧性的工作。
- **软件运行失效或出现问题，往往只是潜在错误的外部表现**，而外部表现与内在原因之间常常没有明显的联系。如果要找出真正的原因，排除潜在的错误，不是一件易事。
- 可以说，**调试是通过现象，找出原因的一个思维分析的过程。**

调试的步骤

(1) 从错误的外部表现形式入手，确定程序中出错位置；

(2) 研究有关部分的程序，找出错误的内在原因；

(3) 修改设计和代码，以排除这个错误；

(4) 重复进行暴露了这个错误的原始测试或某些有关测试。

- 从技术角度来看，查找错误的难度在于：
  - 现象与原因所处的位置可能相距甚远。
  - 当其它错误得到纠正时，这一错误所表现出的现象可能会暂时消失，但并未实际排除。
  - 现象实际上是由一些非错误原因(例如，舍入不精确)引起的。
  - 现象可能是由于一些不容易发现的人为错误引起的。
  - 错误是由于时序问题引起的，与处理过程无关。
  - 现象是由于难于精确再现的输入状态（例如，实时应用中输入顺序不确定）引起。
  - 现象可能是周期出现的。在软、硬件结合的嵌入式系统中常常遇到。

几种主要的调试方法

调试的关键在于推断程序内部的错误位置及原因。可以采用以下方法：

强行排错

这种调试方法目前使用较多，效率较低。它不需要过多的思考，比较省脑筋。例如：

- **通过内存全部打印来调试**，在这大量的数据中寻找出错的位置。
- **在程序特定部位设置打印语句**，把打印语句插在出错的源程序的各个关键变量改变部位、重要分支部位、子程序调用部位，跟踪程序的执行，监视重要变量的变化。
- **自动调试工具**。利用某些程序语言的调试功能或专门的交互式调试工具，分析程序的动态过程，而不必修改程序。

应用以上任一种方法之前，都应当对错误的征兆进行全面彻底的分析，得出对出错位置及错误性质的推测，再使用一种适当的调试方法来检验推测的正确性。

回溯法调试

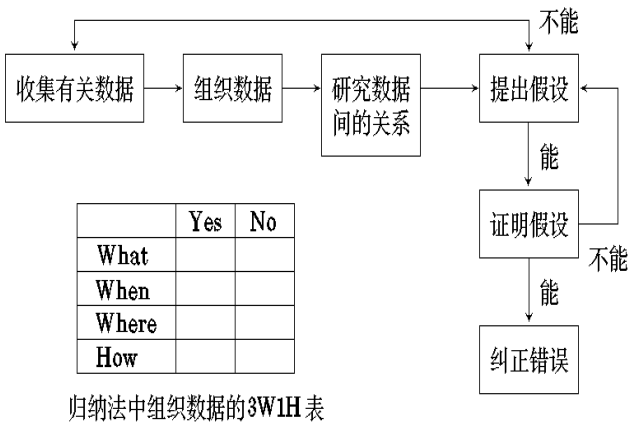
这是在小程序中常用的一种有效的调试方法。

一旦发现了错误，人们先分析错误征兆，确定最先发现“症状”的位置。然后，人工沿程序的控制流程，向回追踪源程序代码，直到找到错误根源或确定错误产生的范围。

- 例如，程序中发现错误处是某个打印语句。通过输出值可推断程序在这一点上变量的值。再从这一点出发，回溯程序的执行过程，反复考虑：“如果程序在这一点上的状态（变量的值）是这样，那么程序在上一点的状态一定是这样...”， 直到找到错误的位置。

归纳法调试

- 归纳法是一种从特殊推断一般的系统化思考方法。归纳法调试的基本思想是：从一些线索(错误征兆)着手，通过分析它们之间的关系来找出错误。
  - **收集有关的数据** 列出所有已知的测试用例和程序执行结果。看哪些输入数据的运行结果是正确的，哪些输入数据的运行结果有错误。



	Yes	No
What		
When		
Where		
How		

归纳法中组织数据的3W1H表

- **组织数据**

由于归纳法是从特殊到一般的推断过程，所以需要组织整理数据，以发现规律。

常以3W1H形式组织可用的数据：

"What" 列出一般现象；

"Where"说明发现现象的地点；

"When" 列出现象发生时所有已知情况；

"How" 说明现象的范围和量级；

"Yes"描述出现错误的3W1H；

"No"作为比较，描述了没有错误的3W1H。通过分析找出矛盾来。

- **提出假设**

分析线索之间的关系，利用在线索结构中观察到的矛盾现象，设计一个或多个关于出错原因的假设。如果一个假设也提不出来，归纳过程就需要收集更多的数据。此时，应当再设计与执行一些测试用例，以获得更多的数据。

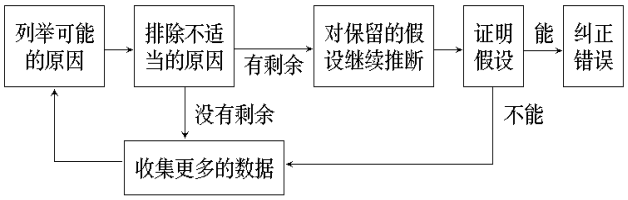
- **证明假设**

把假设与原始线索或数据进行比较，若它能完全解释一切现象，则假设得到证明；否则，就认为假设不合理，或不完全，或是存在多个错误，以致只能消除部分错误。

演绎法调试

演绎法是一种从一般原理或前提出发，经过排除和精化的过程来推导出结论的思考方法。演绎法排错是测试人员首先根据已有的测试用例，设想及枚举出所有可能出错的原因做为假设；然后再用原始测试数据或新的测试，从中逐个排除不可能正确的假设；最后，再用测试数据验证余下的假设确是出错的原因。

– **列举所有可能出错原因的假设**
把所有可能的错误原因列成表。通过它们，可以组织、分析现有数据。
– **利用已有的测试数据，排除不正确的假设**
仔细分析已有的数据，寻找矛盾，力求排除前一步列出所有原因。如果所有原因都被排除了，则需要补充一些数据(测试用例)，以建立新的假设。



– **改进余下的假设**
利用已知的线索，进一步改进余下的假设，使之更具体化，以便可以精确地确定出错位置。
– **证明余下的假设**

调试原则

- 在调试方面，许多原则本质上是心理学方面的问题。调试由两部分组成，调试原则也分成两组。
- **确定错误的性质和位置的原则**
  – 用头脑去分析思考与错误征兆有关的信息。
  – 避开死胡同。
  – 只把调试工具当做辅助手段来使用。利用调试工具，可以帮助思考，但不能代替思考。
  – 避免用试探法，最多只能把它当做最后手段。
- **修改错误的原则**
  – 在出现错误的地方，很可能还有别的错误。

– 修改错误的一个常见失误是只修改了这个错误的征兆或这个错误的表现，而没有修改错误的本身。
– 当心修正一个错误的同时有可能会引入新的错误。
– 修改错误的过程将迫使人们暂时回到程序设计阶段。
– 修改源代码程序，不要改变目标代码。

第七部分　软件维护

- 软件维护的概念
- 软件维护活动
- 程序修改的步骤及修改的副作用
- 软件可维护性
- 提高可维护性的方法

软件维护的概念

- 软件维护的定义
- 影响维护工作量的因素
- 软件维护的策略
- 维护成本

软件维护的定义

- 在软件运行／维护阶段对软件产品进行的修改就是所谓的维护。
- 维护的类型有四种：
  - 改正性维护
  - 适应性维护
  - 完善性维护
  - 预防性维护

改正性维护

- 在软件交付使用后，因开发时测试的不彻底、不完全，必然会有部分隐藏的错误遗留到运行阶段。
- 这些隐藏下来的错误在某些特定的使用环境下就会暴露出来。
- 为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的误使用，应当进行的诊断和改正错误的过程就叫做改正性维护。

适应性维护

- 在使用过程中，
  - 外部环境（新的硬、软件配置）
  - 数据环境（数据库、数据格式、数据输入/输出方式、数据存储介质）

  可能发生变化。
- 为使软件适应这种变化，而去修改软件的过程就叫做适应性维护。

完善性维护

- 在软件的使用过程中，用户往往会对软件提出新的功能与性能要求。
- 为了满足这些要求，需要修改或再开发软件，以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性。
- 这种情况下进行的维护活动叫做完善性维护。

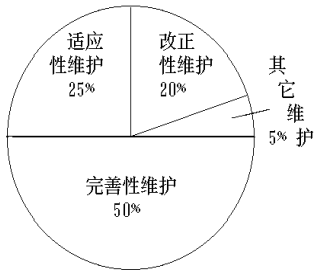
- 实践表明，在几种维护活动中，完善性维护所占的比重最大。即大部分维护工作是改变和加强软件，而不是纠错。
- 完善性维护不一定是救火式的紧急维修，而可以是有计划、有预谋的一种再开发活动。
- 事实证明，来自用户要求扩充、加强软件功能、性能的维护活动约占整个维护工作的50%。

预防性维护

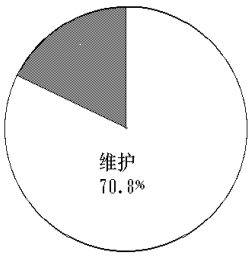
- 预防性维护是为了**提高软件的可维护性、可靠性等**，为以后进一步改进软件打下良好基础。
- 预防性维护定义为：**采用先进的软件工程方法对需要维护的软件或软件中的某一部分（重新）进行设计、编制和测试。**

- 在整个软件维护阶段所花费的全部工作量中，**完善性维护占了几乎一半的工作量。**
- **软件维护活动所花费的工作占整个生存期工作量的70%以上**，这是由于在漫长的软件运行过程中需要不断对软件进行修改，以**改正新发现的错误**、适应新的环境和用户新的要求，这些修改需要花费很多精力和时间，而且有时会引入新的错误。

三类维护占总维护比例



维护在软件生存期所占比例



影响维护工作量的因素

- **在软件的维护过程中，需要花费大量的工作量，从而直接影响了软件维护的成本。**
- **应当考虑有哪些因素影响软件维护的工作量，相应应该采取什么维护策略，才能**有效地维护软件**并**控制维护的成本。

- **系统大小**：系统越大，理解掌握起来越困难。系统越大，所执行功能越复杂。因而需要更多的维护工作量。
- **程序设计语言**：使用强功能的程序设计语言可以控制程序的规模。语言的功能越强，生成程序的模块化和结构化程度越高，所需的指令数就越少，程序的可读性越好。

- **系统年龄：**
  - 老系统随着不断的修改，结构越来越乱；
  - 维护人员经常更换，程序又变得越来越难于理解。
  - 许多老系统在当初并未按照软件工程的要求进行开发，因而没有文档，或文档太少。
  - 在长期的维护过程中文档在许多地方与程序实现变得不一致，在维护时就会遇到很大困难。

- **数据库技术的应用**：使用数据库，可以简单而有效地管理和存储用户程序中的数据，还可以减少生成用户报表应用软件的维护工作量。
- **先进的软件开发技术**：在软件开发时，若使用能使软件结构比较稳定的分析与设计技术，及程序设计技术，如面向对象技术、复用技术等，可减少大量的工作量。

- **其它：**
  - 应用的类型
  - 数学模型
  - 任务的难度
  - 开关与标记、IF嵌套深度、索引或下标数等

  对维护工作量都有影响。
- **许多软件在开发时并未考虑将来的修改，为软件的维护带来许多问题。**

软件维护的策略

- **改正性维护**

  通常要生成100%可靠的软件并不一定合算，成本太高。但通过使用新技术，可大大减少进行改正性维护的需要。

  这些技术包括：**数据库管理系统、软件开发环境、程序自动生成系统、较高级(第四代)的语言。**以及**新的开发方法、软件复用、防错程序设计**及**周期性维护审查**等。

- **适应性维护**

  这一类维护不可避免，可以控制。

  (1) 在配置管理时，把硬件、操作系统和其它相关环境因素的可能变化考虑在内。

  (2) 把与硬件、操作系统，以及其它外围设备有关的程序归到特定的程序模块中。

  (3) 使用内部程序列表、外部文件，以及处理的例行程序包，可为维护时修改程序提供方便。

- **完善性维护**

  利用前两类维护中列举的方法，也可以减少这一类维护。特别是数据库管理系统、程序生成器、应用软件包，可减少维护工作量。此外，建立软件系统的原型，把它在实际系统开发之前提供给用户。用户通过研究原型，进一步完善他们的功能要求，就可以减少以后完善性维护的需要。

维护成本

- **有形的软件维护成本**是花费了多少钱，**无形的维护成本**有更大的影响。
  - 一些**合理的修复或修改请求不能及时安排**，使得客户不满意；
  - 变更的结果**引入新的故障**，使得软件整体质量下降；
  - 把软件人员抽调到维护工作中，干扰了软件开发工作。

- 软件维护的代价是降低了生产率，在做老程序的维护时非常明显。
- 例如，开发每一行源代码耗资25美元，维护每一行源代码需要耗资1000美元。
- 维护工作量包括生产性活动（如分析和评价、设计修改和实现）和“轮转”活动（如力图理解代码在做什么、试图判明数据结构、接口特性、性能界限等）。

维护工作量的模型

$$M = p + Ke^{c-d}$$

- M是维护中消耗的总工作量
- p是上面描述的生产性工作量
- K是一个经验常数
- c是因缺乏好的设计和文档而导致复杂性的度量
- d是对软件熟悉程度的度量。

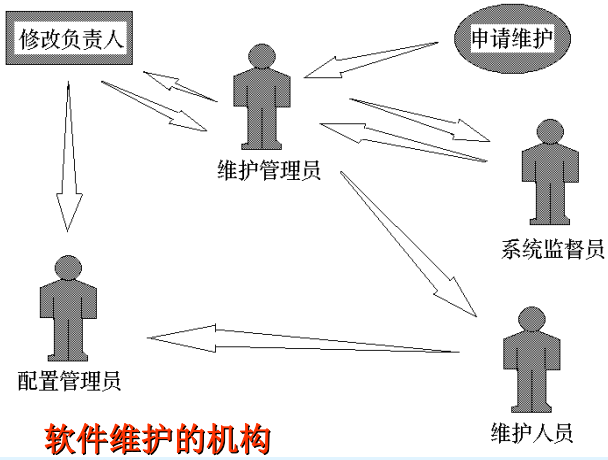
- 模型指明，如果使用了不好的软件开发方法（未按软件工程要求做），原来参加开发的人员或小组不能参加维护，则工作量（及成本）将按指数级增加。

软件维护活动

- 为了有效地进行软件维护，应事先就开始做组织工作。
  - 首先建立维护的机构
  - 申明提出维护申请报告的过程及评价的过程
  - 为每一个维护申请规定标准的处理步骤
  - 建立维护活动的登记制度以及规定评价和评审的标准。

维护机构

- 除了较大的软件开发公司外，通常在软件维护工作方面，并不保持一个正式的组织机构。
- 虽然不要求建立一个正式的维护机构，但是在开发部门确立一个非正式的维护机构则是非常必要的。



软件维护的机构

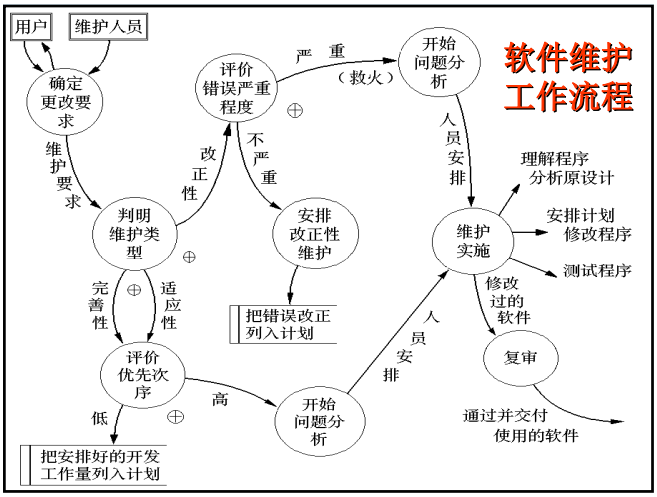
- **维护申请**提交给**维护管理员**，他把申请交给某个**系统监督员**去**评价**。
- 一旦做出评价，由**修改负责人**确定**如何进行修改**。
- 在修改程序的过程中，由**配置管理员**严格把关，**控制修改的范围**，**对软件配置进行审计**。
- 在维护之前，就把责任明确下来，可以减少维护过程中的混乱。

软件维护申请报告

- 维护申请报告或称**软件问题报告**，由**申请维护的用户填写**。
- 用户必须**完整地说明产生错误的情况**，包括**输入数据**、**错误清单**以及**其它有关材料**。
- 如果申请的是适应性维护或完善性维护，用户必须提出一份修改说明书，列出所有希望的修改。

- 维护申请报告将由**维护管理员**和**系统监督员**来研究处理。
- 他们应相应地做出**软件修改报告**，指明：
  - 所需修改变动的性质；
  - 申请修改的优先级；
  - 为满足某个维护申请报告，所需的工作量；
  - 预计修改后的状况.

- 软件修改报告应提交修改负责人，经批准后才能开始进一步安排维护工作。



- 尽管维护申请的类型不同，但都要进行同样的技术工作。
  - **修改软件需求说明**
  - **修改软件设计**
  - **设计评审**
  - **对源程序做必要的修改**
  - **单元测试**
  - **集成测试（回归测试）**
  - **确认测试**
  - **软件配置评审等。**

在每次软件维护任务完成后进行情况评审，对以下问题做一总结：

(1) 在目前情况下，设计、编码、测试中的哪一方面可以改进?

(2) 哪些维护资源应该有但没有?

(3) 工作中主要的或次要的障碍是什么?

(4) 从维护申请的类型来看是否应当有预防性维护?

情况评审对将来的维护工作如何进行会产生重要的影响。

维护档案记录

- 程序名称
- 源程序语句条数
- 机器代码指令条数
- 所用的程序设计语言
- 程序安装的日期
- 程序安装后的运行次数
- 与程序安装后运行次数有关的处理故障次数

- 程序改变的层次及名称
- 修改程序增加的源程序语句条数
- 修改程序减少的源程序语句条数
- 每次修改所付出的“人时”数
- 修改程序的日期
- 软件维护人员的姓名
- 维护申请报告的名称、维护类型
- 维护开始时间和维护结束时间、
- 花费在维护上的累计“人时”数
- 维护工作的净收益等。

维护评价

- **评价维护活动比较困难**，因为缺乏可靠的数据。
- 如果维护的档案记录做得比较好，可以得出一些维护“性能”方面的度量值。
  - **每次程序运行时的平均出错次数；**
  - **花费在每类维护上的总“人时”数；**
  - **每个程序、每种语言、每种维护类型的程序平均修改次数；**
  - **因为维护，增加或删除每个源程序语句所花费的平均“人时”数；**
  - **用于每种语言的平均“人时”数；**
  - **维护申请报告的平均处理时间；**
  - **各类维护申请的百分比。**

据此可对开发技术、语言选择、维护工作计划、资源分配、以及其它许多方面做出判定。

程序修改的步骤及修改的副作用

- **分析和理解程序**
- **修改程序**
- **重新验证程序**

分析和理解程序

- 理解程序的功能和目标；
- 掌握程序的结构信息，即从程序中细分出若干结构成分。如程序系统结构、控制结构、数据结构和输入／输出结构等；
- 了解数据流信息，即涉及到的数据来源何处，在哪里被使用
- 了解控制流信息，即执行每条路径的结果；
- 理解程序的操作（使用）要求；

修改程序

1. 设计程序的修改计划

程序的修改计划要考虑人员和资源的安排。小的修改可以不需要详细的计划，而对于需要耗时数月的修改，就需要计划立案。

2. 修改代码，以适应变化
3. 修改程序的副作用

所谓副作用是指因修改软件而造成的错误或其它不希望发生的情况。副作用有三种：修改代码的副作用、修改数据的副作用、文档的副作用。

重新验证程序

- 在将修改后的程序提交用户之前，需要进行充分的确认和测试，以保证整个修改后程序的正确性。
- 静态确认
  修改软件，伴随着引起新的错误的危险。为了能够做出正确的判断，验证修改后的程序至少需要两个人参加。要检查：

- 计算机确认
  在进行了以上确认的基础上，用计算机对修改程序进行确认测试：
  (1) 确认测试顺序：先对修改部分进行测试，然后隔离修改部分，测试程序的未修改部分，最后再把它们集成起来进行测试。这种测试称为回归测试。
  (2) 准备标准的测试用例。
  (3) 充分利用软件工具帮助重新验证过程。

(4) 在重新确认过程中，需邀请用户参加。

- 维护后的验收——在交付新软件之前，维护主管部门要检验：
  (1) 全部文档是否完备，并已更新；
  (2) 所有测试用例和测试结果已经正确记载；
  (3) 记录软件配置所有副本的工作已经完成；
  (4) 维护工序和责任已经确定。

软件可维护性

软件可维护性的定义

- 软件可维护性是指纠正软件系统出现的错误和缺陷，以及为满足新的要求进行修改、扩充或压缩的容易程度。
- 可维护性、可使用性、可靠性是衡量软件质量的主要质量特性。
- 软件的可维护性是软件开发阶段各个时期的关键目标。

- 目前广泛使用的是用如下的七个特性来衡量程序的可维护性。
  - 可理解性　可使用性
  - 可测试性　可移植性
  - 可修改性　效率
  - 可靠性
- 而且对于不同类型的维护，这七种特性的侧重点也不相同。

在各类维护中的侧重点

	改正性维护	适应性维护	完善性维护
可理解性	o		
可测试性	o		
可修改性	o	o	
可 靠 性	o		
可移植性		o	
可使用性		o	o
效 率			o

可维护性的度量

- 人们一直期望对软件的可维护性做出定量度量，但要做到这一点并不容易。
- 常用的度量一个可维护的程序的七种特性的方法。就是
  - 质量检查表
  - 质量测试
  - 质量标准

- 质量检查表是用于测试程序中某些质量特性是否存在的一个问题清单。
- 评价者针对检查表上的每一个问题，依据自己的定性判断，回答“Yes”或者“No”。
- 质量测试与质量标准则用于定量分析和评价程序的质量。
- 由于许多质量特性是相互抵触的，要考虑几种不同的度量标准，相应地去度量不同的质量特性。

1. 可理解性

- 可理解性表明人们通过阅读源代码和相关文档，了解程序功能及其如何运行的容易程度。
- 一个可理解的程序应具备以下一些特性：模块化，风格一致性，不使用令人捉摸不定或含糊不清的代码，使用有意义的数据名和过程名，结构化，完整性等。

2. 可靠性

- 可靠性表明一个程序按照用户的要求和设计目标，在给定的一段时间内正确执行的概率。
- 关于可靠性，度量的标准主要有：
  - 平均失效间隔时间MTTF
  - 平均修复时间MTTR
  - 有效性A = MTBD/(MTBD+MDT)

度量可靠性的方法

- 根据程序错误统计数字，进行可靠性预测。常用方法是利用一些可靠性模型，根据程序测试时发现并排除的错误数预测平均失效间隔时间MTTF。
- 根据程序复杂性，预测软件可靠性。用程序复杂性预测可靠性，前提条件是可靠性与复杂性有关。因此可用复杂性预测出错率。程序复杂性度量标准可用于预测哪些模块最可能发生错误，以及可能出现的错误类型。

3. 可测试性

- 可测试性表明论证程序正确性的容易程度。程序越简单，证明其正确性就越容易。而且设计合用的测试用例，取决于对程序的全面理解。
- 一个可测试的程序应当是可理解的，可靠的，简单的。
- 用于可测试性度量的检查项目如下：
  - 程序是否模块化？结构是否良好？
  - 程序是否可理解？程序是否可靠？
  - 程序是否能显示任意中间结果？
  - 程序是否能以清楚的方式描述它的输出？
  - 程序是否能及时地按照要求显示所有的输入？
  - 程序是否有跟踪及显示逻辑控制流程的能力？
  - 程序是否能从检查点再启动？
  - 程序是否能显示带说明的错误信息？

4. 可修改性

- 可修改性表明程序容易修改的程度。
- 一个可修改的程序应当是可理解的、通用的、灵活的、简单的。
- 通用性是指程序适用于各种功能变化而无需修改。
- 灵活性是指能够容易地对程序进行修改。

- 测试可修改性的一种定量方法是修改练习。其基本思想是通过做几个简单的修改，来评价修改的难度。
- 设*C*是程序中各个模块的平均复杂性，*n*是必须修改的模块数，*A* 是要修改的模块的平均复杂性。 则修改的难度*D*由下式计算：

$$D = A / C$$

5. 可移植性

- 可移植性表明程序转移到一个新的计算环境的可能性的大小。或者它表明程序可以容易地、有效地在各种各样的计算环境中运行的容易程度。
- 一个可移植的程序应具有结构良好、灵活、不依赖于某一具体计算机或操作系统的性能。
- 用于可移植性度量的检查项目如下：

– 是否是用高级的独立于机器的语言来编写程序?

– 是否使用广泛使用的标准化的程序设计语言来编写程序？是否仅使用了这种语言的标准版本和特性?

– 程序中是否使用了标准的普遍使用的库功能和子程序?

– 程序中是否极少使用或根本不使用操作系统的功能?

–程序在执行之前是否初始化内存?

– 程序在执行之前是否测定当前的输入/输出设备?

– 程序是否把与机器相关的语句分离了出来，集中放在了一些单独的程序模块中，并有说明文件?

– 程序是否结构化?并允许在小一些的计算机上分段(覆盖)运行?

– 程序中是否避免了依赖于字母数字或特殊字符的内部位表示?

6. 效率

- 效率表明一个程序能执行预定功能而又不浪费机器资源的程度。
- 这些机器资源包括内存容量、外存容量、通道容量和执行时间。
- 用于效率度量的检查项目如下：
  - 程序是否模块化?结构是否良好?
  - 是否消除了无用的标号与表达式，以充分发挥编译器优化作用?

– 程序的编译器是否有优化功能?

– 是否把特殊子程序和错误处理子程序都归入了单独的模块中?

– 是否以快速的数学运算代替了较慢的数学运算?

– 是否尽可能地使用了整数运算，而不是实数运算?

– 是否在表达式中避免了混合数据类型的使用，消除了不必要的类型转换?

– 程序是否避免了非标准的函数或子程序的调用?

– 在几条分支结构中，是否最有可能为“真”的分支首先得到测试?

– 在复杂的逻辑条件中，是否最有可能为“真“的表达式首先得到测试?

7. 可使用性

- 从用户观点出发，**可使用性定义为程序方便、实用、及易于使用的程度**。一个可使用的程序应是**易于使用的、能允许用户出错和改变**，并**尽可能不使用户陷入混乱状态的**程序。
- 用于可使用性度量的检查项目如下：
  - **程序是否具有自描述性?**
  - **程序是否能始终如一地按照用户的要求运行?**
  - **程序是否让用户对数据处理有一个满意的和适当的控制?**
  - **程序是否容易学会使用?**
  - **程序是否使用数据管理系统来自动地处理事务性工作和管理格式化、地址分配及存储器组织。**
  - **程序是否具有容错性?**
  - **程序是否灵活?**

其它间接定量度量可维护性的方法

- 问题识别的时间；
- 因管理活动拖延的时间；
- 收集维护工具的时间；
- 分析、诊断问题的时间；
- 修改规格说明的时间；
- 具体的改错或修改的时间；
- 局部测试的时间；
- 集成或回归测试的时间；
- 维护的评审时间；

- 这些数据反映了维护全过程中**检错—纠错—验证**的周期，即**从检测出软件存在的问题开始至修正它们并经回归测试验证这段时间**。
- 可以粗略地认为，**这个周期越短，维护越容易**。

提高可维护性的方法

- 建立明确的软件质量目标和优先级
- 使用提高软件质量的技术和工具
- 进行明确的质量保证审查
- 选择可维护的程序设计语言
- 改进程序的文档

面向对象技术

- **面向对象的概念**
- **面向对象的开发过程**
- **面向对象分析与模型化**
- **面向对象设计**
- **面向对象程序的实现与测试**
- Code与Yourdon面向对象分析与设计技术
- OMT方法

面向对象的概念

- 开发模式
- 什么是面向对象
- 对象
- 类
- 继承

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。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**

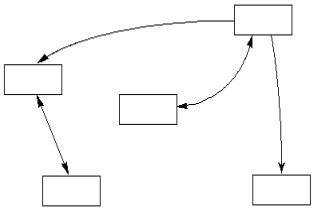
面向过程开发模式的特点

- 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。

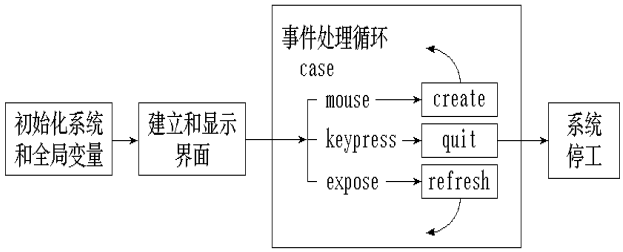
过程性系统

（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



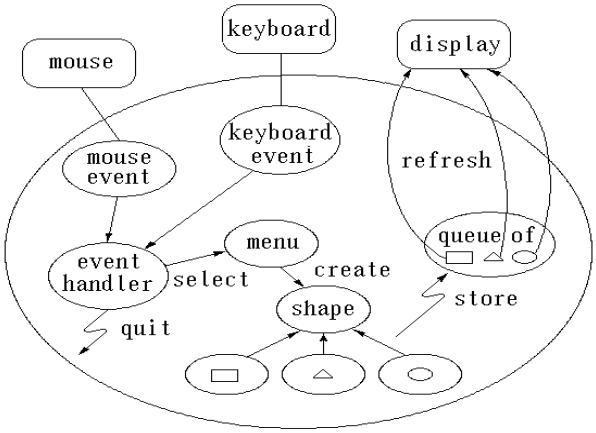
(1) Initialize system;
(2) Create and draw interface;
while QUIT not selected do
case
Mouse event:
create shape structure;
read mouse movements for data;
store newly created shape on list
of shape records;
KeyPress event:
if key = 'q' then exit loop;
else ignore;
Ecpose event:
refresh display by drawing 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

- 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

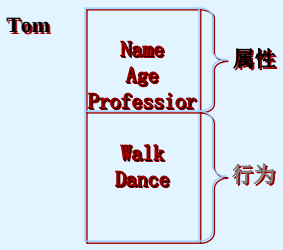
什么是面向对象

- 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

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

- 如：人这个实体
  属性：姓名、年龄、职业
  行为：跑、跳



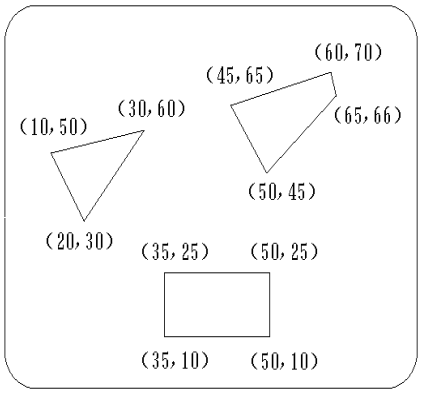
消息（Message）

- 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**。

如:Mycircle是一个Circle类对象

Mycircle.Show(green)是Mycircle对象发出的服务请求，执行在Circle类中所定义的Show操作



(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)

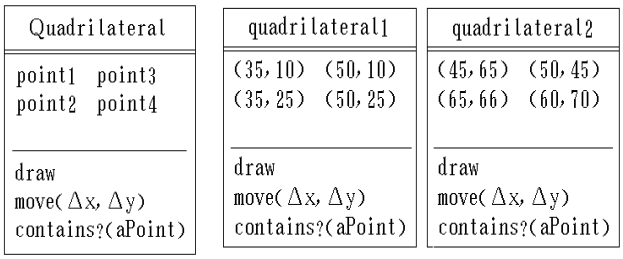
(b) 表示多边形的三个对象

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



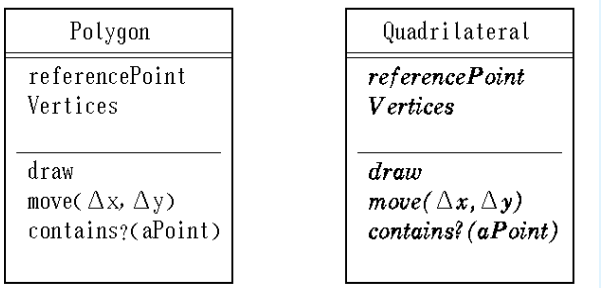
(b) 表示多边形的三个对象

- 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

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

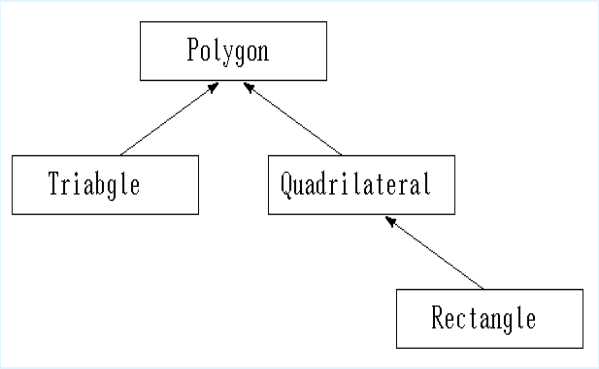


(a) Polygon类　(b) Polygon类的子类Quadrilateral

- 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。

多态性

- **一个操作在不同类中可以有不同的实现方式。**

如：Dance的实现在Male类和Female类中是不同的。

Male类中有Dance( )

Female类中有Dance(news)

重载

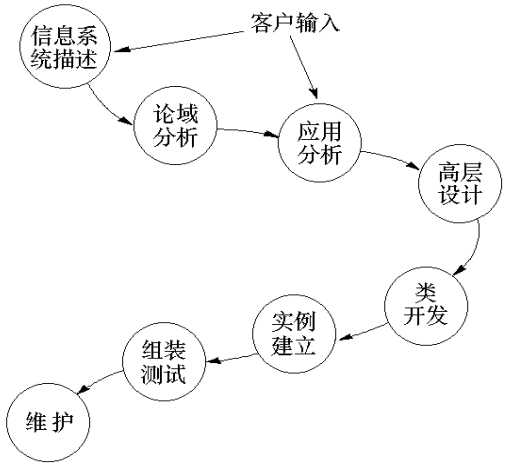
- **在两个类中有相同的行为，但是如果加上作用域的指引来调用即可。**

```
如：void BaseClass::Dance( )
    void SubClass::Dance( )
    Class BaseClass
     {     void Dance( );
     }
   Class SubClass:public BaseClass
     {     void Dance( );
     }
```

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**。

应用生存期



- **分析阶段**

论域分析：对问题敞开思想考虑不加限制，考虑问题论域一个较宽的范围。

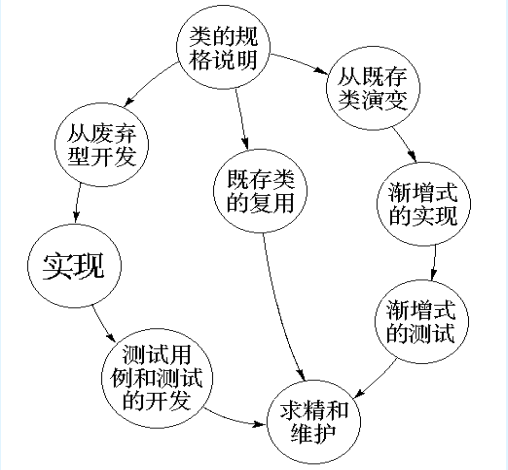
应用分析：针对较具体的应用，当前要解决的问题来分析。

- **高层设计：系统设计与类设计**

其中系统设计应用的顶层视图或开发系统类的界面。

- **类的开发：一组类**
- **实例的建立：对象的实例**
- **组装测试：把系统组装成一个完整的应用来进行。**
- **应用维护：**

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

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

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

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

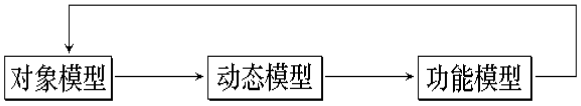
- **一般化和特殊化关系**可用来按层次渐增式地定义抽象（类）。
- 低层抽象是高层抽象的特殊化。
- 这种抽象层次构成论域模型的基础。
- 例如，小汽车，卡车和公共汽车可以归于更一般的概念汽车中。从这个较一般化的概念汽车可以

- **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、

- **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

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。

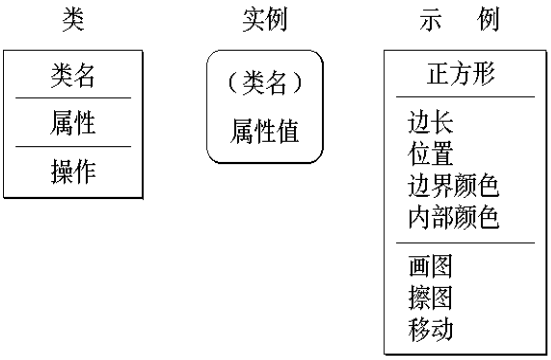


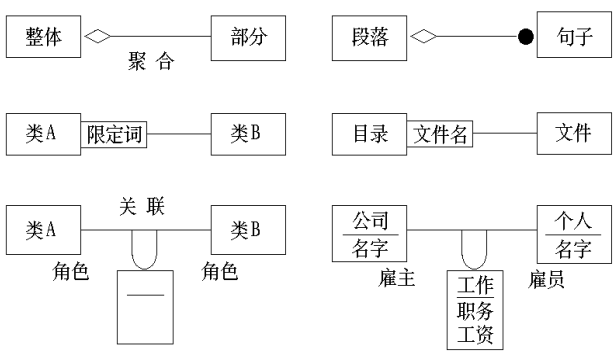
- 这个模型化的过程是一个迭代过

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

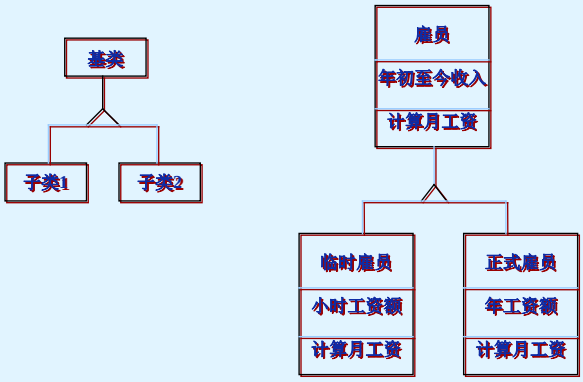
- **聚合，代表整体与部分的关系，这是一种特殊形式的关联。**
- **限定，用以对关联的含义做某种约束。**
- **角色，用来说明关联的一端。由于多数关联具有两个端点，因而涉及到两个角色。**
- **附加的说明对象之间的连接的连接属性。**





