**软件工程**

1. 软件过程

## 软件过程的概念

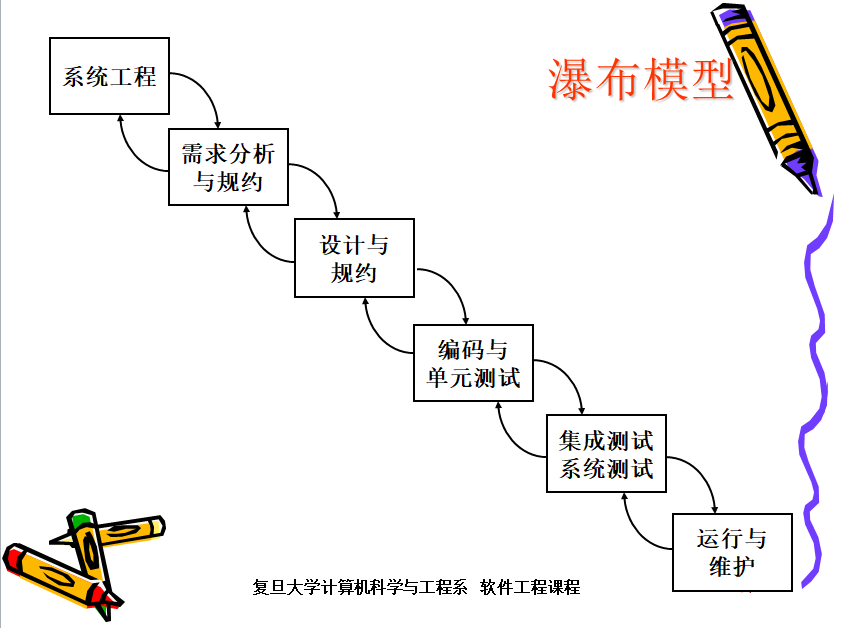
软件过程又称软件生存周期过程，定义了软件组织和人员在软件产品的定义、开发和维护等阶段所实施的一系列活动和任务。同时软件过程也描述了活动及任务的时序关系以及预期目标的途径。

软件生存周期大体可分为如下几个活动：**计算机系统工程、需求分析、设计、编码、测试、运行和维护。**

* 软件过程指软件生存周期中的一系列相关的过程。过程是活动的集合，活动是任务的集合
* 软件过程有三层含义
  + 个体含义，即指软件产品或系统在生存周期中的某一类活动的集合，如软件开发过程，软件管理过程等
  + 整体含义，即指软件产品或系统在所有上述含义下的软件过程的总体
  + 工程含义，即指解决软件过程的工程，它应用软件工程的原则、方法来构造软件过程模型，并结合软件产品的具体要求进行实例化，以及在用户环境下的运作，以此进一步提高软件生产率，降低成本

## 经典软件过程模型的特点（瀑布模型、增量模型、演化模型、统一过程模型）

**瀑布模型**



一个阶段完成后再开始下一个阶段。

特点：过于理想，假设每个环节都可以一次通过。

阶段具有顺序性和依懒性，推迟实现（编码），质量保证（文档+评审）

**演化模型**

特点：演化模型采用迭代的思想，渐进地开发，逐步完整软件产品。

演化模型是一种全局的软件（或产品）生存周期模型。属于迭代开发方法。即根据用户的基本需求，通过快速分析构造出该软件的一个初始可运行版本，这个初始的软件通常称之为原型，然后根据用户在使用原型的过程中提出的意见和建议对原型进行改进，获得原型的新版本。重复这一过程，最终可得到令用户满意的软件产品。

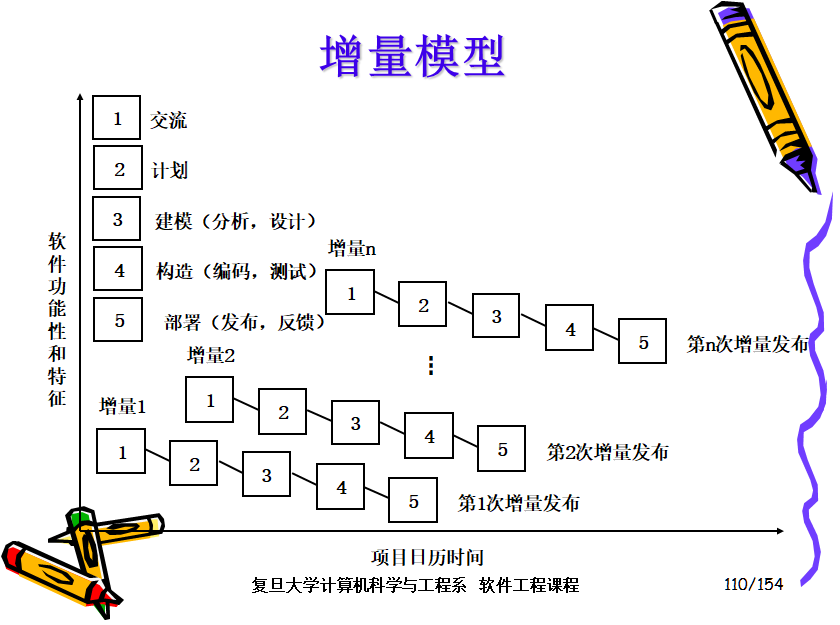
采用演化模型的开发过程，实际上就是从初始的原型逐步演化成最终软件产品的过程。演化模型特别适用于对软件需求缺乏准确认识的情况。

缺点：

如果所有的产品需求在一开始并不完全弄清楚的话，会给总体设计带来困难及削弱产品设计的完整性，并因而影响产品性能的优化及产品的可维护性。

如果缺乏严格的过程管理的话，这个生命周期模型很可能退化为一种原始的无计划的“试－错－改”模式。

**增量模型**



增量模型将产品的开发分成若干个增量进行，每个增量都执行一系列活动，且生产出一个可执行的中间产品，后一个版本是对前一版本的修改和补充，重复增量发布的过程，直至产生最终的完善产品。

特点：增量模型融合了瀑布模型的基本成分（重复地应用）和演化模型的迭代特征。

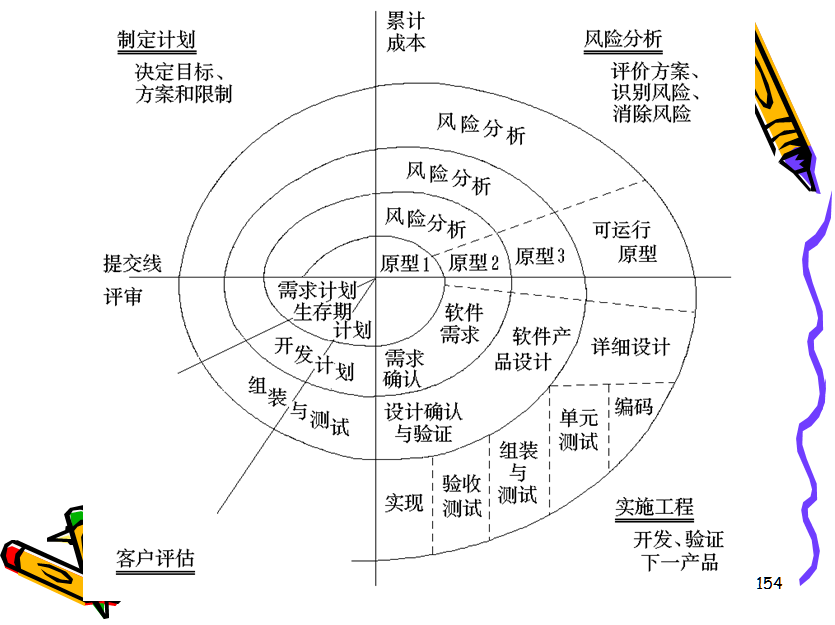
优点：较短时间像用户提交可交互系统，学习适应产品，结构设计开放

缺点：

（风险）若干构建能否合在一起;

很容易退化为边做边改模型，从而是软件过程的控制失去整体性

**螺旋模型**



螺旋模型是瀑布模型和演化模型的结合，并增加了**风险分析**。（时间风险，预算风险，行业竞争等）

特点：首先是对风险的强调。在每个迭代中都包含一个显式的风险分析过程，使得开发人员和用户对本次迭代可能出现的风险有所了解，继而可以采取相应的行为减少或者消除风险的损害。其次，螺旋模型通过把每次迭代映射到4个象限，清晰地定义了里程碑，从而有助于不同角色之间的沟通和协作。然而，过多的迭代次数会增加开发成本，延迟提交时间。

4方面：制定计划；风险分析；实施工程；客户评估；

优点：大型软件有风险控制

缺点：需要有风险评估的经验，不适合契约式开发。

**Rational统一过程模型**

特点：统一过程是一种以用例驱动、以体系结构为核心、迭代及增量的软件过程模型。由UML方法和工具支持，广泛应用于各类面向对象项目。

用例驱动

用例获取系统的功能需求，它们“驱动”需求分析之后的所有阶段的开发。

以体系结构为核心

首先定义一个基础的体系结构，然后将它原型化并加以评估，最后进行精化。

迭代

不要试图一次就定义模型或图的所有细节，开发是逐步进行的，每次迭代增加一些新的信息和细节。每次迭代都要对前次的结果评价，并用于下一次迭代的输入。迭代的过程是提供连续的反馈，这些反馈不仅改善了最终的产品，而是改善了过程本身。

