第一章

软件定义：软件=程序+数据+文档（背）
程序：按事先设计的功能和性能需求执行的指令序列
数据：是程序能正常操纵信息的数据结构
文档：与程序**开发、维护和使用**有关的图文材料

软件的特征：
1、软件是开发的或者是工程化的，并不是制造的
2、软件开发环境对产品影响较大
3、软件开发时间和工作量难以估计
4、软件会多次修改
5、软件的开发进度几乎没有客观衡量标准
6、软件测试非常困难
7、软件不会磨损和老化
8、软件维护易产生新的问题
9、软件生产是简单的拷贝

软件危机：在计算机软件的开发和维护过程中所遇到的一系列严重问题。
集体表现在：项目超出预算，项目超时，软件运行效率低，软件质量差，不符合要求，难以管理，软件成本日益增加，技术进步落后于需求的增长等（感脚会考）
产生原因：
客观：软件本身特点——（逻辑部件，规模庞大）
主观：不正确的开发方法——
（忽视需求分析，错误认为：软件开发=程序编写，轻视软件维护）

软件工程：(记下来)
软件工程定义为：（1）应用系统化的、学科化的、定量的方法，来开发、运行和维护软件，即，将工程应用到软件。（2）对（1）中各种方法的研究。

三要素：工具（工具支持），方法（技术手段），过程（贯穿各环节）

软件工程发展阶段：
1、第一代—传统的软件工程（60-70年代）基本形成软件工程的概念、框架、技术和方法

2、第二代—

对象工程（80年代中到90年代）研究重点转化到面向对象的分析和设计，演化为一种完整的软件开发方法和系统的技术体系

3、第三代——

过程工程（80年代中开始）认识到提高软件生产率，保证软件质量的关键是软件开发和维护中的管理和支持能力

4、第四代——

构件工程（90年代起）可通过现成的可服用构件组装完成，提高效率和质量，降低成本

软件工程7个原则：

1、使用阶段性生命周期计划的管理
2、进行连续的验证
3、保证严格的产品控制
4、使用现代的编程工具/工程实践
5、保持清晰的责任分配
6、用更好更少的人
7、保持过程改进

第二章

软件工程是一种层次化技术

过程模型：

- 定义了若干小的框架活动，为完整的软件开发过程建立了基础
- 每一个活动由一组软件工程动作组成
- 每一个动作都包括一系列相互关联的可考核的任务，并产生一个关键的工作产品

软件生命周期：软件产品或软件系统从设计、投入使用到被淘汰的全过程。（觉得会考哟）

软件过程：沟通、计划、建模、构造、部署

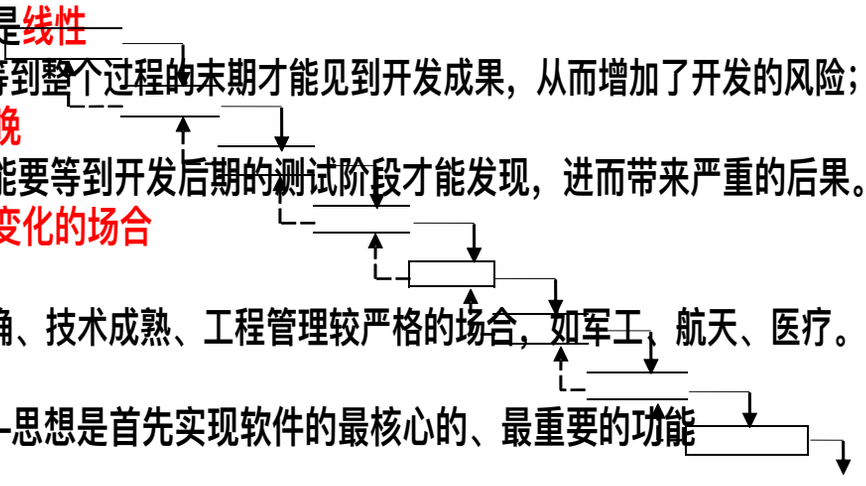
CMMI：初始级，可重复级，已定义级，量化管理级，优化级

软件过程模型：（软件开发模型、软件生存周期模型、软件工程范型）（各自特点和优缺点和比较是重点）

软件过程模型是软件开发全部过程、活动和任务的结构框架。
瀑布模型：（带反馈的瀑布模型）
软件开发过程与软件生命周期是一致的，也称经典的生命周期模型。

特点：
阶段间具有顺序性和依赖性。
推迟实现的观点。
每个阶段必须完成规定的文档； 每个阶段结束前完成文档审查,及早改正错误。
以文档为驱动。
缺点：

增加工作量
各个阶段的划分完全固定，阶段之间产生大量的文档，极大地增加了工作量；
开发风险大
由于开发模型是线性
的，用户只有等到整个过程的末期才能见到开发成果，从而增加了开发的风险；
早起错误发现晚
早期的错误可能要等到开发后期的测试阶段才能发现，进而带来严重的后果。
不适用于需求变化的场合
适用于
：系统需求明确、技术成熟、工程管理较严格的场合，如军工、航天、医疗。