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

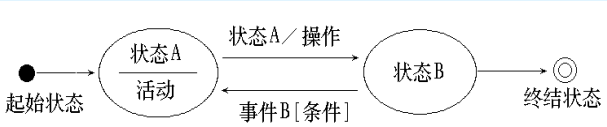


动态模型

- 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，

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

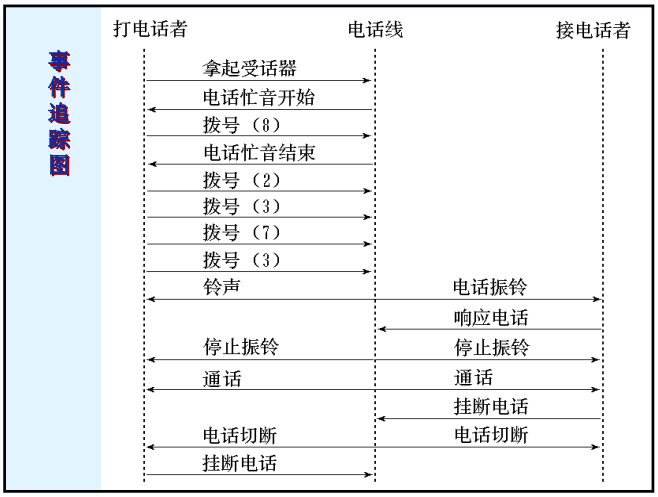
- 一个事件发生在某一时刻
- 每个事件都是单独发生的
- 我们建立事件类，并给每个事件一个名字，以指明共同结构和行为。
- 事件从一个对象向另一个对象传送信息。

- 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)
  - 撤下鼠标按钮(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

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

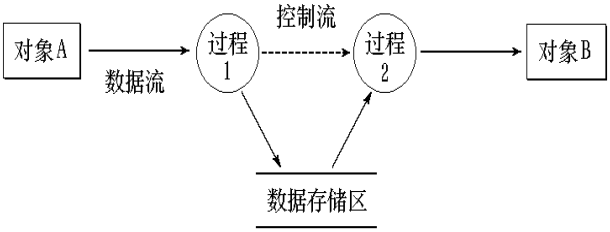
- 状态图叙述一个对象的个体行为，事件追踪图则给出多个对象所表现出来的集体行为。它们从不同侧面来说明同一系统的行为。
- 例如，一个事件追踪图指出某一对象在接受一个事件之后发出另一事件，同一行为在此对

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

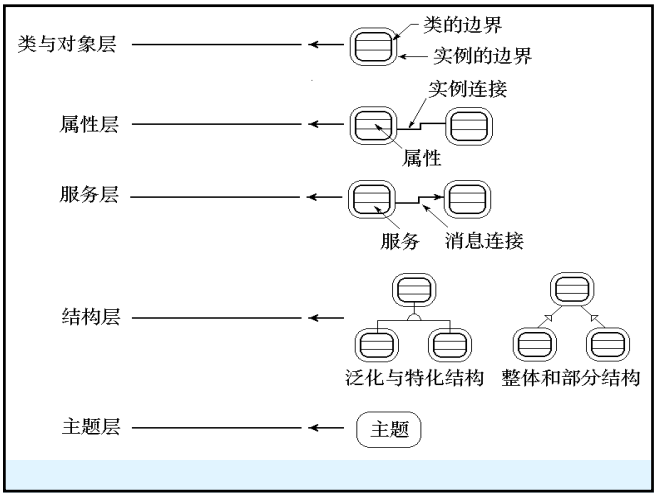
- OOA有两个任务
  - 形式地说明我们所面对的应用问题，最终成为软件系统基本构成的对象，还有系统所必须遵从的，由应用环境所决定的规则和约束。
  - 明确地规定构成系统的对象如何协同合作，完成指定的功能。

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、

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

- **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

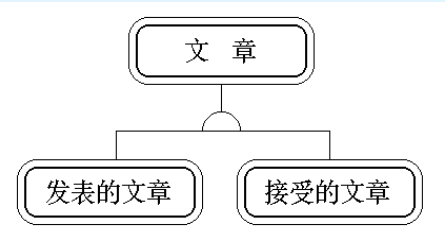
条　　目	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	注　　释
ACCEPTION SUBSCRI PTION				×						SUBSCRIPTION 的属性
ACCOMPANIED PA YMENT	×									对payment 的不同类型不加以区分
ACCOUNTING DEPAR TMENT	×									已超出SBSS 的应用论域
ACTUAL EXPIRATION DATE				×						SUBSCRIPTION 的属性
ADDITIONAL SU BSCRIPTION			×	×						SUBSCRIPTION 的可能属性，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

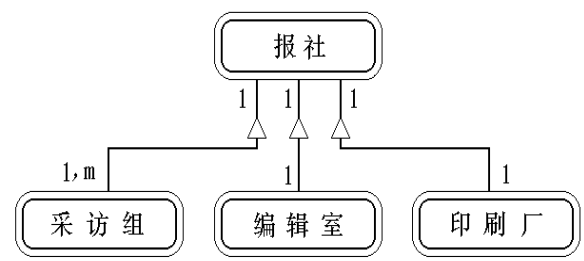
标识结构

- 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**

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**。

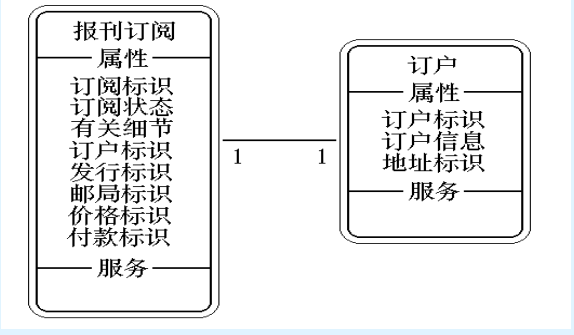
标识属性

- 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。

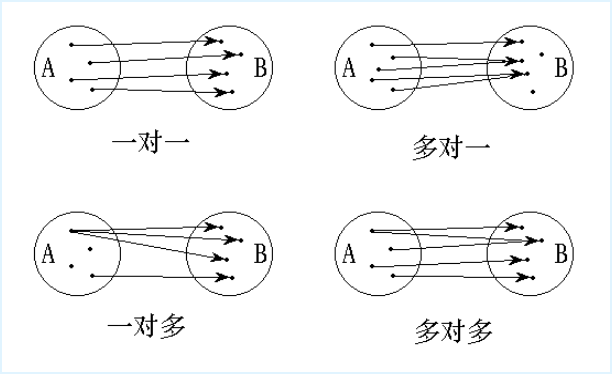
标识属性的方法和策略

- **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



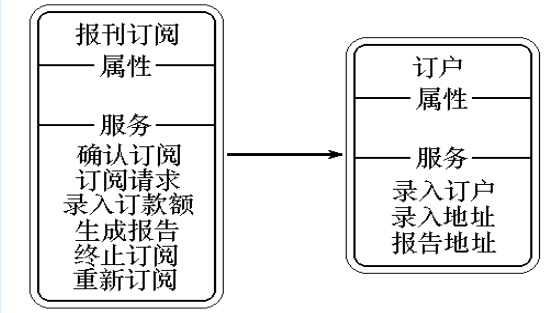
实例连接关系的标识



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

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

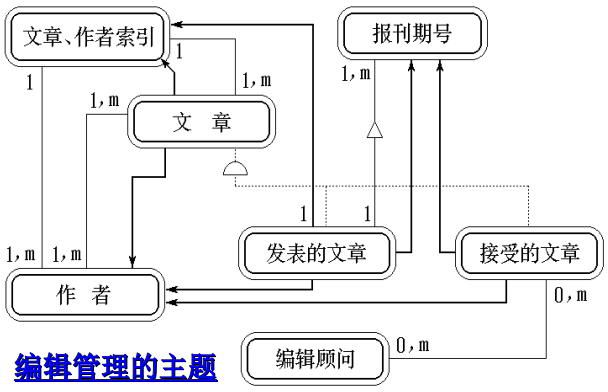
- 两个对象之间可能存在着由于通信需要而形成的关系，这称为消息连接。
- 消息连接表示从一个对象发送消息到另一个对象，由那个对象完成某些处理。它们在图中用箭头表示，方向从发消息的对象指向收消息的对象。

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

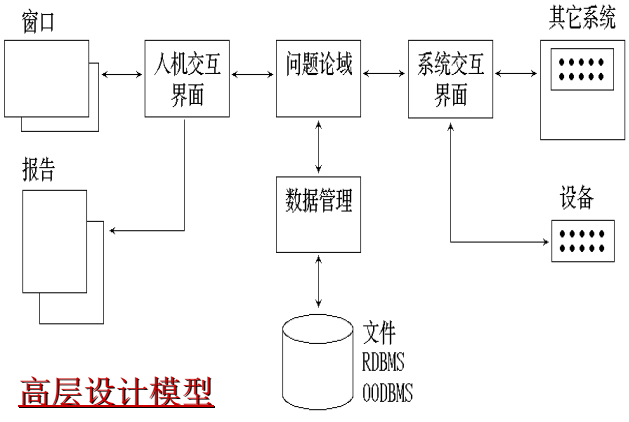
- 将**每一种结构**（包括整体-部分结构、和一般化-特殊化结构）**中最上层的类提升成为主题**；
- **将各不属于任何结构的类提升主题**；
- 检查在相同或类似的问题论域中以前做面向对象分析的结果，看是否有可直接复用的主题。

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

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**

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。

2. 把问题论域相关的类关联起来

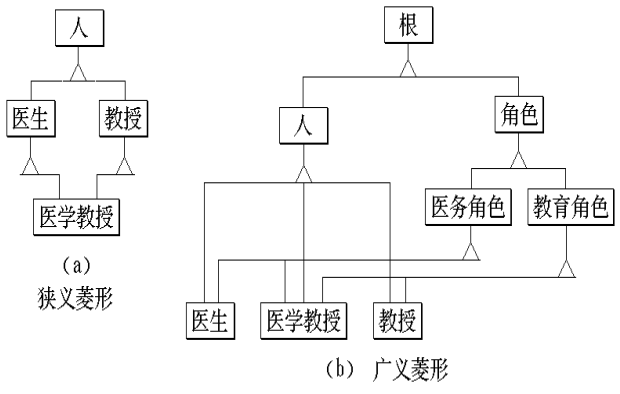
- 在设计时，**从类库中引进一个根类，做为包容类，把所有与问题论域有关的类关联到一起，建立类的层次**。
- 把同一问题论域的一些类集合起来，存于类库中。

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。

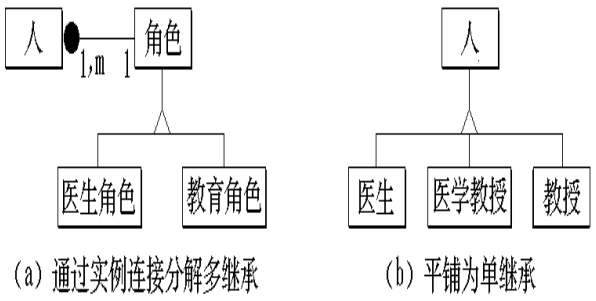
4. 调整继承支持级别

- 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

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

  - **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.

- 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。

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

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

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

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

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

- 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

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。**

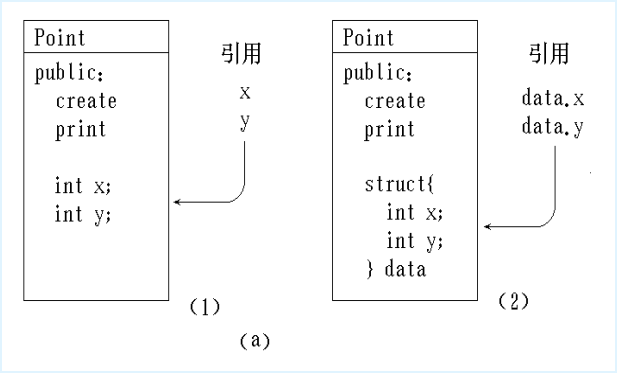
- 可复用的“插接相容性”部件
  - **部件可以在未来的应用中使用。**
  - **界面的标准化**
  - **类的“插接相容性”**
- 可靠的部件
  - **可靠的**(健壮的和正确定义的)**部件。**
  - **每个部件必须经过充分的测试。**
  - **每个操作尽可能小和作用单一。**

- 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。

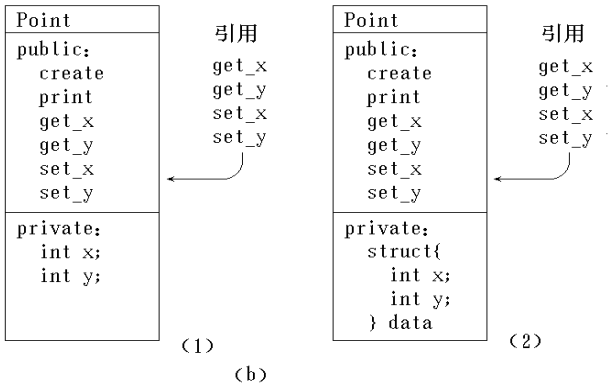
类设计的方针

- 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

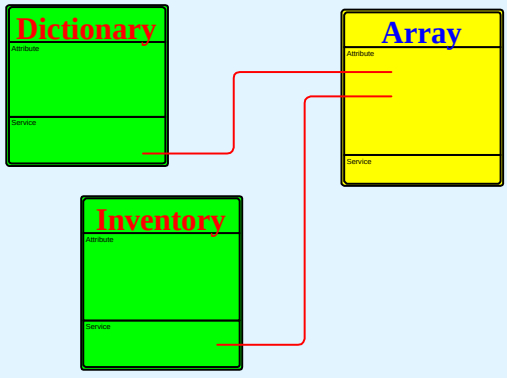


通过界面引用类中的数据



- 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

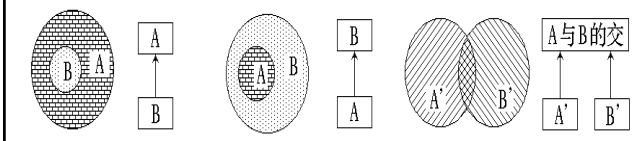
一个面向对象部件库的层次

- 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

  - 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。

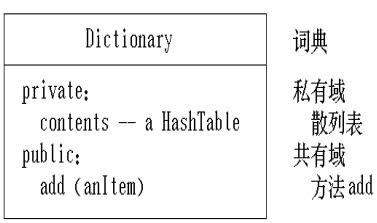
类级关系

- 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

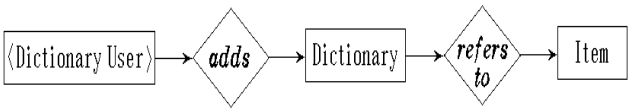
消息(messaging)

- 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。



- **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)

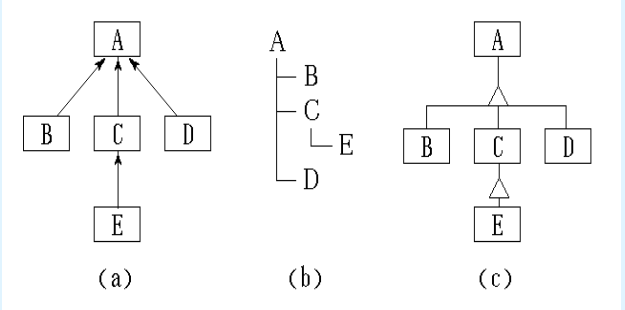
- 组装关系是一个实现级关系，它对应于应用级的聚合关系。
- 它也叫做***component***（部件）或叫做 ***is part of***（是…的一部分）。
- 组装与消息两者都是类间的关系，在这种关系中，一个类的实例将是另一个类的实现的一部分。

- 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)

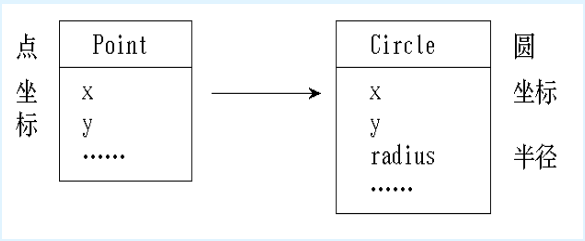
- **继承允许在既存类的基础上定义新的类。**
- 一个新类**B**继承了既存类**A**，则**B**包括了**A**定义的某些行为，以及它自定义的某些附加行为。
- 有多少种面向对象程序设计语言，就有多少种不同的继承实现方式。

继承图



① 针对实现的继承

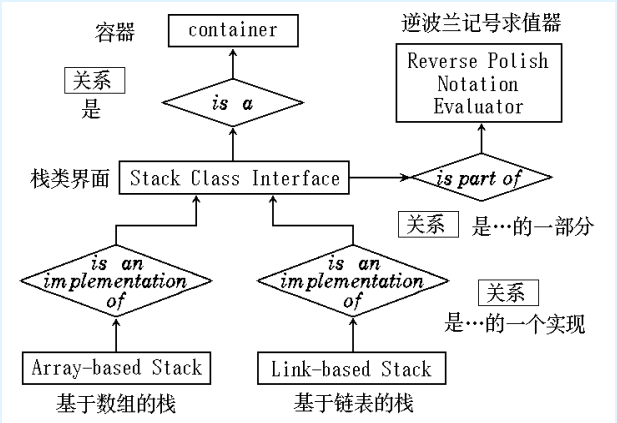
- 两个类之间“针对实现”的继承关系的建立指的是使用既存类的内部表示来做为新类的内部表示的一部分。我们不推荐这种继承方式。
- 考虑使用继承来实现一个**Circle**类，为了定义一个圆，需要定义一个点和一个值，做为圆的圆心和半径。因此，**Point**类可支持**Circle**类的一部分实现。把**Point**当做派生类。



- 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。

—"原封不动"复用

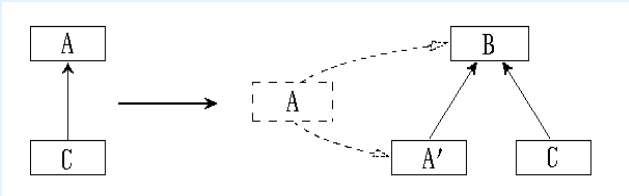
- **所需要的类已经存在**，我们**建立它的一个实例**，用以提供所需要的特性。
- 这个实例可直接被应用利用，或者它可以用来做另一个类的实现部分。
- 通过复用一个既存类，我们可得到不加修改就能工作的已测试的代码。

—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：

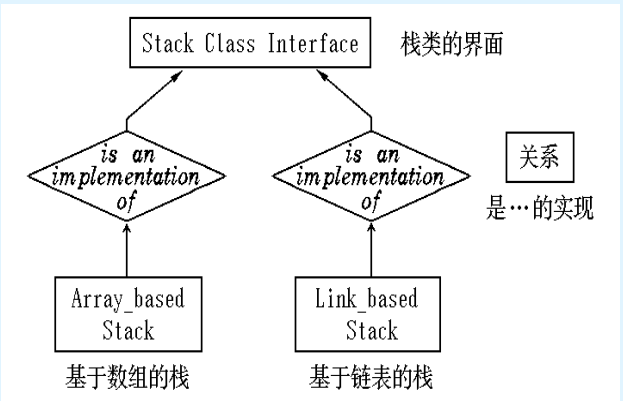
  ERROR_HANDLER.handle

  ("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。

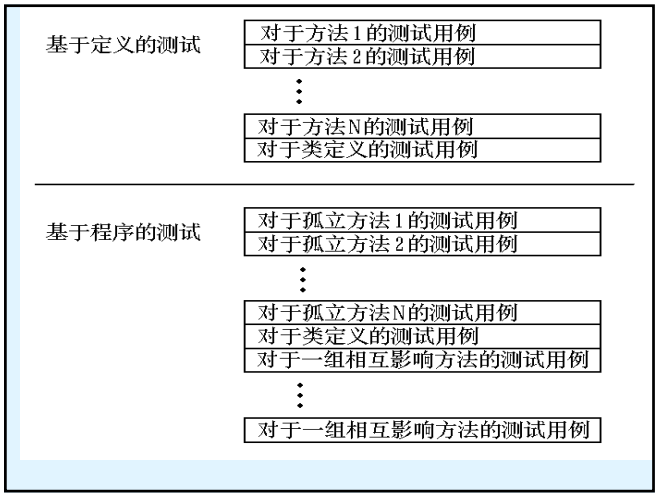
- 测试方法学检测软件中的故障并确定软件是否执行了预定要开发的功能。
- 测试过程包括了一组测试用例的开发，每一个测试用例要求能检验应用的一个特定的元素。还需要分析用各个测试用例执行测试的结果来收集有关软件的信息。

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

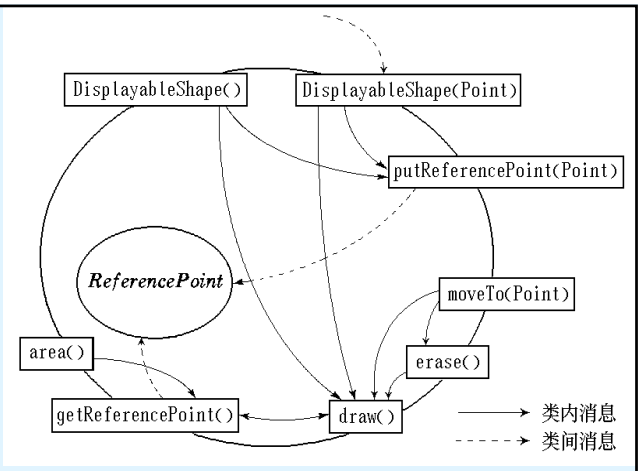
- 基于定义的测试包括两个级别：**类定义**和**服务定义**。
- 类定义
  - 一个类的定义由各个服务的定义和一些表示类的概念的语句组合而成。
  - 例，一个**stack**类包括了服务***push***和***pop***的定义。还表达了**LIFO** 的思想。

- **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。

第二部分　软件计划

- ❖软件的范围
- ❖软件开发中的资源
- ❖软件项目估算
- ❖进度安排
- ❖软件计划文件与复审

软件计划内容

针对不同的目标，软件工程项目需要对各阶段制定相应的工作计划

- 软件开发计划

是软件开发的综合性计划，包括任务、进度、人力、环境、资源和组织。其目的是提供一个框架，项目管理人员对资源、成本以及进度进行合理的估算。在项目开始时完成。

- 质量保证计划

将软件开发的质量要求具体规定为每个开发阶段可以检查的质量保证。

- 软件测试计划

规定测试活动的任务、方法、进度、资源、人员职责。

- 文件编制计划

规定软件开发项目应编写文件的种类、内容、进度、人员职责。

- 用户培训计划

对用户进行技术培训的目标、要求、进度、人员职责。

- 综合支持计划

项目开发过程中所需支持条件

- 软件分发计划

确定软件项目如何提供给用户

软件项目计划的目标

- 软件项目管理人员在开发工作一开始需要进行定量估算。
- 软件项目计划的目标是提供一个能使项目管理人员对资源、成本和进度做出合理估算的框架。
- 这些估算应当在软件项目开始时的一个有限的时间段内做出，并且随着项目的进展定期进行更新。

软件的范围

软件项目计划的第一项活动是确定软件的范围。

- 软件范围包括功能、性能、限制、接口和可靠性。
- 估算开始时，应对软件的功能进行评价，对其进行适当的细化以便提供更详细的细节。由于成本和进度的估算都与功能有关，因此常常采用某种程度的功能分解。

- 性能的考虑包括处理和响应时间的需求。
- 约束条件则标识产品成本、外部硬件、可用存储或其它现有系统对软件的限制。
- 功能、性能和约束必须在一起进行评价。当性能限制不同时，为实现同样的功能，开发工作量可能相差一个数量级。

软件与其它系统元素是相互作用的。要考虑每个接口的性质和复杂性，以确定对开发资源、成本和进度的影响。接口的概念可解释为：

–运行软件的硬件（如处理机与外设）及间接受软件控制的设备（如机器、显示器）；

–必须与新软件链接的现有的软件（如数据库存取例程、子程序包、操作系统）；

–通过终端或其它输入／输出设备使用该软件的人；

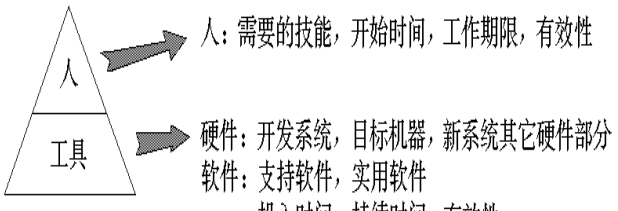
–该软件运行前后的一系列操作过程。

- 对于每一种情况，都必须清楚地了解通过接口的信息转换。

软件开发中的资源

- 软件项目计划的第二个任务是对完成该软件项目所需的资源进行估算。
- 软件开发所需的资源有
- 现成的用以支持软件开发的工具
  ——硬件工具及软件工具
- 最基本的资源
  ——人

软件开发中的资源

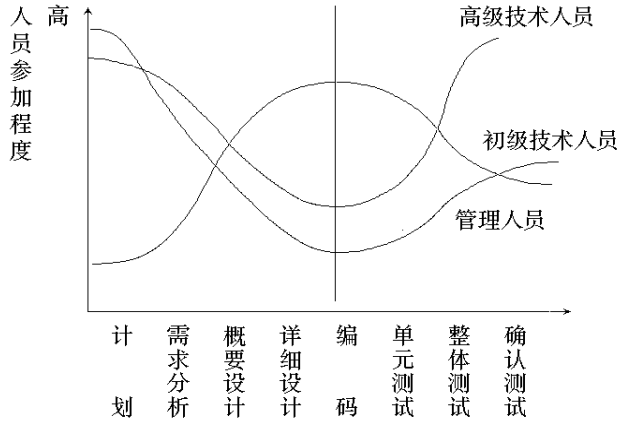


- 通常，对每一种资源，应说明以下四个特性：
  （1）资源的描述；
  （2）资源的有效性说明；
  （3）资源在何时开始需要；
  （4）使用资源的持续时间。
- 最后两个特性统称为时间窗口。对每一个特定的时间窗口，在开始使用它之前就应说明它的有效性。

1. 人力资源

- 在考虑各种软件开发资源时，人是最重要的资源。在安排开发活动时必须考虑人员的技术水平、专业、人数、以及在开发过程各阶段中对各种人员的需要。
- 计划人员首先估算范围并选择为完成开发工作所需要的技能。还要在组织状况（如管理人员、高级软件工程师等）和专业（如通信、数据库、微机等）两方面做出安排。

- 对于一些规模较小的项目（1个人年或者更少），只要向专家做些咨询，也许一个人就可以完成所有的软件工程步骤。
- 对一些规模较大的项目，在整个软件生存期中，各种人员的参与情况是不一样的。下面是各类不同的人员随开发工作的进展在软件工程各个阶段的参与情况的典型曲线。



2. 硬件资源

- 硬件是作为软件开发项目的一种工具而投入的。
  （1）宿主机（Host）— 软件开发时使用的计算机及外围设备；
  （2）目标机（Target）— 运行已开发成功软件的计算机及外围设备；
  （3）其它硬件设备 — 专用软件开发时需要的特殊硬件资源；

- 宿主机连同必要的软件工具构成软件开发系统。通常这样的开发系统能够支持多种用户的需要，且能保持大量的由软件开发小组成员共享的信息。
- 在许多情况下，除了那些很大的系统之外，不一定非要配备专门的开发系统。因此，所谓硬件资源，可以认为是对现存计算机系统的使用，而不是去购买一台新的计算机。宿主机与目标机可以是同一种机型。

3. 软件资源

- 软件工程人员在软件开发期间使用了许多软件工具来帮助开发。这种软件工具集叫做计算机辅助软件工程（CASE）。
  （1）业务系统计划工具集
  （2）项目管理工具集
  （3）支援工具——文档生成工具、网络系统软件、数据库、电子邮件、通报板，以及配置管理工具。

（4）分析和设计工具
（5）编程工具
（6）组装和测试工具
（7）原型化和模拟工具
（8）维护工具
（9）框架工具——这些工具能够提供建立集成项目支撑环境（IPSE）的框架。

4. 软件复用性及软件部件库

- 为了促成软件的复用，以提高软件的生产率和软件产品的质量，可建立可复用的软件部件库。

软件项目估算

- 软件成本和工作量的估算中变化的东西太多，人、技术、环境、政治，都会影响软件的最终成本和开发的工作量。
- 软件项目的估算还是能够通过一系列系统化的步骤，在可接受的风险范围内提供估算结果。
- 成本估算必须“事前”给出。时间越久，我们了解得越多，估算中出现的严重误差就越少。

分解技术

- 当一个待解决的问题过于复杂时，我们可以把它进一步分解，直到分解后的子问题变得容易解决为止。然后，分别解决每一个子问题，并将这些子问题的解答综合起来，从而得到原问题的解答。

LOC和FP（功能点）估算

- 在软件项目估算中，在两个方面使用了LOC和FP数据：
  - 把LOC和FP数据当做一个估算变量，用于量度软件每一个元素的规模。
  - LOC和FP数据作为从过去项目中收集到的基线数据，与其它估算变量联合使用，进行成本和工作量的估算。

- LOC和FP是两个不同的估算技术。两者的共性在于：项目计划人员
  - 给出一个有界的软件范围的叙述
  - 由此叙述尝试把软件分解成一些小的可分别独立进行估算的子功能
  - 对每一个子功能估算其LOC或FP
  - 把基线生产率度量（如LOC / PM（人月）或FP / PM）用做特定的估算变量，导出子功能的成本或工作量
  - 将子功能的估算进行综合后就能得到整个项目的总估算。

- LOC或FP估算技术对于分解所需要的详细程度是不同的。
- 用LOC做为估算变量时，必须进行功能分解，且需要达到很详细的程度。而估算FP时需要的数据是宏观的量，当把FP当做估算变量时不需分解得很详细。
- LOC是直接估算的，而FP是通过估计输入、输出、数据文件、查询和外部接口的数目，以及14种复杂性校正值间接地确定的。

- 项目计划人员可对每一个分解的功能提出一个有代表性的估算值范围。
- 利用历史数据或凭实际经验（当其它的方法失效时），对每个功能分别按最佳的、可能的、悲观的三种情况给出LOC或FP估计值。记作a、m、b。
- 接着计算LOC或FP的期望值 E。

$$E = (a + 4m + b) / 6$$

- 所有子功能的总估算变量值除以相应于该估算变量的平均生产率度量得到项目的总工作量。
- 例如，若假定总的FP估算值是310，基于过去项目的平均FP生产率是5.5FP / PM，则项目的总工作量是：

  工作量 = 310 / 5.5 = 56 PM

  作为LOC和FP估算技术的实例，考察一个为计算机辅助设计（CAD）应用而开发的软件包。

- 系统定义评审指明，软件是在一个工作站上运行，其接口必须使用各种计算机图形设备，包括鼠标器、数字化仪、高分辨率彩色显示器和激光打印机。
- 在这个实例中，使用LOC做为估算变量。
- 根据系统规格说明，软件范围的初步叙述如下

“软件将从操作员那里接收2维或3维几何数据。操作员通过用户界面与CAD系统交互并控制它，这种用户界面将表现出很好的人机接口设计特性。所有的几何数据和其它支持信息保存在一个CAD数据库内。要开发一些设计分析模块以产生在各种图形设备上显示的输出。软件要设计得能控制并与能各种外部设备，包括鼠标器、数字化仪、激光打印机和绘图仪交互。”

经过分解，识别出下列主要软件功能：

- 用户界面和控制功能
- 二维几何分析
- 三维几何分析
- 数据库管理
- 计算机图形显示功能
- 外设控制PC
- 设计分析模块

- 通过分解，可得到如下估算表

估算表

功 能	a 最佳值	m 可能值	b 悲观值	E 期望值	元／行	行／PM	成本（元）	工作量（PM）
用户接口控制	1800	2400	2650	2340	14	315	32760	7.4
二维几何造型	4100	5200	7400	5380	20	220	107600	24.4
三维几何造型	4600	6900	8600	6800	20	220	136000	30.9
数据结构管理	2950	3400	3600	3350	18	240	60300	13.9
计算机图形显示	4050	4900	6200	4950	22	200	108900	24.7
外部设备控制	2000	2100	2450	2140	28	140	59920	15.2
设计分析	6600	8500	9800	8400	18	300	151200	28.0
总 计				33360			656680	144.5

- 从历史的基线数据求出生产率度量，即行／PM和元／行。
- 需要根据复杂性程度的不同，对各功能使用不同的生产率度量值。
- 在表中的成本 = LOC的期望值 E与元／行相乘，工作量 = 用LOC 的期望值 E与行／PM相除。
- 因此可得，该项目总成本的估算值为657,000元，总工作量的估算值为145人月（PM）。

工作量估算

- 工作量估算是估算任何工程开发项目成本的最普遍使用的技术。
- 每一项目任务的解决都需要花费若干工作量 (人日、人月或人年)。
- 每一个工作量单位都对应于一定的货币成本，从而可以由此做出成本估算。

- 工作量估算开始于从软件项目范围抽出软件功能。
- 接着给出为实现每一软件功能所必须执行的一系列软件工程任务，包括需求分析、设计、编码和测试。
- 针对每一软件功能，估算完成各个软件工程任务所需要的工作量（如人月）。同时，把劳动费用率（即成本／单位工作量）加到每个软件工程任务上。

- 对于每个软件工程任务，劳动费用率都可能不同。高级技术人员主要投入到需求分析和早期的设计任务中，而初级技术人员则进行后期设计任务、编码和早期测试工作，他们所需成本比较低。
- 最后一个步骤就是计算每一个功能及软件工程任务的工作量和成本。

- 为了说明工作量估算的使用，考虑上面所介绍的CAD软件。
- 与每个软件工程任务相关的劳动费用率记入表中费用率（元）这一行，这些数据反映了“负担”的劳动成本，即包括公司开销在内的劳动成本。
- 在此例中，需求分析的劳动成本为5,200元／PM，比编码和单元测试的劳动成本高出22%。

工作量估算表

功能 \ 任务	需求分析	设计	编码	测试	总计
用户界面控制	1.0	2.0	0.5	3.5	7.0
二维几何分析	2.0	10.0	4.5	9.5	26.0
三维几何分析	2.5	12.0	6.0	11.0	31.5
数据结构管理	2.0	6.0	3.0	4.0	15.0
图形显示功能	1.5	11.0	4.0	10.5	27.0
外设控制功能	1.5	6.0	3.5	5.0	16.0
设计分析模块	4.0	14.0	5.0	7.0	30.0
总计	14.5	61.0	26.5	50.5	152.5
费用率（元）	5200	4800	4250	4500	
成本（元）	75400	292800	112625	227250	708075

总工作量估算

总成本估算

除特别指出的地方之外，都按人月估算工作量。

软件开发成本估算

- 软件开发成本主要是指**软件开发过程中所花费的工作量及相应的代价**。它不包括原材料和能源的消耗，主要是人的劳动的消耗。
- **人的劳动消耗所需代价就是软件产品的开发成本。**
- 软件产品开发成本的计算方法不同于其它物理产品成本的计算。

- 软件的开发成本是**以一次性开发过程所花费的代价**来计算的。
- 软件开发成本的估算，应是从**软件计划、需求分析、设计、编码、单元测试、组装测试到确认测试**，整个软件开发全过程所花费的代价作为依据的。

软件开发成本估算方法

- **对于一个大型的软件项目，由于项目的复杂性，开发成本的估算不是一件简单的事，要进行一系列的估算处理。主要靠分解和类推。**
- **基本估算方法分为三类。**
  - **自顶向下的估算方法**
  - **自底向上的估计法**
  - **差别估计法**

自顶向下的估算方法

- **这种方法的主要思想是从项目的整体出发，进行类推。**
- **估算人员根据**以前已完成项目所消耗的总成本（或总工作量），**推算将要开发的软件的总成本（或总工作量），然后按比例将它分配到各开发任务单元中去，再来检验它是否能满足要求。**

软件 库存情况更新 开发者 W.Ward 日期 2 / 8 / 82			
阶 段	项目任务	工作量分布(1/53)	小计(1/53)
计划和需求	软件需求定义	5	
	开发计划	1	6
产品设计	产品设计	6	
	初步的用户手册	3	
	测试计划	1	10
详细设计	详细PDL描述	4	
	数据定义	4	
	过程设计	2	
	正式的用户手册	2	12
编码与单元测试	程序编码	6	
	单元测试结果	10	16
组装与联合测试	编写文档	4	
	组装与测试	5	9
总 计			53

- 这种方法的优点是估算工作量小，速度快。
- 缺点是对项目中的特殊困难估计不足，估算出来的成本盲目性大，有时会遗漏被开发软件的某些部分。

自底向上的估计法

- 这种方法的主要思想是把**待开发的软件细分，直到每一个子任务都已经明确所需要的开发工作量**，然后**把它们加起来，得到软件开发的总工作量**。
- 它的优点是估算各个部分的准确性高。缺点是缺少各项子任务之间相互联系所需要的工作量，还缺少许多与软件开发有关的系统级工作量.