在决定每次迭代应做什么时，要考虑这次迭代对系统的最大影响或最高风险。每个迭代周期都是一个小的瀑布模型。

增量

增量开发是在多次迭代的过程中每次增加一些功能(或用例)的开发，每次迭代都包含分析、设计、实现、测试等阶段。

RUP是由Rational公司开发并维护，和一系列软件开发工具紧密集成。RUP蕴含了大量优秀的实践方法，这些经验被称为“最佳实践”。

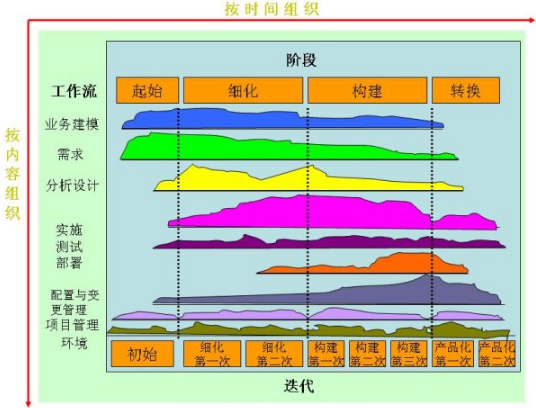
最佳实践包括：

* 迭代式软件开发、
* 需求管理、
* 基于构件的构架应用、
* 建立可视化的软件模型、
* 软件质量验证、
* 软件变更控制等。

RUP的静态结构包括6个核心工作流（业务建模、需求、分析设计、实现、测试、部署）和3个核心支持工作流（配置与变更管理、项目管理和环境）。

RUP（Rational Unified Process）的四个阶段

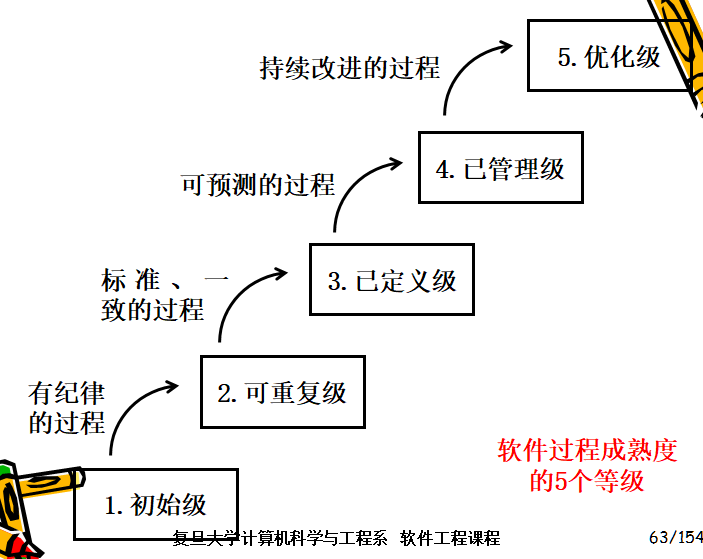
* 初始阶段：大体上的构想，业务案例，范围，和模糊评估。定义系统的业务模型，确定系统的范围。完成后建立目标里程碑。
* 细化阶段：已精化的构想、核心架构的迭代实现、高风险的解决、确定大多数需求和范围以及进行更为实际的评估。建立结构里程碑。
* 构造阶段：对遗留下的风险较低和比较简单的元素进行迭代实现，准备部署。构造产品，并继续演进需求、体系结构和计划，直到产品完成。
* 移交阶段：进行系统部署，系统测试，最终移交给用户。建立发布里程碑。



## 过程评估与CMM/CMMI的基本概念

**CMM的基本概念**

CMM（Capability Maturity Model）即能力成熟度模型，是美国卡耐基梅隆大学软件工程研究所（SEI）在美国国防部资助下于二十世纪八十年代末建立的，用于评价软件机构的软件过程能力成熟度的模型。此模型在建立和发展之初，主要目的在于提供一种评价软件承接方能力的方法，为大型软件项目的招投标活动提供一种全面而客观的评审依据。而发展到后来，又同时被软件组织用于改进其软件过程。



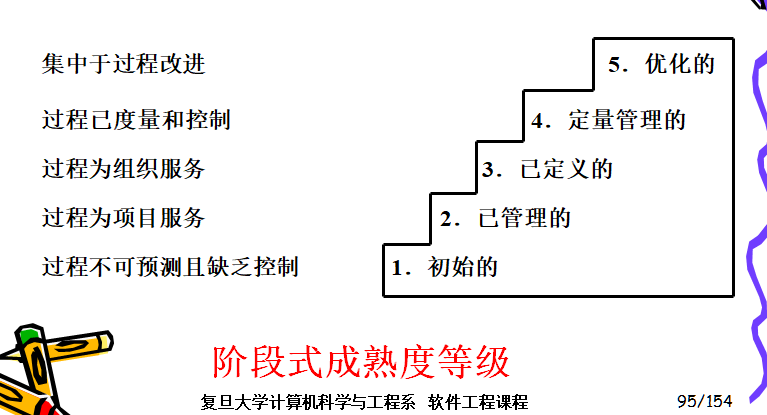
**CMMI的基本概念**

美国国防部、美国国防工业委员会和SEI/CMU于1998年启动CMMI项目，希望CMMI是若干过程模型的综合和改进，是支持多个工程学科和领域的系统的、一致的过程改进框架，能适应现代工程的特点和需要，能提高过程的质量和工作效率。

CMMI模型为每个学科的组合都提供两种表示法：阶段式模型和连续式模型

阶段式模型

阶段式模型的结构类同于软件CMM，它关注组织的成熟度，其成熟度等级如下图所示



## 敏捷宣言与敏捷过程的特点 敏捷宣言

1. 个体和交互 胜过 过程和工具
2. 可以工作的软件 胜过 面面俱到的文档
3. 客户合作 胜过 合同谈判
4. 响应变化 胜过 遵循计划

个人和交互高于过程和工具

不是否定过程和工具的重要性，而是更强调软件开发中人的作用和交流的作用。

可运行软件高于详尽的文档

通过执行一个可运行的软件来了解软件做了什么，远比阅读厚厚的文档要容易得多。

与客户协作高于合同（契约）谈判

要想通过合同谈判的方式，将需求固定下来常常是困难的。敏捷软件开发强调与客户的协作，通过与客户的交流和紧密合作来发现用户的需求。

对变更及时做出反应高于遵循计划

随着项目的进展，需求、业务环境、技术等都可能变化，任务的优先顺序和起止日期也可能因种种原因会改变。因此，项目计划应具有可塑性，有变动的余地。当出现变化时及时做出反应，修订计划以适应变化。

敏捷过程的特点（来自《复旦大学软件工程ppt》）

（1）最优先的是通过尽早地和不断地提交有价值的软件使客户满意

（2）欢迎变化的需求，即使该变化出现在开发的后期，为了提升对客户的竞争优势，Agile过程利用变化作为动力

（3）以几周到几个月为周期，尽快、不断地发布可运行软件

（4）在整个项目过程中，业务人员和开发人员必须天天一起工作

（5）以积极向上的员工为中心建立项目组，给予他们所需的环境和支持，对他们的工作予以充分的信任

（6）项目组内效率最高、最有效的信息传递方式是面对面的交流

（7）测量项目进展的首要依据是可运行的软件

（8）敏捷过程提倡可持续的开发，项目发起者、开发者和用户应能长期保持恒定的速度

（9） 应时刻关注技术上的精益求精和好的设计，以增强敏捷性

（10）简单化是必不可少的，这是尽可能减少不必要工作的艺术

（11）最好的构架、需求和设计出自于自我组织的团队

（12）团队要定期反思怎样才能更有效，并据此调整自己的行为

1. 软件需求

## 软件需求的概念

软件需求是对软件产品或服务所需要具备的外部属性的一种刻画，这些属性应当保证所提供的解决方案能满足用户所需要解决的显示世界问题的需要。

软件需求一般包括三类：

功能性需求： 实现功能；

质量需求： 要求质量属性，如系统性能、可靠性、稳定性等；

约束： 开发语言，系统运行上下文环境等；

## 需求工程的基本过程

需求获取：调查研究

需求提炼：分析建模

需求描述：编写软件需求说明书

需求验证

## 分层数据流模型

它是将提供给用户的业务流程图(“物理模型”)进行功能建模，转化成开发人员能够理解的一系列“逻辑模型”图，即以图形化的方法描绘数据在系统中的流动和处理的过程，这些图都应该用规范的DFD描述。

分层数据流图的设计方法

第一步，画子系统的输入输出。把整个系统视为一个大的加工，然后根据数据系统从哪些外部实体接收数据流，以及系统发送数据流到那些外部实体，就可以画出输入输出图。这张图称为顶层图。