原型模型——思想是首先实现软件的最核心的、最重要的功能

目的：明确并完善需求；研究技术选择方案
结果：抛弃原型、把原型发展成最终产品

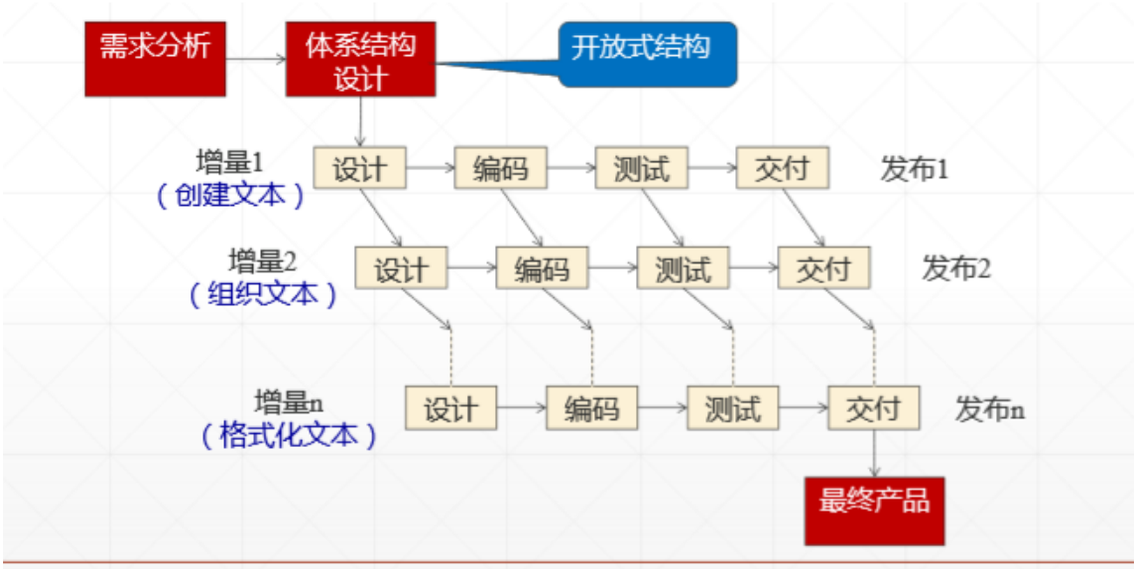
优点：减少需求不明带来的风险

缺点：

1. 构造原型采用的工具和技术不一定主流；
2. 设计者在质量和原型中进行折中；
3. 客户意识不到一些质量问题。

适用场合：客户定义一个总体目标集，但并不清楚系统的具体输入输出
开发者不确定算法的效率、软件与操作系统是否兼容以及客户与计算机的交互方式

增量过程模型（增量模型、RAD）——增量：小而可用的软件



特点

在前面增量的基础上开发后面的增量

每个增量的开发可用瀑布或快速原型模型

迭代的思路

采用了基于时间的线性序列，每个线性序列都会输出该软件的一个“增量”

优点：

1. 增量包概念的引入，以不需要提供完整的需求，只要有一个增量包出现，开发就可以进行。
2. 在项目的初始阶段不需要投入太多的人力资源。
3. 增量可以有效地管理技术风险。