差别估计法

- 这种方法综合了上述两种方法的优点，其主要思想是**把待开发的软件项目与过去已完成的软件项目进行类比，从其开发的各个子任务中区分出类似的部分和不同的部分**。
- **类似的部分按实际量进行计算，不同的部分则采用相应方法进行估算**。
- 这种方法的优点是可以提高估算的准确程度，缺点是不容易明确“类似”的界限。

专家判定技术

- **由多位专家进行成本估算**
- 单独一位专家可能会有种种偏见，譬如有乐观的、悲观的、要求在竞争中取胜的、让大家都高兴的种种愿望及政治因素等。
- **最好由多位专家进行估算，取得多个估算值**。
- 有多种方法把这些估算值合成一个估算值。

- 一种方法是**简单地求各估算值的中值或平均值**。其优点是简便。缺点是可能会由于受一、二个极端估算值的影响而产生严重的偏差。
- 一种方法是**召开小组会，使各位专家们统一于或至少同意某一个估算值**。优点是可以摈弃蒙昧无知的估算值，缺点是一些组员可能会受权威或政治因素的影响。

Deiphi技术

- **标准Deiphi技术**

① 组织者发给每位专家一份**软件系统规格说明书**和一张**记录估算值的表格**，请他们进行估算。

② 专家详细研究软件规格说明书的内容，对该软件提出**三个规模**的估算值，即：

a_i（最小） m_i（可能） b_i（最大）

无记名地填写表格

在填表的过程中，专家互相不进行讨论但可以向组织者提问。

③ 组织者对专家们填在表格中的答复进行整理：

a. 计算各位专家估算的期望值 *Ei*;

b. 对专家的估算结果分类摘要。

专家对此估算值另做一次估算。

④ 在综合专家估算结果的基础上，组织专家再次无记名地填写表格。比较两次估算的结果。若差异很大，则要通过查询找出差异的原因。

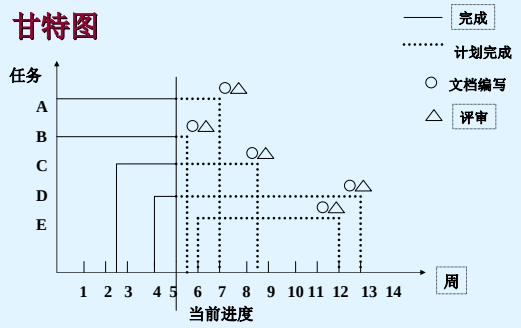
⑤ 上述过程可重复多次。最终可获得一个得到多数专家共识的软件规模（源代码行数）。在此过程中不得进行小组讨论。

- 最后，通过与历史资料进行类比，根据过去完成软件项目的规模和成本等信息，推算出该软件每行源代码所需要的成本。然后再乘以该软件源代码行数的估算值，就可得到该软件的成本估算值。

进度安排

进度安排的图形方法

- 甘特图



- PERT技术和CPM方法

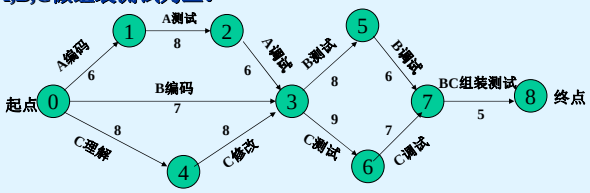
PERT技术叫做计划评审技术

CPM方法叫做关键路径法

它们都是安排开发进度，制定软件开发计划最常用的方法。它们都采用网络图来描述一个项目的任务网络，也就是从一个项目的开始到结束，把应当完成的任务用图的形式表达出来。通常用两张图来表示。一张图给出相见项目的所有任务，另一张图给出应按照什麼次序完成这些任务，给出各任务的衔接。

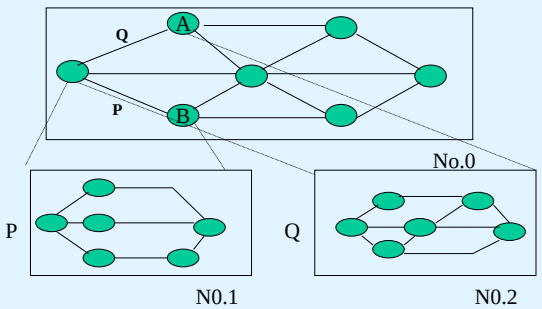
开发模块任务网络图

例：某一项目进入编码阶段考虑如何安排三个模块A,B,C的开发工作，其中A是公用模块，B和C的测试有赖于A的调试C为现成已有的模块，但对它要做理解之后做部分修改。直到A,B,C做组装测试为止。



图中各边表示要完成的任务，数字表示完成该任务的持续时间0为起点，8为终点。

分层任务网络图



在组织较为复杂的项目时或需要对特定的任务进一步做更为详细的计划时,可以使用分层的任务网络图。

软件计划文件与复审

- 软件计划文件
- 软件计划复审

第三部分　软件需求分析

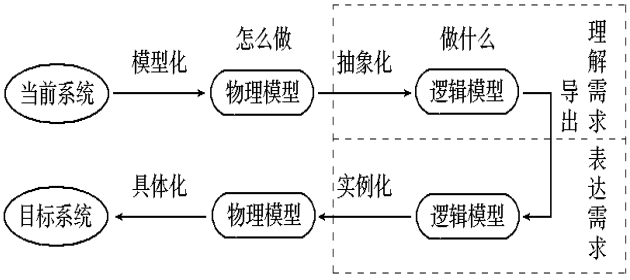
- ❖软件需求分析的任务
- ❖需求分析的过程
- ❖软件需求分析的原则
- ❖软件需求分析方法
- ❖结构化分析方法
- ❖原型化方法
- ❖动态分析方法
- ❖数据及数据库需求

软件需求分析的任务

- 深入描述软件的功能和性能
- 确定软件设计的约束和软件同其它系统元素的接口细节
- 定义软件的其它有效性需求

分析员通过需求分析，逐步细化对软件的要求，描述软件要处理的数据域，给软件开发提供一种可转化为数据设计，结构设计和过程设计的数据与功能表示。制定的 软件需求规格说明还要为评价软件质量提供依据。

- 需求分析研究的对象是软件项目的用户要求
- 准确地表达被接受的用户要求
- 确定被开发软件系统的系统元素
- 将功能和数据结构分配到这些系统元素中



- 需求分析的任务就是借助于当前系统的逻辑模型导出目标系统的逻辑模型，解决目标系统的“做什么”的问题。

- 通常软件开发项目是要实现目标系统的物理模型
- 目标系统的具体物理模型是由它的逻辑模型经实例化，即具体到某个业务领域而得到的

需求分析的过程

(1)问题识别

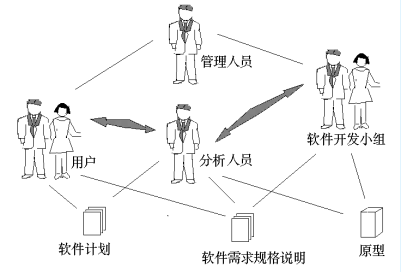
分析人员要研究计划阶段产生的可行性分析报告和软件项目实施计划。

- 从系统的角度来理解软件并评审软件范围是否恰当
- 确定对目标系统的综合要求，即软件的需求
- 提出这些需求实现条件，以及需求应达到的标准

软件需求包括：

- 功能需求
- 性能需求
- 环境需求
- 可靠性需求
- 安全保密要求
- 用户界面需求
- 资源使用需求
- 成本消耗需求
- 开发进度需求
- 预先估计以后系统可能达到的目标

问题识别的另一项工作是建立分析所需要的通信途径，以保证能顺利地对问题进行分析。



（2）分析与综合

需求分析的第二步工作是问题分析和方案的综合。

- 从数据流和数据结构出发，逐步细化所有的软件功能，找出系统各元素之间的联系、接口特性和设计上的约束，分析它们是否满足功能要求，是否合理。剔除其不合理的部分，增加其需要部分。最终综合成系统的解决方案，给出目标系统的详细逻辑模型。

常用的分析方法

- 面向数据流的结构化分析方法（SA）
- 面向数据结构的Jackson方法（JSD）
- 面向对象的分析方法（OOA）等
- 建立动态模型的状态迁移图或Petri网

(3) 编制需求分析阶段的文档

- 软件需求说明书
- 数据要求说明书
- 初步的用户手册
- 修改、完善与确定软件开发实施计划

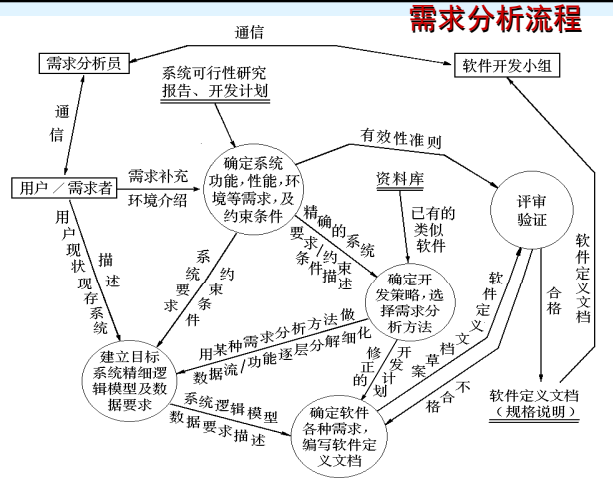
(4) 需求分析评审

- 系统定义的目标是否与用户的要求一致；
- 系统需求分析阶段提供的文档资料是否齐全；
- 文档中的所有描述是否完整、清晰、准确反映用户要求；
- 与所有其它系统成分的重要接口是否都已经描述；

- 被开发项目的数据流与数据结构是否足够，确定；
- 所有图表是否清楚，在不补充说明时能否理解；
- 主要功能是否已包括在规定的软件范围之内，是否都已充分说明；
- 设计的约束条件或限制条件是否符合实际；
- 开发的技术风险是什么；

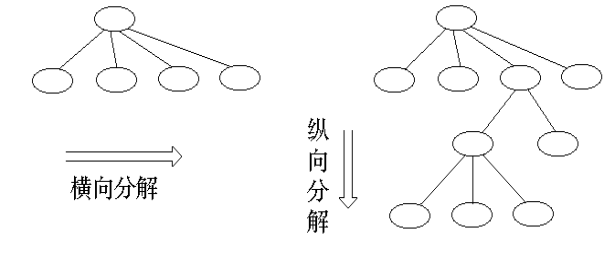
- 是否考虑过软件需求的其它方案；
- 是否考虑过将来可能会提出的软件需求；
- 是否详细制定了检验标准，它们能否对系统定义是否成功进行确认；

需求分析流程



软件需求分析的原则

- 必须能够表达和理解问题的数据域和功能域
- 必须按自顶向下，逐层分解的方式对问题进行分解和不断细化
- 要给出系统的逻辑视图和物理视图



软件需求分析方法

- 需求分析方法由对软件问题的数据域和功能域的系统分析过程及其表示方法组成
- 大多数的需求分析方法是由信息驱动的
- 数据域有三种属性：数据流、数据内容和数据结构。

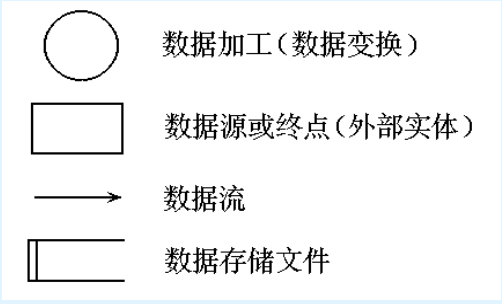
结构化分析方法

- 结构化分析是面向数据流进行需求分析的方法
- 结构化分析方法适合于数据处理类型软件的需求分析

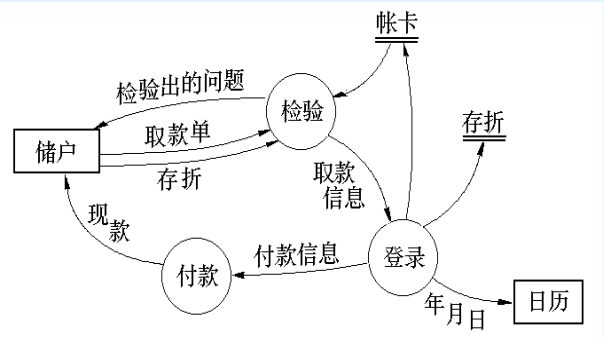
- 具体来说，结构化分析方法就是用抽象模型的概念，按照软件内部数据传递、变换的关系，自顶向下逐层分解，直到找到满足功能要求的所有可实现的软件为止
- 结构化分析方法使用工具：数据流图，数据词典，结构化英语，判定表与判定树

数据流图

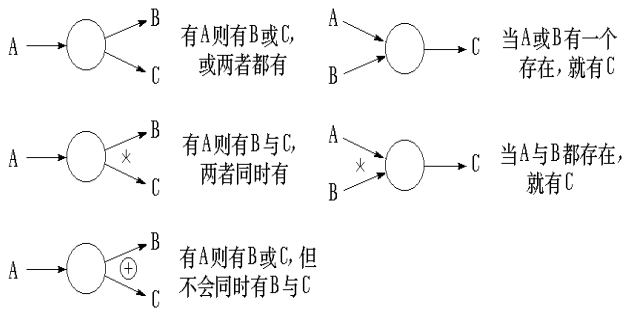
- 数据流图中的主要图形元素



例：办理取款手续的数据流图



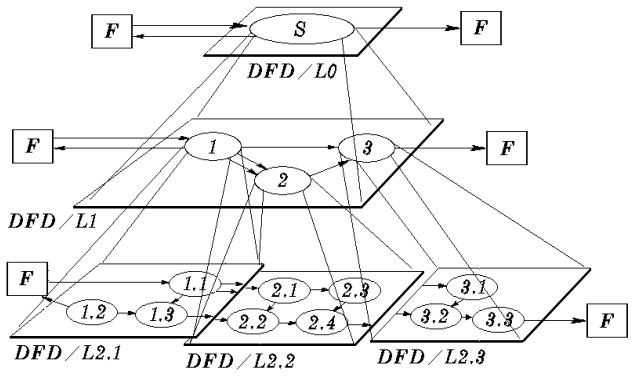
数据流与数据加工之间的关系



分层的数据流图

- 为了表达数据处理过程的数据加工情况，需要采用层次结构的数据流图。按照系统的层次结构进行逐步分解，并以分层的数据流图反映这种结构关系，能清楚地表达和容易理解整个系统

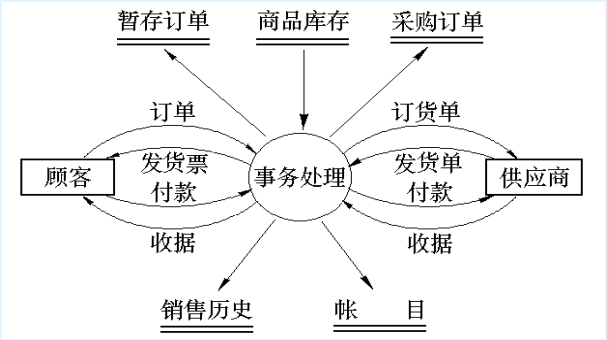
分层数据流图



- 在多层数据流图中，**顶层流图**仅包含**一个加工**，它代表被开发系统。它的输入流是该系统的输入数据，输出流是系统所输出数据
- **底层流图**是指其**加工不需再做分解**的数据流图，它处在最底层
- **中间层流图**则表示**对其上层父图的细化**。它的每一加工可能继续细化，形成子图。

结构化分析方法步骤示例

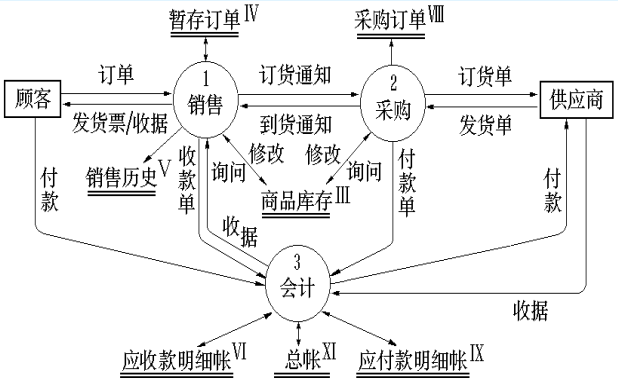
商店业务处理系统

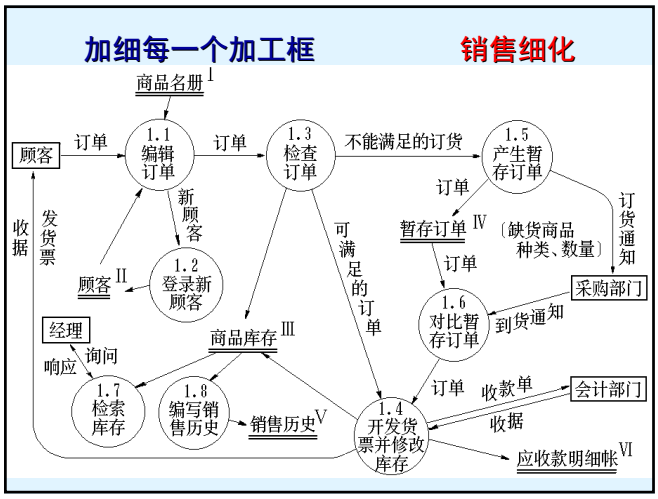


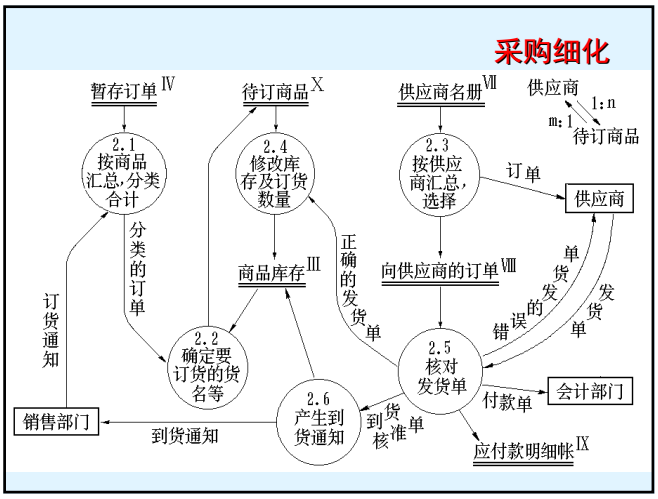
- **这个数据流图只是一个高层的系统逻辑模型，它反映了目标系统要实现的功能**
- **数据流图绘制步骤**
  - **首先确定系统的输入和输出**
  - **根据商店业务，画出顶层数据流图，以反映最主要业务处理流程**

- **经过分析，商店业务处理的主要功能应当有销售、采购、会计三大项。主要数据流输入的源点和输出终点是顾客和供应商。**
- **然后从输入端开始，根据商店业务工作流程，画出数据流流经的各加工框，逐步画到输出端，得到第一层数据流图**

第一层数据流图







检查和修改数据流图的原则

- 数据流图上所有图形符号只限于前述四种基本图形元素
- 数据流图的主图必须包括前述四种基本元素，缺一不可
- 数据流图的主图上的数据流必须封闭在外部实体之间
- 每个加工至少有一个输入数据流和一个输出数据流

- 在数据流图中，需按层给加工框编号。编号表明该加工所处层次及上下层的亲子关系
- 规定任何一个数据流子图必须与它上一层的一个加工对应，两者的输入数据流和输出数据流必须一致。此即父图与子图的平衡
- 可以在数据流图中加入物质流，帮助用户理解数据流图

- 图上每个元素都必须有名字
- 数据流图中不可夹带控制流
- 初画时可以忽略琐碎的细节，以集中精力于主要数据流

数据词典

- 数据词典与数据流图配合，能清楚地表达数据处理的要求
- 词条描述 —— 对于在数据流图中每一个被命名的图形元素，均加以定义，其内容有:名字，别名或编号，分类，描述，定义，位置，其它，等

（1）数据流词条描述

- 数据流名：
- 说明：简要介绍作用即它产生的原因和结果
- 数据流来源：来自何方
- 数据流去向：去向何处
- 数据流组成：数据结构
- 数据量流通量：数据量，流通量

（2）数据元素词条描述

- 数据元素名：
- 类型：数字（离散值，连续值），文字（编码类型）
- 长度：
- 取值范围：
- 相关的数据元素及数据结构：

（3）数据文件词条描述

- 数据文件名：
- 简述：存放的是什么数据
- 输入数据：
- 输出数据：
- 数据文件组成：数据结构
- 存储方式：顺序，直接，关键码
- 存取频率：

（4）加工逻辑词条描述

- 加工名：
- 加工编号：反映该加工的层次
- 简要描述：加工逻辑及功能简述
- 输入数据流：
- 输出数据流：
- 加工逻辑：简述加工程序，加工顺序

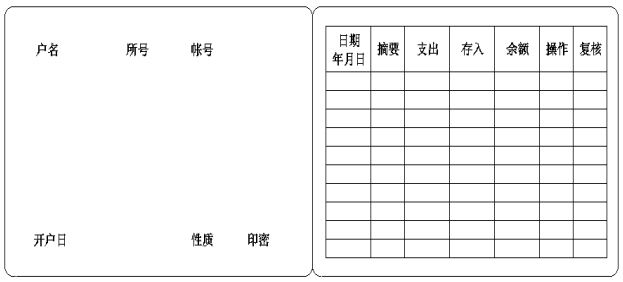
（5）源点及汇(终)点词条描述

- 名称：外部实体名
- 简要描述：什么外部实体
- 有关数据流：
- 数目：

数据结构的描述

符 号	含 义	举 例
=	被定义为	
+	与	$x = a+b$
[...,...] 或 [...|...]	或	$x = [a,b]$，$x = [a\|b]$
{ ... }或 m{...}n	重复	$x = \{a\}$，$x = 3\{a\}8$
(...)	可选	$x = (a)$
"..."	基本数据元素	$x =$ "a"
..	连结符	$x = 1..9$

存折格式



- 存折＝户名＋所号＋帐号＋开户日＋性质＋（印密）＋1{存取行}50
- 户名＝2{字母}24
- 所号＝“001”..“999”
- 帐号＝“00000001”..“99999999”
- 开户日＝年＋月＋日
- 性质＝“1”..“6” 注：“1”表示普通户，“5”表示工资户等
- 印密＝“0” 注：印密在存折上不显示
- 存取行＝日期＋（摘要）＋支出＋存入＋余额＋操作＋复核

基本加工逻辑说明

- 对数据流图的每一个基本加工，必须有一个基本加工逻辑说明
- 基本加工逻辑说明必须描述基本加工如何把输入数据流变换为输出数据流的加工规则
- 加工逻辑说明必须描述实现加工的策略而不是实现加工的细节
- 加工逻辑说明中包含的信息应是充足的，完备的，有用的，没有重复的多余信息

用于写加工逻辑说明的工具

- 结构化英语
- 判定表
- 判定树

（1）结构化英语(PDL)语言

- 结构化英语的词汇表由
  - 英语命令动词
  - 数据词典中定义的名字
  - 有限的自定义词
  - 逻辑关系词 IF_THEN_ELSE、CASE_OF 、 WHILE_DO、REPEAT_UNTIL等组成。

- 是一种介于自然语言和形式化语言之间的语言
- 语言的正文用基本控制结构进行分割，加工中的操作用自然语言短语来表示
- 其基本控制结构有三种：
  - 简单陈述句结构：避免复合语句；
  - 重复结构：WHILE_DO 或 REPEAT_UNTIL结构。
  - 判定结构：IF_THEN_ELSE 或 CASE_OF结构；

商店业务处理系统中“检查发货单”

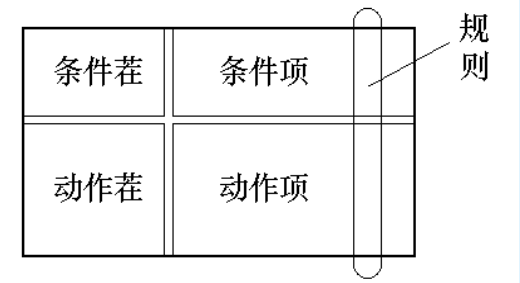
```
IF 发货单金额超过$500 THEN
    IF 欠款超过了60天 THEN
        在偿还欠款前不予批准
    ELSE （欠款未超期）
        发批准书，发货单
    ENDIF
ELSE （发货单金额未超过$500）
    IF 欠款超过60天 THEN
        发批准书，发货单及赊欠报告
ELSE （欠款未超期）
        发批准书，发货单
    ENDIF
ENDIF
```

（2）判定表

- 如果数据流图的加工需要依赖于多个逻辑条件的取值，使用判定表来描述比较合适

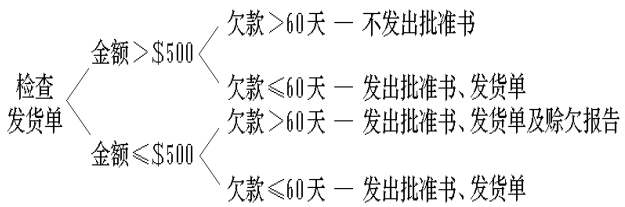
以“检查发货单”为例

		1	2	3	4
条件	发货单金额	＞$500	＞$500	≤$500	≤$500
	赊欠情况	＞60天	≤60天	＞60天	≤60天
操作	不发出批准书	√			
	发出批准书		√	√	√
	发出发货单		√	√	√
	发出赊欠报告			√	



（3）判定树

- 判定树也是用来表达加工逻辑的一种工具。有时侯它比判定表更直观。



原型化方法

- 在开发初期，要想得到一个完整准确的规格说明不是一件容易的事。特别是对一些大型的软件项目。
- 用户往往对系统只有一个模糊的想法，很难完全准确地表达对系统的全面要求。

- 软件开发者对于所要解决的应用问题认识更是模糊不清
- 随着开发工作向前推进，用户可能会产生新的要求，或因环境变化，要求系统也能随之变化；开发者又可能在设计与实现的过程中遇到些没有预料到的实际困难，需要以改变需求来解脱困境。

- 因此规格说明难以完善、需求的变更、以及通信中的模糊和误解，都会成为软件开发顺利推进的障碍。
- 为了解决这些问题，逐渐形成了软件系统的快速原型的概念。

软件原型的分类

- 在软件开发中，原型是软件的一个早期可运行的版本，它反映最终系统的部分重要特性。
  - 探索型：目的是要弄清对目标系统的要求，确定所希望的特性，并探讨多种方案的可行性。

  - 实验型：这种原型用于大规模开发和实现之前，考核方案是否合适，规格说明是否可靠。
  - 进化型：这种原型的目的不在于改进规格说明，而是将系统建造得易于变化，在改进原型的过程中，逐步将原型进化成最终系统。

原型使用策略

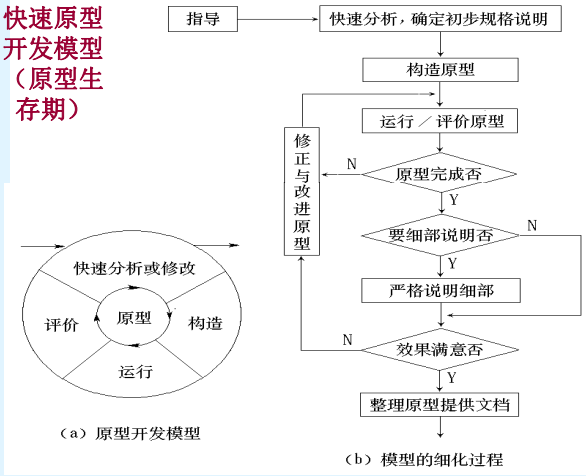
- 废弃策略
- 追加策略

建立快速原型，进行系统的分析和构造的好处：

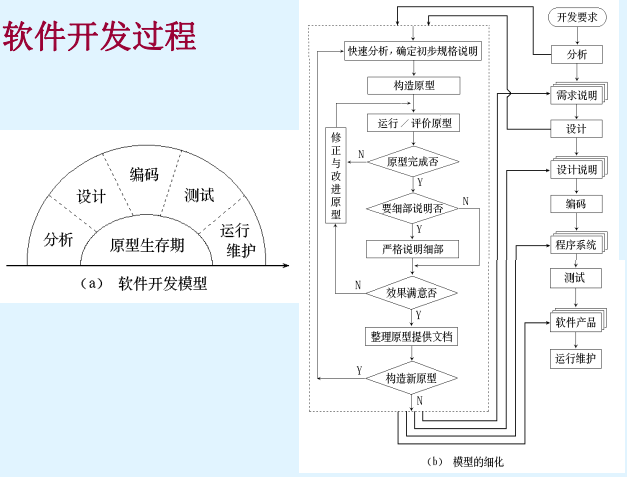
- 增进软件人员和用户对系统服务需求的理解，使比较含糊的具有不确定性的软件需求（主要是功能）明确化。
- 软件原型化方法提供了一种有力的学习手段。

– 使用原型化方法，可以容易地确定系统的性能，确认各项主要系统服务的可应用性，确认系统设计的可行性，确认系统作为产品的结果。
– 软件原型的最终版本，有的可以原封不动地成为产品，有的略加修改就可以成为最终系统的一个组成部分，这样有利于建成最终系统。

快速原型开发模型（原型生存期）



软件开发过程



软件复用技术

- 利用可复用的模块，做出适当的组合，就可得到快速构造的原型系统。
- 为了快速地构造原型，这些模块首先必须有简单而清晰的界面；其次它们应当尽量不依赖其它的模块或数据结构；第三，它们应具有一些通用的功能。

- 软件复用的范围尚无严格定义，它包容了用来开发软件的任何信息的复用。包括现有软件开发方法论的复用；软件要求，规格说明和设计的复用；源代码，模块和操作系统的复用；文档的复用；分析数据的复用；测试信息的复用；维护信息数据库的复用等等。软件工具与支撑环境的复用也属于软件复用工作的范围。

- 软件复用的范围基本上规为五个层次：
  - 复用数据
  - 复用模块
  - 复用结构
  - 复用设计
  - 复用规格说明

软件复用技术

- 合成技术：构件是复用的基础。构件方法以抽象数据类型为基础，将功能与数据结构封装在构件内部，构件可以是对某一个函数，过程，子程序，数据类型，算法等可复用软件成分的抽象。

形成大构件的方法：

连接；

消息传递和继承；

管道机制

生成技术：生成技术利用可复用的模式，通过生成程序产生一个新的程序或程序段，产生的程序可以看成是程序的实例。两种复用模式：

代码模式：将可复用的代码模式存于生成器内，通过特定的参数替换，生成抽象软件模块的具体实现。

规则模式：利用变换规则集合。其变换方法中通常采用高级的规格说明语言，形式化地给出软件的需求规格说明，利用程序变换系统把采用高级的规格说明语言编写的程序转化成某种可执行语言的程序。

系统动态分析

- 系统的需求规格说明通常是用自然语言来叙述的，但是用自然语言描述往往会出现歧义性。
- 为了直观地分析系统的动作，从特定的视点出发描述系统的行为，需要采用动态分析的方法。

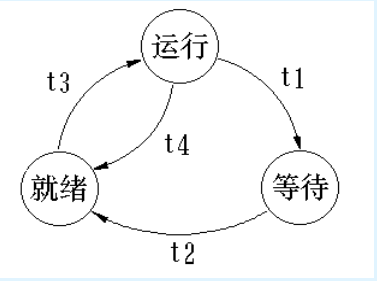
最常用的动态分析方法

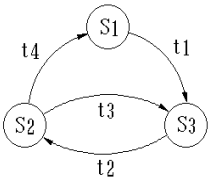
- 状态迁移图
- 时序图
- Petri网

状态迁移图

- 状态迁移图是描述系统的状态如何相应外部的信号进行推移的一种图形表示。
  - 圆圈“○”表示可得到的系统状态
  - 箭头“→”表示从一种状态向另一种状态的迁移。

例如，当有多个申请占用CPU运行的进程时，有关CPU分配的进程的状态迁移。





（a）状态迁移图

状态 / 事件	S1	S2	S3
t1	S3		
t2			S2
t3		S3	
t4		S1	

（b） 状态迁移表

- 可得到的状态＝就绪，运行，等待
- 生成的事件＝t1，t2， t3， t4
  - t1 — 中断事件 • t2 — 中断已处理
  - t3 — 分配CPU • t4 — 用完CPU时间

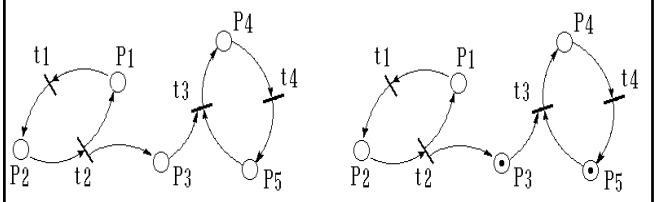
状态迁移图的优点

- 状态之间的关系能够直观地捕捉到
- 由于状态迁移图的单纯性，能够机械地分析许多情况，可很容易地建立分析工具

Petri网

- Petri网已广泛地应用于硬件与软件系统的开发中，它适用于描述与分析相互独立、协同操作的处理系统，也就是并发执行的处理系统。

- Petri网简称PNG (Petri Net Graph)，是一种有项图，它有两种结点：
  - –位置(*place*)：符号为“○”，它用来表示系统的状态。
  - –转移(*transition*)：符号为 “|”或
  - –“—”, 它用来表示系统中的事件。
  - –图中的有向边表示对转移的输入，或由转移的输出



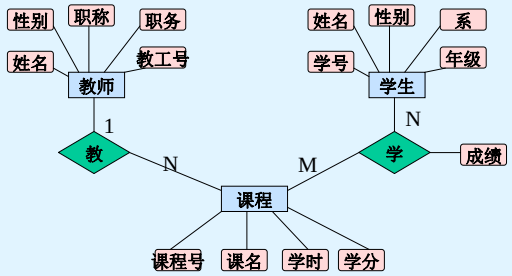
（a）Petri网 （b）加了标记的Petri网

- 标记，或称令牌(*token*)，是表明系统当前处于什么状态的标志

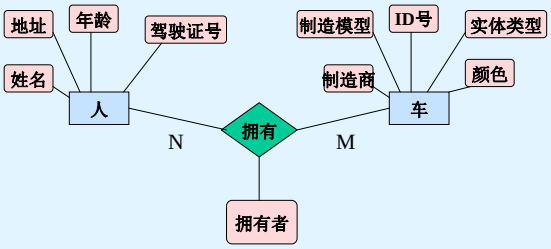
- 图中，表示位置P3与P5的圆圈中间点了一个黑点，称为标记。标记在位置上的出现表明了处理要求的到来。也就是说事件t3激发的两个前提都已具备，转移t3激发。
- 作为执行结果，位置p3与p5上的标记移去，移到了位置p4上。反过来，当激发产生的结果有几个时，将随机地选择一个结果输出，并把作为结果的位置的状态加上标记。

E-R方法（Entity-Relationship Approach）实体关系模型

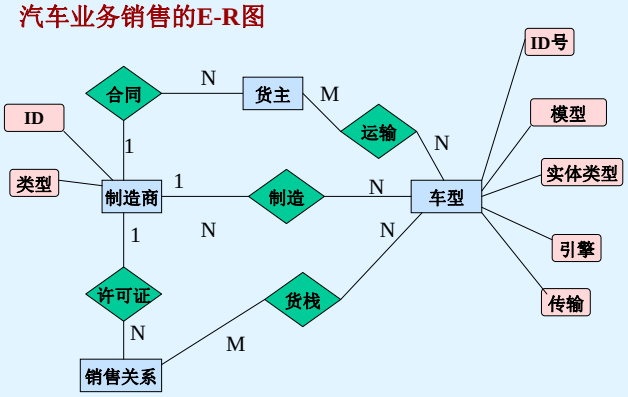
教师-学生-课程E-R 图



人与车关系E-R 图



汽车业务销售的E-R图



数据结构的规范化

三个条件：

- 表格中每个信息项必须是一个不可分割的数据项。
- 表格中每一列中所有信息项必是同一类型，名字各异
- 表格中各行互不相同

不满足以上关系的叫做非规范化关系

例：教学管理

有三个实体型，课程，学生，教师

学生（学号，姓名，性别，年龄，专业，籍贯）

教师（职工号，姓名，年龄，职称，工资级别，工资）

课程（课程号，课程名，学分，学时，课程类别）

为了表示实体型之间的关系，又建立两个关系：

选课（学号，课程号，听课出勤率，作业完成率，分数）

教课（职工号，课程号）

这五个关系组成了数据库模型。在每个关系中，属性加下划线指明关键字。

关系的规范化程度通常按属性间的依赖程度来区分，并以范式来表达。

判断规范化程度的条件：

1) 关系中所有属性都是“单纯域”，即不出现表中有表；
2) 非主属性完全函数依赖于关键字
3) 非主属性相互独立，即任何非主属性间不存在函数依赖

如果满足关系1)则为第一范式（1NF））

如果满足关系1)和2)为第二范式(2NF)

如果满足关系1)，2)和3)为第三范式（3NF)

第四部分　软件设计

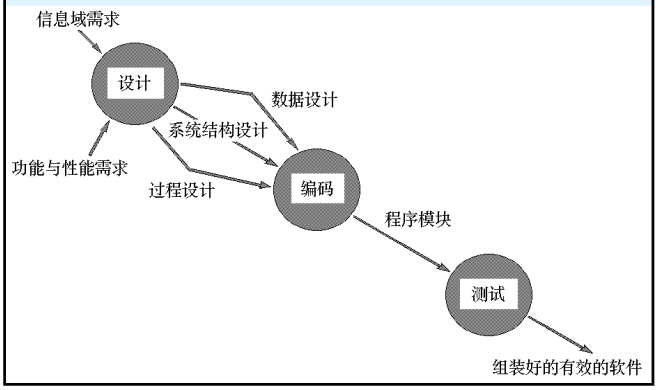
- ❖软件设计的目标和任务
- ❖软件设计基础
- ❖模块独立性
- ❖结构化设计方法
- ❖数据设计和文件设计
- ❖过程设计