第二步，画子系统的内部。把顶层图的加工分解成若干个加工，并用数据流将这些加工连接起来，使得顶层图的输入数据经过若干加工处理后，变成顶层图的输出数据流。这张图称为0层图。从一个加工画出一张数据流图的过程就是对加工的分解。 可以用下述方法来确定加工：在数据流的组成或值发生变化的地方应该画出一个加工，这个加工的功能就是实现这一变化，也可以根据系统的功能决定加工。 确定数据流的方法 用户把若干数据当作一个单位来处理（这些数据一起到达、一起处理）时，可以把这些数据看成一个数据流。 关于数据存储 对于一些以后某个时间要使用的数据，可以组织成为一个数据存储来表示。

第三步，画加工的内部。把每个加工看作一个小系统，把加工的输入输出数据流看成小系统的输入输出流。于是可以象画0层图一样画出每个小系统的加工的DFD图。

第四步，画子加工的分解图。对第三步分解出来的DFD图中的每个加工，重复第三步的分解过程，直到图中尚未分解的加工都是足够简单的（即不可再分解）。至此，得到了一套分层数据流图。 第五步，对数据流图和加工编号

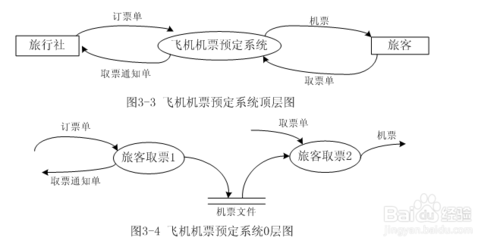
对于一个软件系统，其数据流图可能有许多层，每一层又有许多张图。为了区分不同的加工和不同的DFD子图，应该对每张图进行编号，以便于管理。

● 顶层图只有一张，图中的加工也只有一个，所以不必为其编号。

● 0层图只有一张，图中的加工号分别是0.1、0.2、…，或者1， 2 。

● 子图就是父图中被分解的加工号。

● 子图中的加工号是由图号、圆点和序号组成，如：1.12，1.3 等等。



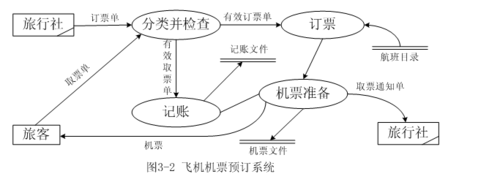
数据流图有四种基本图形符号：

：箭头，表示数据流；

〇：圆或椭圆，表示加工；

= ：双杠，表示数据存储；

□：方框，表示数据的源点或终点。



## 用例和场景建模及其UML表达（用例图、活动图、泳道图、顺序图）

**用例图**

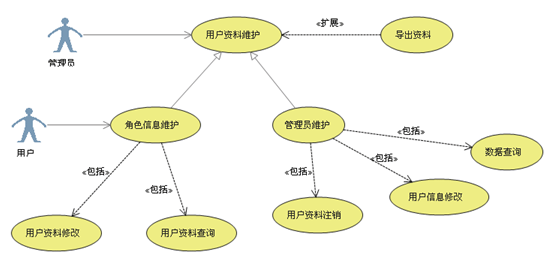
也被称为用户模型图，是从软件的需求分析的到最终实现的第一步，它是从客户角度来描述系统功能的。它包含三个基本组件：参与者(使用系统的人或事物)、用例(代表系统的某项完整的功能，在图形中使用椭圆型表示)、关系(关联、泛化、包含、扩展)。

扩展关系：如果一个功能在完成的时候，偶尔会执行另外一个功能，使用扩展关系表示。

泛化关系：表示同一个业务的不同技术实现。其实就是继承关系的一种。

包含关系：是指一个用例可以含有其他用例具有的行为。

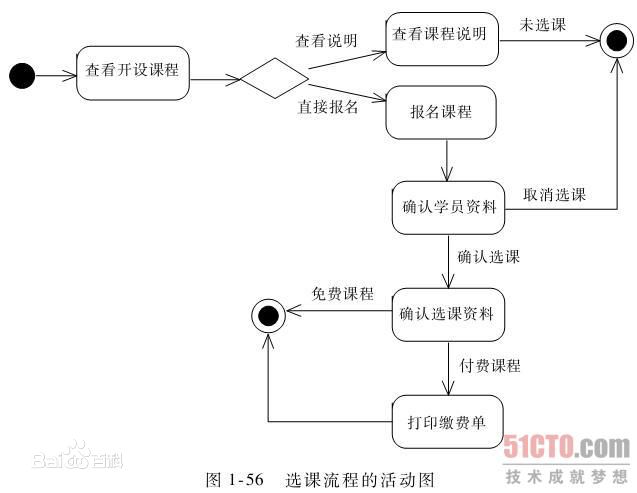




**活动图**

它用于描述系统的活动，判定点和分支等。活动中的动作状态，原子的、不可已中断的动作。并在此动作完成后向另一个动作转变。

分支与合并。分支在软件系统中很常见：用于表示对象类具有的条件行为。用一个布尔型的表达式真假来判定动作的流向，合并有两个如转换一个出转换。分支有一个如转换两个出转换。

分叉与汇合：分叉又来描述并发线程。每个分叉可以有一个输入的转换和两个或多个输出转换。汇合代表两个或多个并发控制流的同步发生。当所有流都到达汇合点后，程序才能继续前进。

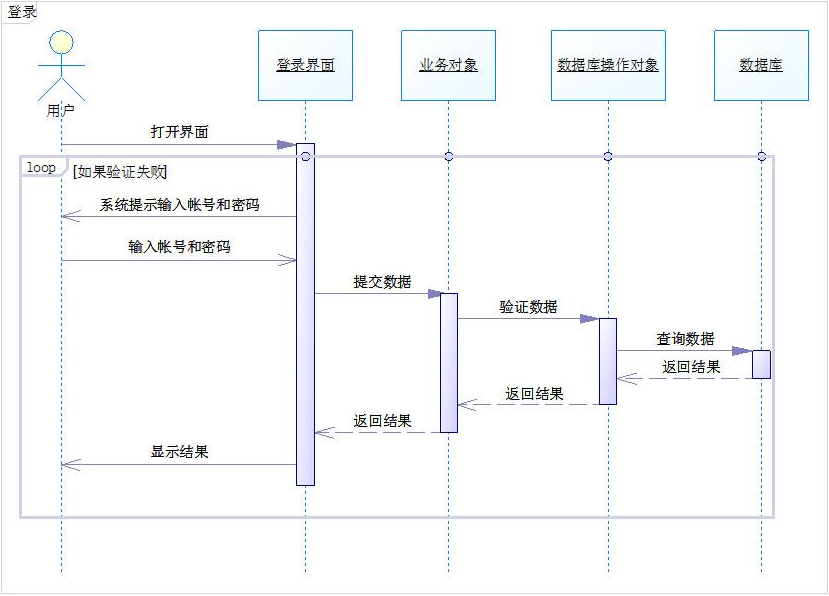
**泳道图**

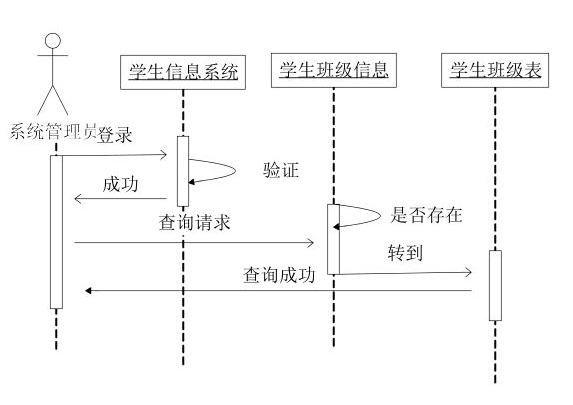
泳道图是将模型中的活动按照职责组织起来。这种分配可以通过将活动组织成用线分开的不同区域来表示。由于它们的外观的缘故，这些区域被称作泳道。它可以方便的描述企业的各种业务流程，能够直观地描述系统的各活动之间的逻辑关系，利于用户理解业务逻辑。



**顺序图**

时序图用于描述对象之间的传递信息的时间顺序。即用例中的行为顺序。当执行一个用例时，时序图中的每一条消息对应了一个类中操作或者引起转换的触发事件。纵轴表示时间时间轴向下延伸。横轴代表协作中的各个独立对象。对象存在时。消息用从一个对象的生命线到另个对象的生命线的箭头表示。箭头以时间的顺序在图中上下排列。



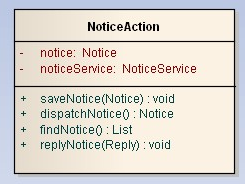


## 数据模型建模及其UML表达（类图）

类图（Class Diagram）: 类图是面向对象系统建模中最常用和最重要的图，是定义其它图的基础。类图主要是用来显示系统中的类、接口以及它们之间的静态结构和关系的一种静态模型。