缺点：

每个增量必须提供一些系统功能，这使得开发者很难根据客户需求给出大小适合的增量。

软件必须具备开放式体系结构

已退化成边做边改的方式，是软件过程控制失去整体性

适用场合：需求可能发生变化、较大风险、尽快投入市场

RAD——

瀑布模型的“高速”变体。通过基于组件的构建方法实现快速开发。

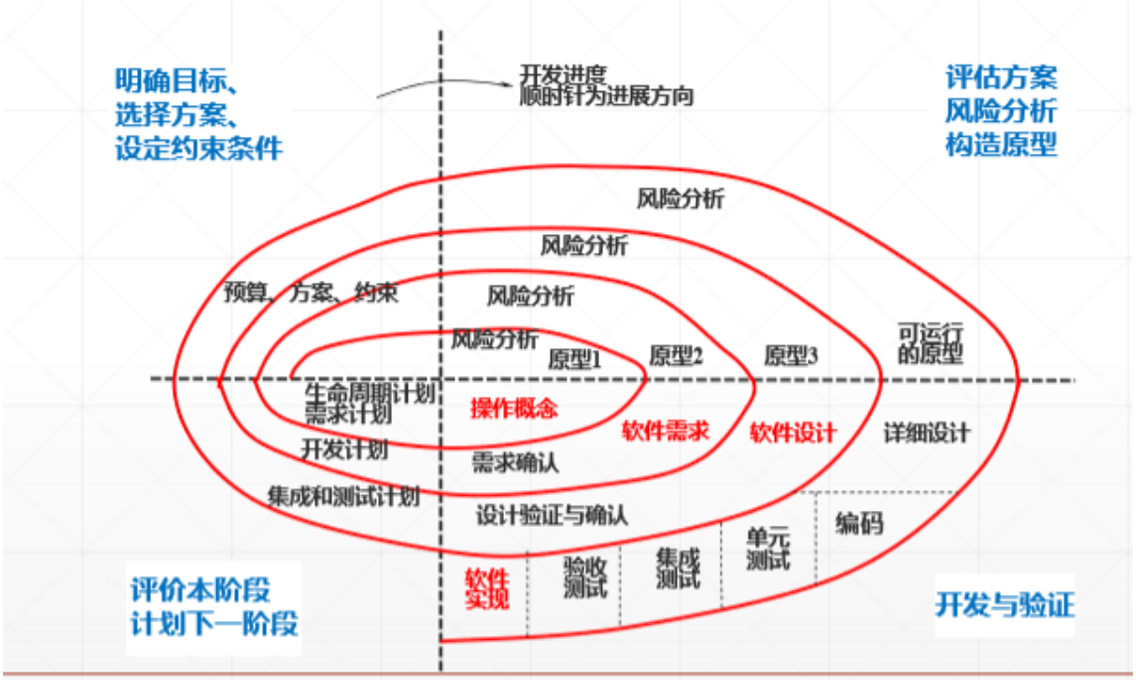
缺点：

1）对大型项目而言，RAD 需要足够的人力资源。

2）开发者和客户都要实现承诺，否则将导致失败。

3）并非所有系统都适合（不能合理模块化的系统、高性能需求并且要调整构件接口的、技术风险很高的系统均不适合）。

螺旋模型——
开发活动与风险管理的结合，强调风险管理，因此该模型适用于大型系统的开发。



优点：
螺旋模型特别强调原型的可扩充性和可修改性，原型的进化贯穿整个软件生存周期，这将有助于目标软件的适应能力。
支持用户需求的动态变化。
原型可看作形式的可执行的需求规格说明，易于为用户和开发人员共同理解，还可作为继续开发的基础，并为用户参与所有关键决策提供了方便。
螺旋模型为项目管理人员及时调整管理决策提供了方便，进而可降低开发风险。
缺点：
如果每次迭代的效率不高，致使迭代次数过多，将会增加成本并推迟提交时间；
使用该模型需要有相当丰富的风险评估经验和专门知识，要求开发队伍水平较高。
适用于：
支持需求不明确、特别是大型软件系统的开发，并支持面向规格说明、面向过程、面向对象等多种软件开发方法，是一种具有广阔前景的模型。

喷泉模型——
喷泉模型是一种以用户需求为动力，以对象为驱动的模型，主要用于描述面向对

象的软件开发过程。

优点

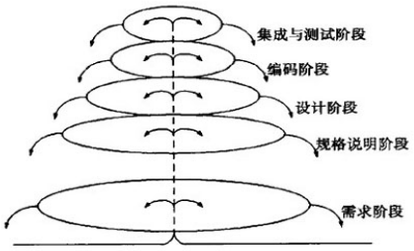


图　喷泉模型

：该模型的各个阶段没有明显的界限，开发人员可以同步进行开发。其优点是可以提高软件项目开发效率，节省开发时间，适应于面向对象的软件开发过程。

缺点

：喷泉模型在各个开发阶段是重叠的，在开发过程中需要大量的开发人员，因此不利于项目的管理。此外这种模型要求严格管理文档，使得审核的难度加大，尤其是面对可能随时加入各种信息、需求与资料的情况。

基于构件的模型（重用和集成）：

统一过程模型：

面向对象方法学、使用统一建模语言UML

动态视角、静态视角、时间视角

适合大团队大项目

敏捷开发：

参考原则（看看就行啦~）

1. 在前期需求明确的情况下，尽量采用瀑布模型或改进的瀑布模型。
2. 在用户无系统使用经验，需求分析人员技能不足情况下一定要借助原型。
3. 在不确定因素很多，很多东西前面无法计划的情况下尽量采用增量迭代和螺旋模型。
4. 在需求不稳定的情况下尽量采用增量迭代模型。
5. 在资金和成本无法一次到位的情况下可采用增量模型，软件产品多个版本进行发布。
6. 对于完成多个独立功能开发可以在需求分析阶段就进行功能并行，但每个功能内部都应该遵循瀑布模型。
7. 对于全新系统的开发必须在总体设计完成后再开始增量或并行。
8. 对于编码人员经验较少的情况下建议不要采用敏捷或迭代等生命周期模型。
9. 增量、迭代和原型可以综合使用，但每一次增量或迭代都必须有明确的交付和出口原则。