软件设计的目标和任务

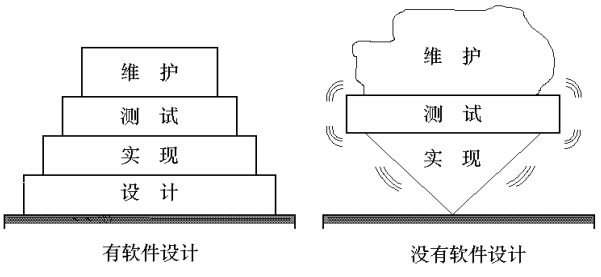
- 根据用信息域表示的软件需求，以及功能和性能需求，进行
  - 数据设计
  - 系统结构设计
  - 过程设计。

- 数据设计侧重于数据结构的定义。
- 系统结构设计定义软件系统各主要成份之间的关系。
- 过程设计则是把结构成分转换成软件的过程性描述。在编码步骤，根据这种过程性描述，生成源程序代码，然后通过测试最终得到完整有效的软件。

开发阶段的信息流



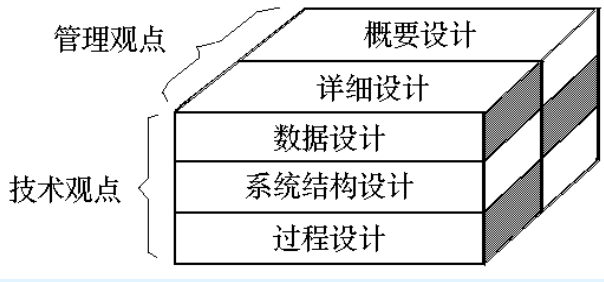
- 软件设计是后续开发步骤及软件维护工作的基础。如果没有设计，只能建立一个不稳定的系统结构



软件设计任务

- 软件设计是一个把软件需求变换成软件表示的过程。
- 从工程管理的角度来看，软件设计分两步完成。
  - 概要设计，将软件需求转化为数据结构和软件的系统结构。
  - 详细设计，即过程设计。通过对结构表示进行细化，得到软件的详细的数据结构和算法。

从管理和技术两个不同角度对设计的认识



软件设计过程（概要设计）

（1）．制定规范

- 在进入软件开发阶段之初，首先应为软件开发组制定在设计时应该共同遵守的标准，以便协调组内各成员的工作。包括:

- 阅读和理解软件需求说明书，确认用户要求能否实现，明确实现的条件，从而确定设计的目标，以及它们的优先顺序
- 根据目标确定最合适的设计方法
- 规定设计文档的编制标准
- 规定编码的信息形式，与硬件，操作系统的接口规约，命名规则

(2)．软件系统结构的总体设计

- 基于功能层次结构建立系统。
  - 采用某种设计方法，将系统按功能划分成模块的层次结构
  - 确定每个模块的功能
  - 建立与已确定的软件需求的对应关系
  - 确定模块间的调用关系
  - 确定模块间的接口即模块间传递的信息
  - 评估模块划分的质量及导出模块结构的规则

（3）．处理方式设计

- 确定为实现系统的功能需求所必需的算法，评估算法的性能
- 确定为满足系统的性能需求所必需的算法和模块间的控制方式。主要性能指标：
  - 周转时间（从输入—处理—输出结果为止整个）
  - 响应时间（用户向计算机发出请求之后，一次输入输出的时间）
  - 吞吐量（单位时间内能够处理的数据量）
  - 精度（运算精确度的要求）
- 确定外部信号的接收发送形式

（4）．数据结构设计

确定软件涉及的文件系统的结构以及数据库的模式、子模式，进行数据完整性和安全性的设计

- 确定输入，输出文件的详细的数据结构
- 结合算法设计，确定算法所必需的逻辑数据结构及其操作
- 确定对逻辑数据结构所必需的那些操作的程序模块(软件包)

- 若需要与操作系统或调度程序接口所必须的控制表等数据时，确定其详细的数据结构和使用规则
- 数据的保护性设计

防卫性设计：

一致性设计：

冗余性设计：

（5）. 可靠性设计

- **可靠性设计也叫做质量设计**
- **在运行过程中，为了适应环境的变化和用户新的要求，需经常对软件进行改造和修正。在软件开发的一开始就要确定软件可靠性和其它质量指标，考虑相应措施，以使得软件易于修改和易于维护**

（6）. 编写概要设计阶段的文档

- **概要设计阶段完成时应编写以下文档：**
  - **概要设计说明书**
  - **数据库设计说明书**
  - **用户手册**
  - **制定初步的测试计划**

（7）. 概要设计评审

- 可追溯性：确认该设计是否复盖了所有已确定的软件需求，软件每一成份是否可追溯到某一项需求
- 接口：确认该软件的内部接口与外部接口是否已经明确定义。模块是否满足高内聚和低耦合的要求。模块作用范围是否在其控制范围之内
- 风险：确认该设计在现有技术条件下和预算范围内是否能按时实现

- 实用性：确认该设计对于需求的解决方案是否实用
- 技术清晰度：确认该设计是否以一种易于翻译成代码的形式表达
- 可维护性：确认该设计是否考虑了方便未来的维护
- 质量：确认该设计是否表现出良好的质量特征

- 各种选择方案：看是否考虑过其它方案，比较各种选择方案的标准是什么
- 限制：评估对该软件的限制是否现实，是否与需求一致
- 其它具体问题：对于文档、可测试性、设计过程. . 等进行评估

详细设计

- 在详细设计过程中，需要完成的工作是：
  (1).确定软件各个组成部分内的算法以及各部分的内部数据组织
  (2).选定某种过程的表达形式来描述各种算法。
  (3).进行详细设计的评审

软件设计基础

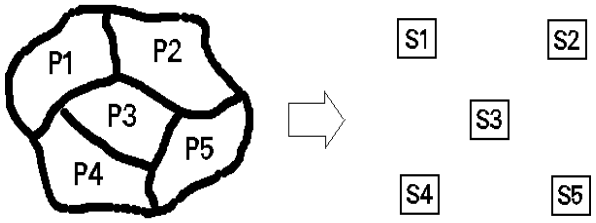
- 自顶向下，逐步细化
- 软件结构
- 程序结构
- 结构图
- 模块化
- 抽象化
- 信息隐蔽

自顶向下，逐步细化

- 将软件的体系结构按自顶向下方式，对各个层次的过程细节和数据细节逐层细化，直到用程序设计语言的语句能够实现为止，从而最后确立整个的体系结构。

软件结构

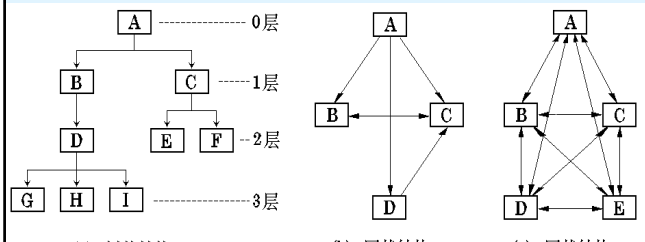
- 软件结构包括两部分。程序的模块结构和数据的结构
- 软件的体系结构通过一个划分过程来完成。该划分过程从需求分析确立的目标系统的模型出发，对整个问题进行分割，使其每个部分用一个或几个软件成份加以解决，整个问题就解决了



需要通过软件解决的"问题"　　软件的"解决方案"

程序结构

- 程序结构表明了程序各个部件(模块)的组织情况，是软件的过程表示。

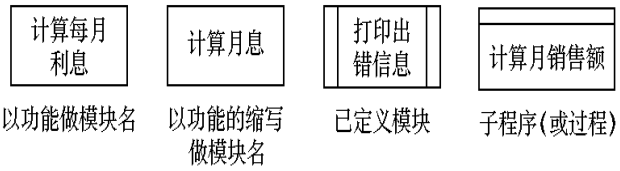


(a) 树状结构　(b) 网状结构　(c) 网状结构

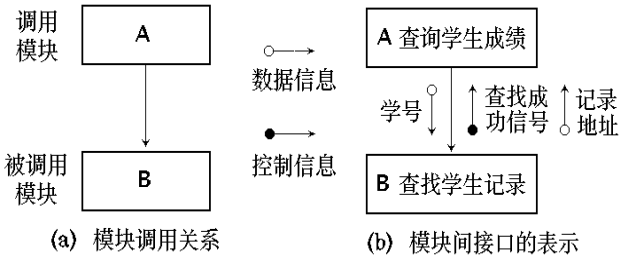
结构图

- 结构图反映程序中模块之间的层次调用关系和联系：它以特定的符号表示模块、模块间的调用关系和模块间信息的传递

① 模块：模块用矩形框表示，并用模块的名字标记它。

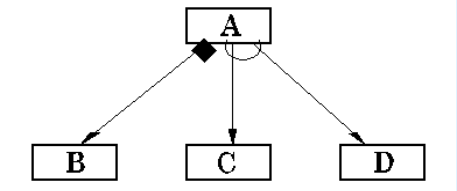


② 模块的调用关系和接口：模块之间用单向箭头联结，箭头从调用模块指向被调用模块，表示调用模块调用了被调用模块。



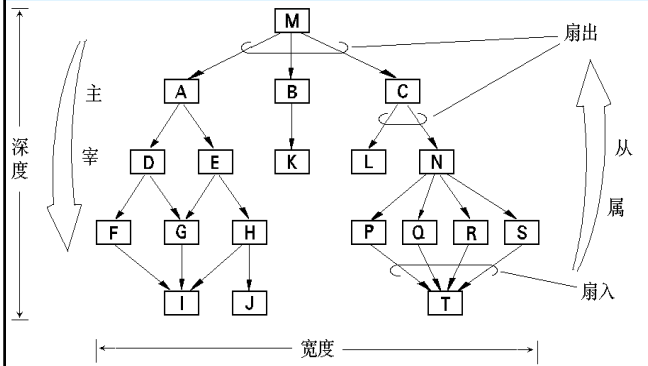
③ 模块间的信息传递：当一个模块调用另一个模块时，调用模块把数据或控制信息传送给被调用模块，以使被调用模块能够运行。而被调用模块在执行过程中又把它产生的数据或控制信息回送给调用模块

④ 在模块A的箭头尾部标以一个菱形符号，表示模块A有条件地调用另一个模块B。当一个在调用箭头尾部标以一个弧形符号，表示模块A反复调用模块C和模块D。



条件调用和循环调用

软件系统的分层模块结构图



模块化

- 软件系统的模块化是指整个软件被划分成若干单独命名和可编址的部分，称之为模块。这些模块可以被组装起来以满足整个问题的需求。
- 把问题／子问题的分解与软件开发中的系统／子系统或系统／模块对应起来，就能够把一个大而复杂的软件系统划分成易于理解的比较单纯的模块结构。

抽象化

- 软件系统进行模块设计时，可有不同的抽象层次。
- 在最高的抽象层次上，可以使用问题所处环境的语言概括地描述问题的解法。
- 在较低的抽象层次上，则采用过程化的方法。

(1) 过程的抽象

在软件工程中，从系统定义到实现，每进展一步都可以看做是对软件解决方法的抽象化过程的一次细化。

- 在软件需求分析阶段，用“问题所处环境的为大家所熟悉的术语”来描述软件的解决方法。
- 在从概要设计到详细设计的过程中，抽象化的层次逐次降低。当产生源程序时到达最低抽象层次。

例：开发一个CAD软件时的三种抽象层次

- 抽象层次Ⅰ. 用问题所处环境的术语来描述这个软件：
  该软件包括一个计算机绘图界面，向绘图员显示图形，以及一个数字化仪界面，用以代替绘图板和丁字尺。所有直线、折线、矩形、圆及曲线的描画、所有的几何计算、所有的剖面图和辅助视图都可以用这个CAD软件实现……。

- 抽象层次Ⅱ. 任务需求的描述。

```
CAD SOFTWARE TASKS
  user interaction task;
  2-D drawing creation task;
  graphics display task;
  drawing file management task;
end.
```

在这个抽象层次上，未给出“怎样做”的信息，不能直接实现。

- 抽象层次Ⅲ. 程序过程表示。以2-D(二维)绘图生成任务为例:

```
PROCEDURE：2-D drawing creation
REPEAT UNTIL (drawing creation task terminates)
  DO WHILE (digitizer interaction occurs)
  digitizer interface task;
    DETERMINE drawing request CASE;
      line：        line drawing task;
      rectangle：   rectangle drawing task;
      circle：      circle drawing task;
      ……
```

(2) 数据抽象

在不同层次上描述数据对象的细节，定义与该数据对象相关的操作。例如，在CAD软件中，定义一个叫做drawing的数据对象。可将drawing规定为一个抽象数据类型，定义它的内部细节为:

- TYPE drawing IS STRUCTURE
  DEFIND
    number IS STRING LENGTH(12);
    geometry DEFIND ……
    notes IS STRING LENGTH(256);
    BOM DEFIND
  END drawing TYPE;

- 数据抽象drawing本身由另外一些数据抽象，如geometry、BOM (bill of materials) 构成
- 定义drawing的抽象数据类型之后，可引用它来定义其它数据对象，而不必涉及drawing的内部细节
- 例如，定义:
  blue-print IS INSTANCE OF drawing;
  或
- schematic IS INSTANCE OF drawing;

信息隐蔽

- 由 parnas 方法提倡的信息隐蔽是指，每个模块的实现细节对于其它模块来说是隐蔽的。也就是说，模块中所包含的信息（包括数据和过程）不允许其它不需要这些信息的模块使用。

模块的独立性

- 模块（Module）

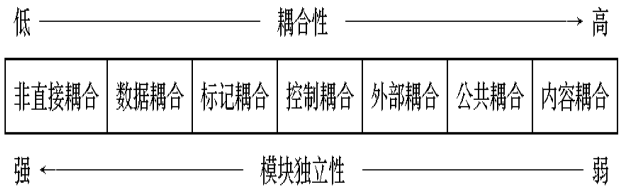
  “模块”，又称“组件”。它一般具有如下三个基本属性:
  - 功能：描述该模块实现什么功能
  - 逻辑：描述模块内部怎么做
  - 状态：该模块使用时的环境和条件

- 在描述一个模块时，还必须按模块的外部特性与内部特性分别描述
- 模块的外部特性
  - 模块的模块名、参数表、其中的输入参数和输出参数，以及给程序以至整个系统造成的影响
- 模块的内部特性
  - 完成其功能的程序代码和仅供该模块内部使用的数据

- **模块独立性**
  - 模块独立性，是指软件系统中每个模块只涉及软件要求的具体的子功能，而和软件系统中其它的模块的接口是简单的
  - 例如，若一个模块只具有单一的功能且与其它模块没有太多的联系，则称此模块具有模块独立性
  - 一般采用两个准则度量模块独立性。即模块间耦合和模块内聚

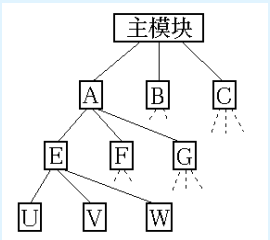
  - 耦合是模块之间的互相连接的紧密程度的度量。
  - 内聚是模块功能强度(一个模块内部各个元素彼此结合的紧密程度)的度量。
  - 模块独立性比较强的模块应是高内聚低耦合的模块。

模块间的耦合



非直接耦合(Nondirect Coupling)

如果两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的，这就是非直接耦合。这种耦合的模块独立性最强。



数据耦合 (Data Coupling)

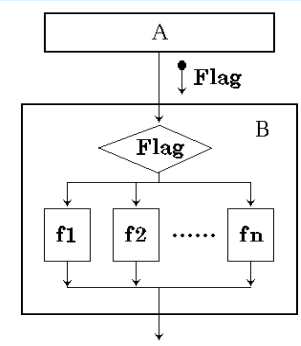
如果一个模块访问另一个模块时，彼此之间是通过简单数据参数（不是控制参数、公共数据结构或外部变量）来交换输入、输出信息的，则称这种耦合为数据耦合。

标记耦合 (Stamp Coupling)

如果一组模块通过参数表传递记录信息，就是标记耦合。这个记录是某一数据结构的子结构，而不是简单变量。

控制耦合 (Control Coupling)

如果一个模块通过传送开关、标志、名字等控制信息，明显地控制选择另一模块的功能，就是控制耦合。



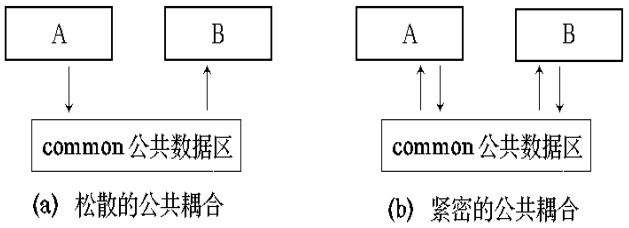
外部耦合（External Coupling）

一组模块都访问**同一全局简单变量**而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

公共耦合（Common Coupling）

若一组模块都访问**同一个公共数据环境**，则它们之间的耦合就称为公共耦合。公共的数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。

- 公共耦合的复杂程度随耦合模块的个数增加而显著增加。若只是**两模块间有公共数据环境**，则公共耦合有两种情况。松散公共耦合和紧密公共耦合。



(a) 松散的公共耦合　　(b) 紧密的公共耦合

内容耦合 (Content Coupling)

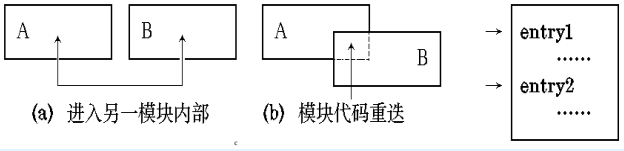
如果发生下列情形，两个模块之间就发生了内容耦合

(1) 一个模块直接访问另一个模块的内部数据;

(2) 一个模块不通过正常入口转到另一模块内部;

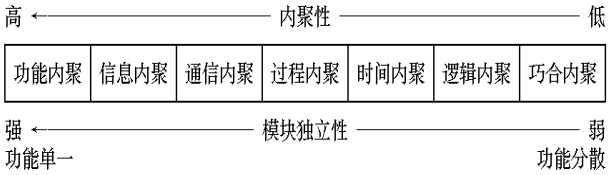
(3) 两个模块有一部分程序代码重迭(只可能出现在汇编语言中);

(4) 一个模块有多个入口。



(a) 进入另一模块内部　　(b) 模块代码重迭

模块内聚



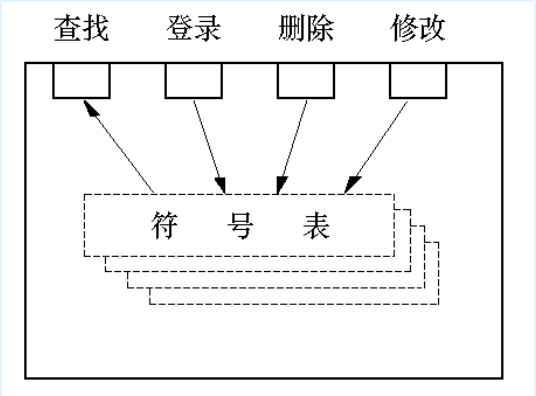
功能内聚 (Functional Cohesion)

一个模块中各个部分都是**完成某一具体功能**必不可少的组成部分，或者说该模块中所有部分都是为了完成一项具体功能而协同工作，紧密联系，不可分割的。则称该模块为功能内聚模块。

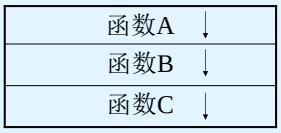
函数A-处理1
函数B-处理2
函数C-处理3

信息内聚 (Informational Cohesion)

这种模块**完成多个功能，各个功能都在同一数据结构上操作**，每一项功能有一个唯一的入口点。这个模块将根据不同的要求，确定该执行哪一个功能。由于这个模块的所有功能都是基于同一个数据结构（符号表），因此，它是一个信息内聚的模块。

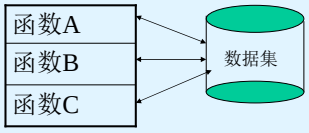


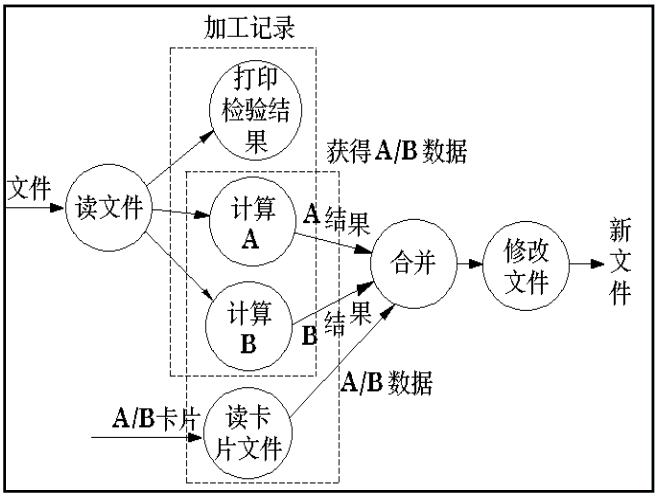
- 信息内聚模块可以看成是多个功能内聚模块的组合，并且达到信息的隐蔽。即把某个数据结构、资源或设备隐蔽在一个模块内，不为别的模块所知晓。



通信内聚 (Communication Cohesion)

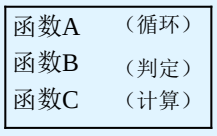
如果一个模块内各功能部分都使用了相同的输入数据，或产生了相同的输出数据，则称之为通信内聚模块。通常，通信内聚模块是通过数据流图来定义的。





过程内聚（Procedural Cohesion）

使用流程图做为工具设计程序时，把流程图中的某一部分划出组成模块，就得到过程内聚模块。例如，把流程图中的循环部分、判定部分、计算部分分成三个模块，这三个模块都是过程内聚模块。

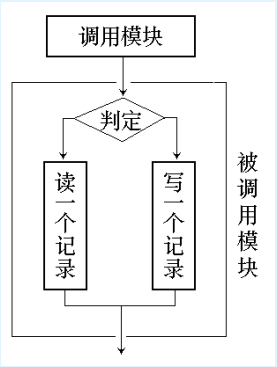


时间内聚（Classical Cohesion）

时间内聚又称为经典内聚。这种模块大多为多功能模块，但模块的各个功能的执行与时间有关，通常要求所有功能必须在同一时间段内执行。例如初始化模块和终止模块。

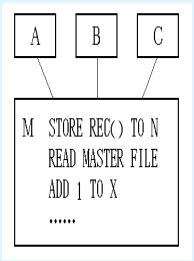
逻辑内聚（Logical Cohesion）

这种模块把几种相关的功能组合在一起，每次被调用时，由传送给模块的判定参数来确定该模块应执行哪一种功能。



巧合内聚（Coincidental Cohesion）

巧合内聚又称为偶然内聚。当模块内各部分之间没有联系，或者即使有联系，这种联系也很松散，则称这种模块为巧合内聚模块，它是内聚程度最低的模块。



函数A	
函数B	函数C
函数D	函数E

结构化设计方法

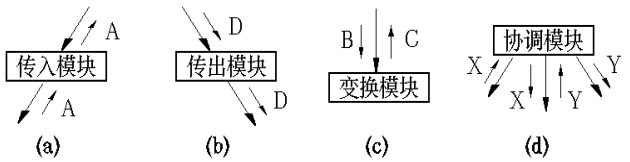
- 首先研究、分析和审查数据流图。从软件的需求规格说明中弄清数据流加工的过程，对于发现的问题及时解决。
- 然后根据数据流图决定问题的类型。数据处理问题典型的类型有两种：**变换型**和**事务型**。针对两种不同的类型分别进行分析处理。

- 由数据流图推导出系统的初始结构图。
- 利用一些启发式原则来改进系统的初始结构图，直到得到符合要求的结构图为止。
- 修改和补充数据词典。
- 制定测试计划。

在系统结构图中的模块

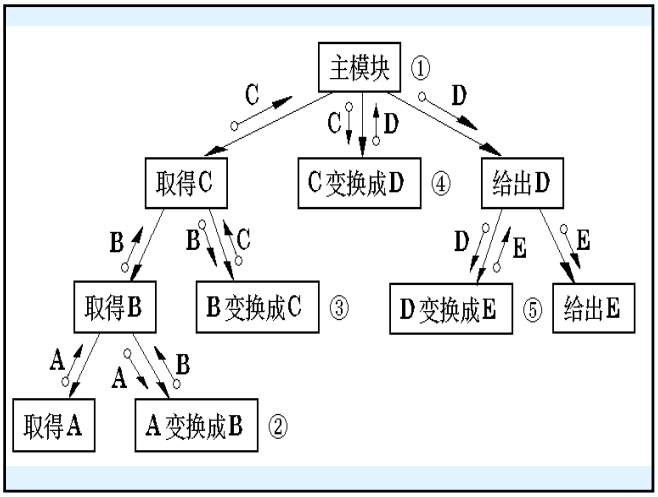
- 传入模块 — 从下属模块取得数据，经过某些处理，再将其传送给上级模块。它传送的数据流叫做逻辑输入数据流。
- 传出模块 — 从上级模块获得数据，进行某些处理，再将其传送给下属模块。它传送的数据流叫做逻辑输出数据流。

- 变换模块 — 它从上级模块取得数据，进行特定的处理，转换成其它形式，再传送回上级模块。它加工的数据流叫做变换数据流。
- 协调模块 — 对所有下属模块进行协调和管理的模块。



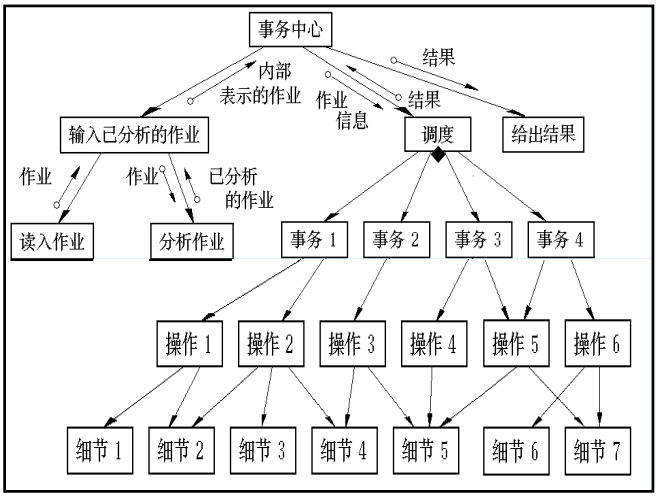
变换型系统结构图

- 变换型数据处理问题的工作过程大致分为三步，即取得数据，变换数据和给出数据。
- 相应于取得数据、变换数据、给出数据，变换型系统结构图由输入、中心变换和输出等三



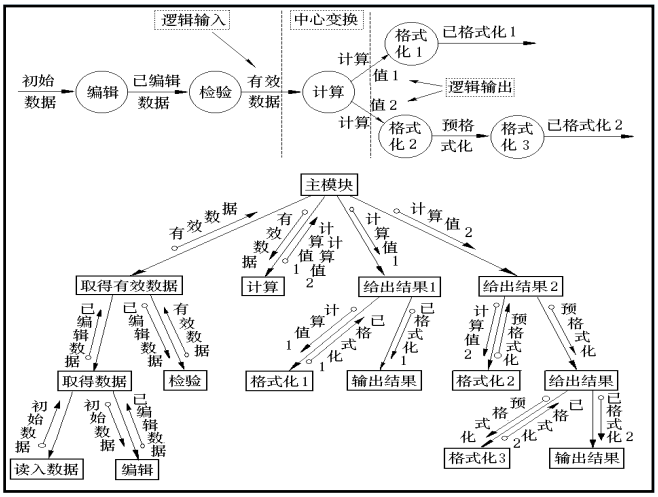
事务型系统结构图

- 它接受一项事务，根据事务处理的特点和性质，选择分派一个适当的处理单元，然后给出结果。
- 在事务型系统结构图中，事务中心模块按所接受的事务的类型，选择某一事务处理模块执行。各事务处理模块并列。每个事务处理模块可能要调用若干个操作模块，而操作模块又可能调用若干个细节模块。

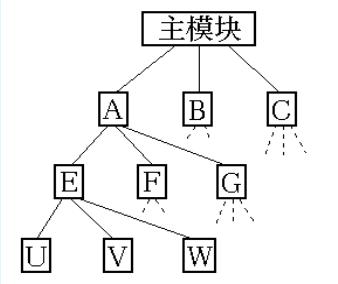


变　换　分　析

- 变换分析方法由以下四步组成：
  - 重画数据流图；
  - 区分有效（逻辑）输入、有效（逻辑）输出和中心变换部分；
  - 进行一级分解，设计上层模块；
  - 进行二级分解，设计输入、输出和中心变换部分的中、下层模块。



① 在选择模块设计的次序时，必须对一个模块的全部直接下属模块都设计完成之后，才能转向另一个模块的下层模块的设计。



② 在设计下层模块时，应考虑模块的耦合和内聚问题，以提高初始结构图的质量。

③ 使用“黑箱”技术：在设计当前模块时，先把这个模块的所有下层模块定义成“黑箱”，在设计中利用它们时，暂时不考虑其内部结构和实现。在这一步定义好的“黑箱”，在下一步就可以对它们进行设计和加工。这样，又会导致更多的“黑箱”。最后，全部“黑箱”的内容和结构应完全被确定。

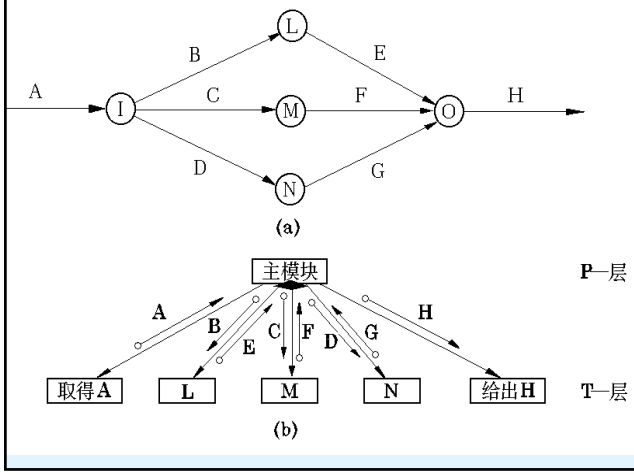
④ 在模块划分时，一个模块的直接下属模块一般在5个左右。如果直接下属模块超过10个，可设立中间层次。

⑤ 如果出现了以下情况，就停止模块的功能分解：

- 当模块不能再细分为明显的子任务时；
- 当分解成用户提供的模块或程序库的子程序时；
- 当模块的界面是输入／输出设备传送的信息时；
- 当模块不宜再分解得过小时。

事　务　分　析

- 在很多软件应用中，存在某种作业数据流，它可以引发一个或多个处理，这些处理能够完成该作业要求的功能。这种数据流就叫做事务。
- 与变换分析一样，事务分析也是从分析数据流图开始，自顶向下，逐步分解，建立系统到结构图。



事务分析过程

① 识别事务源

利用数据流图和数据词典，从问题定义和需求分析的结果中，找出各种需要处理的事务。通常，事务来自物理输入装置。有时，设计人员还必须区别系统的输入、中心加工和输出中产生的事务。

② 规定适当的事务型结构
在确定了该数据流图具有事务型特征之后，根据模块划分理论，建立适当的事务型结构。

③ 识别各种事务和它们定义的操作
从问题定义和需求分析中找出的事务及其操作所必需的全部信息，对于系统内部产生的事务，必须仔细地定义它们的操作。

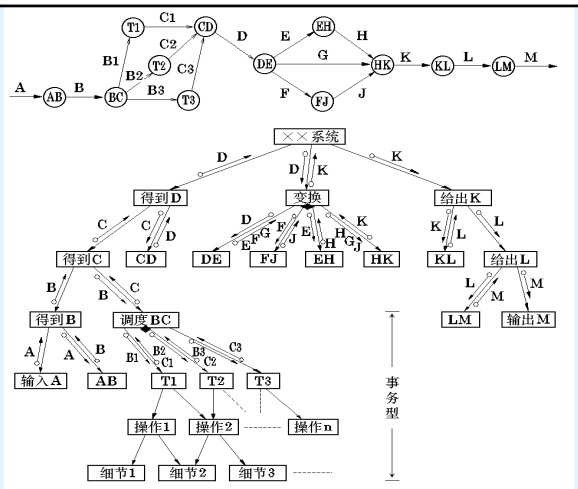
④ 注意利用公用模块
在事务分析的过程中，如果不同事务的一些中间模块可由具有类似的语法和语义的若干个低层模块组成，则可以把这些低层模块构造成公用模块。

⑤ 对每一事务，或对联系密切的一组事务，建立一个事务处理模块；如果发现在系统中有类似的事务，可以把它们组成一个事务处理模块。

⑥ 对事务处理模块规定它们全部的下层操作模块

⑦ 对操作模块规定它们的全部细节模块

变换分析是软件系统结构设计的主要方法。一般，一个大型的软件系统是变换型结构和事务型结构的混合结构。所以，我们通常利用以变换分析为主，事务分析为辅的方式进行软件结构设计。



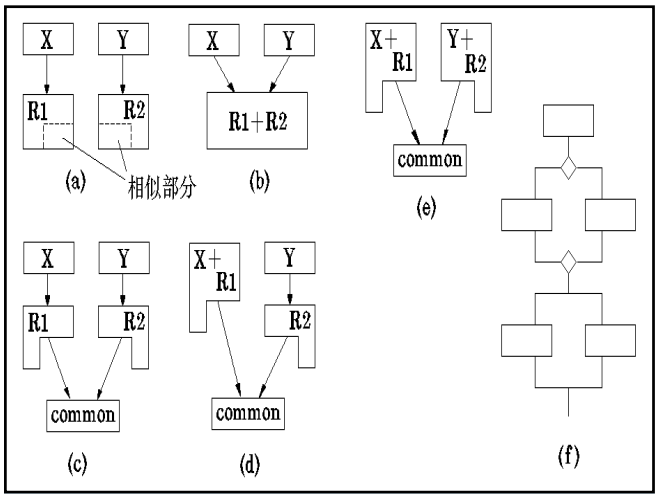
软件模块结构的改进

- **模块功能的完善化**

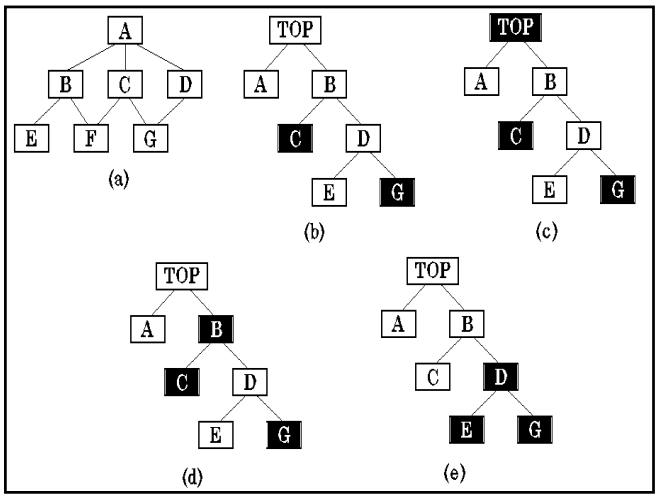
一个完整的模块应当有以下几部分：
① 执行规定的功能的部分；
② 出错处理的部分。当模块不能完成规定的功能时，必须回送出错标志，出现例外情况的原因。
③ 如果需要返回一系列数据给它的调用者，在完成数据加工或结束时， 应当给它的调用者返回一个结束状态标志。

- **消除重复功能，改善软件结构**

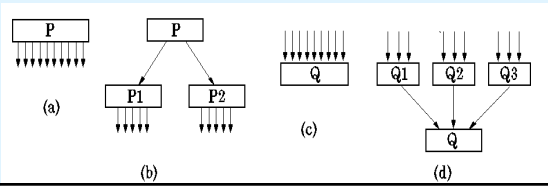
① 完全相似：在结构上完全相似，可能只是在数据类型上不一致。此时可以采取完全合并的方法。
② 局部相似：找出其相同部分，分离出去，重新定义成一个独立的下一层模块。还可以与它的上级模块合并。



- **模块的作用范围应在控制范围之内**
  - 模块的控制范围包括它本身及其所有的从属模块。
  - 模块的作用范围是指模块内一个判定的作用范围，凡是受这个判定影响的所有模块都属于这个判定的作用范围。
  - 如果一个判定的作用范围包含在这个判定所在模块的控制范围之内，则这种结构是简单的，否则，它的结构是不简单的。

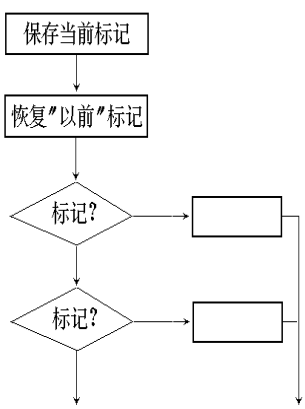


- **尽可能减少高扇出结构，随着深度增大扇入。**
  如果一个模块的扇出数过大，就意味着该模块过分复杂，需要协调和控制过多的下属模块。应当适当增加中间层次的控制模块。



- **模块的大小要适中**
  模块的大小，可以用模块中所含语句的数量的多少来衡量。把模块的大小限制在一定的范围之内。通常规定其语句行数在50～100左右，保持在一页纸之内，最多不超过500行。

- **设计功能可预测的模块，但要避免过分受限制的模块**
  - 一个功能可预测的模块，不论内部处理细节如何，但对相同的输入数据，总能产生同样的结果。但是，如果模块内部蕴藏有一些特殊的鲜为人知的功能时，这个模块就可能是不可预测的。对于这种模块，如果调用者不小心使用，其结果将不可预测。



(a) 不可预测的模块

```
PROC INERT-TRING (……)
/* Module will insert any string
less than 10 characters ……
```

(b) 过分受限制的模块

– 如果一个模块的局部数据结构的大小、控制流的选择或者与外界(人、硬软件)的接口模式被限制死了，则很难适应用户新的要求或环境的变更。

– 为了能够适应将来的变更，软件模块中局部数据结构的大小应当是可控制的，控制流的选择对于调用者来说，应当是可预测的。而与外界的接口应当是灵活的。

- **软件包应满足设计约束和可移植性**

为了使得软件包可以在某些特定的环境下能够安装和运行，对软件包提出了一些设计约束和可移植的要求。例如，设计约束有时要求一个程序段在存储器中覆盖自身。当这种情况出现时，设计出来的软件程序结构不得不根据重复程度、访问频率、调用间隔等等特性，重新加以组织。