类图的3个基本组件：类名、属性、方法。

（+：表示public； -：表示private； #：表示protected（friendly也归入这类））



类与类之间关系的表示方式

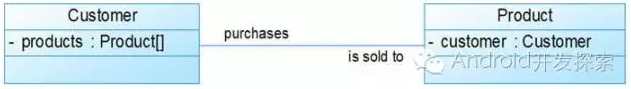
1. 关联关系（单向关联、双向关联和自关联）

（1）单向关联



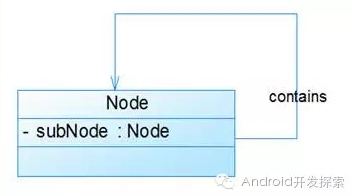
我们可以看到，在UML类图中单向关联用一个带箭头的直线表示。上图表示每个顾客都有一个地址，这通过让Customer类持有一个类型为Address的成员变量类实现。

（2）双向关联



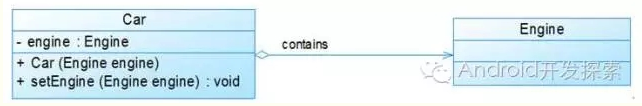
从上图中我们很容易看出，所谓的双向关联就是双方各自持有对方类型的成员变量。在UML类图中，双向关联用一个不带箭头的直线表示。上图中在Customer类中维护一个Product[]数组，表示一个顾客购买了那些产品；在Product类中维护一个Customer类型的成员变量表示这个产品被哪个顾客所购买。

（3）自关联



自关联在UML类图中用一个带有箭头且指向自身的直线表示。上图的意思就是Node类包含类型为Node的成员变量，也就是“自己包含自己”。

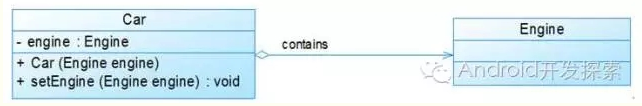
2、聚合关系



上图中的Car类与Engine类就是聚合关系（Car类中包含一个Engine类型的成员变量）。由上图我们可以看到，UML中聚合关系用带空心菱形和箭头的直线表示。聚合关系强调是“整体”包含“部分”，但是“部分”可以脱离“整体”而单独存在。比如上图中汽车包含了发动机，而发动机脱离了汽车也能单独存在。

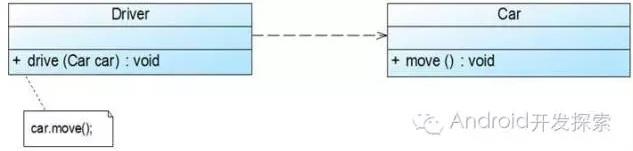
3、组合关系

组合关系与聚合关系见得最大不同在于：这里的“部分”脱离了“整体”便不复存在。比如下图：



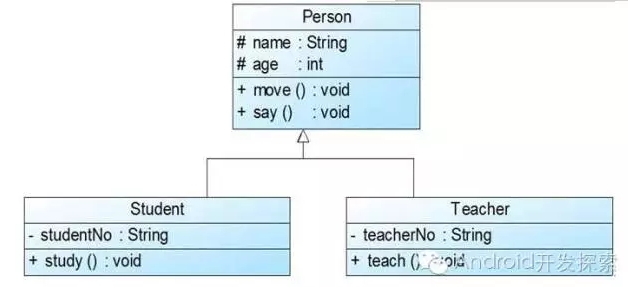
显然，嘴是头的一部分且不能脱离了头而单独存在。在UML类图中，组合关系用一个带实心菱形和箭头的直线表示。

4、依赖关系



从上图我们可以看到，Driver的drive方法只有传入了一个Car对象才能发挥作用，因此我们说Driver类依赖于Car类。在UML类图中，依赖关系用一条带有箭头的虚线表示。

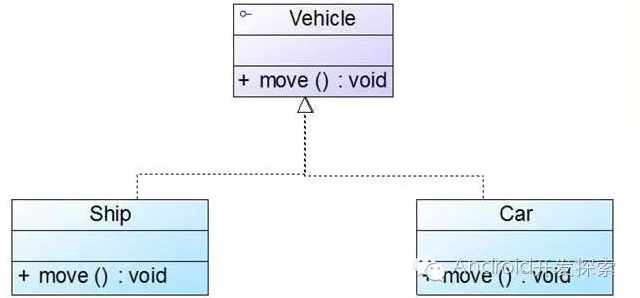
5、继承关系



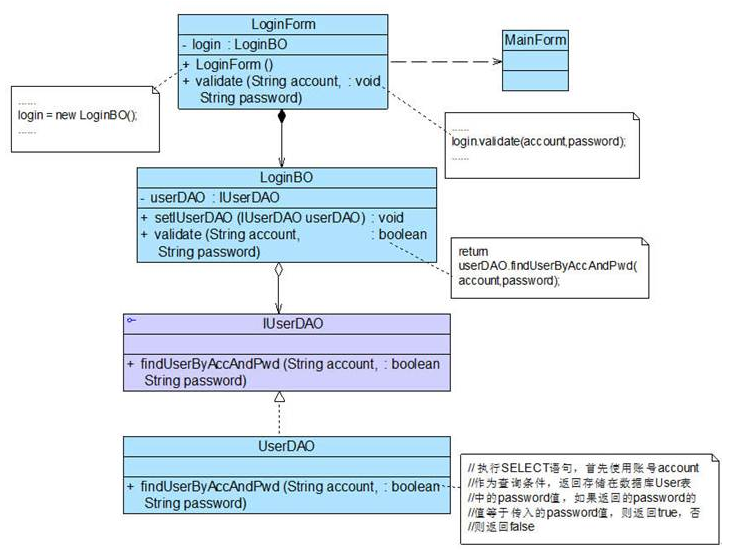
继承关系对应的是extend关键字，在UML类图中用带空心三角形的直线表示，如下图所示中，Student类与Teacher类继承了Person类。

6、接口实现关系

这种关系对应implement关键字，在UML类图中用带空心三角形的虚线表示。如下图中，Car类与Ship类都实现了Vehicle接口。

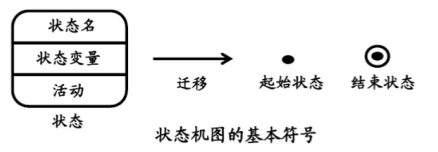


**【例】**用户通过登录界面(LoginForm)输入账号和密码，系统将输入的账号和密码与存储在数据库(User)表中的用户信息进行比较，验证用户输入是否正确，如果输入正确则进入主界面(MainForm)，否则提示“输入错误”。

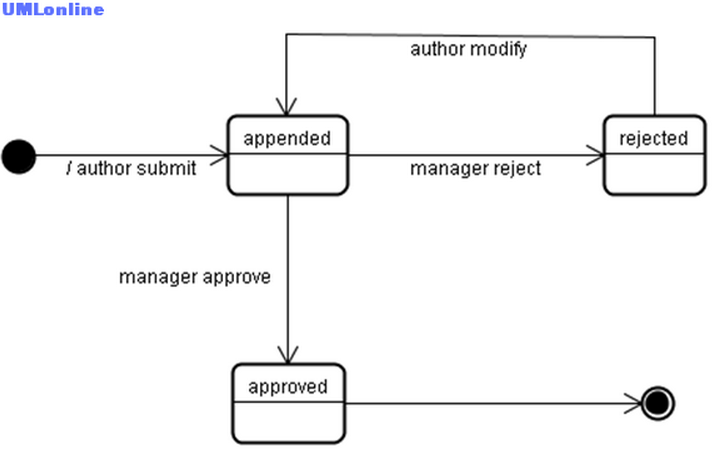


## 行为模型建模及其UML表达（状态机图）

利用状态机可以精确地描述对象的行为。从对象的初始状态起，开始响应事件并执行某些动作，这些事件引起状态的转换；对象在新状态下又开始响应事件和执行动作，如此连续进行直到终结状态。



**【例】**请假条



1. 软件设计与构造

## 软件体系结构及体系结构风格的概念

软件体系结构关注系统的一个或多个结构，包含软件构件、这些构件的对外可见的性质以及它们之间的关系

Bass提出体系结构重要的三个关键理由：

**①**方便利益相关人员的交流

②有利于系统设计的前期决策

③可传递的系统级抽象

* 常见的软件体系结构
  + 单主机结构
  + C/S（Client/Server）结构
  + B/S（Browser/Server）结构

**软件体系结构的风格：**

* 绝大多数可以被归类为相对小数量的体系结构风格之一
* 每种风格描述一种系统范畴，范畴包括：

①一些实现系统所需的功能的部件（如数据库、计算模块）；

②一组用来连接部件“通信、协调和合作”的“连接子”；

③定义部件之间怎样整合的系统约束；

④使设计者能够理解整个系统属性并分析已知属性的语义模型。

## 设计模式的概念

在许多面向对象系统中，存在一些类和对象的重复出现的模式。这些模式求解特定的设计问题，使面向对象设计更灵活，并最终可复用。这些模式帮助设计者复用以前成功的设计，设计者可以把这些模式应用到新的设计中