瀑布模型、增量模型、原型模型和螺旋模型的联系和区别

比较项	瀑布模型	原型模型	增量模型	螺旋模型
是否事先定义大部分需求	是	无	无	是
迭代类型	无	过程级	活动级	逻辑级
每次迭代输出是否为产品	是	无	无	是
驱动方式	文档	用户	用户	风险
迭代周期内的过程模型	瀑布模型	瀑布模型	原型模型	可选

举例说明：若一个软件包含A、B、C、D四个功能，那么
瀑布模型输出结果为ABCD
原型模型输出结果为A’B’C’D’->ABCD
增量模型输出结果为A->AB->ABC->ABCD
螺旋模型输出结果为A1B1C1D1->A2B2C2D2->……->ABCD

第三章(这才是重点好不亲~)

需求分析——确定系统必须具有的功能和性能，系统要求的运行环境，并且预测系统发展的前景。换句话说需求就是以一种清晰、简洁、一致且无二义性的方式，对一个待开发系统中各个有意义方面的陈述的一个集合。

任务——建立模型、编写需求规格说明书

需求分析步骤——需求获取 、需求提炼 、需求描述（撰写需求规格说明书）、需求验证

需求分析的核心在于建立分析模型
需求建模图形工具（记住下面这张表啦~）

	面向过程的需求分析	面向对象的需求分析
数据模型	实体-联系图（ERD） 数据字典（DD）	类图、类关系图
功能模型	数据流图（DFD）	用例图
行为模型	状态变迁图（STD）	活动图、时序图、状态图

需求验证：

(1) 有效性检查

检查不同用户使用不同功能的有效性。

(2) 一致性检查

在文档中，需求之间不应该冲突。

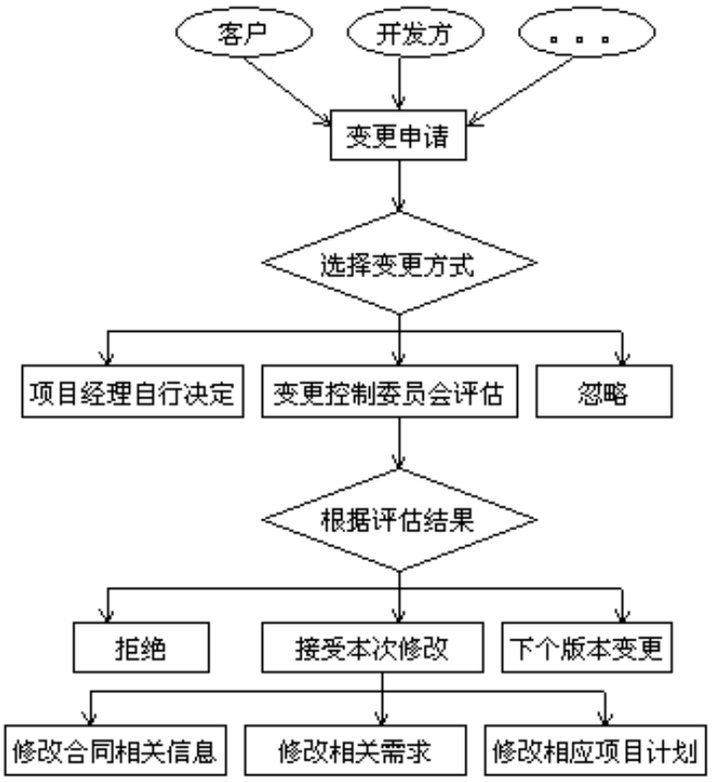
(3) 完备性检查

需求文档应该包括所有用户想要的功能和约束。

(4) 现实性检查

检查保证能利用现有技术实现需求。

需求变更管理：

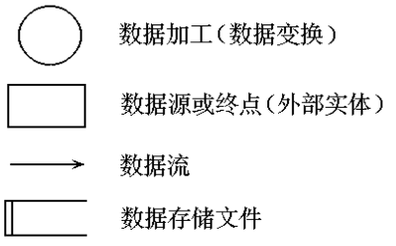


面向过程的分析方法——
面向数据流进行需求分析的方法（感脚数据流图最重要了。。）

结构化分析方法:

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

面向数据流
数据流图（加工规约）



检查和修改数据流图的原则

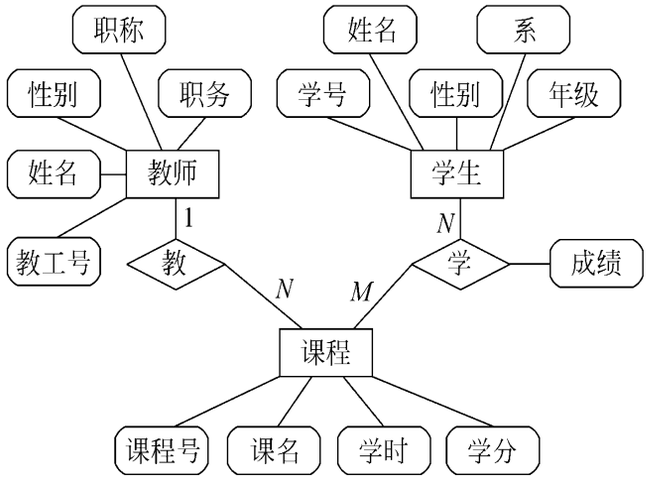
- 数据流图上所有图形符号只限于前述四种基本图形元素
- 数据流图的主图必须包括前述四种基本元素，缺一不可
- 数据流图的主图上的数据流必须封闭在外部实体之间
- 每个加工至少有一个输入数据流和一个输出数据流
- 在数据流图中，需按层给加工框编号。编号表明该加工所处层次及上下层的亲子关系
- 规定任何一个数据流子图必须与它上一层的一个加工对应，两者的输入数据流和输出数据流必须一致。此即父图与子图的平衡
- 图上每个元素都必须有名字
- 数据流图中不可夹带控制流
- 初画时可以忽略琐碎的细节，以集中精力于主要数据流

ER图 ---- 是用来建立数据模型的工具

数据模型 ----

是一种面向问题的数据模型，是按照用户的观点对数据建立的模型。它描述了从用户角度看到的数据，反映了用户的现实环境，而且与在软件系统中的实现方法无关。

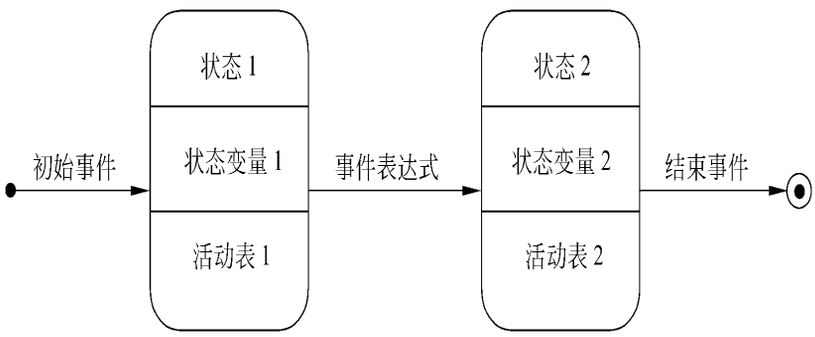
数据对象及其属性、彼此间的关系



数据字典

订票单
名字: 订票单
数据类型: 航班日期 + 目的地 + 出发地 + 航班号
作为输出流的转换列表: 无
作为输入流的转换列表: 预定机票
使用说明: 必须给出各个数据项
解释性说明: 无
缺省值: 出发地 = 填写本地

行为模型——状态变迁图（控制规约）



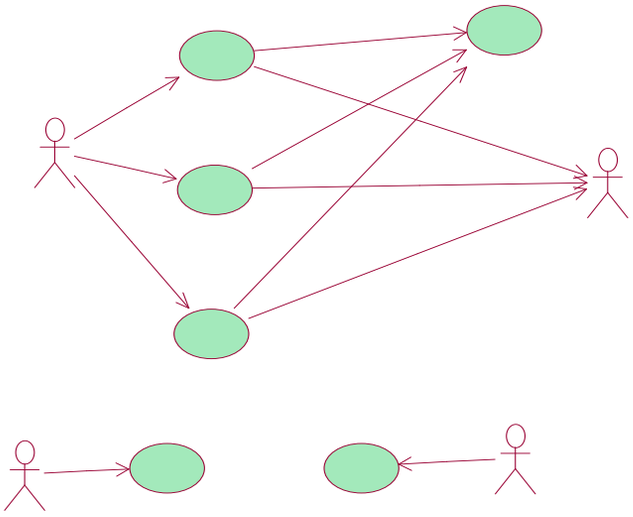
通过描绘系统的状态及引起系统状态转换的事件，来表示系统的行为。此外，状态图还指明了作为特定事件的结果系统将做哪些动作(例如，处理数据)。