设 计 的 后 处 理

- **为每一个模块写一份处理说明**
- **为每一个模块提供一份接口说明**
- **确定全局数据结构和局部数据结构**
- **指出所有的设计约束和限制**
- **进行概要设计的评审**
- **进行设计的优化(如果需要和可能的话)**

❖数据设计及文件设计

- 数据设计的原则
- 文件设计

❖数 据 设 计 的 原 则

- R. S. Pressman数据设计的过程

– 为在需求分析阶段所确定的数据对象选择逻辑表示，需要对不同结构进行算法分析，以便选择一个最有效的结构；设计对于这种逻辑数据结构的一组操作，以实现各种所期望的运算。

– 确定对逻辑数据结构所必需的那些操作的程序模块(软件包)，以便限制或确定各个数据设计决策的影响范围。

Pressman提出了一组原则，用来定义和设计数据。实际上，在进行需求分析时往往就开始了数据设计。

(1). 用于软件的系统化方法也适用于数据。在导出、评审和定义软件的需求和软件系统结构时，必须定义和评审其中所用到的数据流、数据对象及数据结构的表示。应当考虑几种不同的数据组织方案，还应当分析数据设计给软件设计带来的影响。

(2). 确定所有的数据结构和在每种数据结构上施加的操作。设计有效的数据结构，必须考虑到要对该数据结构进行的各种操作。

(3). 应当建立一个数据词典并用它来定义数据和软件的设计。数据词典清楚地说明了各个数据之间的关系和对数据结构内各个数据元素的约束。

(4). 低层数据设计的决策应推迟到设计过程的后期进行。在进行需求分析时确定的总体数据组织，应在概要设计阶段加以细化，在详细设计阶段才规定具体的细节。

(5). 数据结构的表示只限于那些必须直接使用该数据结构内数据的模块才能知道。此原则就是信息隐蔽和与此相关的耦合性原则。

(6). 应当建立一个存放有效数据结构及相关操作的库。数据结构应当设计成为可复用的。建立一个存有各种可复用的数据结构模型的部件库。

(7). 软件设计和程序设计语言应当支持抽象数据类型的定义和实现。

以上原则适用于软件工程的定义阶段和开发阶段。“清晰的信息定义是软件开发成功的关键”。

❖文 件 设 计

文件设计的过程，主要分两个阶段。第一个阶段是文件的逻辑设计，主要在概要设计阶段实施。

(1) 整理必须的数据元素：

在软件设计中所使用的数据，有长期的，有短期的，还有临时的。它们都可以存放在文件中，在需要时对它们进行访问。因此首先必须整理应存储的数据元素，给它们一个易于理解的名字，指明其类型和位数，以及其内容涵义。

(2) 分析数据间的关系：

分析在业务处理中哪些数据元素是同时使用的。把同时使用次数多的数据元素归纳成一个文件进行管理。分析数据元素的内容，研究数据元素与数据元素之间的逻辑关系，根据分析，弄清数据元素的含义及其属性。

(3) 确定文件的逻辑设计：

根据数据关联性分析，明确哪些数据元素应当归于一组进行管理，把应当归于一组的数据元素进行统一布局，产生文件的逻辑设计。应用关系模型设计文件的逻辑结构时，必须使其达到第三范式(3NF)，以减少数据的冗余，提高存取的效率。

顾客文件

数据元素名	属性	长度	备注
顾客号码	X	6	
顾客姓名	K	15	
住址1	K	10	省，市
住址2	K	10	区，街
住址3	K	10	门牌号
电话号码	X	12	
邮政编码	X	6	

商品文件

数据元素名	属性	长度	备注
商品号码	X	8	3英文+
商 品 名	K	20	5数字
单　　价	N	7	
单　　位	X	3	
期首库存量	N	7	
现在库存量	N	7	

X：英文字母＋数字；　K：汉字；　N：数字

第二个阶段是文件的物理设计，主要在软件的详细设计阶段实施

(4) 理解文件的特性：

对于文件的逻辑规格说明，研究从业务处理的观点来看所要求的一些特性，包括文件的使用率、追加率和删除率，以及保护和保密等。考虑需要采用什么文件组织形式。

(5) 确定文件的组织方式；

一般要根据文件的特性，来确定文件的组织方式。

(6)确定文件的存储介质；

(7)确定文件的记录格式；

(8) 估算存取时间和存储容量。

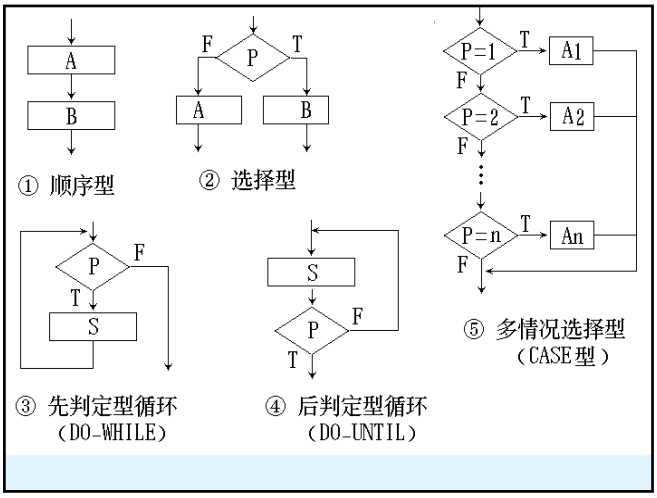
过 程 设 计

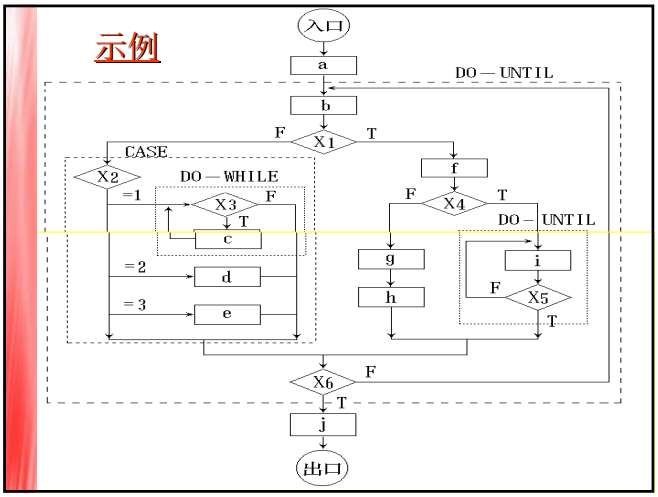
- 从软件开发的工程化观点来看，在使用程序设计语言编制程序以前，需要对所采用算法的逻辑关系进行分析，设计出全部必要的过程细节，并给予清晰的表达。这就是过程设计的任务。

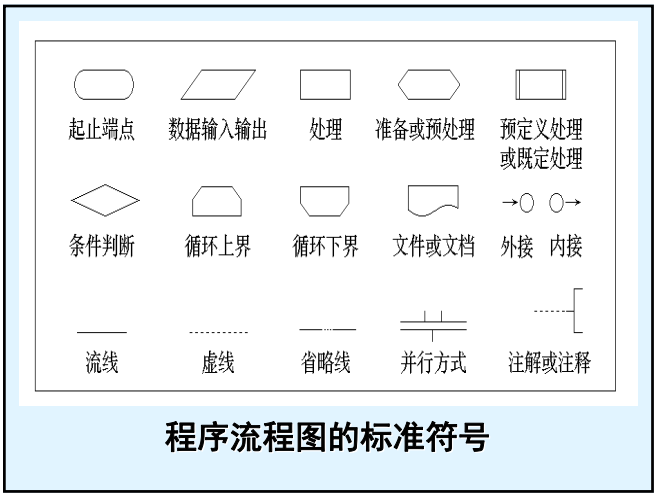
- 在过程设计阶段，要决定各个模块的实现算法，并精确地表达这些算法。表达过程规格说明的工具叫做详细设计工具，它可以分为以下三类：
  - 图形工具
  - 表格工具
  - 语言工具

程序流程图

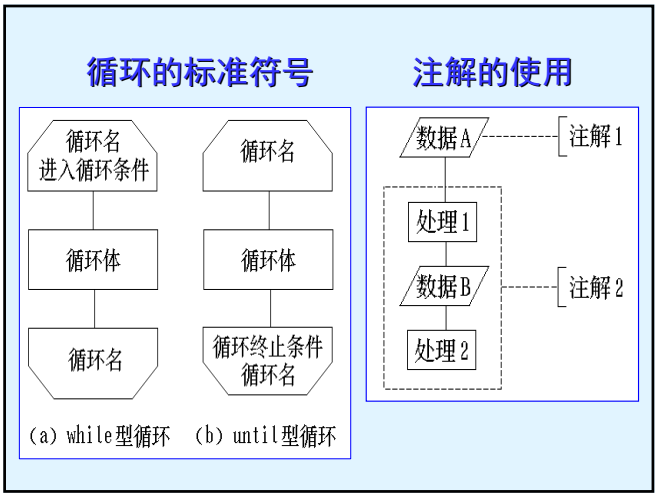
- 程序流程图也称为程序框图，程序流程图使用五种基本控制结构是：



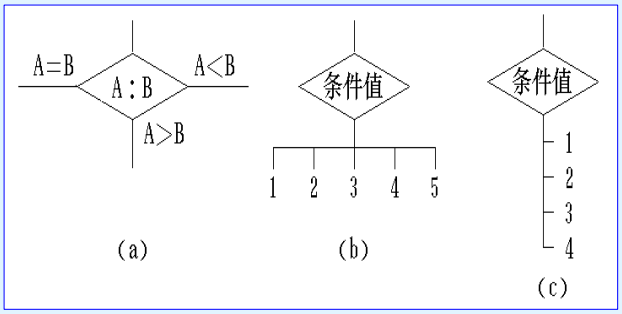




程序流程图的标准符号

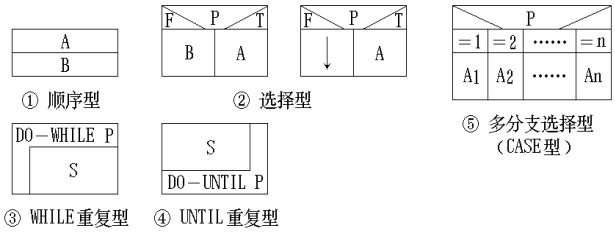


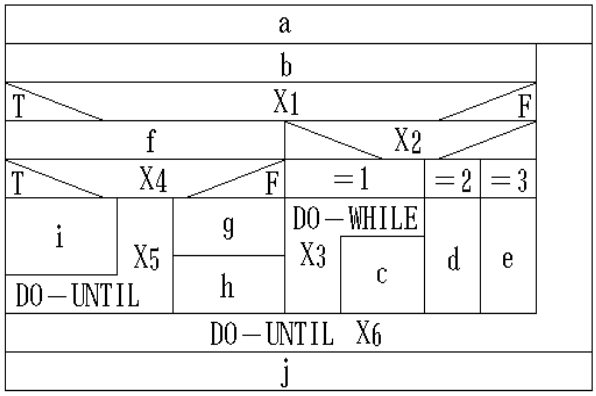
多出口判断



N-S图

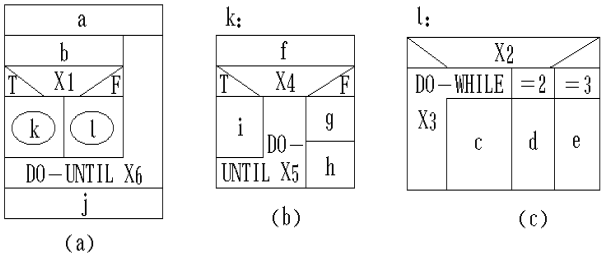
- N-S图也叫做盒图。五种基本控制结构由五种图形构件表示。





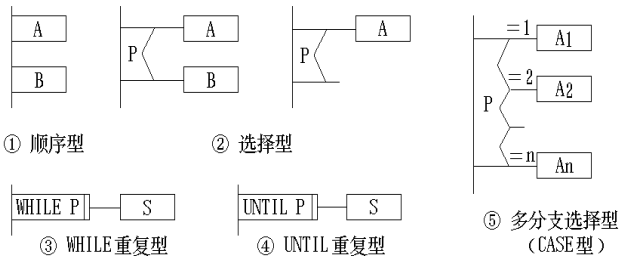
示例

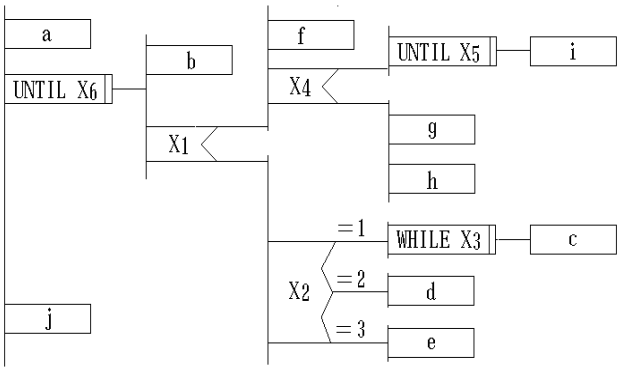
N-S图的嵌套定义形式



问题分析图(PAD)

- PAD也设置了五种基本控制结构的图式，并允许递归使用。



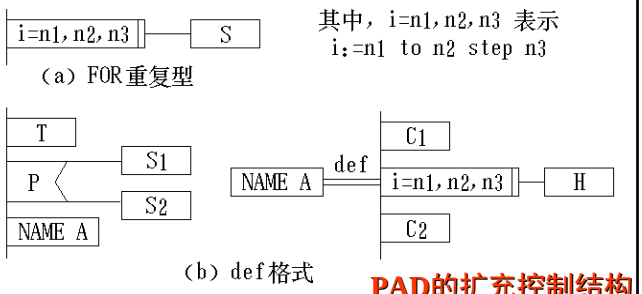


PAD描述的示例

对应于增量型循环结构

for *i* := *n*1 **to** *n*2 **step** *n*3 **do**

在PAD中有相应的循环控制结构

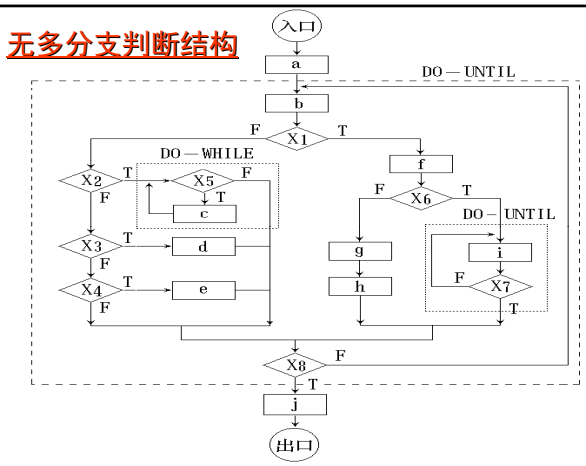


PAD的扩充控制结构

判定表

- 判定表用于表示程序的静态逻辑
- 在判定表中的条件部分给出所有的两分支判断的列表，动作部分给出相应的处理
- 要求将程序流程图中的多分支判断都改成两分支判断

无多分支判断结构



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
X1	T	T	T	T	T	F	F	F	F	F	F	F	F	F
X2	–	–	–	–	–	T	T	T	F	F	F	F	F	F
X3	–	–	–	–	–	–	–	–	F	F	T	T	F	F
X4	–	–	–	–	–	–	–	–	F	F	–	–	T	T
X5	–	–	–	–	–	T	F	F	–	–	–	–	–	–
X6	T	T	T	F	F	–	–	–	–	–	–	–	–	–
X7	T	T	F	–	–	–	–	–	–	–	–	–	–	–
X8	T	F	–	T	F	–	T	F	F	T	T	F	T	F
a	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
b	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
c	–	–	–	–	–	Y	–	–	–	–	–	–	–	–
d	–	–	–	–	–	–	–	–	–	–	Y	Y	–	–
e	–	–	–	–	–	–	–	–	–	–	–	–	Y	Y
f	Y	Y	Y	Y	Y	–	–	–	–	–	–	–	–	–
g	–	–	–	Y	Y	–	–	–	–	–	–	–	–	–
h	–	–	–	Y	Y	–	–	–	–	–	–	–	–	–
i	Y	Y	Y	–	–	–	–	–	–	–	–	–	–	–
j	Y	–	–	Y	–	–	Y	–	–	Y	Y	–	Y	–

建立判定表的步骤

- 列出与一个具体过程(或模块)有关的所有处理。
- 列出过程执行期间的所有条件(或所有判断)。
- 将特定条件取值组合与特定的处理相匹配，消去不可能发生的条件取值组合。
- 将右部每一纵列规定为一个处理规则，即对于某一条件取值组合将有什么动作。

PDL (Program Design Language)

- PDL是一种用于描述功能模块的算法设计和加工细节的语言。称为设计程序用语言。它是一种伪码。
- 伪码的语法规则分为“外语法”和“内语法”。
- PDL具有严格的关键字外语法，用于定义控制结构和数据结构，同时它的表示实际操作和条件的内语法又是灵活自由的，可使用自然语言的词汇。

PDL的特点

- 提供全部结构化控制结构、数据说明和模块特征。能对PDL正文进行结构分割，使之变得易于理解。
- 为了区别关键字，规定关键字一律大写，其它单词一律小写。或者规定关键字加下划线，或者规定它们为黑体字。

- 内语法使用自然语言来描述处理特性。内语法比较灵活，只要写清楚就可以，不必考虑语法错，以利于人们可把主要精力放在描述算法的逻辑上。
- 有数据说明机制，包括简单的(如标量和数组)与复杂的(如链表和层次结构)的数据结构。
- 有子程序定义与调用机制，用以表达各种方式的接口说明。

第五部分　程序编码

- **结构化程序设计**
- **程序设计风格**
- **程序效率**
- **程序复杂性度量**

- 做为软件工程过程的一个阶段，**程序编码是设计的继续**。
- 程序设计语言的特性和程序设计风格会深刻地影响软件的质量和可维护性。
- 为了保证程序编码的质量，程序员必须深刻地理解、熟练地掌握并正确地运用程序设计语言的特性。此外，还要求源程序具有良好的结构性和良好的程序设计风格。

结构化程序设计

结构化程序设计主要包括两方面：

(1) 在编写程序时，强调**使用几种基本控制结构**，通过组合嵌套，形成程序的控制结构。尽可能避免使用GOTO语句。

(2) 在程序设计过程中，尽量**采用自顶向下和逐步细化**的原则，由粗到细，一步步展开。

结构化程序设计的主要原则

- 使用语言中的**顺序**、**选择**、**重复**等有限的基本控制结构表示程序逻辑。
- 选用的控制结构只准许有**一个入口**和**一个出口**。
- 程序语句组成**容易识别的块**，每块只有**一个入口**和**一个出口**。
- 复杂结构应该用基本控制结构进行组合嵌套来实现。

- 语言中没有的控制结构，可用一段等价的程序段模拟，但要求该程序段在整个系统中应前后一致。
- **严格控制GOTO语句**，仅在下列情形才可使用：
  ① 用一个非结构化的程序设计语言去实现一个结构化的构造。
  ② 若不使用GOTO语句就会使程序功能模糊。
  ③ 在某种可以改善而不是损害程序可读性的情况下。

自顶向下，逐步求精

- 在详细设计和编码阶段，应当采取自顶向下，逐步求精的方法。
- 把一个模块的功能逐步分解，细化为一系列具体的步骤，进而翻译成一系列用某种程序设计语言写成的程序。

自顶向下，逐步求精方法的优点

- 符合人们解决复杂问题的普遍规律。可提高软件开发的成功率和生产率
- 用先全局后局部，先整体后细节，先抽象后具体的逐步求精的过程开发出来的程序具有清晰的层次结构，程序容易阅读和理解

- 程序自顶向下，逐步细化，分解成一个树形结构。在同一层的节点上的细化工作相互独立。有利于编码、测试和集成
- 程序清晰和模块化，使得在修改和重新设计一个软件时，可复用的代码量最大
- 每一步工作仅在上层节点的基础上做不多的设计扩展，便于检查
- 有利于设计的分工和组织工作。

程序设计风格

- 程序实际上也是一种供人阅读的文章，有一个文章的风格问题。应该使程序具有良好的风格。
  - 源程序文档化
  - 数据说明
  - 语句结构
  - 输入／输出方法

源 程 序 文 档 化

- 标识符的命名
- 安排注释
- 程序的视觉组织

符号名的命名

- 符号名即标识符，包括模块名、变量名、常量名、标号名、子程序名、数据区名以及缓冲区名等。
- 这些名字应能反映它所代表的实际东西，应有一定实际意义。
- 例如，表示次数的量用**Times**，表示总量的用**Total**，表示平均值的用**Average**，表示和的量用**Sum**等。

- 名字不是越长越好，应当选择精炼的意义明确的名字。必要时可使用缩写名字，但这时要注意缩写规则要一致，并且要给每一个名字加注释。同时，在一个程序中，一个变量只应用于一种用途。

程序的注释

- 夹在程序中的注释是程序员与日后的程序读者之间通信的重要手段。
- 注释决不是可有可无的。
- 一些正规的程序文本中，注释行的数量占到整个源程序的1／3到1／2，甚至更多。
- 注释分为序言性注释和功能性注释。

序言性注释

- 通常置于每个程序模块的开头部分，**它应当给出程序的整体说明**，对于理解程序本身具有引导作用。有些软件开发部门对序言性注释做了明确而严格的规定，要求程序编制者逐项列出

功能性注释

- 功能性注释嵌在源程序体中，用以描述其后的语句或程序段是在做什么工作，或是执行了下面的语句会怎么样。而不要解释下面怎么做。
- 例如，

```
/* ADD AMOUNT TO TOTAL */
TOTAL = AMOUNT＋TOTAL
```

不好。

- 如果注明把月销售额计入年度总额，便使读者理解了下面语句的意图：

```
/* ADD MONTHLY-SALES TO
ANNUAL-TOTAL */
TOTAL = AMOUNT＋TOTAL
```

- 要点
  - 描述一段程序，而不是每一个语句；
  - 用缩进和空行，使程序与注释容易区别；
  - 注释要正确。

视觉组织　空格、空行和移行

- 恰当地利用**空格**，可以**突出运算的优先性**，避免发生运算的错误。
- 例如，将表达式

```
(A＜－17)ANDNOT(B＜＝49)ORC
```

写成

```
(A＜－17) AND  NOT (B＜＝49)
OR  C
```

- 自然的程序段之间可用**空行**隔开；

- **移行**也叫做**向右缩格**。它是指程序中的各行不必都在左端对齐，都从第一格起排列。这样做使程序完全分不清层次关系。
- 对于**选择语句**和**循环语句**，把其中的程序段语句向右做**阶梯式移行**。使程序的逻辑结构更加清晰。
- 例如，两重选择结构嵌套，写成下面的移行形式，层次就清楚得多。

```
IF（…） THEN
    IF（…） THEN
        ……
    ELSE
        ……
    ENDIF
    ……
ELSE
 ……
ENDIF
```

数据说明

- 在设计阶段已经确定了数据结构的组织及其复杂性。在编写程序时，则需要注意数据说明的风格。
- 为了使程序中数据说明更易于理解和维护，必须注意以下几点。
  1. 数据说明的次序应当规范化
  2. 说明语句中变量安排有序化
  3. 使用注释说明复杂数据结构

数据说明的次序应当规范化

- 数据说明次序规范化，使数据属性容易查找，也有利于测试，排错和维护。
- 原则上，数据说明的次序与语法无关，其次序是任意的。但出于阅读、理解和维护的需要，最好使其规范化，使说明的先后次序固定。

说明语句中变量安排有序化

- 当多个变量名在一个说明语句中说明时，应当对这些变量按字母的顺序排列。带标号的全程数据(如FORTRAN的公用块)也应当按字母的顺序排列。
- 例如，把

  integer size, length, width, cost, price

  写成

  integer cost, length, price , size, width

使用注释说明复杂数据结构

- 如果设计了一个复杂的数据结构，应当使用注释来说明在程序实现时这个数据结构的固有特点。
- 例如，对PL/1的链表结构和Pascal中用户自定义的数据类型，都应当在注释中做必要的补充说明。

语句结构

- 在设计阶段确定了软件的逻辑流结构，但构造单个语句则是编码阶段的任务。语句构造力求简单，直接，不能为了片面追求效率而使语句复杂化。

1. 在一行内只写一条语句

- 在一行内只写一条语句，并且采取适当的移行格式，使程序的逻辑和功能变得更加明确。
- 许多程序设计语言允许在一行内写多个语句。但这种方式会使程序可读性变差。因而不可取。

- 例如，有一段排序程序

```
FOR I:=1 TO N－1 DO BEGIN
T:=I；FOR J:=I＋1 TO N DO IF A[J]
＜A[T] THEN T:=J；IF T≠I THEN
BEGIN WORK:=A[T]；
A[T]:=A[I]；A[I]:=WORK；END
END；
```

- 由于一行中包括了多个语句，掩盖了程序的循环结构和条件结构，使其可读性变得很差。

```
FOR I:=1 TO N-1 DO      //改进布局
  BEGIN
    T:=I;
    FOR J:=I＋1 TO N DO
      IF A[J]＜A[T] THEN T:=J;
    IF T≠I THEN
      BEGIN
        WORK:=A[T];
        A[T]:=A[I];
        A[I]:=WORK;
      END
  END;
```

2. 程序编写首先应当考虑清晰性

- 程序编写首先应当考虑清晰性，不要刻意追求技巧性，使程序编写得过于紧凑。
- 例如，有一个用Pascal语句写出的程序段：

```
A[I]:=A[I]＋A[T];
A[T]:=A[I]－A[T];
A[I]:=A[I]－A[T];
```

- 此段程序可能不易看懂，有时还需用实际数据试验一下。
- 实际上，这段程序的功能就是交换A[I]和A[T]中的内容。目的是为了节省一个工作单元。如果改一下：

```
WORK:=A[T];
A[T]:=A[I];
A[I]:=WORK;
```

就能让读者一目了然了。

3. 程序要能直截了当地说明程序员的用意。

4. 除非对效率有特殊的要求，程序编写要做到**清晰第一，效率第二**。不要为了追求效率而丧失了清晰性。事实上，**程序效率的提高主要应通过选择高效的算法**来实现。

5. 首先要保证**程序正确**，然后才要求**提高速度**。反过来说，在使程序高速运行时，首先要保证它是正确的。

6. **避免使用临时变量**而使可读性下降。例如，有的程序员为了追求效率，往往喜欢把表达式

A[I]＋1 / A[I];

写成 **AI＝A[I];**

X＝AI＋1 / AI;

这样将一句分成两句写，会产生意想不到的问题。

7. **让编译程序做简单的优化。**
8. **尽可能使用库函数**
9. **避免不必要的转移。同时如果能保持程序可读性，则不必用 GO TO语句。**
10. **尽量只采用三种基本的控制结构来编写程序。除顺序结构外，使用IF-THEN-ELSE来实现选择结构；使用DO-UNTIL或DO-WHILE来实现循环结构。**

11. 避免使用空的ELSE语句和IF… THEN IF…的语句。这种结构容 易使读者产生误解。例如，

```
IF ( CHAR＞＝'A' ) THEN
 IF ( CHAR＜＝'Z' ) THEN
 PRINT “This is a letter。”
 ELSE
 PRINT “This is not a letter。”
```

可能产生二义性问题。

12. **避免采用过于复杂的条件测试。**
13. **尽量减少使用“否定”条件的条件语句。例如，如果在程序中出现**

```
IF NOT ( ( CHAR＜‘0’ ) OR
( CHAR＞‘9’ ) ) THEN ……
```

改成

```
IF ( CHAR＞＝'0' ) AND
( CHAR＜＝'9' ) THEN ……
```

不要让读者绕弯子想。

14. 尽可能用通俗易懂的**伪码**来描述程序的流程，然后再翻译成必须使用的语言。
15. 数据结构要有利于程序的简化。
16. 要**模块化**，使模块功能尽可能单一化，模块间的耦合能够清晰可见。
17. 利用**信息隐蔽**，确保每一个模块的独立性。

18. 从**数据**出发去构造程序。
19. 不要修补不好的程序，要重新编写。也不要一味地追求代码的复用，要重新组织。
20. 对太大的程序，要分块编写、测试，然后再集成。
21. 对递归定义的数据结构尽量使用递归过程。

输入和输出

- **输入和输出信息是与用户的使用直接相关的。输入和输出的方式和格式应当尽可能方便用户的使用。一定要避免因设计不当给用户带来的麻烦。**
- **因此，在软件需求分析阶段和设计阶段，就应基本确定输入和输出的风格。系统能否被用户接受，有时就取决于输入和输出的风格。**

- 不论是**批处理的输入／输出方式**，还是**交互式的输入／输出方式**，在设计和程序编码时都应考虑下列原则：

1. 对所有的输入数据都要进行检验，识别错误的输入，以保证每个数据的有效性；
2. 检查输入项的各种重要组合的合理性，必要时报告输入状态信息；
3. 使得输入的步骤和操作尽可能简单，并保持简单的输入格式；

4. 输入数据时，应允许使用自由格式输入；
5. 应允许缺省值；
6. 输入一批数据时，最好使用输入结束标志，而不要由用户指定输入数据数目；
7. 在交互式输入输入时，要在屏幕上使用提示符明确提示交互输入的请求，指明可使用选择项的种类和取值范围。同时，在数据输入的过程中和输入结束时，也要在屏幕上给出状态信息；

8. 当程序设计语言对输入／输出格式有严格要求时，应保持输入格式与输入语句的要求的一致性；
9. 给所有的输出加注解，并设计输出报表格式。
输入／输出风格还受到许多其它因素的影响。如输入／输出设备（例如终端的类型，图形设备，数字化转换设备等）、用户的熟练程度、以及通信环境等。

程序效率

- **讨论效率的准则**

程序的效率是指**程序的执行速度**及**程序所需占用的内存的存储空间**。程序编码是最后提高运行速度和节省存储的机会，因此在此阶段不能不考虑程序的效率。让我们首先明确讨论程序效率的几条准则

- 效率是一个性能要求，应当在需求分析阶段给出。**软件效率以需求为准**，不应以人力所及为准。
- 好的设计可以提高效率。
- 程序的**效率与程序的简单性**相关。
- 一般说来，任何对效率无重要改善，且对程序的简单性、可读性和正确性不利的程序设计方法都是不可取的。

算法对效率的影响

- 源程序的**效率与详细设计阶段确定的算法的效率直接有关**。在详细设计翻译转换成源程序代码后，算法效率反映为程序的执行速度和存储容量的要求。
- 设计向程序转换过程中的指导原则：

① 在编程序前，尽可能化简有关的算术表达式和逻辑表达式；

② 仔细检查算法中的嵌套的循环，尽可能将某些语句或表达式移到循环外面；

③ 尽量避免使用多维数组；

④ 尽量避免使用指针和复杂的表；

⑤ 采用“快速”的算术运算；

⑥ 不要混淆数据类型，避免在表达式中出现类型混杂；

⑦ 尽量采用整数算术表达式和布尔表达式；

⑧ 选用等效的高效率算法；

- 许多编译程序具有“优化”功能，可以自动生成高效率的目标代码。

影响存储器效率的因素

- 在大中型计算机系统中，存储限制不再是主要问题。在这种环境下，对**内存采取基于操作系统的分页功能的虚拟存储管理**。**存储效率与操作系统的分页功能直接有关**。

- 采用结构化程序设计，**将程序功能合理分块，使每个模块或一组密切相关模块的程序体积大小与每页的容量相匹配**，可减少页面调度，减少内外存交换，提高存储效率。

- 在微型计算机系统中，存储器的容量对软件设计和编码的制约很大。因此**要选择可生成较短目标代码且存储压缩性能优良的编译程序**，有时需采用汇编程序。
- 提高存储器效率的关键是程序的简单性。

影响输入／输出的因素

- 输入／输出可分为两种类型：
  - 面向人(操作员)的输入／输出
  - 面向设备的输入／输出
- 如果操作员能够十分方便、简单地录入输入数据，或者能够十分直观、一目了然地了解输出信息，则可以说面向人的输入／输出是高效的。

- 关于面向设备的输入/输出，可以提出一些提高输入/输出效率的指导原则：
  - 输入/输出的请求应当最小化；
  - 对于所有的输入/输出操作，**安排适当的缓冲区**，以减少频繁的信息交换。
  - 对辅助存储(例如磁盘)，**选择尽可能简单的，可接受的存取方法**；

  - 对辅助存储的输入/输出，应当**成块传送**；
  - **对终端或打印机的输入/输出，应考虑设备特性**，尽可能改善输入/输出的质量和速度；
  - 任何不易理解的，对改善输入/输出效果关系不大的措施都是不可取的；
  - 任何不易理解的所谓“超高效”的输入/输出是毫无价值的；

程序复杂性度量

- 程序复杂性主要指**模块内程序的复杂性**。它直接关联到软件开发费用的多少，开发周期的长短和软件内部潜伏错误的多少。
- 减少程序复杂性，可提高软件的简单性和可理解性，并使软件开发费用减少，开发周期缩短，软件内部潜藏错误减少。

复杂性度量需要满足的假设

- 为了度量程序复杂性，要求：
  - 它可以用来计算任何一个程序的复杂性；
  - 对于不合理的程序，例如对于长度动态增长的程序，或者对于原则上无法排错的程序，不应当使用它进行复杂性计算；
  - 如果程序中指令条数、附加存储量、计算时间增多，不会减少程序的复杂性。

代码行度量法

- 源代码行数度量法基于两个前提：
  - **程序复杂性随着程序规模的增加不均衡地增长；**
  - **控制程序规模的方法最好是采用分而治之的办法。将一个大程序分解成若干个简单的可理解的程序段。**

- 方法的基本考虑是统计一个程序模块的源代码行数目，并以源代码行数做为程序复杂性的度量。
- 设每行代码的出错率为每100行源程序中可能有的错误数目。

McCabe度量法

- McCabe度量法，又称环路复杂性度量，是一种基于程序控制流的复杂性度量方法。
- 它基于一个程序模块的程序图中环路的个数，因此计算它先要画出程序图。
- 程序图是退化的程序流程图。流程图中每个处理都退化成一个结点，流线变成连接不同结点的有向弧。

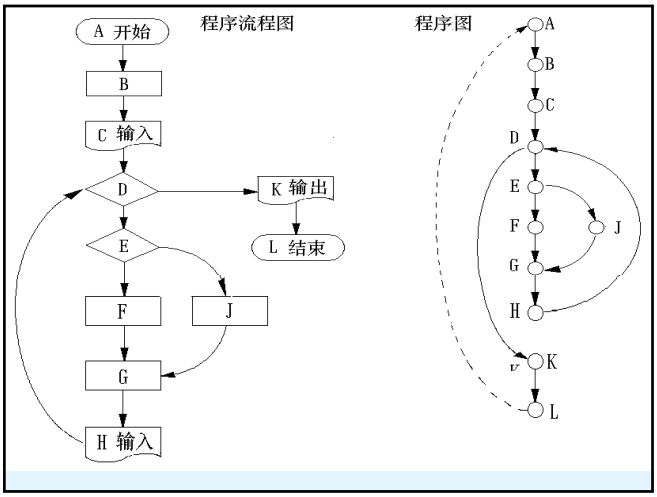
- 程序图仅描述程序内部的控制流程，完全不表现对数据的具体操作，以及分支和循环的具体条件。
- **计算环路复杂性的方法：**根据图论，在一个强连通的有向图G中，环的个数由以下公式给出：

  $V(G)=m-n+p$

  其中，$V(G)$是有向图G中环路个数，m是图G中弧数，n是图G中结点数，p是图G中的强连通分量个数。

- 为使图成为强连通图，从图的入口点到出口点加一条用虚线表示的有向边，使图成为强连通图。这样就可以使用上式计算环路复杂性。
- 在例示中，结点数$n=11$，弧数$m=13$，$p=1$，则有

  $V(G)=m-n+p=13-11+1=3.$
- 等于程序图中弧所封闭的区域数。



- 这种度量的缺点是：
  - 对于不同种类的控制流的复杂性不能区分
  - 简单IF语句与循环语句的复杂性同等看待
  - 嵌套IF语句与简单CASE语句的复杂性是一样的
  - 模块间接口当成一个简单分支一样处理
  - 一个具有1000行的顺序程序与一行语句的复杂性相同

第六部分　软件测试

- 软件测试的定义
- 软件测试基础
- 软件测试用例设计
- 软件测试策略
- 软件测试种类
- 程序调试

软件测试的定义

软件测试是在软件投入运行前，对软件需求分析，设计规格说明和编码的最终复审，是软件质量保证的关键步骤。

如果下定义：软件测试是为了发现错误而执行程序的过程。或者说软件测试是根据软件开发各阶段的规格说明和程序的内部结构而精心设计一批测试用例，并利用这些测试用例去运行程序，以发现程序错误的过程。

软件测试的基础

- 软件测试的目的
- 软件测试的原则
- 软件测试的对象
- 测试信息流
- 测试与软件开发各阶段的关系

软件测试的目的

- 基于不同的立场，存在着两种完全不同的测试目的。
- 从用户的角度出发，普遍希望通过软件测试暴露软件中隐藏的错误和缺陷，以考虑是否可接受该产品。
- 从软件开发者的角度出发，则希望测试成为表明软件产品中不存在错误的过程，验证该软件已正确地实现了用户的要求，确立人们对软件质量的信心。

Myers软件测试目的

(1) 测试是程序的执行过程，目的在于发现错误；

(2) 一个好的测试用例在于能发现至今未发现的错误；

(3) 一个成功的测试是发现了至今未发现的错误的测试。

- 换言之，测试的目的是
  - 系统地找出软件中潜在的各种错误和缺陷。
  - 能够证明软件的功能和性能与需求说明相符合。
  - 测试不能表明软件中不存在错误，它只能说明软件中存在错误。