## 模块化设计的基本思想及概念（抽象、分解、模块化、封装、信息隐藏、功能独立）；

**抽象，**是在软件设计的规模逐渐增大的情况下，控制复杂性的基本策略，**抽象的过程**是从特殊到一般的过程，上层概念是下层概念的抽象，下层概念是上层概念的精化和细化

**逐步求精，**把问题的求解过程分解成若干步骤或阶段，每步都比上步更精化，更接近问题的解法

**模块化，**即把软件按照规定原则，划分为一个个较小的，相互独立的但又相互关联的部件，实际上是系统分解和抽象的过程

**封装（encapsulation）**是一种信息隐蔽技术，用户只能看见对象封装界面上的信息，对象的内部实现对用户是隐蔽的。封装的目的是使对象的使用者和生产者分离，使对象的定义和实现分开

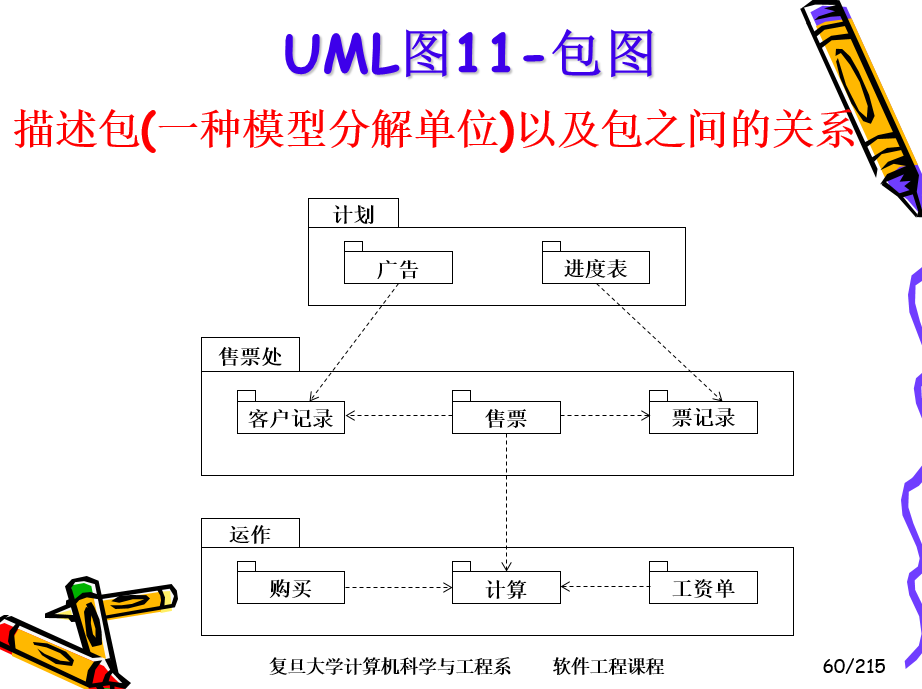
**信息隐藏**，每个模块的实现细节对于其它模块来说应该是隐蔽的，块中所包含的信息（包括数据和过程）不允许其它不需要这些信息的模块使用

**功能独立**是模块化、抽象、信息隐藏和局部化等概念的直接结果。开发功能专一的且避免与其他模块过多交互的的模块可以实现功能独立

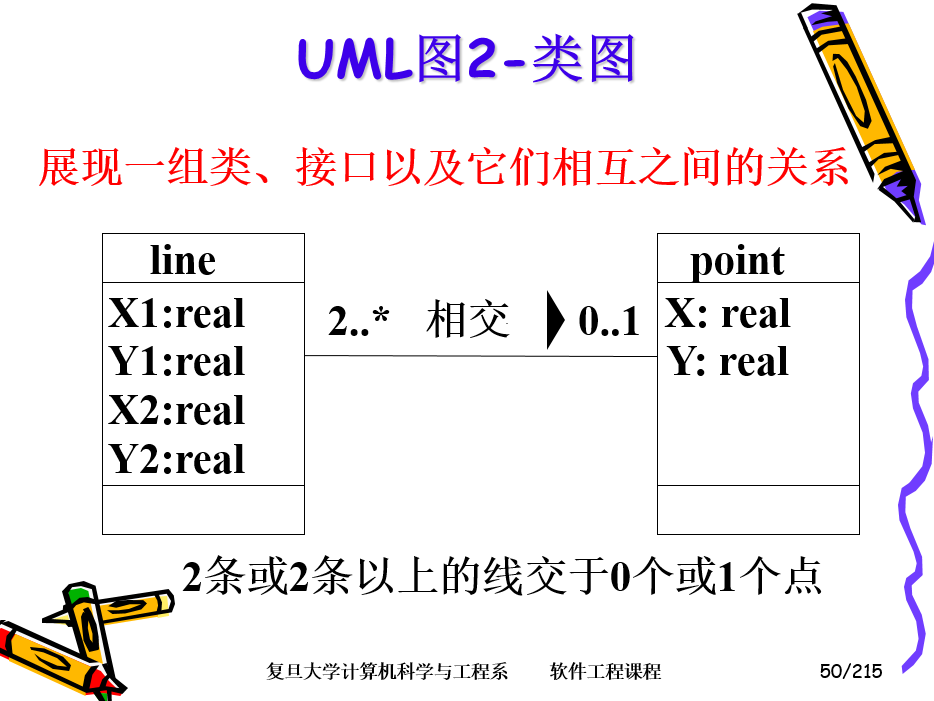
## 软件重构的概念

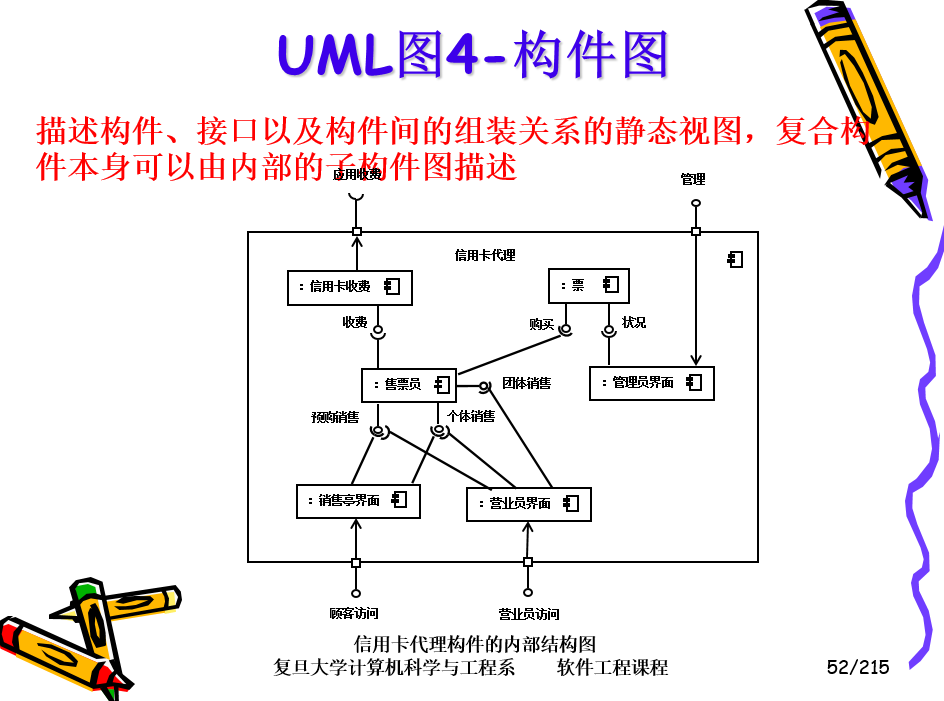
**软件重构**是指在不改变软件的功能和外部可见性的情况下，为了改善软件的结构，提高软件的清晰性、可扩展性和可重用性而对其进行的改造