面向对象的分析方法（就考考用例图啦~）

- 就像
  数据流图作为结构化分析的建模语言，模块结构图作为结构化总体设计的建模语言一样，UML是面向对象的系统分析与设计的建模语言
  ，不要将它理解为一种方法论或是一种开发过程
- (即UML不能用来直接开发程序）。

功能模型——用例图

-



用例是系统向用户提供一个有价值的结果的某项功能

参与者：一个参与者可以代表一个人、一个计算机子系统、硬件设备或者时间等角色

用例：

对一组动作序列的描述，系统通过执行这一组动作序列为参与者产生一个可观察的结果

例：ATM机

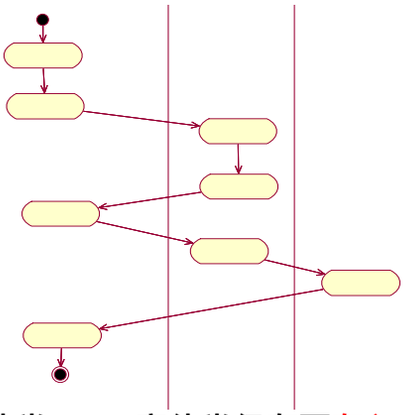
数据模型——类图

类图中的基本关系包括：关联关系，聚合关系，组合关系，依赖关系，泛化关系等。

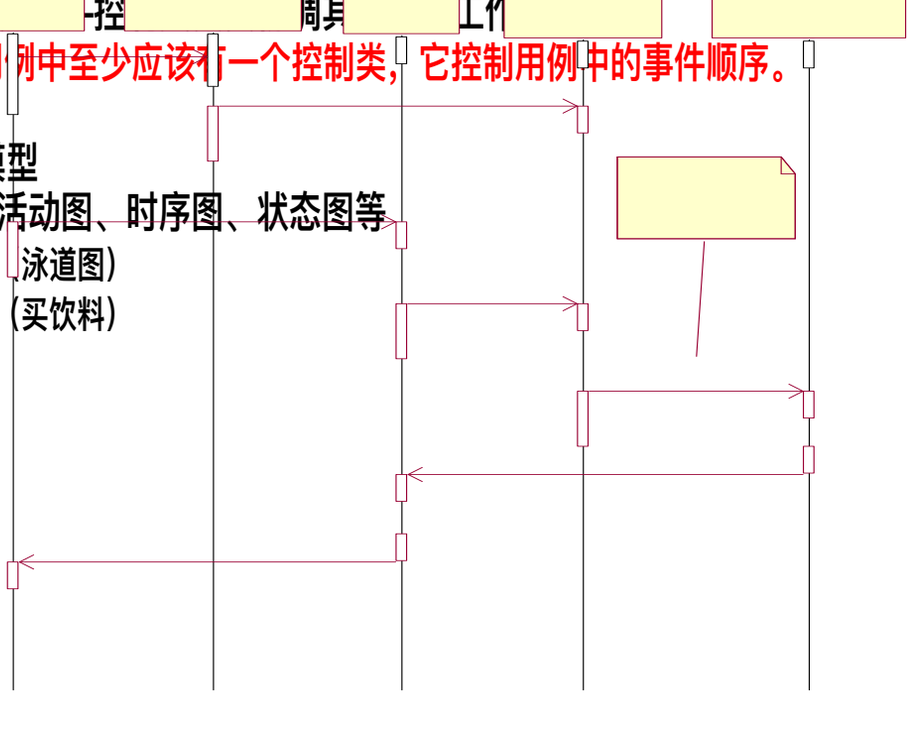
在聚合关系中，类A是类B的一部分，但是类A可以独立存在
在组合关系中，类A控制类B的生命周期意味着类B的存在依赖于类A。
依赖关系体现在某个类的方法使用另一个类作为参数
泛化也就是继承关系，泛化关系描述了超类与子类之间的关系，超类又叫做基类，子类又叫做派生类。

根据功能将类分为实体类、边界类和控制类。

边界类——
位于系统与外界的交界处，包括所有的窗体、报表、系统硬件接口、与其它系统的接口。



实体类——实体类保存要存入永久存储体的信息
。实体类通常在事件流或交互图中，是对用户最有意义的类。
控 -控 周其 工作
每个用例中至少应该有一个控制类，它控制用例中的事件顺序。