软件测试的原则

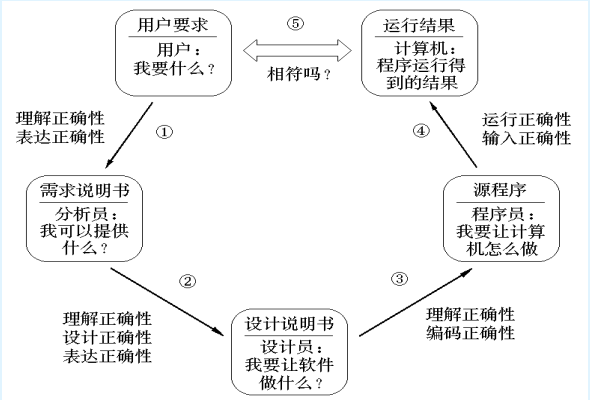
1. 应当把“尽早地和不断地进行软件测试”作为软件开发者的座右铭。
2. 测试用例应由测试输入数据和对应的预期输出结果这两部分组成。
3. 程序员应避免检查自己的程序。
4. 在设计测试用例时，应当包括合理的输入条件和不合理的输入条件。
5. 充分注意测试中的群集现象。
   经验表明，测试后程序中残存的错误数目与该程序中已发现的错误数目成正比。
6. 严格执行测试计划，排除测试的随意性。
7. 应当对每一个测试结果做全面检查。
8. 妥善保存测试计划，测试用例，出错统计和最终分析报告，为维护提供方便。

软件测试的对象

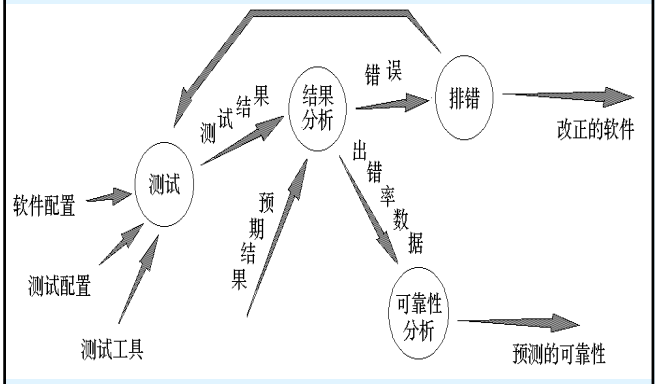
- 软件测试并不等于程序测试。软件测试应贯穿于软件定义与开发的整个期间。
- 需求分析、概要设计、详细设计以及程序编码等各阶段所得到的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

- 为把握软件开发各个环节的正确性，需要进行各种确认和验证工作。
- 确认(Validation)，是一系列的活动和过程，目的是想证实在一个给定的外部环境中软件的逻辑正确性。
  - 需求规格说明的确认
  - 程序的确认
- 验证(Verification)，试图证明在软件生存期各个阶段，以及阶段间的逻辑协调性、完备性和正确性。

软件生存期各阶段之间需要保持的正确性



测试信息流



测试信息流

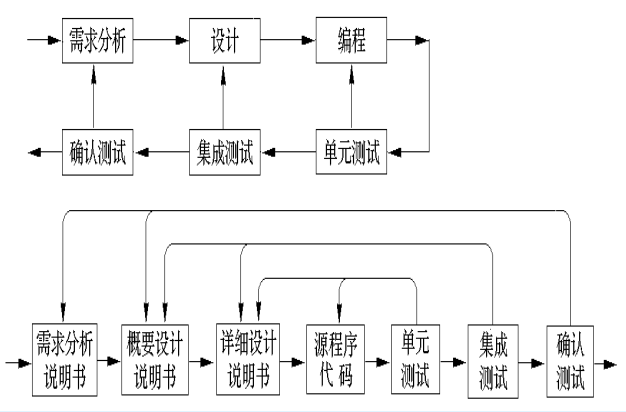
- **软件配置**：软件需求规格说明、软件设计规格说明、源代码等；
- **测试配置**：测试计划、测试用例、测试程序等；
- **测试工具**：测试数据自动生成程序、静态分析程序、动态分析程序、测试结果分析程序、以及驱动测试的测试数据库等等。

- **测试结果分析**：比较实测结果与预期结果，评价错误是否发生。
- **排错(调试)**：对已经发现的错误进行错误定位和确定出错性质，并改正这些错误，同时修改相关的文档。
- **修正后的文档再测试**：直到通过测试为止。

- 通过收集和分析测试结果数据，对软件建立可靠性模型
- 利用可靠性分析，评价软件质量：
  - **软件的质量和可靠性达到可以接受的程度；**
  - **所做的测试不足以发现严重的错误；**
- 如果测试发现不了错误，可以肯定，测试配置考虑得不够细致充分，错误仍然潜伏在软件中。

测试与软件开发各阶段的关系

- 软件开发过程是一个自顶向下，逐步细化的过程
- 软件计划阶段定义软件作用域
- 软件需求分析建立软件信息域、功能和性能需求、约束等
- 软件设计把设计用某种程序设计语言转换成程序代码
- 测试过程是依相反顺序安排的自底向上，逐步集成的过程。



软件测试用例设计

- 两种常用的测试方法
  - 黑盒测试
  - 白盒测试

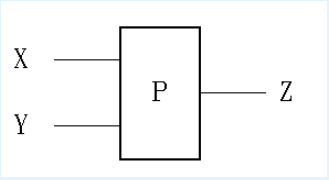
黑盒测试

- 这种方法是把**测试对象**看做**一个黑盒子**，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。
- 黑盒测试又叫做**功能测试**或**数据驱动测试**。

- 黑盒测试方法是在程序接口上进行测试，主要是为了发现以下错误：
  - **是否有不正确或遗漏了的功能?**
  - **在接口上，输入能否正确地接受?能否输出正确的结果?**
  - **是否有数据结构错误或外部信息(例如数据文件)访问错误?**
  - **性能上是否能够满足要求?**
  - **是否有初始化或终止性错误?**

- 用黑盒测试发现程序中的错误，必须在**所有可能的输入条件和输出条件**中确定测试数据，来检查程序是否都能产生正确的输出。
- 但这是**不可能**的。

- 假设一个**程序P**有**输入量X**和**Y**及**输出量Z**。在字长为32位的计算机上运行。若**X**、**Y**取整数，按黑盒方法进行穷举测试：
- 可能采用的测试数据组：
  $2^{32} \times 2^{32} = 2^{64}$

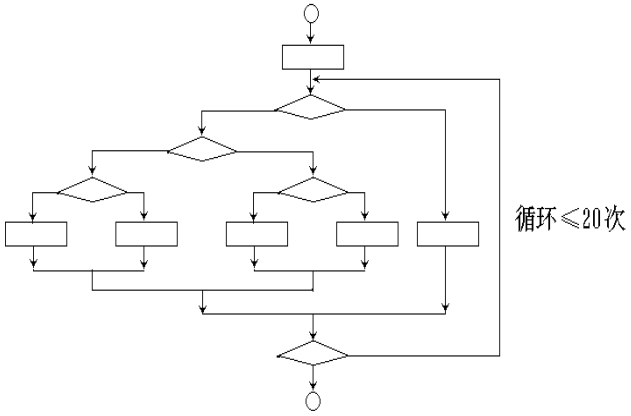


- 如果测试一组数据需要1毫秒，一年工作365×24小时，完成所有测试需5亿年。

白盒测试

- 此方法**把测试对象看做一个透明的盒子**，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。
- 通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。因此白盒测试又称为结构测试或逻辑驱动测试。

- 软件人员使用白盒测试方法，主要想对程序模块进行如下的检查：
  - **对程序模块的所有独立的执行路径至少测试一次；**
  - **对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次；**
  - **在循环的边界和运行界限内执行循环体；**
  - **测试内部数据结构的有效性，等。**



- 对一个具有**多重选择和循环嵌套**的程序，**不同的路径数目可能是天文数字**。给出一个小程序的流程图，它包括了一个执行**20**次的循环。
- 包含的不同执行路径数达5^{20}条，对每一条路径进行测试需要**1**毫秒，假定一年工作365×24小时，要想把所有路径测试完，需**3170**年。

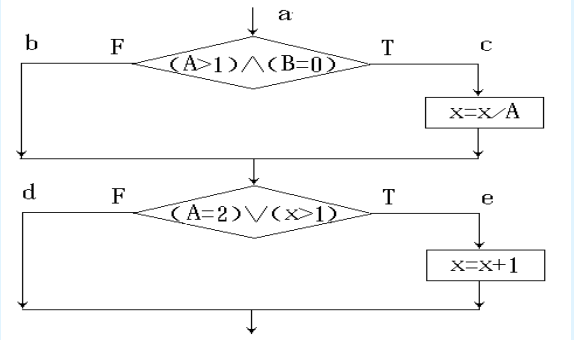
白盒测试的测试用例设计

逻辑覆盖

逻辑覆盖是以**程序内部的逻辑结构为基础**的设计测试用例的技术。它属白盒测试。

- **语句覆盖**
- **判定覆盖**
- **条件覆盖**
- **判定—条件覆盖**
- **条件组合覆盖**
- **路径覆盖。**

举例：所有路径为：L1(a->c->e) ,L2(a->b->d), L3(a->b->e), L4(a->c->d)



$$L1(a \to c \to e)$$
$$= \{(A>1)\ and\ (B=0)\}\ and\ \{(A=2)\ or\ (X/A>1)\}$$
$$= (A>1)\ and\ (B=0)\ and\ (A=2)\ or$$
$$(A>1)\ and\ (B=0)\ and\ (X/A>1)$$
$$= (A=2)\ and\ (B=0)\ or$$
$$(A>1)\ and\ (B=0)\ and\ (X/A>1)$$

$$L2(a \to b \to d)$$
$$= \overline{\{(A>1)\ and\ (B=0)\}}\ and\ \overline{\{(A=2)\ or\ (X>1)\}}$$
$$= \{\overline{(A>1)}\ or\ \overline{(B=0)}\}\ and\ \{\overline{(A=2)}\ and\ \overline{(X>1)}\}$$
$$= \overline{(A>1)}\ and\ \overline{(A=2)}\ and\ \overline{(X>1)}\ or$$
$$\overline{(B=0)}\ and\ \overline{(A=2)}\ and\ \overline{(X>1)}$$
$$= (A \le 1)\ and\ (X \le 1)\ or$$
$$(B \ne 0)\ and\ (A \ne 2)\ and\ (X \le 1)$$

L3(a→b→c)

$= \overline{\{(A>1)\ and\ (B=0)\}}\ and\ \{(A=2)\ or\ (X>1)\}$

$= \{\overline{(A>1)}\ or\ \overline{(B=0)}\}\ and\ \{(A=2)\ or\ (X>1)\}$

$= \overline{(A>1)}\ \text{and}\ (X>1)\ \text{or}$

$\overline{(B=0)}\ \text{and}\ (A=2)\ \text{or}\ \overline{(B=0)}\ \text{and}\ (X>1)$

$= (A \le 1)\ and\ (X>1)\ or$

$(B \ne 0)\ and\ (A=2)\ or\ (B \ne 0)\ and\ (X>1)$

L4 (a → c → d)

$= \{(A>1)\ and\ (B=0)\}\ and\ \overline{\{(A=2)\ or\ (X/A>1)\}}$

$= (A>1)\ and\ (B=0)\ and\ (A \ne 2)\ and\ (X/A \le 1)$

依据以上推导出来的结果就可以设计满足要求的测试用例。

语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得**每一可执行语句至少执行一次**。
- 在图例中，正好所有的可执行语句都在**路径L1**上，所以选择**路径 L1**设计测试用例，就可以覆盖所有的可执行语句。

- 测试用例的设计格式如下
  【输入的(A, B, X)，输出的(A, B, X)】
- 为图例设计满足**语句覆盖**的测试用例是:
  【(2, 0, 4)，(2, 0, 3)】
  覆盖 ace【L1】

$$(A=2)\ and\ (B=0)\ or\ (A>1)\ and\ (B=0)\ and\ (X/A>1)$$

判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得**程序中每个判断的取真分支和取假分支至少经历一次**。
- 判定覆盖又称为**分支覆盖**。
- 对于图例，如果选择**路径L1**和**L2**，就可得满足要求的测试用例:

- **【(2, 0, 4)，(2, 0, 3)】覆盖 ace【L1】**
  【(1, 1, 1)，(1, 1, 1)】覆盖 abd【L2】

$$(A=2)\ and\ (B=0)\ or\ (A>1)\ and\ (B=0)\ and\ (X/A>1)$$

$$(A \le 1)\ and\ (X \le 1)\ or\ (B \ne 0)\ and\ (A \ne 2)\ and\ (X \le 1)$$

- 如果选择路径L3和L4，还可得另一组可用的测试用例:
  【(2, 1, 1)，(2, 1, 2)】覆盖 abe【L3】
  【(3, 0, 3)，(3, 1, 1)】覆盖 acd【L4】

$$(A \le 1)\ and\ (X > 1)\ or\ (B \ne 0)\ and\ (A = 2)\ or\ (B \ne 0)\ and\ (X > 1)$$

$$(A > 1)\ and\ (B = 0)\ and\ (A \ne 2)\ and\ (X/A \le 1)$$

条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中**每个判断的每个条件的可能取值至少执行一次**。
- 在图例中，我们事先可对所有条件的取值加以标记。例如，
- 对于第一个判断：
  - 条件 A>1 取真为 T_1，取假为 $\overline{T_1}$
    条件 B=0 取真为 T_2，取假为 $\overline{T_2}$

- 对于第二个判断：
  - 条件A=2 取真为T_3，取假为 $\overline{T_3}$
    条件X>1 取真为T_4，取假为 $\overline{T_4}$

测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 0, 1)，(1, 0, 1)】	L2(b, d)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$

或

测 试 用 例	覆盖分支	条件取值
【(1, 0, 3)，(1, 0, 4)】	L3(b, e)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1T_2T_3T_4$

判定—条件覆盖

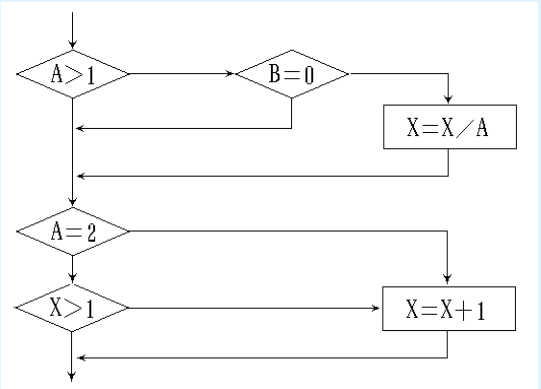
- 判定—条件覆盖就是设计足够的测试用例，使得**判断中每个条件的所有可能取值至少执行一次**，同时**每个判断中的每个条件的可能取值至少执行一次**。

测 试 用 例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】	L2(b, d)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$

$$(A = 2)\ and\ (B = 0)\ or\ (A > 1)\ and\ (B = 0)\ and\ (X/A > 1)$$

$$(A \le 1)\ and\ (X \le 1)\ or\ (B \ne 0)\ and\ (A \ne 2)\ and\ (X \le 1)$$

由多个基本判断组成的流程图



条件组合覆盖

- 条件组合覆盖就是设计足够的测试用例，运行被测程序，使得**每个判断的所有可能的条件取值组合至少执行一次**。

记 ① A＞1, B＝0 作 $T_1\overline{T_2}$
② A＞1, B≠0 作 T_1T_2
③ A≯1, B＝0 作 $\overline{T_1}\overline{T_2}$
④ A≯1, B≠0 作 $\overline{T_1}T_2$

⑤ A＝2, X＞1 作 $T_3\overline{T_4}$
⑥ A＝2, X≯1 作 T_3T_4
⑦ A≠2, X＞1 作 $\overline{T_3}\overline{T_4}$
⑧ A≠2, X≯1 作 $\overline{T_3}T_4$

测 试 用 例	覆盖条件	覆盖组合
【(2, 0, 4), (2, 0, 3)】	(L1) $T_1\overline{T_2}T_3\overline{T_4}$	①, ⑤
【(2, 1, 1), (2, 1, 2)】	(L3) $T_1T_2\overline{T_3}T_4$	②, ⑥
【(1, 0, 3), (1, 0, 4)】	(L3) $\overline{T_1}T_2\overline{T_3}\overline{T_4}$	③, ⑦
【(1, 1, 1), (1, 1, 1)】	(L2) $\overline{T_1}\overline{T_2}\overline{T_3}T_4$	④, ⑧

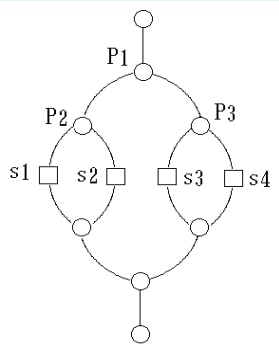
路径测试

- 路径测试就是设计足够的测试用例，**覆盖程序中所有可能的路径**。

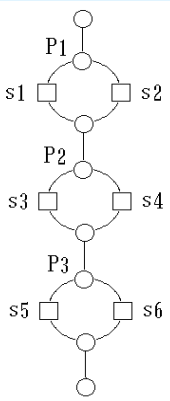
测 试 用 例	通过路径	覆盖条件
【(2, 0, 4), (2, 0, 3)】	ace (L1)	$T_1T_2T_3T_4$
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$\overline{T_1}T_2\overline{T_3}T_4$
【(3, 0, 3), (3, 0, 1)】	acd (L3)	$T_1\overline{T_2}T_3T_4$

条 件 测 试 路 径 选 择

- 当程序中判定多于一个时，形成的分支结构可以分为两类：**嵌套型分支结构**和**连锁型分支结构**。
- 对于嵌套型分支结构，若有**n**个判定语句，需要**n+1**个测试用例；
- 对于连锁型分支结构， 若有**n**个判定语句，需要有2^n个测试用例，覆盖它的2^n条路径。当**n**较大时将无法测试。



（a）嵌套型分支结构



（b）连锁型分支结构

循环测试路径选择

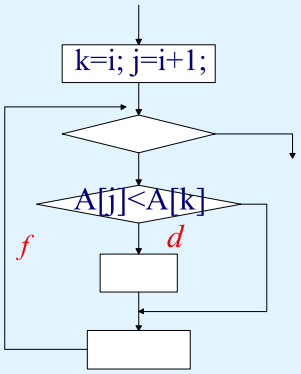
- 循环分为4种不同类型：**简单循环**、**连锁循环**、**嵌套循环**和**非结构循环**。

(1) 简单循环

① 零次循环：从循环入口到出口
② 一次循环：检查循环初始值
③ 二次循环：检查多次循环
④ m次循环： 检查在多次循环
⑤ 最大次数循环、比最大次数多一次、少一次的循环。

例：求最小值

```
k = i;
  ❶
for ( j = i+1;  j <= n;  j++ )
       ❷          ❸      ❻
   if ( A[j] < A[k] ) then  k = j;
          ❹              ❺
```



测试用例选择

循环	i	n	A[i]	A[i+1]	A[i+2]	k	路 径
0	1	1				i	ac
1	1	2	1	2		i	abefc
			2	1		i+1	abdfc
2	1	3	1	2	3	i	abefefc
			2	3	1	i+2	abefdfc
			3	2	1	i+2	abdfdfc
			3	1	2	i+1	abdfefc

d 改 k 的值，e 不改 k 的值

(2) 嵌套循环

① 对最内层循环做简单循环的全部测试。所有其它层的循环变量置为最小值；

② 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。。

③ 反复进行，直到所有各层循环测试完毕。

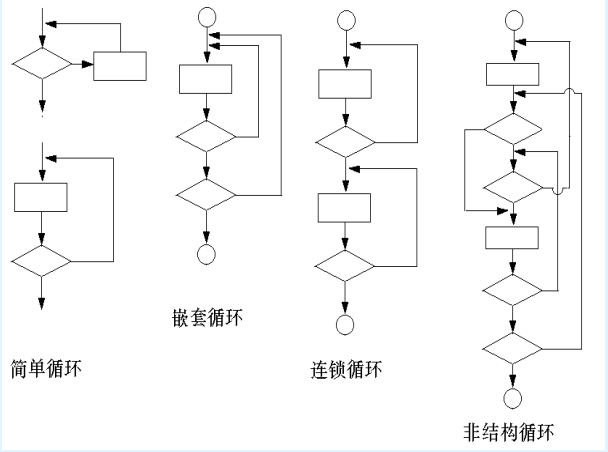
④ 对全部各层循环同时取最小循环次数，或者同时取最大循环次数

(3) 连锁循环

如果各个循环互相独立，则可以用与简单循环相同的方法进行测试。但如果几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。

(4) 非结构循环

这一类循环应该使用结构化程序设计方法重新设计测试用例。



黑盒测试的测试用例设计

- 等价类划分
- 边界值分析
- 错误推测法
- 因果图

等价类划分

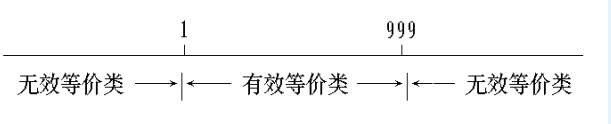
- 等价类划分是一种典型的黑盒测试方法，使用这一方法时，**完全不考虑程序的内部结构，只依据程序的规格说明来设计测试用例**。
- 等价类划分方法**把所有可能的输入数据**，即程序的输入域**划分成若干部分**，然后**从每一部分中选取少数有代表性的数据做为测试用例**。

- 使用这一方法设计测试用例要经历**划分等价类**（列出等价类表）和**选取测试用例**两步。
- **划分等价类**
  等价类是指某个输入域的子集合。在该子集合中，**各个输入数据对于揭露程序中的错误都是等效的**。测试某等价类的代表值就等价于对这一类其它值的测试。

- 等价类的划分有两种不同的情况：
  ① **有效等价类**：是指对于程序的规格说明来说，是合理的，有意义的输入数据构成的集合。
  ② **无效等价类**：是指对于程序的规格说明来说，是不合理的，无意义的输入数据构成的集合。
- 在设计测试用例时，要同时考虑有效等价类和无效等价类的设计。

- 划分等价类的原则。
  (1) 如果输入条件规定了取值范围，或值的个数，则可以确立一个有效等价类和两个无效等价类。

- 例如，在程序的规格说明中，对输入条件有一句话：
  “…… 项数可以从1到999 ……”
  则有效等价类是**“1≤项数≤999”**
  两个无效等价类是**“项数＜1”**或**“项数＞999”**。在数轴上表示成:



(2) 如果输入条件规定了输入值的集合，或者是规定了“必须如何”的条件，这时可确立一个有效等价类和一个无效等价类。

- 例如，在Pascal语言中对变量标识符规定为“以字母打头的……串”。那么所有以字母打头的构成有效等价类，而不在此集合内（不以字母打头）的归于无效等价类。

(3) 如果输入条件是一个布尔量，则可以确定一个有效等价类和一个无效等价类。

(4) 如果规定了输入数据的一组值，而且程序要对每个输入值分别进行处理。这时可为 每一个输入值确立一个有效等价类，此外针对这组值确立一个无效等价类，它是所有不允许的输入值的集合。

- 例如，在教师上岗方案中规定对教授、副教授、讲师和助教分别计算分数，做相应的处理。因此可以确定4个有效等价类为教授、副教授、讲师和助教，一个无效等价类，它是所有不符合以上身分的人员的输入值的集合。

(5) 如果规定了输入数据必须遵守的规则，则可以确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

- 例如，**Pascal**语言规定 “一个语句必须以分号‘;’结束”。这时，可以确定一个有效等价类 “以‘;’结束”，若干个无效等价类 “以‘:’结束”、“以‘,’结束”、“以‘ ’结束”、“以**LF**结束”等。
- 确立测试用例
  在确立了等价类之后，建立等价类表，列出所有划分出的等价类。

输入条件	有效等价类	无效等价类
……	……	……
……	……	……

- 再从划分出的等价类中按以下原则选择测试用例：
  (1) 为每一个等价类规定一个唯一编号；
  (2) 设计一个新的测试用例，使其**尽可能多地覆盖尚未被覆盖的有效等价类**，重复这一步，直到所有的有效等价类都被覆盖为止；
  （3）设计一个新的测试用例，使其**仅覆盖一个尚未被覆盖的无效等价类**，重复这一步，直到所有的无效等价类都被覆盖为止。

- 用等价类划分法设计测试用例的实例
  在某一**PASCAL**语言版本中规定：“标识符是由字母开头，后跟字母或数字的任意组合构成。有效字符数为8个，最大字符数为80个。”
  并且规定：“标识符必须先说明，再使用。”“在同一说明语句中，标识符至少必须有一个。”

用等价类划分的方法，建立输入等价类表：

输入条件	有效等价类	无效等价类
标识符个数	1个（1），多个（2）	0个（3）
标识符字符数	1~8个（4）	0个（5），>8个（6），>80个（7）
标识符组成	字母（8），数字（9）	非字母数字字符（10），保留字（11）
第一个字符	字母（12）	非字母（13）
标识符使用	先说明后使用（14）	未说明已使用（15）

- 下面选取了9个测试用例，它们覆盖了所有的等价类。

① VAR x，T1234567：REAL；
BEGIN x := 3.414；
T1234567 := 2.732；
……
(1), (2), (4), (8), (9), (12), (14)

② VAR ： REAL； (3)

③ VAR x，： REAL； (5)

④ VAR T12345678：REAL； (6)

⑤ VAR T12345……：REAL； (7)
多于80个字符

⑥ VAR T$：CHAR； (10)

⑦ VAR GOTO：INTEGER； (11)

⑧ VAR 2T：REAL； (13)

⑨ VAR PAR：REAL； (15)
BEGIN ……
PAP := SIN (3.14 * 0.8) / 6；

边界值分析

- 边界值分析也是一种黑盒测试方法，是对等价类划分方法的补充。
- 人们从长期的测试工作经验得知，**大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部**。因此针对各种边界情况设计测试用例，可以查出更多的错误。

- 比如，在做三角形计算时，要输入三角形的三个边长：A、B和C。 我们应注意到这三个数值应当满足
A＞0、B＞0、C＞0、
A＋B＞C、A＋C＞B、B＋C＞A，
才能构成三角形。但如果把六个不等式中的任何一个大于号“＞”错写成大于等于号“≥”，那就不能构成三角形。问题恰出现在容易被疏忽的边界附近。

- 这里所说的边界是指，相当于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。
- 使用边界值分析方法设计测试用例，首先应确定边界情况。**应当选取正好等于，刚刚大于，或刚刚小于边界的值做为测试数据**，而不是选取等价类中的典型值或任意值做为测试数据。

错误推测法

- 人们也可以靠经验和直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的例子。这就是错误推测法。
- 错误推测法的基本想法是：**列举出程序中所有可能有的错误和容易发生错误的特殊情况，根据它们选择测试用例**。

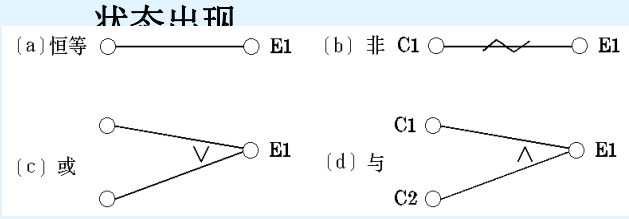
因果图

- 因果图的适用范围
  如果在测试时必须考虑**输入条件的各种组合**，可使用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来设计测试用例，这就需要利用因果图。
  因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

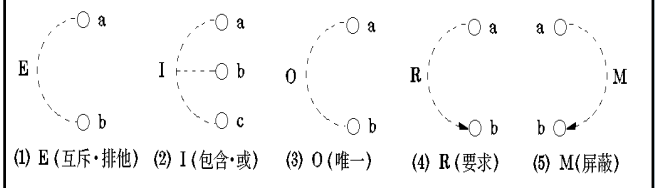
- 用因果图生成测试用例的基本步骤
  (1) 分析软件规格说明描述中，哪些是原因 (即输入条件或输入条件的等价类)，哪些是结果 (即输出条件)，并给每个原因和结果赋予一个标识符。
  (2) 分析软件规格说明描述中的语义，找出原因与结果之间，原因与原因之间对应的是什么关系? 根据这些关系，画出因果图。

(3) 由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出现。为表明这些特殊情况，在因果图上用一些记号标明约束或限制条件。
(4) 把因果图转换成判定表。
(5) 把判定表的每一列拿出来作为依据，设计测试用例。

- 在因果图中出现的基本符号
  通常在因果图中用Ci表示原因，用Ei表示结果，各结点表示状态，可取值“0”或“1”。“0”表示某状态不出现，“1”表示某状态出现



- 表示约束条件的符号
  为了表示原因与原因之间，结果与结果之间可能存在的约束条件，在因果图中可以附加一些表示约束条件的符号。



- 例如，有一个处理单价为5角钱的饮料的自动售货机软件测试用例的设计。其规格说明如下：

若投入5角钱或1元钱的硬币，押下〖橙汁〗或〖啤酒〗的按钮，则相应的饮料就送出来。若售货机没有零钱找，则一个显示〖零钱找完〗的红灯亮，这时在投入1元硬币并押下按钮后，饮料不送出来而且1元硬币也退出来；若有零钱找，则显示〖零钱找完〗的红灯灭，在送出饮料的同时退还5角硬币。”

(1) 分析这一段说明，列出原因和结果

原因: 1. 售货机有零钱找
2. 投入1元硬币
3. 投入5角硬币
4. 押下橙汁按钮
5. 押下啤酒按钮

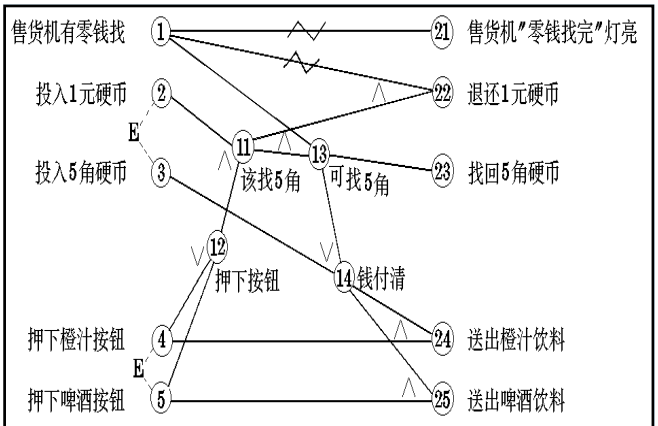
建立中间结点，表示处理中间状态
11. 投入1元硬币且押下饮料按钮
12. 押下〖橙汁〗或〖啤酒〗的按钮
13. 应当找5角零钱并且售货机有零钱找
14. 钱已付清

结果： 21. 售货机〖零钱找完〗灯亮
22. 退还1元硬币
23. 退还5角硬币
24. 送出橙汁饮料
25. 送出啤酒饮料

(2) 画出因果图。所有原因结点列在左边，所有结果结点列在右边。

(3) 由于 2 与 3，4 与 5 不能同时发生，分别加上约束条件E。

(4) 因果图

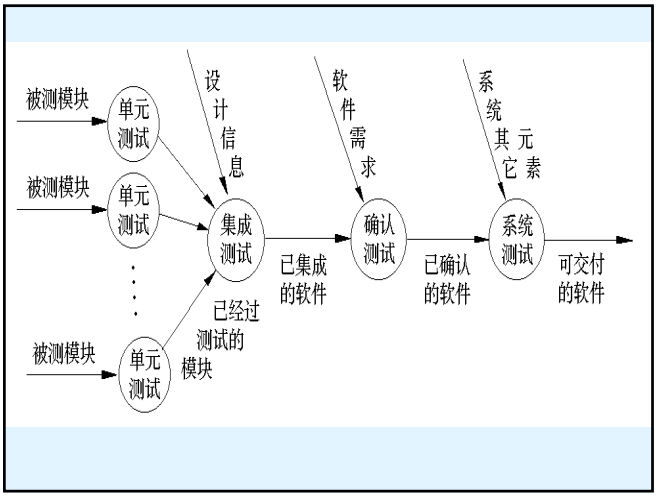


(5) 转换成判定表

序号		1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1	2
条件	①	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	②	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
	③	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0
	④	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	⑤	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
中间结果	⑪						1	1	0			0	0	0		0	0	0				1	1	0		0	0	0		0	0	0
	⑫						1	1	0			1	1	0		1	1	0				1	1	0		1	1	0		1	1	0
	⑬						1	1	0			0	0	0		0	0	0				0	0	0		0	0	0		0	0	0
	⑭						1	1	0			1	1	1		0	0	0				0	0	0		1	1	1		0	0	0
结果	㉑						0	0	0			0	0	0		0	0	0				1	1	1		1	1	1		1	1	1
	㉒						0	0	0			0	0	0		0	0	0				1	1	0		0	0	0		0	0	0
	㉓						1	1	0			0	0	0		0	0	0				0	0	0		0	0	0		0	0	0
	㉔						1	0	0			1	0	0		0	0	0				0	0	0		1	0	0		0	0	0
	㉕						0	1	0			0	1	0		0	0	0				0	0	0		0	1	0		0	0	0
测试用例							Y	Y	Y			Y	Y	Y		Y	Y					Y	Y	Y		Y	Y	Y		Y	Y	

软件测试的策略

- 测试过程按4个步骤进行，即单元测试、组装测试、确认测试和系统测试。
- 开始是单元测试，集中对用源代码实现的每一个程序单元进行测试，检查各个程序模块是否正确地实现了规定的功能。



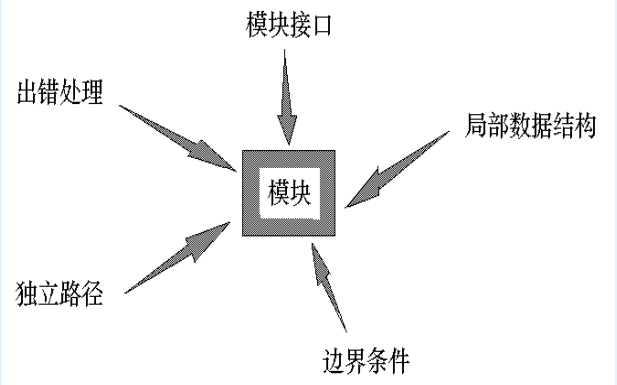
- **组装测试**把已测试过的模块组装起来，主要对与设计相关的软件体系结构的构造进行测试。
- **确认测试**则是要检查已实现的软件是否满足了需求规格说明中确定了的各种需求，以及软件配置是否完全、正确。
- **系统测试**把已经经过确认的软件纳入实际运行环境中，与其它系统成份组合在一起进行测试。

单元测试 (Unit Testing)

- 单元测试又称模块测试，**是针对软件设计的最小单位 — 程序模块，进行正确性检验**的测试工作。其目的在于发现各模块内部可能存在的各种差错。
- 单元测试需要**从程序的内部结构出发设计测试用例**。多个模块可以平行地独立进行单元测试。

1. 单元测试的内容

- 在单元测试时，测试者需要依据详细设计说明书和源程序清单，了解该模块的I/O条件和模块的逻辑结构，主要采用白盒测试的测试用例，辅之以黑盒测试的测试用例，使之对任何合理的输入和不合理的输入，都能鉴别和响应。



(1) 模块接口测试

- 在单元测试的开始，应对**通过被测模块的数据流**进行测试。测试项目包括：
  - 调用本模块的输入参数是否正确；
  - 本模块调用子模块时输入给子模块的参数是否正确；
  - 全局量的定义在各模块中是否一致；

- 在做内外存交换时要考虑：
  - 文件属性是否正确；
  - OPEN与CLOSE语句是否正确；
  - 缓冲区容量与记录长度是否匹配；
  - 在进行读写操作之前是否打开了文件；
  - 在结束文件处理时是否关闭了文件；
  - 正文书写 / 输入错误，
  - I / O错误是否检查并做了处理。

(2) 局部数据结构测试

- 不正确或不一致的数据类型说明
- 使用尚未赋值或尚未初始化的变量
- 错误的初始值或错误的缺省值
- 变量名拼写错或书写错
- 不一致的数据类型
- 全局数据对模块的影响

(3) 路径测试

- 选择适当的测试用例，对模块中重要的执行路径进行测试。
- 应当设计测试用例查找由于错误的计算、不正确的比较或不正常的控制流而导致的错误。
- 对基本执行路径和循环进行测试可以发现大量的路径错误。

(4) 错误处理测试

- 出错的描述是否难以理解
- 出错的描述是否能够对错误定位
- 显示的错误与实际的错误是否相符
- 对错误条件的处理正确与否
- 在对错误进行处理之前，错误条件是否已经引起系统的干预等

(5) 边界测试

- 注意数据流、控制流中刚好等于、大于或小于确定的比较值时出错的可能性。对这些地方要仔细地选择测试用例，认真加以测试。
- 如果对模块运行时间有要求的话，还要专门进行关键路径测试，以确定最坏情况下和平均意义下影响模块运行时间的因素。

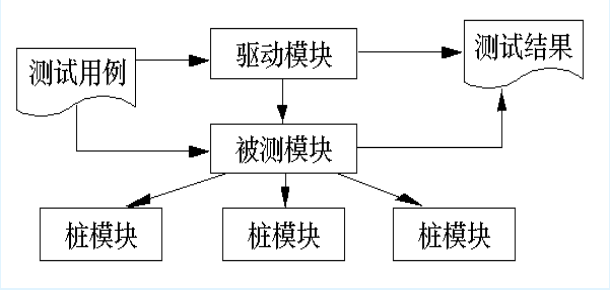
2. 单元测试的步骤

- 模块并不是一个独立的程序，在考虑测试模块时，同时要考虑它和外界的联系，用一些辅助模块去模拟与被测模块相联系的其它模块。
  - 驱动模块 (driver)
  - 桩模块 (stub) —— 存根模块