## 软件体系结构的UML建模（包图、类图、构件图、顺序图、部署图）

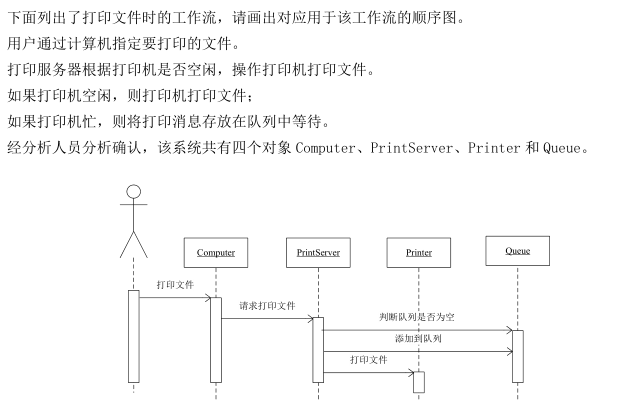


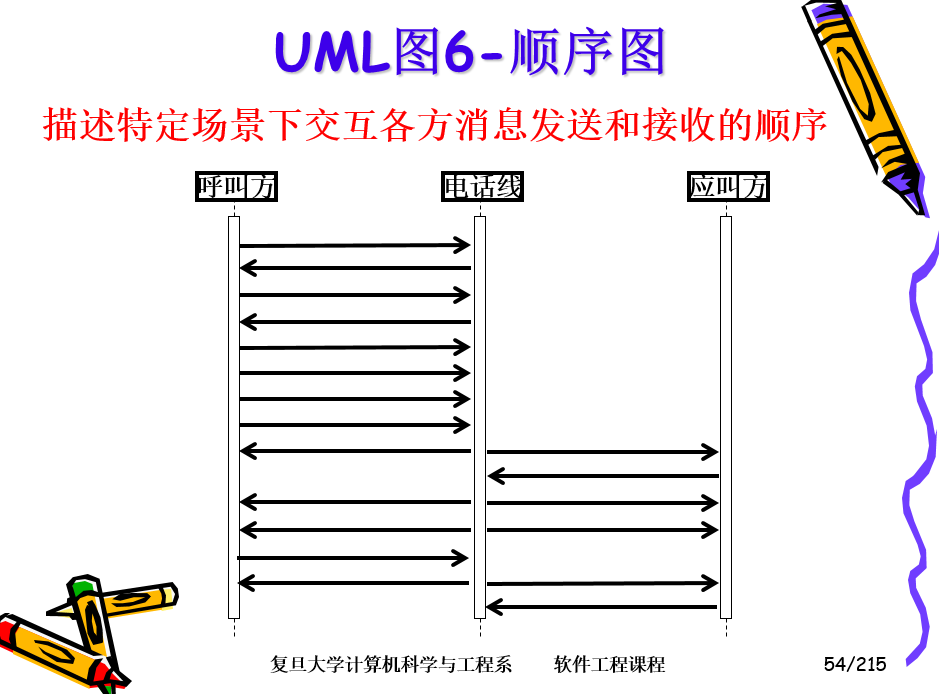
类图: <https://blog.csdn.net/monkey_d_meng/article/details/6005764>





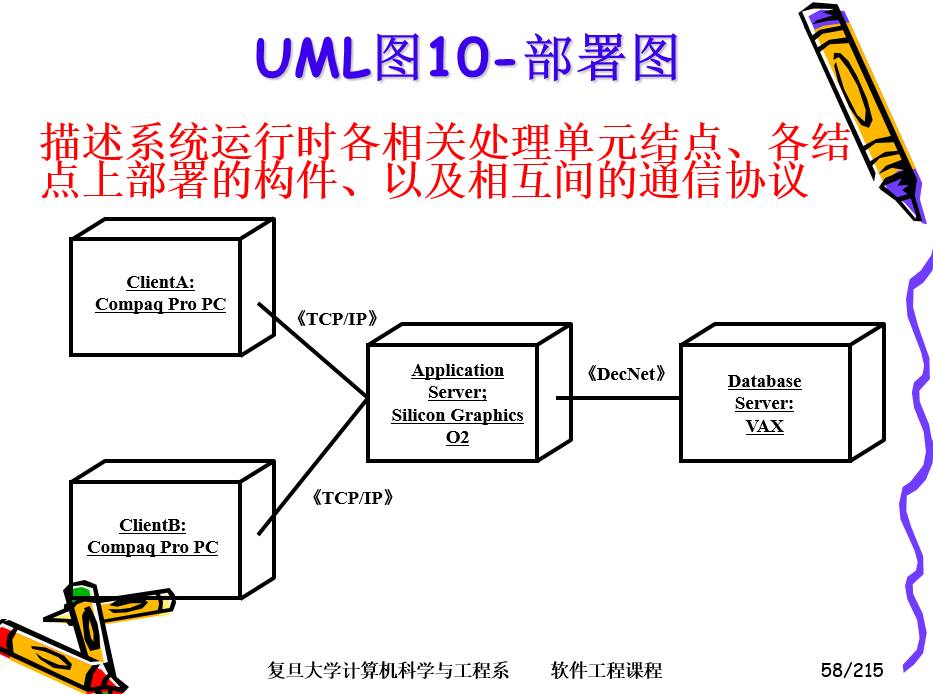
顺序图（重点）：用来显示对象之间发送消息的顺序，以及对象之间的交互。例题如下：





部署图(略，考察几率低):展现系统中硬件和软件的物理机构

画法：<https://blog.csdn.net/wangyongxia921/article/details/8250129>



## 接口的概念；

首先接口不是类，而是对某个类的一组需求描述。它主要用于描述某个类应该具备什么样的功能，实现（implements）了某接口的类必须遵从接口描述的统一格式进行定义。一个类可以实现一个或者多个接口，实现了某接口的类必须重写接口中包含的所有方法。

## 面向对象设计原则（开闭原则、Liskov替换原则、依赖转置原则、接口隔离原则）；

**开闭原则**:

类的改动是通过增加代码进行的,而不是修改源代码。对扩展开放，对修改关闭。

**里氏代换原则**:

任何抽象类（父类）出现的地方都可以用他的实现类（子类）进行替换，实际就是虚拟机制，语言级别实现面向对象功能

**依赖倒转原则**:

依赖于抽象(接口),不要依赖具体的实现(类)，也就是针对接口编程。

1.高层模块不应该依赖低层模块，两者都应该依赖其抽象（抽象类或接口）

2.抽象不应该依赖细节（具体实现）

3.细节（具体实现）应该依赖抽象。

三种实现方式:

1.通过构造函数传递依赖对象

2.通过setter方法传递依赖对象

3.接口声明实现依赖对象

**接口隔离原则**:

不应该强迫用户的程序依赖他们不需要的接口方法。一个接口应该只提供一种对外功能，不应该把所有操作都封装到一个接口中去。

分离接口的两种实现方法：

1.使用委托分离接口。（Separation through Delegation）

2.使用多重继承分离接口。（Separation through Multiple Inheritance）

## 内聚与耦合的概念、常见的内聚和耦合类型。

**功能独立性可以由两项指标来衡量：内聚度与耦合度**

* **内聚（cohesion）**是一个模块内部各个元素彼此结合的紧密程度的度量

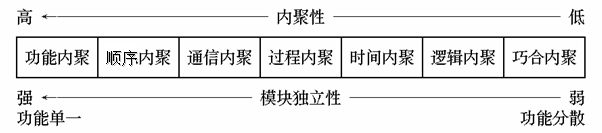
**耦合(coupling)**是模块之间的相对独立性（互相连接的紧密程度）的度量

“高内聚，低耦合”

起因：模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准――耦合性和内聚性。  
    耦合性也称块间联系。指软件系统结构中各模块间相互联系紧密程度的一种度量。模块之间联系越紧密，其耦合性就越强，模块的独立性则越差。模块间耦合高低取决于模块间接口的复杂性、调用的方式及传递的信息。

    内聚性又称块内联系。指模块的功能强度的度量，即一个模块内部各个元素彼此结合的紧密程度的度量。若一个模块内各元素（语名之间、程序段之间）联系的越紧密，则它的内聚性就越高。

    耦合性与内聚性是模块独立性的两个定性标准，将软件系统划分模块时，尽量做到高内聚低耦合，提高模块的独立性，为设计高质量的软件结构奠定基础。



**1) 巧合内聚（偶然内聚）：**将几个模块中没有明确表现出独立功能的相同程序代码段独立出来建立的模块称为巧合内聚模块。

**2) 逻辑内聚 ：**指完成一组逻辑相关任务的模块，调用该模块时，由传送给模块的控制型参数来确定该模块应执行哪一种功能。

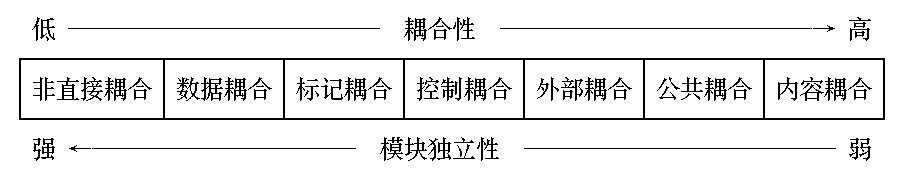
**3) 时间内聚：**指一个模块中的所有任务必须在同一时间段内执行。例如初始化模块和终止模块。

**4) 过程内聚 ：**指一个模块完成多个任务，这些任务必须按指定的过程（procedural）执行。

**5) 通信内聚 ：**指一个模块内所有处理元素都集中在某个数据结构的一块区域中。

**6) 顺序内聚：**指一个模块完成多个功能，这些功能又必须顺序执行。

**7) 功能内聚 ：**指一个模块中各个部分都是为完成一项具体功能而协同工作，紧密联系，不可分割的。



**1) 内容耦合 ：**如果一个模块直接访问另一个模块的内部数据；或者一个模块不通过正常入口转到另一模块内部；或者两个模块有一部分程序代码重迭；或者一个模块有多个入口，则两个模块之间就发生了内容耦合。

**2) 公共耦合 ：**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。

**3) 外部耦合 ：**指模块间通过软件之外的环境联结（如I/O将模块耦合到特定的设备、格式、通信协议上）时，称为外部耦合。

**4) 控制耦合：**如果一个模块传送给另一个模块的参数中包含了控制信息，该控制信息用于控制接收模块中的执行逻辑，则称为控制耦合。

**5) 标记耦合：**两个模块之间通过参数表传递一个数据结构的一部分（如某一数据结构的子结构），就是标记耦合。

**6) 数据耦合：**两个模块之间仅通过参数表传递简单数据，则称为数据耦合。

**7) 非直接耦合 ：**如果两个模块之间没有直接关系，即它们中的任何一个都不依赖于另一个而能独立工作，这种耦合称为非直接耦合。

1. 软件测试

## 软件测试及测试用例的概念

测试是一个为了发现错误而执行程序的过程

* 一个好的测试用例是指很可能找到迄今为至尚未发现的错误的测试用例
* 一个成功的测试是指揭示了迄今为至尚未发现的错误的测试
* 根据这个测试目的，我们应该排除对测试的错误观点，设计合适的测试用例，用尽可能少的测试用例，来发现尽可能多的软件错误