行为模型
——活动图、时序图、状态图等
活动图（泳道图）
时序图（买饮料）

状态图

一个完整的用例（瞄一眼就行……）

用例：在ATM上提款
主要参与者：取款者
目标：从与银行卡关联的账户中取出一定数目的现金
前提条件：取款者将银行卡正确插入ATM机，并成功登录该卡账户
触发器：取款者选择“取款”功能
场景：
取款者：选择“取款”功能
取款者：选择取款金额或输入取款金额并确认
ATM机：与银行主机请求取款，清点并吐出相应数目现金，并记录取款成功
取款者：取走钞票，打印凭条，并选择“退卡”或其他功能
异常：
ATM机现金不足：失效“取款”功能或提示错误原因，用例失败
凭条打印有故障：提示并询问用户是否继续
取款金额不符合要求（如：非100的整数倍、大于单次取款上限或当日上限等）：提示错误原因，要求重新输入
网络通讯故障：提示错误原因，用例失败
取款金额大于账户余额：提示错误原因，要求重新输入
吐现金故障：通知银行主机取消取款，用例失败
取款者未及时输入超过30秒：ATM机吞卡，用例失败
优先级：必须的，必须被实现
何时可用：首次增量
使用频率：每天多次

使用方式：通过ATM交互面板
次要参与者：技术支持人员，银行主机
次要参与者使用方式：电话线（技术支持人员）；有线网络（银行主机）

第四章（不好意思，这个也是重点……）

软件设计的定义（是连接用户需求和软件技术的桥梁）：

软件设计定义为软件系统或组件的架构、构件、接口和其他特性的定义过程及该过程的结果。

软件设计的任务：以软件需求规格说明书为依据，着手实现软件的需求，并将设计的结果反映在“设计规格说明书”文档中。

软件设计的重要性：是软件开发阶段的第一步，最终影响软件实现的成败和软件维护的难易程度。

设计由软件需求分析过程中获得信息驱动，采用可重复使用的方法导出

八个设计技术

抽象、体系结构、设计模式、模块化、信息隐藏、功能独立、细化、重构（哪八个，记下）

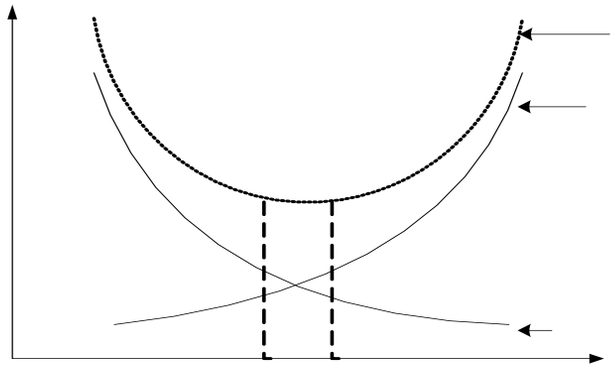
抽象：参数化、规范化
体系结构：软件的整体结构好和体系结构为系统提供概念上完整性的方式
体系结构：结构模型、框架模型、动态模型、过程模型、功能模型
模块化：软件被划分为命名和功能相对独立的多个组件（通常称为模块），通过这些组件的集成来满足问题的需求 .

什么是模块和模块化思想?