驱动模块 (driver) —— 相当于所测模块的主程序。它接收测试数据，把这些数据传送给所测模块，最后再输出实测结果。

桩模块 (stub) —— 存根模块。用以代替所测模块调用的子模块。

单元测试的测试环境



组装测试（Integrated Testing）

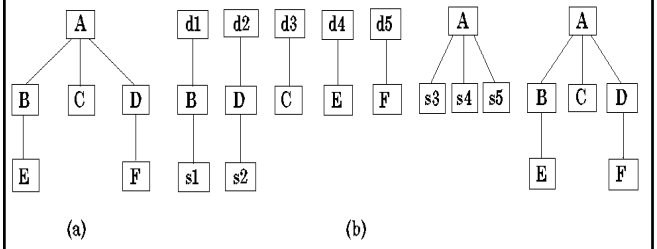
- 组装测试 (集成测试、联合测试）
- 通常，在单元测试的基础上，需要将所有模块按照设计要求组装成为系统。这时需要考虑的问题是：
  - 在把各个模块连接起来的时侯，**穿越模块接口的数据**是否会丢失；
  - **一个模块的功能是否会对另一个模块的功能产生不利的影响**；
  - **各个子功能组合起来，能否达到预期要求的父功能**；
  - **全局数据结构是否有问题**；
  - **单个模块的误差累积起来，是否会放大**，从而达到不能接受的程度。

在单元测试的同时可进行组装测试，发现并排除在模块连接中可能出现的问题，最终构成要求的软件系统。

- 子系统的组装测试特别称为**部件测试**，它所做的工作是要找出组装后的**子系统与系统需求规格说明之间**的不一致。
- 通常，把模块组装成为系统的方式有两种
  - 一次性组装方式
  - 增殖式组装方式

1. 一次性组装方式 (big bang)

- 它是一种非增殖式组装方式。也叫做整体拼装。
- 使用这种方式，首先对每个模块分别进行模块测试，然后再把所有模块组装在一起进行测试，最终得到要求的软件系统。

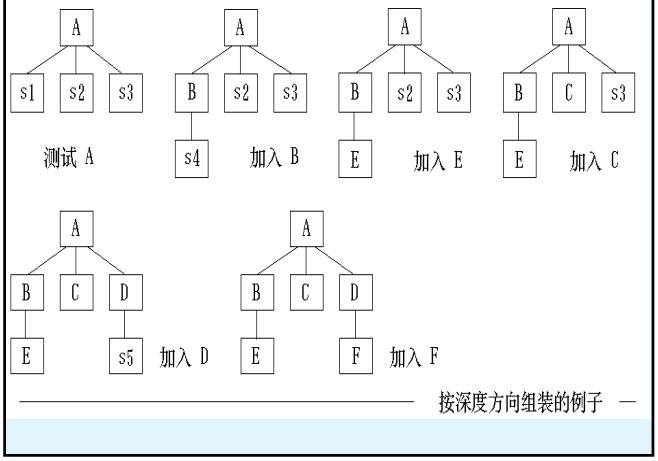


2. 增殖式组装方式

- 这种组装方式又称**渐增式组装**
- 首先对一个个模块进行模块测试，然后将这些模块逐步组装成较大的系统
- 在组装的过程中边连接边测试，以发现连接过程中产生的问题
- 通过增殖逐步组装成为要求的软件系统。

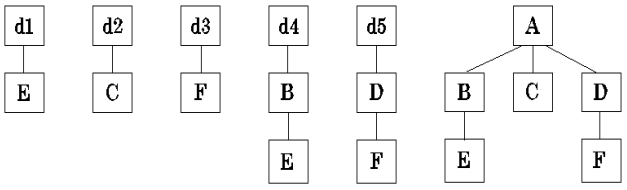
(1) 自顶向下的增殖方式

- 这种组装方式将模块**按系统程序结构，沿控制层次自顶向下进行组装**。
- 自顶向下的增殖方式在测试过程中较早地验证了主要的控制和判断点。
- 选用按深度方向组装的方式，可以首先实现和验证一个完整的软件功能。



(2) 自底向上的增殖方式

- 这种组装的方式是从**程序模块结构的最底层的模块开始组装和测试**。
- 因为模块是自底向上进行组装，对于一个给定层次的模块，它的子模块（包括子模块的所有下属模块）已经组装并测试完成，所以**不再需要桩模块**。在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到。



- 自顶向下增殖的方式和自底向上增殖的方式各有优缺点。
- 一般来讲，一种方式的优点是另一种方式的缺点。

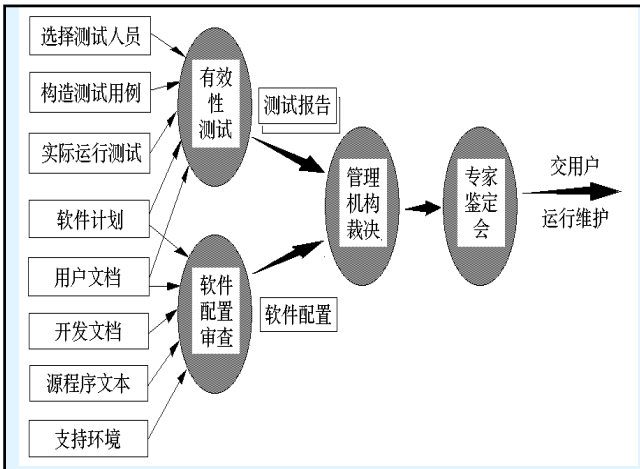
(3) 混合增殖式测试

- **衍变的自顶向下的增殖测试**
  - 首先对输入 / 输出模块和引入新算法模块进行测试;
  - 再自底向上组装成为功能相当完整且相对独立的子系统;
  - 然后由主模块开始自顶向下进行增殖测试。

- **自底向上–自顶向下的增殖测试**
  - 首先对含读操作的子系统自底向上直至根结点模块进行组装和测试;
  - 然后对含写操作的子系统做自顶向下的组装与测试。
- **回归测试**
  - 这种方式采取自顶向下的方式测试被修改的模块及其子模块;
  - 然后将这一部分视为子系统，再自底向上测试。

确认测试（Validation Testing）

- 确认测试又称**有效性测试**。任务是验证软件的功能和性能及其它特性是否与用户的要求一致。
- 对软件的功能和性能要求在软件需求规格说明书中已经明确规定。它包含的信息就是软件确认测试的基础。



1. 进行有效性测试（黑盒测试）

- 有效性测试是在模拟的环境 (可能就是开发的环境) 下，运用黑盒测试的方法，验证被测软件是否满足需求规格说明书列出的需求。
- 首先制定测试计划，规定要做测试的种类。还需要制定一组测试步骤，描述具体的测试用例。

- 通过实施预定的测试计划和测试步骤，确定
  - 软件的特性是否与需求相符;
  - 所有的文档都是正确且便于使用;
  - 同时，对其它软件需求，例如可移植性、兼容性、出错自动恢复、可维护性等，也都要进行测试

- 在全部软件测试的测试用例运行完后，所有的测试结果可以分为两类：
  - **测试结果与预期的结果相符**。这说明软件的这部分功能或性能特征与需求规格说明书相符合，从而这部分程序被接受。
  - **测试结果与预期的结果不符**。这说明软件的这部分功能或性能特征与需求规格说明不一致，因此要为它提交一份问题报告。

2. 软件配置复查

- 软件配置复查的目的是保证
  - 软件配置的所有成分都齐全；
  - 各方面的质量都符合要求；
  - 具有维护阶段所必需的细节；
  - 而且已经编排好分类的目录。
- 应当严格遵守用户手册和操作手册中规定的使用步骤，以便检查这些文档资料的完整性和正确性。

3. α 测试和 β 测试

- 在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。
- **α 测试**是由一个**用户在开发环境下进行的测试，**也可以是**公司内部的用户在模拟实际操作环境下进行的测试。**

- **α 测试**的目的是评价软件产品的**FLURPS**（即功能、局域化、可使用性、可靠性、性能和支持）。尤其注重产品的界面和特色。
- **α 测试**可以从软件产品编码结束之时开始，或在模块（子系统）测试完成之后开始，也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。

- **β 测试**是由软件的**多个用户在实际使用环境下进行的测试**。这些用户返回有关错误信息给开发者。
- 测试时，开发者通常不在测试现场。因而，**β 测试**是在开发者无法控制的环境下进行的软件现场应用。
- 在 β 测试中，由用户记下遇到的所有问题，包括真实的以及主观认定的，定期向开发者报告。

- **β 测试**主要衡量产品的**FLURPS**。着重于产品的支持性，包括文档、客户培训和支持产品生产能力。
- 只有当**α 测试**达到一定的可靠程度时，才能开始**β 测试**。它处在整个测试的最后阶段。同时，产品的所有手册文本也应该在此阶段完全定稿。

4.验收测试（Acceptance Testing）

- 在通过了系统的有效性测试及软件配置审查之后，就应开始系统的验收测试。
- 验收测试是以用户为主的测试。软件开发人员和QA（质量保证）人员也应参加。
- 由用户参加设计测试用例，使用生产中的实际数据进行测试。

- 在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。
- 确认测试应交付的文档有：
  - 确认测试分析报告
  - 最终的用户手册和操作手册
  - 项目开发总结报告。

系统测试（System Testing）

- 系统测试，是将通过确认测试的软件，**作为整个基于计算机系统的一个元素**，与计算机硬件、外设、某些支持软件、数据和人员等其它系统元素结合在一起，**在实际运行环境下**，对计算机系统进行一系列的组装测试和确认测试。
- 系统测试的目的在于**通过与系统的需求定义作比较，发现软件与系统的定义不符合或与之矛盾的地方**。

测试种类

- 软件测试是由一系列不同的测试组成。主要目的是对以计算机为基础的系统进行充分的测试。

功能测试

功能测试是在规定的一段时间内运行软件系统的所有功能，以验证这个软件系统有无严重错误。

可靠性测试

如果系统需求说明书中有对可靠性的要求，则需进行可靠性测试。

① 平均失效间隔时间 MTBF (Mean Time Between Failures) 是否超过规定时限?

② 因故障而停机的时间 MTTR (Mean Time To Repairs) 在一年中应不超过多少时间。

强度测试

强度测试是要检查在系统运行环境不正常乃至发生故障的情况下，系统可以运行到何种程度的测试。

- 强度测试的一个变种就是**敏感性测试**。在程序有效数据界限内一个小范围内的一组数据可能引起极端的或不平稳的错误处理出现，或者导致极度的性能下降的情况发生。此测试用以发现可能引起这种不稳定性或不正常处理的某些数据组合。

性能测试

性能测试是要检查系统是否满足在需求说明书中规定的性能。特别是对于实时系统或嵌入式系统。

性能测试常常**需要与强度测试结合起来进行，并常常要求同时进行硬件和软件检测。**

- 通常，对软件性能的检测表现在以下几个方面：**响应时间**、**吞吐量**、**辅助存储区**，例如缓冲区，工作区的大小等、**处理精度**，等等。

恢复测试

恢复测试是要证实在**克服硬件故障**(包括掉电、硬件或网络出错等)**后，系统能否正常地继续进行工作**，并不对系统造成任何损害。

- 为此，可采用各种人工干预的手段，模拟硬件故障，故意造成软件出错。并由此检查：
  - **错误探测功能**——系统能否发现硬件失效与故障；
  - 能否**切换或启动备用的硬件**；
  - 在故障发生时能否**保护正在运行的作业和系统状态**；
  - 在系统恢复后能否**从最后记录下来的无错误状态开始继续执行作业**，等等。
  - **掉电测试**：其目的是测试软件系统在发生电源中断时能否**保护当时的状态且不毁坏数据，然后在电源恢复时从保留的断点处重新进行操作。**

启动／停止测试

这类测试的目的是验证**在机器启动及关机阶段，软件系统正确处理的能力。**

这类测试包括

- **反复启动软件系统** (例如，操作系统自举、网络的启动、应用程序的调用等)
- **在尽可能多的情况下关机**。

配置测试

- 这类测试是要检查**计算机系统内各个设备或各种资源之间的相互联结和功能分配中的错误。**
- 它主要包括以下几种：
  - **配置命令测试**：验证全部配置命令的可操作性（有效性）；特别对最大配置和最小配置要进行测试。软件配置和硬件配置都要测试。
  - **循环配置测试**：证明对每个设备物理与逻辑的，逻辑与功能的每次循环置换配置都能正常工作。
  - **修复测试**：检查每种配置状态及哪个设备是坏的。并用自动的或手工的方式进行配置状态间的转换。

- **安全性测试**

  安全性测试是要检验**在系统中已经存在的系统安全性、保密性措施是否发挥作用，有无漏洞**。
- 力图破坏系统的保护机构以进入系统的主要方法有以下几种：
  - 正面攻击或从侧面、背面攻击系统中易受损坏的那些部分；
  - 以系统输入为突破口，利用输入的容错性进行正面攻击；
  - 申请和占用过多的资源压垮系统，以破坏安全措施，从而进入系统；
  - 故意使系统出错，利用系统恢复的过程，窃取用户口令及其它有用的信息；
  - 通过浏览残留在计算机各种资源中的垃圾（无用信息），以获取如口令，安全码，译码关键字等信息；
  - 浏览全局数据，期望从中找到进入系统的关键字；
  - 浏览那些逻辑上不存在，但物理上还存在的各种记录和资料等。

可使用性测试

- 可使用性测试主要从使用的**合理性**和**方便性**等角度对软件系统进行检查，发现人为因素或使用上的问题。
- 要保证在足够详细的程度下，**用户界面便于使用；对输入量可容错、响应时间和响应方式合理可行、输出信息有意义、正确并前后一致；出错信息能够引导用户去解决问题；软件文档全面、正规、确切**。

可支持性测试

这类测试是要验证**系统的支持策略对于公司与用户方面是否切实可行**。

- 它所采用的方法是
  - **试运行支持过程**(如对有错部分打补丁的过程，热线界面等)；
  - 对其结果进行**质量分析**；
  - **评审诊断工具**；
  - **维护过程、内部维护文档**；
  - **修复一个错误所需平均最少时间**。

安装测试

安装测试的目的**不是找软件错误，而是找安装错误。**

- 在安装软件系统时，会有多种选择。
  - 要分配和装入文件与程序库
  - 布置适用的硬件配置
  - 进行程序的联结。
- 而安装测试就是要找出在这些安装过程中出现的错误。

- 安装测试是在系统安装之后进行测试。它要检验：
  - 用户选择的一套任选方案是否相容；
  - 系统的每一部分是否都齐全；
  - 所有文件是否都已产生并确有所需要的内容；
  - 硬件的配置是否合理，等等。

过程测试

- 在一些大型的系统中，部分工作由软件自动完成，其它工作则需由各种人员，包括操作员，数据库管理员，终端用户等，按一定规程同计算机配合，靠人工来完成。
- 指定由人工完成的过程也需经过仔细的检查，这就是所谓的过程测试。

互连测试

- 互连测试是要验证两个或多个不同的系统之间的互连性。

兼容性测试

- 这类测试主要想验证软件产品在不同版本之间的兼容性。有两类基本的兼容性测试：
  - 向下兼容
  - 交错兼容

容量测试

- 容量测试是要检验系统的能力最高能达到什么程度。例如，
  - 对于编译程序，让它处理特别长的源程序；
  - 对于操作系统，让它的作业队列“满员”；
  - 对于信息检索系统，让它使用频率达到最大。

  在使系统的全部资源达到“满负荷”的情形下，测试系统的承受能力。

文档测试

这种测试是检查用户文档(如用户手册)的清晰性和精确性。

- 用户文档中所使用的例子必须在测试中一一试过，确保叙述正确无误。

调试（Debug）

- 软件调试是在进行了成功的测试之后才开始的工作。它与软件测试不同，调试的任务是进一步诊断和改正程序中潜在的错误。
- 调试活动由两部分组成：
  - 确定程序中可疑错误的确切性质和位置。
  - 对程序(设计,编码)进行修改，排除这个错误。

- 调试工作是一个具有很强技巧性的工作。
- 软件运行失效或出现问题，往往只是潜在错误的外部表现，而外部表现与内在原因之间常常没有明显的联系。如果要找出真正的原因，排除潜在的错误，不是一件易事。
- 可以说，调试是通过现象，找出原因的一个思维分析的过程。

调试的步骤

(1) 从错误的外部表现形式入手，确定程序中出错位置；

(2) 研究有关部分的程序，找出错误的内在原因；

(3) 修改设计和代码，以排除这个错误；

(4) 重复进行暴露了这个错误的原始测试或某些有关测试。

- 从技术角度来看，查找错误的难度在于：
  - 现象与原因所处的位置可能相距甚远。
  - 当其它错误得到纠正时，这一错误所表现出的现象可能会暂时消失，但并未实际排除。
  - 现象实际上是由一些非错误原因(例如，舍入不精确)引起的。

  - 现象可能是由于一些不容易发现的人为错误引起的。
  - 错误是由于时序问题引起的，与处理过程无关。
  - 现象是由于难于精确再现的输入状态（例如，实时应用中输入顺序不确定）引起。
  - 现象可能是周期出现的。在软、硬件结合的嵌入式系统中常常遇到。

几种主要的调试方法

调试的关键在于推断程序内部的错误位置及原因。可以采用以下方法：

强行排错

这种调试方法目前使用较多，效率较低。它不需要过多的思考，比较省脑筋。例如：

- 通过内存全部打印来调试，在这大量的数据中寻找出错的位置。

  - 在程序特定部位设置打印语句，把打印语句插在出错的源程序的各个关键变量改变部位、重要分支部位、子程序调用部位，跟踪程序的执行，监视重要变量的变化。
  - 自动调试工具。利用某些程序语言的调试功能或专门的交互式调试工具，分析程序的动态过程，而不必修改程序。

应用以上任一种方法之前，都应当对错误的征兆进行全面彻底的分析，得出对出错位置及错误性质的推测，再使用一种适当的调试方法来检验推测的正确性。

回溯法调试

这是在小程序中常用的一种有效的调试方法。

一旦发现了错误，人们先分析错误征兆，确定最先发现“症状”的位置。

然后，人工沿程序的控制流程，向回追踪源程序代码，直到找到错误根源或确定错误产生的范围。

- 例如，程序中发现错误处是某个打印语句。通过输出值可推断程序在这一点上变量的值。再从这一点出发，回溯程序的执行过程，反复考虑：“如果程序在这一点上的状态（变量的值）是这样，那么程序在上一点的状态一定是这样...”， 直到找到错误的位置。

归纳法调试

- 归纳法是一种从特殊推断一般的系统化思考方法。归纳法调试的基本思想是：从一些线索(错误征兆)着手，通过分析它们之间的关系来找出错误。
  - **收集有关的数据** 列出所有已知的测试用例和程序执行结果。看哪些输入数据的运行结果是正确的，哪些输入数据的运行结果有错误。

- **组织数据**

由于归纳法是从特殊到一般的推断过程，所以需要组织整理数据，以发现规律。

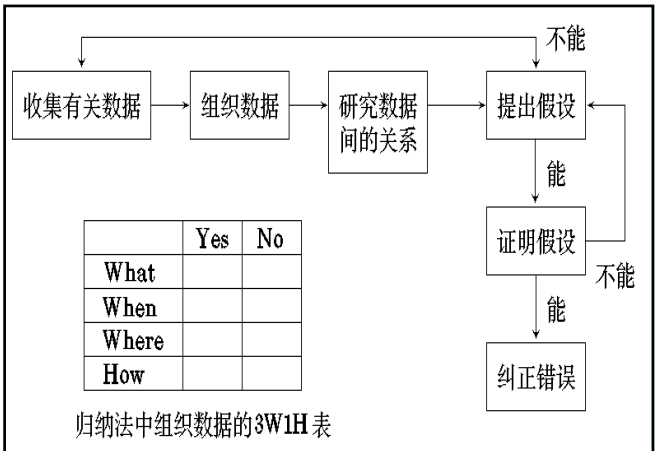
常以3W1H形式组织可用的数据：

“What” 列出一般现象；

“Where”说明发现现象的地点；

“When” 列出现象发生时所有已知情况；

“How” 说明现象的范围和量级；



	Yes	No
What		
When		
Where		
How		

归纳法中组织数据的3W1H表

“Yes”描述出现错误的3W1H；

“No”作为比较，描述了没有错误的3W1H。通过分析找出矛盾来。

- **提出假设**

分析线索之间的关系，利用在线索结构中观察到的矛盾现象，设计一个或多个关于出错原因的假设。如果一个假设也提不出来，归纳过程就需要收集更多的数据。此时，应当再设计与执行一些测试用例，以获得更多的数据。

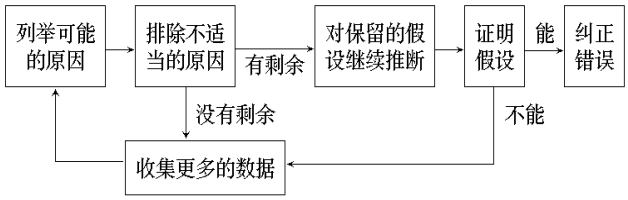
- **证明假设**

把假设与原始线索或数据进行比较，若它能完全解释一切现象，则假设得到证明；否则，就认为假设不合理，或不完全，或是存在多个错误，以致只能消除部分错误。

演绎法调试

演绎法是一种从一般原理或前提出发，经过排除和精化的过程来推导出结论的思考方法。演绎法排错是测试人员首先根据已有的测试用例，设想及枚举出所有可能出错的原因做为假设；然后再用原始测试数据或新的测试，从中逐个排除不可能正确的假设；最后，再用测试数据验证余下的假设确是出错的原因。

- **列举所有可能出错原因的假设**
  把所有可能的错误原因列成表。通过它们，可以组织、分析现有数据。
- **利用已有的测试数据，排除不正确的假设**
  仔细分析已有的数据，寻找矛盾，力求排除前一步列出所有原因。如果所有原因都被排除了，则需要补充一些数据(测试用例)，以建立新的假设。



- **改进余下的假设**
  利用已知的线索，进一步改进余下的假设，使之更具体化，以便可以精确地确定出错位置。
- **证明余下的假设**

调试原则

- 在调试方面，许多原则本质上是心理学方面的问题。调试由两部分组成，调试原则也分成两组。
- **确定错误的性质和位置的原则**
  - 用头脑去分析思考与错误征兆有关的信息。
  - 避开死胡同。

- 只把调试工具当做辅助手段来使用。利用调试工具，可以帮助思考，但不能代替思考。
- 避免用试探法，最多只能把它当做最后手段。

- **修改错误的原则**
  - 在出现错误的地方，很可能还有别的错误。

- 修改错误的一个常见失误是只修改了这个错误的征兆或这个错误的表现，而没有修改错误的本身。
- 当心修正一个错误的同时有可能会引入新的错误。
- 修改错误的过程将迫使人们暂时回到程序设计阶段。
- 修改源代码程序，不要改变目标代码。

第七部分　软件维护

- 软件维护的概念
- 软件维护活动
- 程序修改的步骤及修改的副作用
- 软件可维护性
- 提高可维护性的方法

软件维护的概念

- 软件维护的定义
- 影响维护工作量的因素
- 软件维护的策略
- 维护成本

软件维护的定义

- 在软件运行／维护阶段对软件产品进行的修改就是所谓的维护。
- 维护的类型有四种：
  - 改正性维护
  - 适应性维护
  - 完善性维护
  - 预防性维护

改正性维护

- 在软件交付使用后，因开发时测试的不彻底、不完全，必然会有部分隐藏的错误遗留到运行阶段。
- 这些隐藏下来的错误在某些特定的使用环境下就会暴露出来。
- 为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的误使用，应当进行的诊断和改正错误的过程就叫做改正性维护。

适应性维护

- 在使用过程中，
  - 外部环境（新的硬、软件配置）
  - 数据环境（数据库、数据格式、数据输入/输出方式、数据存储介质）

  可能发生变化。
- 为使软件适应这种变化，而去修改软件的过程就叫做适应性维护。

完善性维护

- 在软件的使用过程中，用户往往会对软件提出新的功能与性能要求。
- 为了满足这些要求，需要修改或再开发软件，以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性。
- 这种情况下进行的维护活动叫做完善性维护。

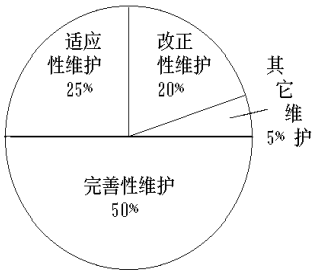
- 实践表明，在几种维护活动中，完善性维护所占的比重最大。**即大部分维护工作是改变和加强软件，而不是纠错**。
- 完善性维护不一定是救火式的紧急维修，而可以**是有计划、有预谋的一种再开发活动**。
- 事实证明，来自用户要求扩充、加强软件功能、性能的维护活动约占整个维护工作的50%。

预防性维护

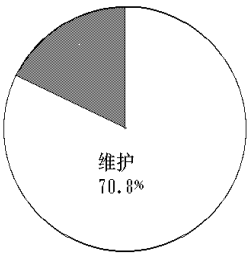
- 预防性维护是为了**提高软件的可维护性、可靠性等**，为以后进一步改进软件打下良好基础。
- 预防性维护定义为：**采用先进的软件工程方法对需要维护的软件或软件中的某一部分（重新）进行设计、编制和测试。**

- 在整个软件维护阶段所花费的全部工作量中，**完善性维护占了几乎一半的工作量。**
- **软件维护活动所花费的工作占整个生存期工作量的70%以上**，这是由于在漫长的软件运行过程中需要不断对软件进行修改，以**改正新发现的错误**、适应新的环境和用户新的要求，这些修改需要花费很多精力和时间，而且有时会引入新的错误。

三类维护占总维护比例



维护在软件生存期所占比例



影响维护工作量的因素

- **在软件的维护过程中，需要花费大量的工作量，从而直接影响了软件维护的成本。**
- **应当考虑有哪些因素影响软件维护的工作量，相应应该采取什么维护策略，才能**有效地维护软件**并**控制维护的成本**。**

- **系统大小**：系统越大，理解掌握起来越困难。系统越大，所执行功能越复杂。因而需要更多的维护工作量。
- **程序设计语言**：使用强功能的程序设计语言可以控制程序的规模。语言的功能越强，生成程序的模块化和结构化程度越高，所需的指令数就越少，程序的可读性越好。

- **系统年龄：**
  - 老系统随着不断的修改，结构越来越乱；
  - 维护人员经常更换，程序又变得越来越难于理解。
  - 许多老系统在当初并未按照软件工程的要求进行开发，因而没有文档，或文档太少。
  - 在长期的维护过程中文档在许多地方与程序实现变得不一致，在维护时就会遇到很大困难。

- **数据库技术的应用**：使用数据库，可以简单而有效地管理和存储用户程序中的数据，还可以减少生成用户报表应用软件的维护工作量。
- **先进的软件开发技术**：在软件开发时，若使用能使软件结构比较稳定的分析与设计技术，及程序设计技术，如面向对象技术、复用技术等，可减少大量的工作量。

- **其它：**
  - 应用的类型
  - 数学模型
  - 任务的难度
  - 开关与标记、IF嵌套深度、索引或下标数等

  对维护工作量都有影响。
- **许多软件在开发时并未考虑将来的修改，为软件的维护带来许多问题。**

软件维护的策略

- **改正性维护**

  通常要生成100％可靠的软件并不一定合算，成本太高。但通过使用新技术，可大大减少进行改正性维护的需要。

  这些技术包括：**数据库管理系统、软件开发环境、程序自动生成系统、较高级(第四代)的语言。**以及**新的开发方法、软件复用、防错程序设计**及**周期性维护审查**等。

- **适应性维护**

  这一类维护不可避免，可以控制。

  (1) 在配置管理时，把硬件、操作系统和其它相关环境因素的可能变化考虑在内。

  (2) 把与硬件、操作系统，以及其它外围设备有关的程序归到特定的程序模块中。

  (3) 使用内部程序列表、外部文件，以及处理的例行程序包，可为维护时修改程序提供方便。

- **完善性维护**

  利用前两类维护中列举的方法，也可以减少这一类维护。特别是数据库管理系统、程序生成器、应用软件包，可减少维护工作量。此外，建立软件系统的原型，把它在实际系统开发之前提供给用户。用户通过研究原型，进一步完善他们的功能要求，就可以减少以后完善性维护的需要。

维护成本

- 有形的软件维护成本是花费了多少钱，无形的维护成本有更大的影响。
  - 一些合理的修复或修改请求不能及时安排，使得客户不满意；
  - 变更的结果引入新的故障，使得软件整体质量下降；
  - 把软件人员抽调到维护工作中，干扰了软件开发工作。

- 软件维护的代价是降低了生产率，在做老程序的维护时非常明显。
- 例如，开发每一行源代码耗资25美元，维护每一行源代码需要耗资1000美元。
- 维护工作量包括生产性活动（如分析和评价、设计修改和实现）和“轮转”活动（如力图理解代码在做什么、试图判明数据结构、接口特性、性能界限等）。

维护工作量的模型

$$M = p + Ke^{c-d}$$

- M是维护中消耗的总工作量
- p是上面描述的生产性工作量
- K是一个经验常数
- c是因缺乏好的设计和文档而导致复杂性的度量
- d是对软件熟悉程度的度量。

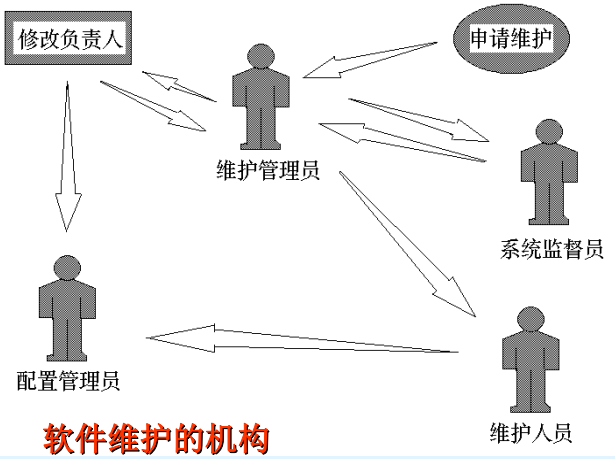
- 模型指明，如果使用了不好的软件开发方法（未按软件工程要求做），原来参加开发的人员或小组不能参加维护，则工作量（及成本）将按指数级增加。

软件维护活动

- 为了有效地进行软件维护，应事先就开始做组织工作。
  - 首先建立维护的机构
  - 申明提出维护申请报告的过程及评价的过程
  - 为每一个维护申请规定标准的处理步骤
  - 建立维护活动的登记制度以及规定评价和评审的标准。

维护机构

- 除了较大的软件开发公司外，通常在软件维护工作方面，并不保持一个正式的组织机构。
- 虽然不要求建立一个正式的维护机构，但是在开发部门确立一个非正式的维护机构则是非常必要的。



软件维护的机构

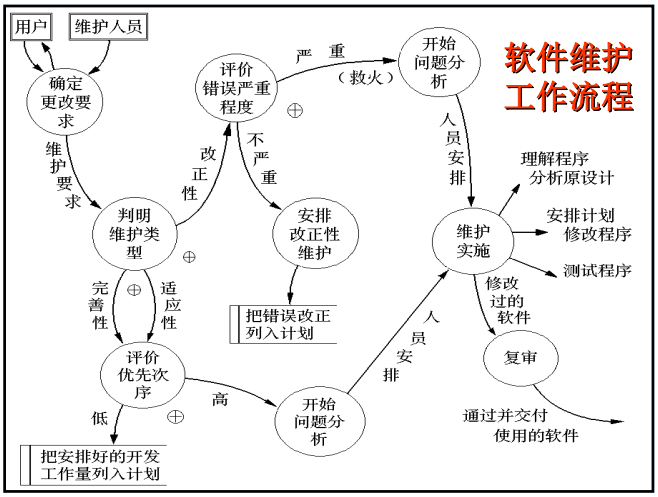
- 维护申请提交给维护管理员，他把申请交给某个系统监督员去评价。
- 一旦做出评价，由修改负责人确定如何进行修改。
- 在修改程序的过程中，由配置管理员严格把关，控制修改的范围，对软件配置进行审计。
- 在维护之前，就把责任明确下来，可以减少维护过程中的混乱。

软件维护申请报告

- 维护申请报告或称软件问题报告，由申请维护的用户填写。
- 用户必须完整地说明产生错误的情况，包括输入数据、错误清单以及其它有关材料。
- 如果申请的是适应性维护或完善性维护，用户必须提出一份修改说明书，列出所有希望的修改。

- 维护申请报告将由维护管理员和系统监督员来研究处理。
- 他们应相应地做出软件修改报告，指明：
  - 所需修改变动的性质；
  - 申请修改的优先级；
  - 为满足某个维护申请报告，所需的工作量；
  - 预计修改后的状况.

- 软件修改报告应提交修改负责人，经批准后才能开始进一步安排维护工作。



软件维护工作流程

- 尽管维护申请的类型不同，但都要进行同样的技术工作。
  - **修改软件需求说明**
  - **修改软件设计**
  - **设计评审**
  - **对源程序做必要的修改**
  - **单元测试**
  - **集成测试（回归测试）**
  - **确认测试**
  - **软件配置评审等**。

在每次软件维护任务完成后进行情况评审，对以下问题做一总结：

(1) 在目前情况下，设计、编码、测试中的哪一方面可以改进?

(2) 哪些维护资源应该有但没有?

(3) 工作中主要的或次要的障碍是什么?

(4) 从维护申请的类型来看是否应当有预防性维护?

情况评审对将来的维护工作如何进行会产生重要的影响。

维护档案记录

- 程序名称
- 源程序语句条数
- 机器代码指令条数
- 所用的程序设计语言
- 程序安装的日期
- 程序安装后的运行次数
- 与程序安装后运行次数有关的处理故障次数

- 程序改变的层次及名称
- 修改程序增加的源程序语句条数
- 修改程序减少的源程序语句条数
- 每次修改所付出的“人时”数
- 修改程序的日期
- 软件维护人员的姓名
- 维护申请报告的名称、维护类型
- 维护开始时间和维护结束时间、
- 花费在维护上的累计“人时”数
- 维护工作的净收益等。

维护评价

- **评价维护活动比较困难**，因为缺乏可靠的数据。
- 如果维护的档案记录做得比较好，可以得出一些维护“性能”方面的度量值。
  - **每次程序运行时的平均出错次数；**
  - **花费在每类维护上的总“人时”数；**
  - **每个程序、每种语言、每种维护类型的程序平均修改次数；**
  - **因为维护，增加或删除每个源程序语句所花费的平均“人时”数；**
  - **用于每种语言的平均“人时”数；**
  - **维护申请报告的平均处理时间；**
  - **各类维护申请的百分比。**

据此可对开发技术、语言选择、维护工作计划、资源分配、以及其它许多方面做出判定。

程序修改的步骤及修改的副作用

- 分析和理解程序
- 修改程序
- 重新验证程序

分析和理解程序

- 理解程序的功能和目标；
- 掌握程序的结构信息，即从程序中细分出若干结构成分。如程序系统结构、控制结构、数据结构和输入／输出结构等；
- 了解数据流信息，即涉及到的数据来源何处，在哪里被使用
- 了解控制流信息，即执行每条路径的结果；
- 理解程序的操作（使用）要求；

修改程序

1. 设计程序的修改计划

程序的修改计划要考虑人员和资源的安排。小的修改可以不需要详细的计划，而对于需要耗时数月的修改，就需要计划立案。

2. 修改代码，以适应变化
3. 修改程序的副作用

所谓副作用是指因修改软件而造成的错误或其它不希望发生的情况。副作用有三种：修改代码的副作用、修改数据的副作用、文档的副作用。

重新验证程序

- 在将修改后的程序提交用户之前，需要进行充分的确认和测试，以保证整个修改后程序的正确性。
- 静态确认
  修改软件，伴随着引起新的错误的危险。为了能够做出正确的判断，验证修改后的程序至少需要两个人参加。要检查：

- 计算机确认
  在进行了以上确认的基础上，用计算机对修改程序进行确认测试：
  (1) 确认测试顺序：先对修改部分进行测试，然后隔离修改部分，测试程序的未修改部分，最后再把它们集成起来进行测试。这种测试称为回归测试。
  (2) 准备标准的测试用例。
  (3) 充分利用软件工具帮助重新验证过程。

(4) 在重新确认过程中，需邀请用户参加。

- 维护后的验收——在交付新软件之前，维护主管部门要检验：
  (1) 全部文档是否完备，并已更新；
  (2) 所有测试用例和测试结果已经正确记载；
  (3) 记录软件配置所有副本的工作已经完成；
  (4) 维护工序和责任已经确定。

软件可维护性

软件可维护性的定义

- **软件可维护性是指**纠正软件系统出现的错误和缺陷，以及为满足新的要求进行修改、扩充或压缩的容易程度。
- **可维护性、可使用性、可靠性是衡量软件质量的主要质量特性。**
- **软件的可维护性是**软件开发阶段各个时期的关键目标。

- 目前广泛使用的是用如下的七个特性来衡量程序的可维护性。
  - **可理解性**　**可使用性**
  - **可测试性**　**可移植性**
  - **可修改性**　**效率**
  - **可靠性**
- 而且对于不同类型的维护，这七种特性的侧重点也不相同。

在各类维护中的侧重点

	改正性维护	适应性维护	完善性维护
可理解性	o		
可测试性	o		
可修改性	o	o	
可 靠 性	o		
可移植性		o	
可使用性		o	o
效 率			o

可维护性的度量

- 人们一直期望**对软件的可维护性做出定量度量**，但要做到这一点并不容易。
- 常用的度量一个可维护的程序的七种特性的方法。就是
  - **质量检查表**
  - **质量测试**
  - **质量标准**

- **质量检查表**是用于测试程序中某些质量特性是否存在的一个问题清单。
- 评价者针对检查表上的每一个问题，依据自己的定性判断，回答“Yes”或者“No”。
- **质量测试**与**质量标准**则用于定量分析和评价程序的质量。
- 由于许多质量特性是相互抵触的，要**考虑几种不同的度量标准**，相应地去度量不同的质量特性。

1. 可理解性

- 可理解性表明人们通过阅读源代码和相关文档，了解程序功能及其如何运行的容易程度。
- 一个可理解的程序应具备以下一些特性：模块化，风格一致性，不使用令人捉摸不定或含糊不清的代码，使用有意义的数据名和过程名，结构化，完整性等。

2. 可靠性

- 可靠性表明一个程序按照用户的要求和设计目标，在给定的一段时间内正确执行的概率。
- 关于可靠性，度量的标准主要有：
  - 平均失效间隔时间MTTF
  - 平均修复时间MTTR
  - 有效性A = MTBD/(MTBD+MDT)

度量可靠性的方法

- 根据程序错误统计数字，进行可靠性预测。常用方法是利用一些可靠性模型，根据程序测试时发现并排除的错误数预测平均失效间隔时间MTTF。
- 根据程序复杂性，预测软件可靠性。用程序复杂性预测可靠性，前提条件是可靠性与复杂性有关。因此可用复杂性预测出错率。程序复杂性度量标准可用于预测哪些模块最可能发生错误，以及可能出现的错误类型。

3. 可测试性

- 可测试性表明论证程序正确性的容易程度。程序越简单，证明其正确性就越容易。而且设计合用的测试用例，取决于对程序的全面理解。
- 一个可测试的程序应当是可理解的，可靠的，简单的。
- 用于可测试性度量的检查项目如下：
  - 程序是否模块化？结构是否良好？
  - 程序是否可理解？程序是否可靠？
  - 程序是否能显示任意中间结果？
  - 程序是否能以清楚的方式描述它的输出？
  - 程序是否能及时地按照要求显示所有的输入？
  - 程序是否有跟踪及显示逻辑控制流程的能力？
  - 程序是否能从检查点再启动？
  - 程序是否能显示带说明的错误信息？

4. 可修改性

- 可修改性表明程序容易修改的程度。
- 一个可修改的程序应当是可理解的、通用的、灵活的、简单的。
- 通用性是指程序适用于各种功能变化而无需修改。
- 灵活性是指能够容易地对程序进行修改。

- 测试可修改性的一种定量方法是**修改练习**。其基本思想是**通过做几个简单的修改，来评价修改的难度**。
- **设*C*是程序中各个模块的平均复杂性，*n*是必须修改的模块数，*A* 是要修改的模块的平均复杂性。 则修改的难度*D*由下式计算：**

$$D = A / C$$

5. 可移植性

- **可移植性表明程序转移到一个新的计算环境的可能性的大小**。或者它表明程序可以容易地、有效地在各种各样的计算环境中运行的容易程度。
- 一个可移植的程序应具有**结构良好、灵活、不依赖于某一具体计算机或操作系统的性能**。
- 用于可移植性度量的检查项目如下：
  - **是否是用高级的独立于机器的语言来编写程序?**
  - **是否使用广泛使用的标准化的程序设计语言来编写程序？是否仅使用了这种语言的标准版本和特性?**
  - **程序中是否使用了标准的普遍使用的库功能和子程序?**
  - **程序中是否极少使用或根本不使用操作系统的功能?**
  - **程序在执行之前是否初始化内存?**
  - **程序在执行之前是否测定当前的输入/输出设备?**
  - **程序是否把与机器相关的语句分离了出来，集中放在了一些单独的程序模块中，并有说明文件?**
  - **程序是否结构化? 并允许在小一些的计算机上分段(覆盖)运行?**
  - **程序中是否避免了依赖于字母数字或特殊字符的内部位表示?**

6. 效率

- **效率表明一个程序能执行预定功能而又不浪费机器资源的程度**。
- 这些机器资源包括**内存容量**、**外存容量**、**通道容量**和**执行时间**。
- 用于效率度量的检查项目如下：
  - **程序是否模块化? 结构是否良好?**
  - **是否消除了无用的标号与表达式，以充分发挥编译器优化作用?**
  - **程序的编译器是否有优化功能?**
  - **是否把特殊子程序和错误处理子程序都归入了单独的模块中?**
  - **是否以快速的数学运算代替了较慢的数学运算?**
  - **是否尽可能地使用了整数运算，而不是实数运算?**
  - **是否在表达式中避免了混合数据类型的使用，消除了不必要的类型转换?**

- 程序是否避免了非标准的函数或子程序的调用?
- 在几条分支结构中，是否最有可能为“真”的分支首先得到测试?
- 在复杂的逻辑条件中，是否最有可能为“真“的表达式首先得到测试?