测试用例（Test Case）是为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径或核实是否满足某个特定需求。

## 单元测试、集成测试、确认测试、系统测试、回归测试的概念

单元测试 (Unit Testing)

* 单元测试又称模块测试，它着重对软件设计的最小单元（软件构件或模块）进行测试
* 单元测试根据设计描述，对重要的控制路径进行测试，以发现构件或模块内部的错误
* 单元测试通常采用白盒测试，并且多个构件或模块可以并行进行测试
* 这里将构件或模块统一称为模块

集成测试（Integrated Testing）

集成测试 也称组装测试、联合测试。经单元测试后，每个模块都能独立工作，但把它们放在一起往往不能正常工作，主要问题在于：

* 数据可能在通过接口时丢失
* 一个模块可能对另一个模块产生非故意的、有害的影响（即副作用）
* 当子功能被组合起来时，可能不能达到期望的主功能
* 单个模块可以接受的不精确性（如误差），连接起来后可能会扩大到无法接受的程度
* 全局数据结构可能会存在问题

回归测试（Regression Testing）

在集成测试过程中，每当增加一个（或一组）新模块时，原先已集成的软件就发生了改变。新的数据流路径被建立，新的I/O操作可能出现，还可能激活新的控制逻辑，这些改变可能使原本正常的功能产生错误

当测试时发现错误后，需修改程序；或者在软件维护时也需修改程序。这些对程序的修改也可能使原本正常的功能产生错误

回归测试就是对已经进行过测试的测试用例子集的重新测试，以确保对程序的改变和修改，没有传播非故意的副作用

确认测试（Validation Testing）

确认测试的标准

确认测试以软件需求规约为依据，以发现软件与需求不一致的错误。主要检查软件是否实现了规约规定的全部功能要求，文档资料是否完整、正确、合理，其他的需求，如可移植性、可维护性、兼容性、错误恢复能力等是否满足

确认测试的结果可分为两类：

满足需求规约要求的功能或性能特性，用户可以接受

发现与需求规约有偏差，此时需列出问题清单

系统测试（System Testing）

系统测试是对整个基于计算机的系统进行的一系列测试

系统测试的种类很多，每种测试都有不同的目的，它们从不同的角度测试计算机系统是否被正常地集成，并完成相应的功能

常用的系统测试包括：

* 恢复测试（recovery testing）
* 安全保密性测试（security testing）
* 压力测试（stress testing）
* 性能测试（performance testing）

## 调试的概念、调试与测试的关系

测试的目的是发现错误，调试（也称排错）的目的是确定错误的原因和准确位置，并加以纠正

测试是旨在发现软件“表面”的不当行为和属性，而调试是寻找这个表象下面的内因

## 测试覆盖度的概念

测试覆盖度评估是衡量阶段性软件测试执行状态的重要手段之一，来确定测试是否达到事先设定的测试任务完成的标准

## 白盒测试、黑盒测试的概念

白盒测试（又称为结构测试）把测试对象看作一个透明的盒子，测试人员根据程序内部的逻辑结构及有关信息设计测试用例，检查程序中所有逻辑路径是否都按预定的要求正确地工作

白盒测试主要用于对模块的测试，包括：

* 程序模块中的所有独立路径至少执行一次
* 对所有逻辑判定的取值（“真”与“假”）都至少测试一次
* 在上下边界及可操作范围内运行所有循环
* 测试内部数据结构的有效性等

黑盒测试（又称行为测试）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需求

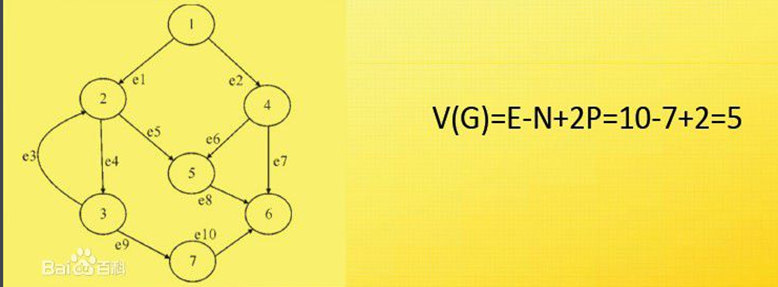
黑盒测试可用于各种测试，它试图发现以下类型的错误：

* 不正确或遗漏的功能
* 接口错误，如输入/输出参数的个数、类型等
* 数据结构错误或外部信息(如外部数据库)访问错误
* 性能错误
* 初始化和终止错误

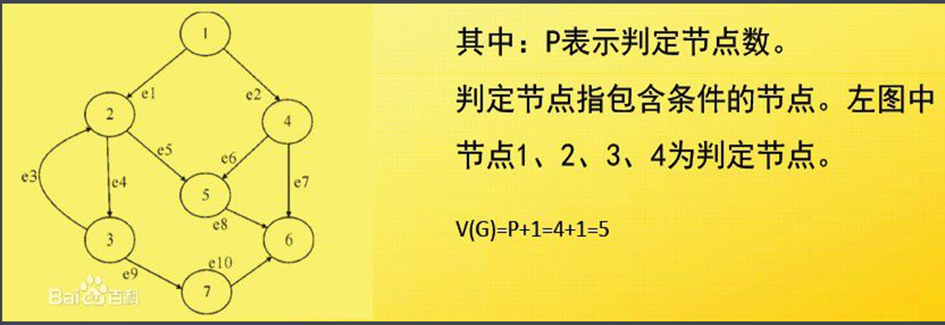
## 代码圈复杂度的计算方法

圈复杂度是一种代码复杂度的衡量标准。在软件测试的概念里，圈复杂度用来衡量一个模块判定结构的复杂程度，数量上表现为独立线性路径条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难以测试和维护，根据经验，程序的可能错误和高的圈复杂度有着很大关系。它的计算方法为：

1. V(G)=e-n+2p。e表示控制流图中边的数量，n表示控制流图中节点的数量，p表示图的连接组件数目（图的组件数是相连节点的最大集合），因为控制流图都是连通的，所以p永远为1



1. V(G)=区域数=判定节点数+1



1. V(G)=R。R代表平面被控制流图划分成的区域数

## 白盒测试中的基本路径测试方法

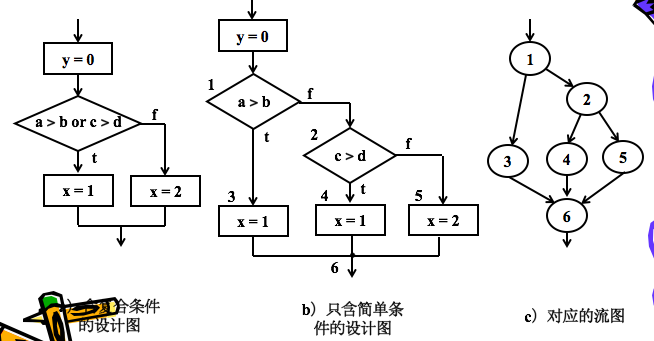
在实际问题中，一个不太复杂的程序，特别是包含循环的程序，其路径数可能非常大。因此测试常常难以做到覆盖程序中的所有路径，为此，我们希望把测试的程序路径数压缩到一定的范围内

基本路径测试是Tom McCabe提出的一种白盒测试技术，这种方法首先根据程序或设计图画出控制流图，并计算其区域数，然后确定一组独立的程序执行路径（称为基本路径），最后为每一条基本路径设计一个测试用例

流图由结点和边组成，分别用圆和箭头表示。设计图中一个连续的处理框（对应于程序中的顺序语句）序列和一个判定框（对应于程序中的条件控制语句）映射成流图中的一个结点，设计图中的箭头（对应于程序中的控制转向）映射成流图中的一条边。对于设计图中多个箭头的交汇点可以映射成流图中的一个结点（空结点）

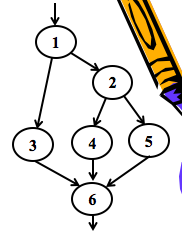


上述映射的前提是设计图的判定中不包含复合条件。如果设计图的判定中包含了复合条件，那么必须先将其转换成等价的简单条件设计图

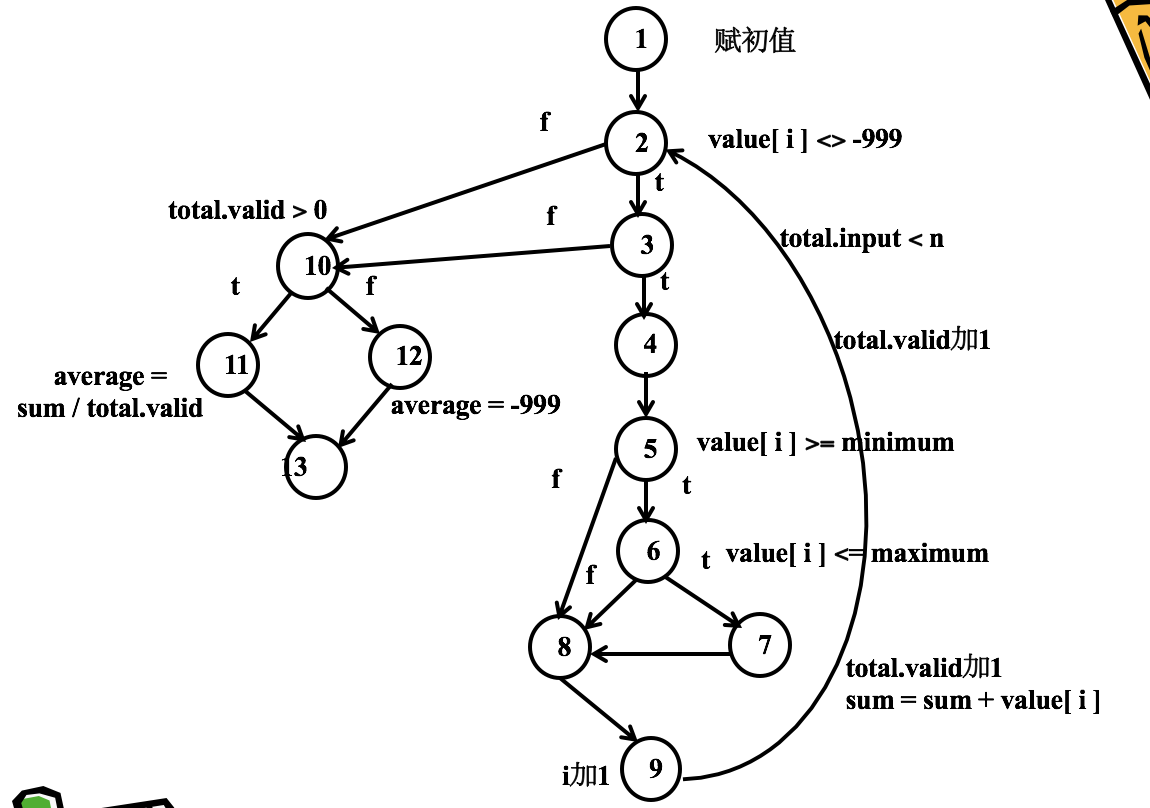


我们把流图中由结点和边组成的闭合部分称为一个区域（region），在计算区域数时，图的外部部分也作为一个区域。例如，右图所示的流图的区域数为3

独立路径是指程序中至少引进一个新的处理语句序列或一个新条件的任一路径，在流图中，独立路径至少包含一条在定义该路径之前未曾用到过的边。在基本路径测试时，独立路径的数目就是流图的区域数

* ****

例如，对一个PDL程序进行基本路径测试，该程序的功能是：最多输入N个值（以-999为输入结束标志），计算位于给定范围内的那些值（称为有效输入值）的平均值，以及输入值的个数和有效值的个数。

其区域数为6，我们选取独立路径如下：

* 路径1：1-2-10-11-13
* 路径2：1-2-10-12-13
* 路径3：1-2-3-10-11-13
* 路径4：1-2-3-4-5-8-9-2-10-12-13
* 路径5：1-2-3-4-5-6-8-9-2-10-12-13
* 路径6：1-2-3-4-5-6-7-8-9-2-10-11-13

为每一条独立路径设计测试用例。

假设：n = 5；minimum = 0；maximum = 100。

路径1： 1-2-10-11-13

测试数据：value = [ 90，-999，0，0，0]

预期结果：Average = 90，total.input = 1，total.valid = 1

路径2： 1-2-10-12-13

测试数据：value = [ -999 ，0，0，0，0]

预期结果：Average = -999，total.input = 0，total.valid = 0

路径3： 1-2-3-10-11-13

测试数据：value = [ -1，90，70，-1，80]

预期结果：Average = 80，total.input = 5，total.valid = 3

路径4： 1-2-3-4-5-8-9-2-10-12-13

测试数据：value = [-1，-2，-3，-4，-999]

预期结果：Average = -999，total.input = 4，total.valid = 0

路径5： 1-2-3-4-5-6-8-9-2-10-12-13

测试数据：value = [ 120，110，101，-999，0]

预期结果：Average = -999，total.input = 3，total.valid = 0

路径6： 1-2-3-4-5-6-7-8-9-2-10-11-13

测试数据：value = [ 95，90，70，65，-999]

预期结果：Average = 80，total.input = 4，total.valid = 4

值得注意的是，某些独立路径（如例中的路径1和路径3）不能以独立的方式进行测试，此时，这些路径必须在其他的独立路径测试中被覆盖

## 黑盒测试中的等价类划分方法

由于不能穷举所有可能的输入数据来进行测试，所以只能选择少量有代表性的输入数据，来揭露尽可能多的程序错误

等价类划分方法将所有可能的输入数据划分成若干个等价类，然后在每个等价类中选取一个代表性的数据作为测试用例

等价类是指输入域的某个子集，该子集中的每个输入数据对揭露软件中的错误都是等效的，测试等价类的某个代表值就等价于对这一类其他值的测试。也就是说，如果该子集中的某个输入数据能检测出某个错误，那么该子集中的其他输入数据也能检测出同样的错误；反之，如果该子集中的某个输入数据不能检测出错误，那么该子集中的其他输入数据也不能检测出错误

等价类划分方法把输入数据分为有效输入数据和无效输入数据

有效输入数据指符合规格说明要求的合理的输入数据，主要用来检验程序是否实现了规格说明中的功能

无效输入数据指不符合规格说明要求的不合理或非法的输入数据，主要用来检验程序是否做了规格说明以外的事

在确定输入数据等价类时，常常还要分析输出数据的等价类，以便根据输出数据等价类导出输入数据等价类

**等价类划分设计测试用例的步骤**

第一步，确定等价类  
根据软件的规格说明，对每一个输入条件（通常是规格说明中的一句话或一个短语）确定若干个有效等价类和若干个无效等价类。

|  |  |  |
| --- | --- | --- |
| 输入条件 | 有效等价类 | 无效等价类 |
|  |  |  |

确定等价类的规则：  
 (1) 如果输入条件**规定了取值范围**，则可以确定一个有效等价类（输入值在此范围内）和两个无效等价类（输入值小于最小值及大于最大值）

例如，规定输入的考试成绩在0..100之间，则有效等价类是“0 ≤ 成绩 ≤ 100”，无效等价类是“成绩 < 0”和“成绩 > 100”

(2) 如果输入条件**规定了值的个数**，则可以确定一个有效等价类（输入值的个数等于规定的个数）和两个无效等价类（输入值的个数小于规定的个数和大于规定的个数）

例如，规定输入构成三角形的3条边，则有效等价类是“输入边数 = 3”，无效等价类是“输入边数 < 3”和“输入边数 > 3”

(3) 如果输入条件**规定了输入值的集合（即离散值**），而且程序对不同的输入值做不同的处理，那么每个允许的值都确定为一个有效等价类，另外还有一个无效等价类（任意一个不允许的值）

例如，规定输入的考试成绩为优、良、中、及格、不及格，则可确定5个有效等价类和一个无效等价类

(4) 如果输入条件**规定了输入值必须遵循的规则**，那么可确定一个有效等价类（符合此规则）和若干个无效等价类（从各个不同的角度违反此规则）

例如，在程序语言中对变量标识符规定为“以字母开头的……串”。那么有效等价类是“以字母开头的串”，而无效等价类有“以数字开头的串”、“以标点符号开头的串”…等

(5) 如果输入条件**规定输入数据是整型**，那么可以确定三个有效等价类（正整数、零、负整数）和一个无效等价类（非整数）

(6) 如果输入条件**规定处理的对象是表格**，那么可以确定一个有效等价类（表有一项或多项）和一个无效等价类（空表）

第二步，设计测试用例  
在确定了等价类之后，建立等价类表，列出所有划分出的等价类，并为每个有效等价类和无效等价类编号

|  |  |  |
| --- | --- | --- |
| 输入条件 | 有效等价类 | 无效等价类 |
|  |  |  |

利用等价类设计测试用例的步骤：

(1) 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

(2) 为每个无效等价类设计一个新的测试用例

用等价类划分法设计测试用例的实例：  
某编译程序的规格说明中关于标识符的规定如下：标识符是由字母开头，后跟字母或数字的任意组合构成；标识符的字符数为1∽8个；标识符必须先说明后使用；一个说明语句中至少有一个标识符；保留字不能用作变量标识符

用等价类划分方法，建立输入等价类表



下面选取9个测试用例，它们覆盖了所有的等价类