7. 可使用性

- 从用户观点出发，**可使用性定义为程序方便、实用、及易于使用的程度**。一个可使用的程序应是**易于使用的、能允许用户出错和改变**，并**尽可能不使用户陷入混乱状态的**程序。
- 用于可使用性度量的检查项目如下：
  - 程序是否具有自描述性?

- 程序是否能始终如一地按照用户的要求运行?
- 程序是否让用户对数据处理有一个满意的和适当的控制?
- 程序是否容易学会使用?
- 程序是否使用数据管理系统来自动地处理事务性工作和管理格式化、地址分配及存储器组织。
- 程序是否具有容错性?
- 程序是否灵活?

其它间接定量度量可维护性的方法

- 问题识别的时间；
- 因管理活动拖延的时间；
- 收集维护工具的时间；
- 分析、诊断问题的时间；
- 修改规格说明的时间；
- 具体的改错或修改的时间；
- 局部测试的时间；
- 集成或回归测试的时间；
- 维护的评审时间；

- 这些数据反映了维护全过程中**检错—纠错—验证**的周期，即**从检测出软件存在的问题开始至修正它们并经回归测试验证这段时间**。
- 可以粗略地认为，**这个周期越短，维护越容易**。

提高可维护性的方法

- 建立明确的软件质量目标和优先级
- 使用提高软件质量的技术和工具
- 进行明确的质量保证审查
- 选择可维护的程序设计语言
- 改进程序的文档

面向对象技术

- 面向对象的概念
- 面向对象的开发过程
- 面向对象分析与模型化
- 面向对象设计
- 面向对象程序的实现与测试
- Code与Yourdon面向对象分析与设计技术
- OMT方法

面向对象的概念

- 开发模式
- 什么是面向对象
- 对象
- 类
- 继承

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。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。

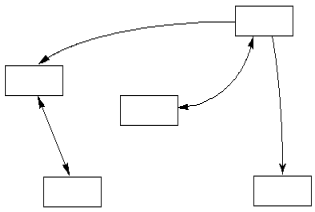
面向过程开发模式的特点

- 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。

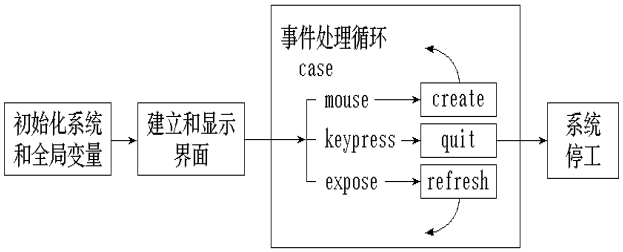
过程性系统

（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



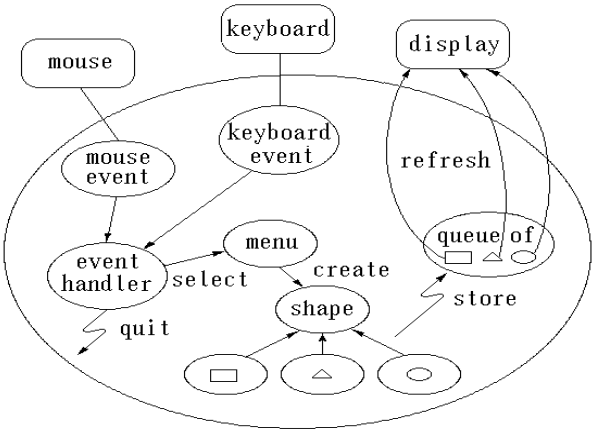
(1) Initialize system;
(2) Create and draw interface;
while QUIT not selected do
case
Mouse event:
create shape structure;
read mouse movements for data;
store newly created shape on list of shape records;
KeyPress event:
if key = 'q' then exit loop;
else ignore;
Ecpose event:
refresh display by drawing 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。

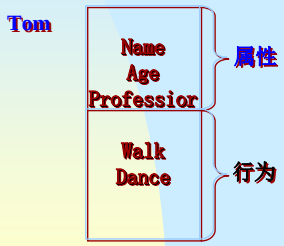
什么是面向对象

- 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。

- 如：人这个实体
  属性：姓名、年龄、职业
  行为：跑、跳



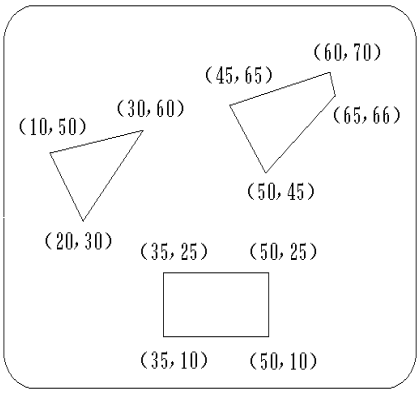
消息（Message）

- 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。

如:Mycircle是一个Circle类对象

Mycircle.Show(green)是Mycircle 对象发出的服务请求，执行在Circle类中所定义的Show操作



(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)

(b) 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。**

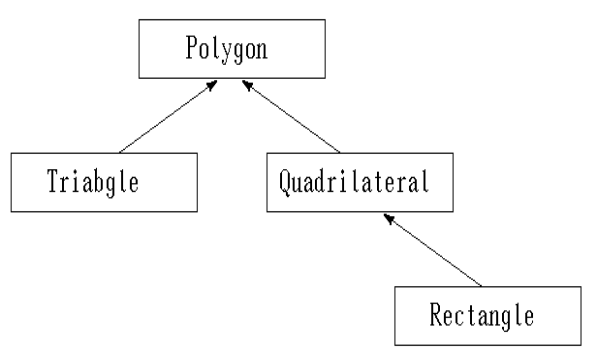
Polygon	Quadrilateral
referencePoint Vertices	*referencePoint* *Vertices*
draw move(Δx, Δy) contains?(aPoint)	*draw* *move(Δx,Δy)* *contains?(aPoint)*

(a) Polygon类　　(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。

多态性

- **一个操作在不同类中可以有不同的实现方式。**

如：Dance的实现在Male类和Female类中是不同的。

Male类中有Dance( )

Female类中有Dance(news)

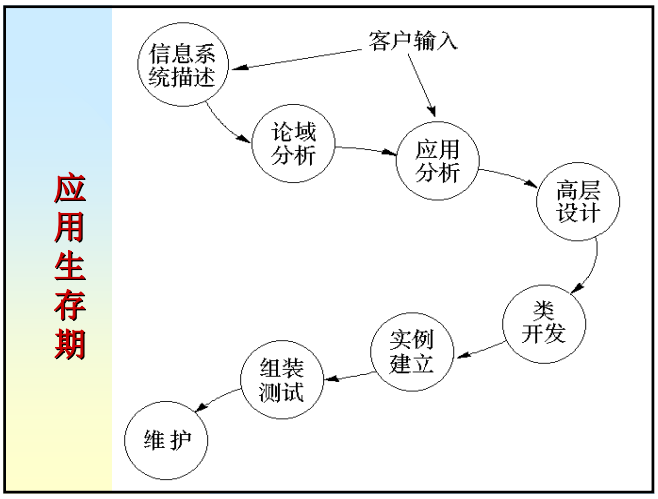
重载

- **在两个类中有相同的行为，但是如果加上作用域的指引来调用即可。**

```
如：void BaseClass::Dance( )
    void SubClass::Dance( )
    Class BaseClass
     {     void Dance( );
     }
   Class SubClass:public BaseClass
     {     void Dance( );
     }
```

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。



- 分析阶段

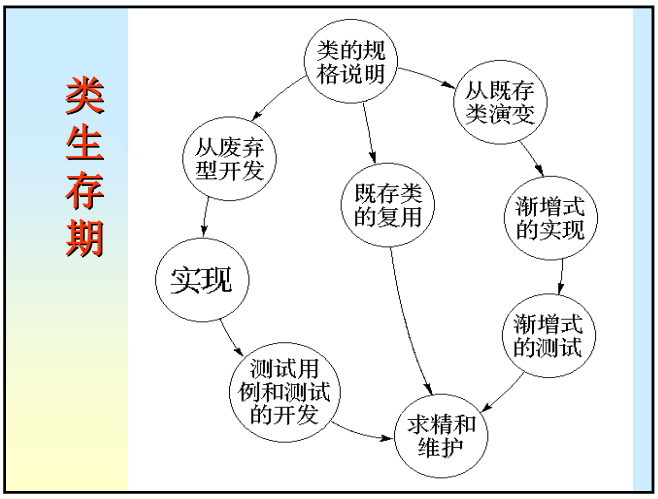
论域分析：对问题敞开思想考虑不加限制，考虑问题论域一个较宽的范围。

应用分析：针对较具体的应用，当前要解决的问题来分析。

- 高层设计：系统设计与类设计

其中系统设计应用的顶层视图或开发系统类的界面。

- 类的开发：一组类
- 实例的建立：对象的实例
- 组装测试：把系统组装成一个完整的应用来进行。
- 应用维护：



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

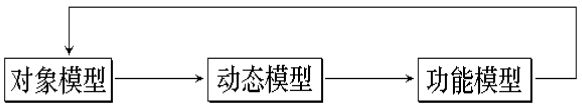
- **一般化和特殊化关系**可用来按层次渐增式地定义抽象（类）。
- 低层抽象是高层抽象的特殊化。
- 这种抽象层次构成论域模型的基础。
- 例如，小汽车，卡车和公共汽车可以归于更一般的概念汽车中。从这个较一般化的概念汽车可以定义其它较特殊的抽象：赛车，面包车和牵引车。

- **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。

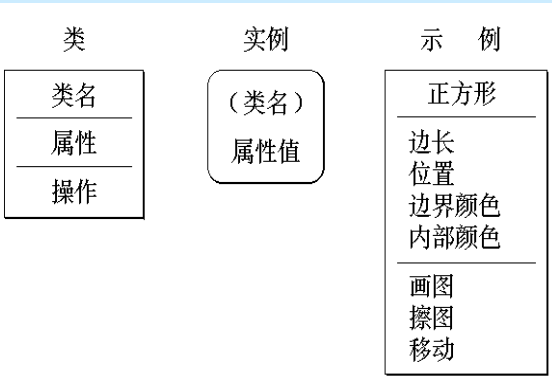


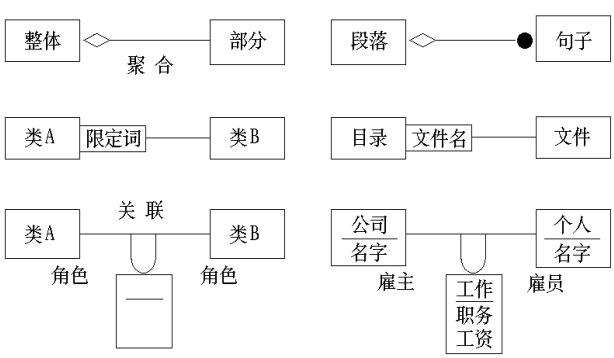
- 这个模型化的过程是一个迭代过程。

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。

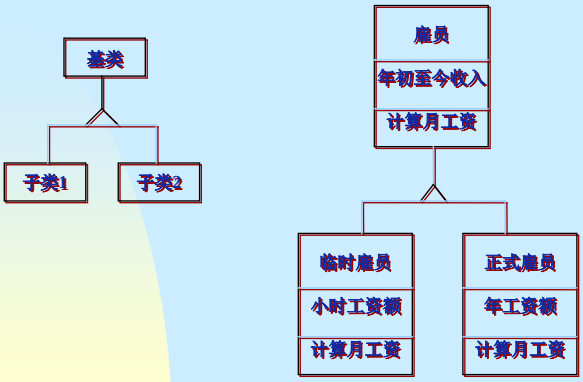
- **聚合，代表整体与部分的关系，这是一种特殊形式的关联。**
- **限定，用以对关联的含义做某种约束。**
- **角色，用来说明关联的一端。由于多数关联具有两个端点，因而涉及到两个角色。**
- **附加的说明对象之间的连接的连接属性。**





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。

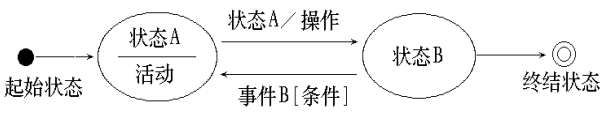


动态模型

- 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。

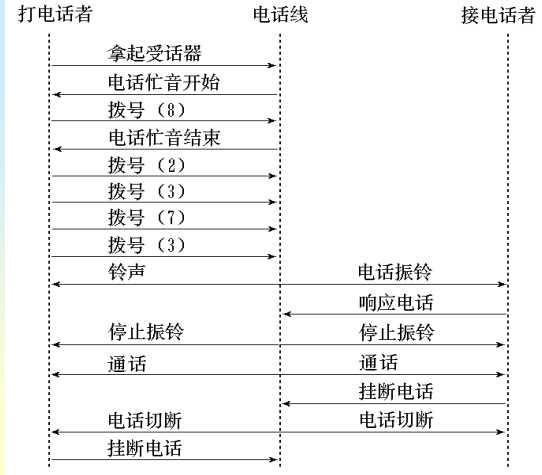
- 有些事件类可能传送的是简单的信号“要发生某件事”，而有些事件类则可能传送的是数据值。由事件传送的数据值叫做属性。
  - 列车出发(线路、班次、城市)
  - 掀下鼠标按钮(按钮、位置)
  - 拿起电话受话器
  - 数字拨号(数字)

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 .....

事件追踪图



状态图与事件追踪图的关系

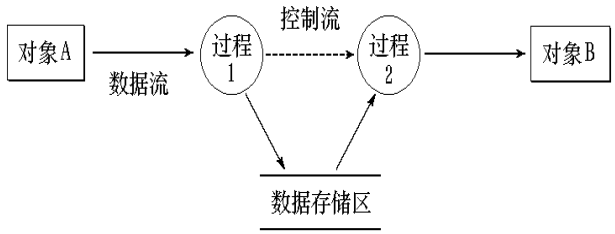
- 状态图叙述一个对象的个体行为，事件追踪图则给出多个对象所表现出来的集体行为。它们从不同侧面来说明同一系统的行为。
- 例如，一个事件追踪图指出某一对象在接受一个事件之后发出另一事件，同一行为在此对象的状态图中也应当有所表示。

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

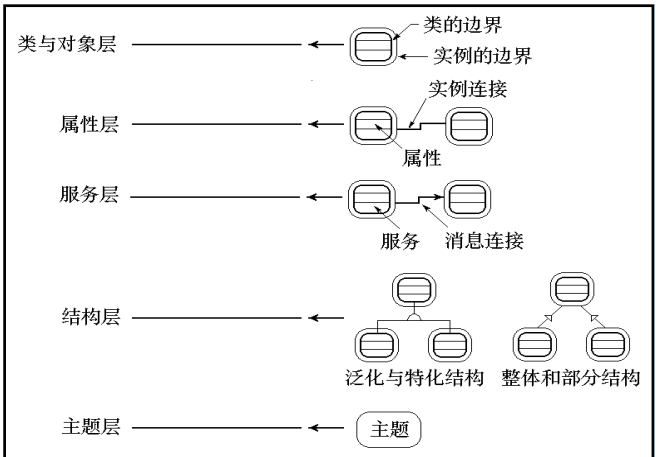
- OOA有两个任务
  - 形式地说明我们所面对的应用问题，最终成为软件系统基本构成的对象，还有系统所必须遵从的，由应用环境所决定的规则和约束。
  - 明确地规定构成系统的对象如何协同合作，完成指定的功能。

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

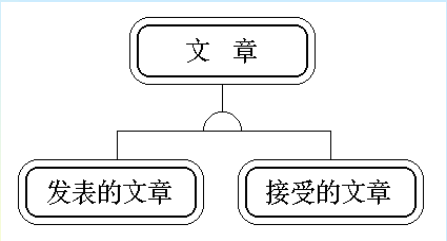
条　　目	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	注　　释
ACCEPTION SUBSCRI PTION				×						SUBSCRIPTION 的属性
ACCOMPANIED PA YMENT	×									对payment 的不同类型不加以区分
ACCOUNTING DEPAR TMENT	×									已超出SBSS 的应用论域
ACTUAL EXPIRATION DATE				×						SUBSCRIPTION 的属性
ADDITIONAL SU BSCRIPTION			×	×						SUBSCRIPTION 的可能属性，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

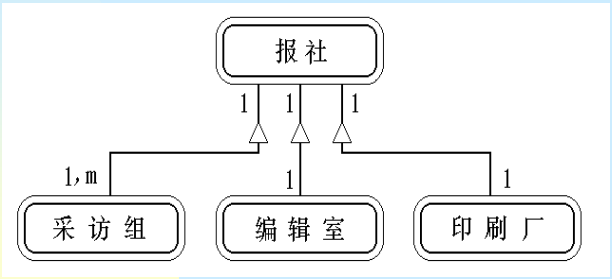
标识结构

- 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**。

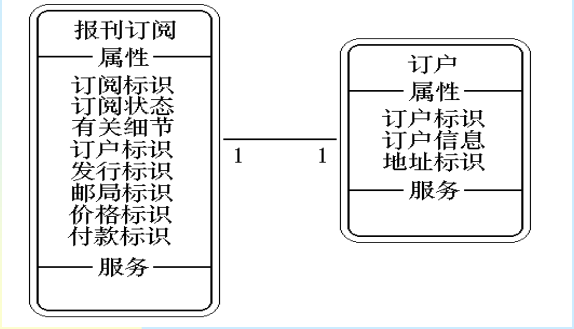
标识属性

- 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。

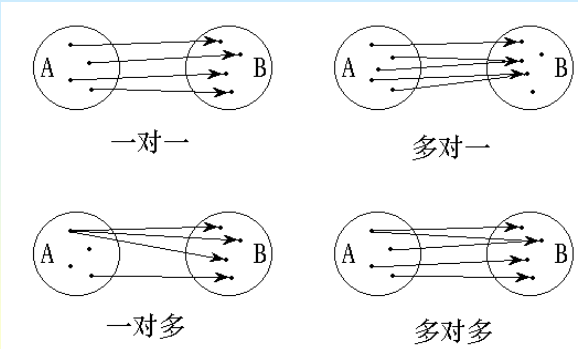
标识属性的方法和策略

- 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



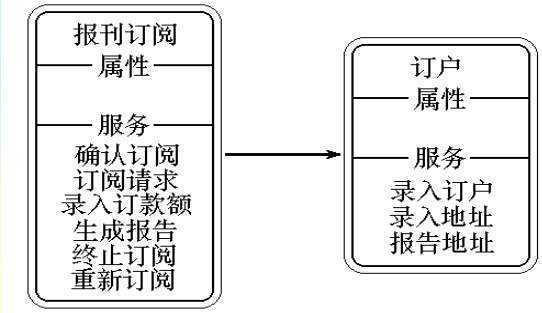
实例连接关系的标识



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

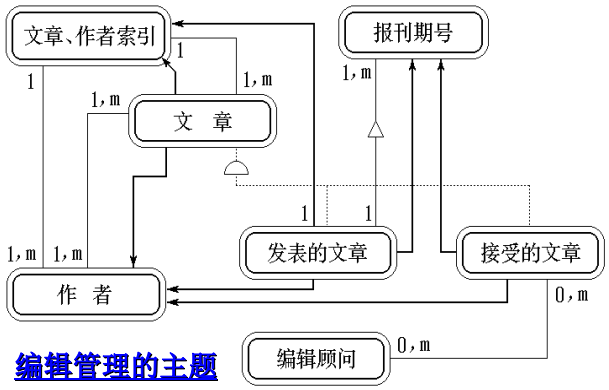
- 两个对象之间可能存在着由于通信需要而形成的关系，这称为消息连接。
- 消息连接表示从一个对象发送消息到另一个对象，由那个对象完成某些处理。它们在图中用箭头表示，方向从发消息的对象指向收消息的对象。

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

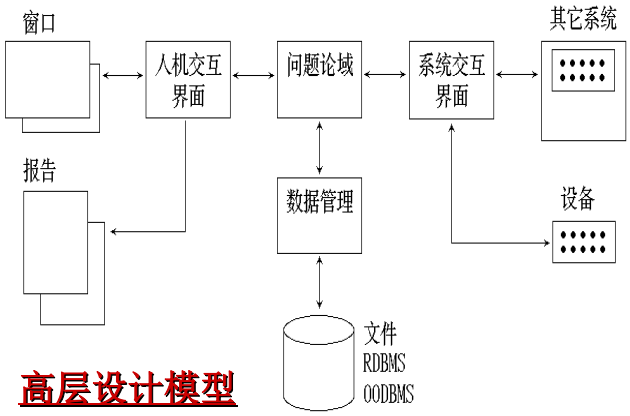
- 将每一种结构（包括整体-部分结构、和一般化-特殊化结构）中最上层的类提升成为主题；
- 将各不属于任何结构的类提升主题；
- 检查在相同或类似的问题论域中以前做面向对象分析的结果，看是否有可直接复用的主题。

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

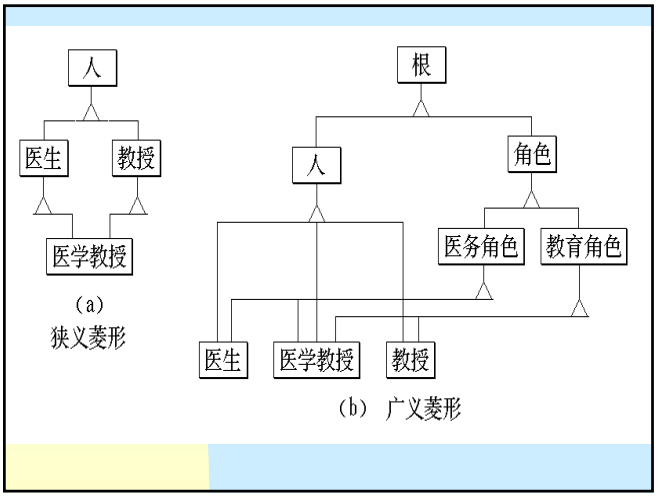
- 在设计时，从类库中引进一个根类，做为包容类，把所有与问题论域有关的类关联到一起，建立类的层次。
- 把同一问题论域的一些类集合起来，存于类库中。

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。

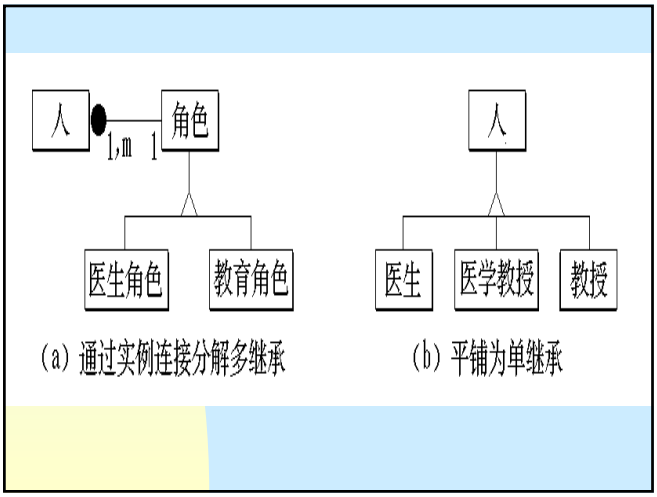
4. 调整继承支持级别

- 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。



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。

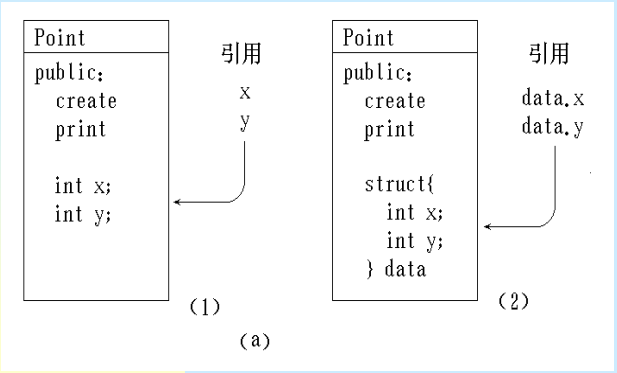
- 可复用的“插接相容性”部件
  - 部件可以在未来的应用中使用。
  - 界面的标准化
  - 类的“插接相容性”
- 可靠的部件
  - 可靠的(健壮的和正确定义的)部件。
  - 每个部件必须经过充分的测试。
  - 每个操作尽可能小和作用单一。

- 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。

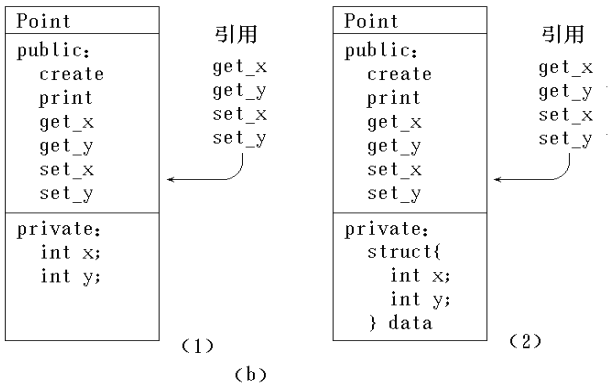
类设计的方针

- 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

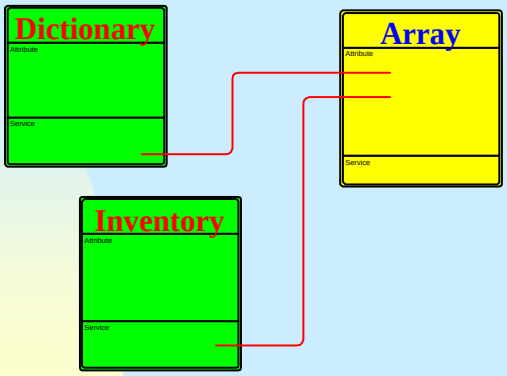


通过界面引用类中的数据



- 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

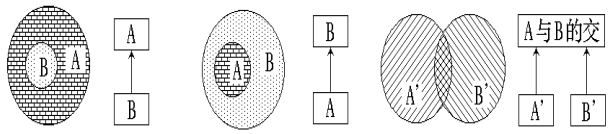
- 特定组的部件（一个小组为他们自己组内所有成员使用而开发）
- 特定项目的部件（一个小组为某一个项目而开发）
- 特定问题论域的部件（购自某一个特定论域的软件销售商）
- 一般部件（购自专门提供部件的销售商）
- 特定语言原操作（购自一个编译器的销售商）

- 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。

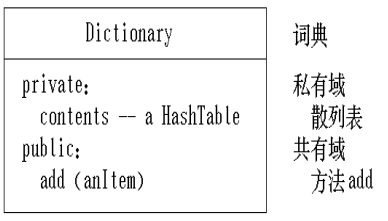
类级关系

- 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

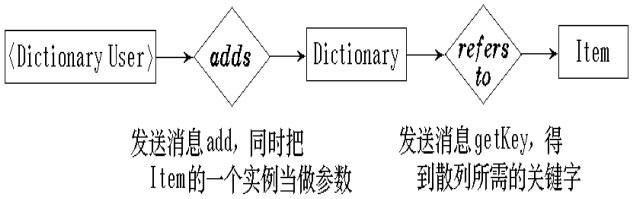
消息(messaging)

- 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。



- 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)

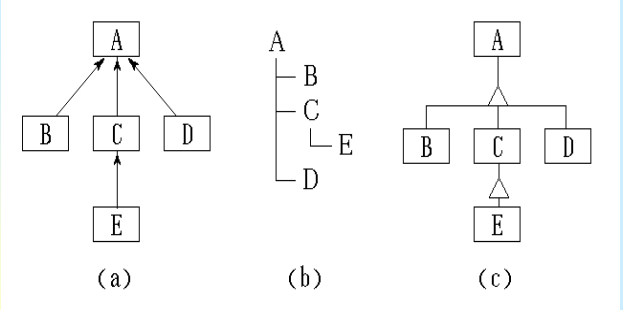
- 组装关系是一个实现级关系，它对应于应用级的聚合关系。
- 它也叫做*component*（部件）或叫做 *is part of*（是…的一部分）。
- 组装与消息两者都是类间的关系，在这种关系中，一个类的实例将是另一个类的实现的一部分。

- 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)

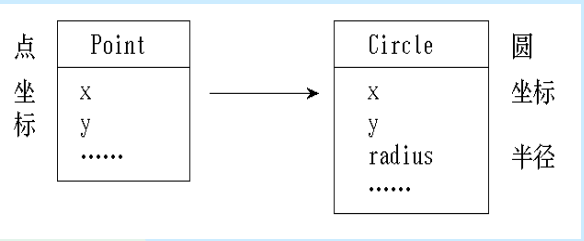
- 继承允许在既存类的基础上定义新的类。
- 一个新类B继承了既存类A，则B包括了A定义的某些行为，以及它自定义的某些附加行为。
- 有多少种面向对象程序设计语言，就有多少种不同的继承实现方式。

继承图



① 针对实现的继承

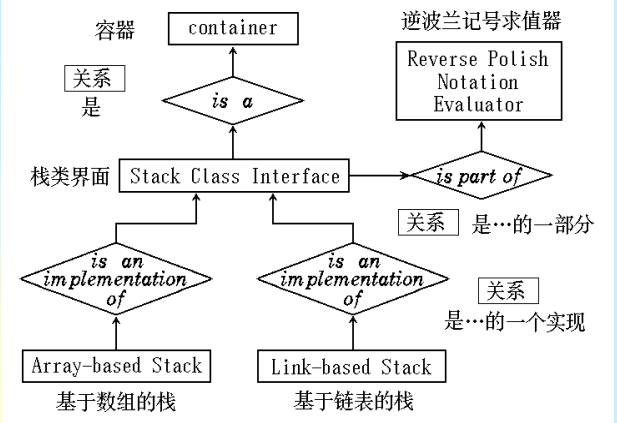
- 两个类之间“针对实现”的继承关系的建立指的是使用既存类的内部表示来做为新类的内部表示的一部分。我们不推荐这种继承方式。
- 考虑使用继承来实现一个Circle类，为了定义一个圆，需要定义一个点和一个值，做为圆的圆心和半径。因此，Point类可支持Circle类的一部分实现。把Point当做派生类。



- 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。

—“原封不动”复用

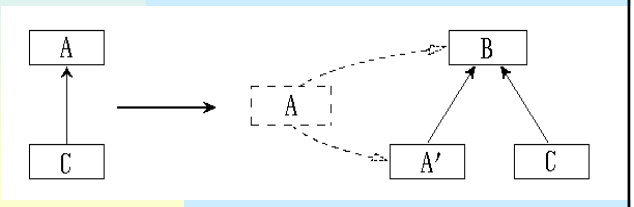
- 所需要的类已经存在，我们建立它的一个实例，用以提供所需要的特性。
- 这个实例可直接被应用利用，或者它可以用来做另一个类的实现部分。
- 通过复用一个既存类，我们可得到不加修改就能工作的已测试的代码。

—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。



—"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：

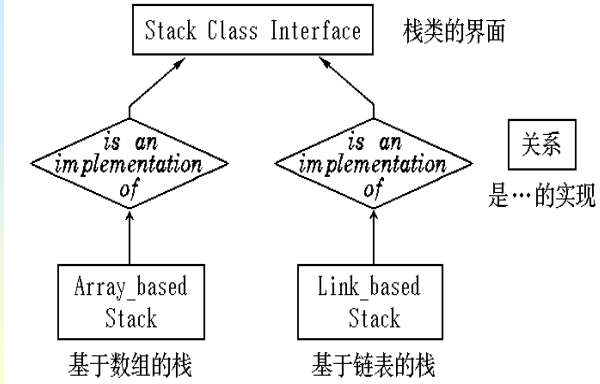
```
ERROR_HANDLER.handle
        ("Message to be printed", 1);
```

- *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。

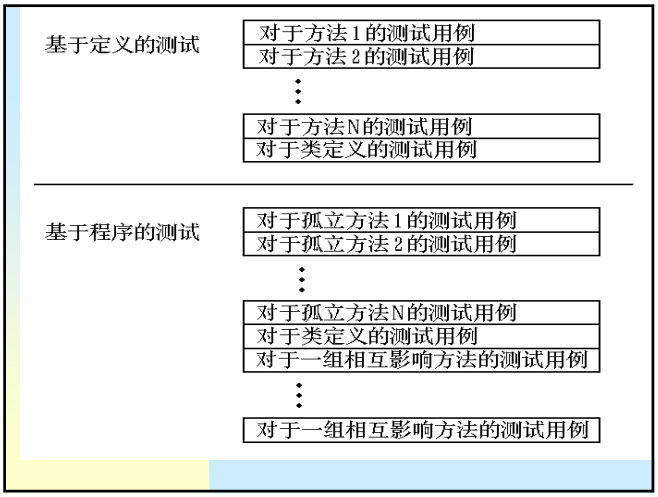
- 测试方法学检测软件中的故障并确定软件是否执行了预定要开发的功能。
- 测试过程包括了一组测试用例的开发，每一个测试用例要求能检验应用的一个特定的元素。还需要分析用各个测试用例执行测试的结果来收集有关软件的信息。

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

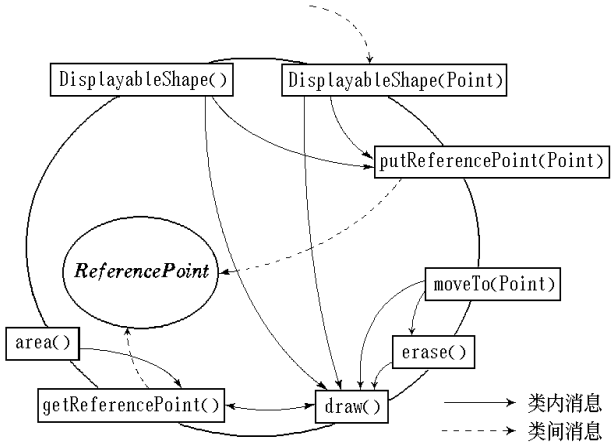
- 基于定义的测试包括两个级别：类定义和服务定义。
- 类定义
  - 一个类的定义由各个服务的定义和一些表示类的概念的语句组合而成。
  - 例，一个stack类包括了服务*push*和*pop*的定义。还表达了 LIFO 的思想。

- 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。