采取自顶向下的方式，逐层把软件系统划分成若干可单独命名和可编址的部分－“模块”，每个模块完成一个特定的子功能；所有模块按某种方法组成一个整体，完

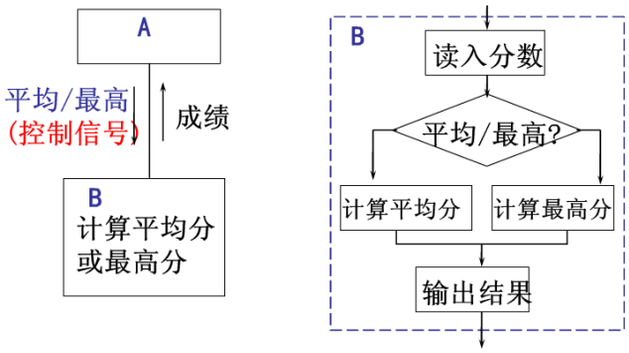
成整个系统所要求的功能。

模块化优点:
模块化使软件容易测试和调试，因而有助提高软件的可靠性。
模块化能提高软件的可修改性。



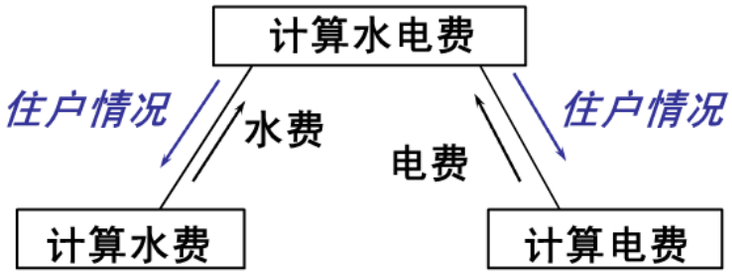
模块化有助于软件开发工程的组织管理。

如何确定最小代价区间M（怎么求来着？）
信息隐藏原则——
模块应该具有彼此相互隐藏的特性，保证模块内的信息不可以被不需要这些信息的其他模块访问
特点
抽象有助于定义构成软件的过程（或信息）实体。
信息隐藏原则定义和隐藏了模块内的过程细节和模块内的本地数据结构。
功能独立
每个模块只解决了需求中特定的子功能，并从程序结构的其他部分看该模块具有简单的接口
优点：易于开发——功能被划分，接口被简化
易于维护（和测试）——次生影响有限，错误传递减少，模块重用
定性衡量标准（内聚和耦合是什么总要知道吧。。。）
耦合性：模块之间的相互依赖程度
（1）数据耦合：一模块访问另一模块时，通过数据参数交换输入输出信息
（2）控制耦合：模块之间传递的是控制信息（控制被调用模块的内部逻辑）

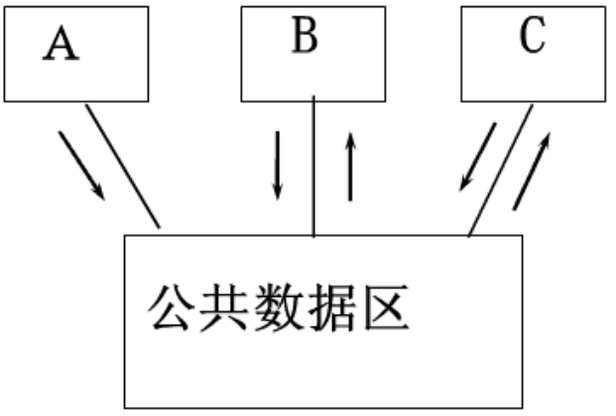


(3) 特征耦合:两个模块通过传递数据结构加以联系，或都与一个数据结构有关系

。



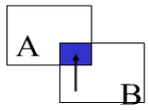
(4) 公共环境耦合:一组模块引用同一个公用数据区



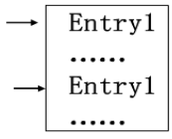
(5) 内容耦合:

A、一模块直接访问另一模块的内部信息

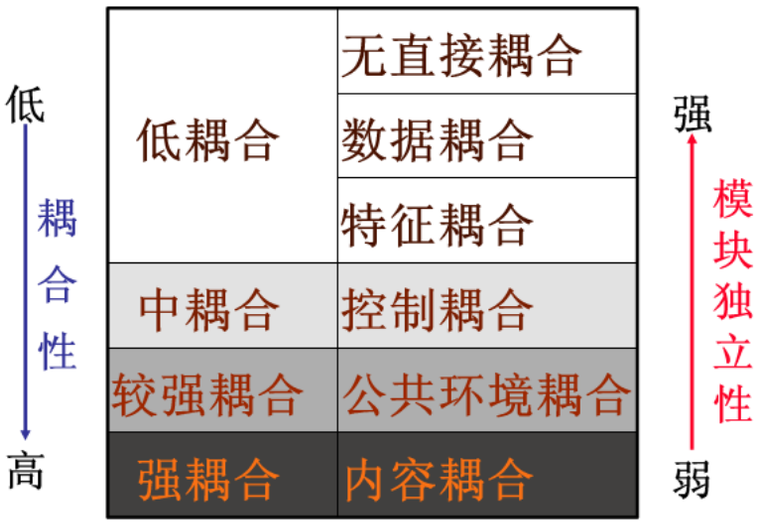
B、一个模块不通过正常入口转换到另一个模块内部



C、两个模块有一部分代码重叠



D、一个模块有多入口模块



内聚性：模块内部个元素之间联系的紧密程度。

内聚性有六种类型：偶然内聚、逻辑内聚、时间内聚、通信内聚、顺序内聚、功能内聚。

偶然内聚指一个模块内的各处理元素之间没有任何联系。这是内聚程度最差的内聚。

逻辑内聚

指模拟内执行几个逻辑上相似的功能，通过参数确定该模块完成一个功能。

把需要同时执行的动作组合在一起形成的模块为时间内聚模块。

重构：不改变组件功能和行为条件下，简化组件设计

模块独立性强 = 高内聚低耦合

界面设计：

允许用户操作控制、减少用户记忆负担、保持界面一致

环境分析确定了用户接口操作的物理结构和社会结构

面向过程的系统设计=数据设计+体系结构设计+接口设计+组件设计

面向过程的系统设计

数据处理问题典型的类型：变换型和事务型

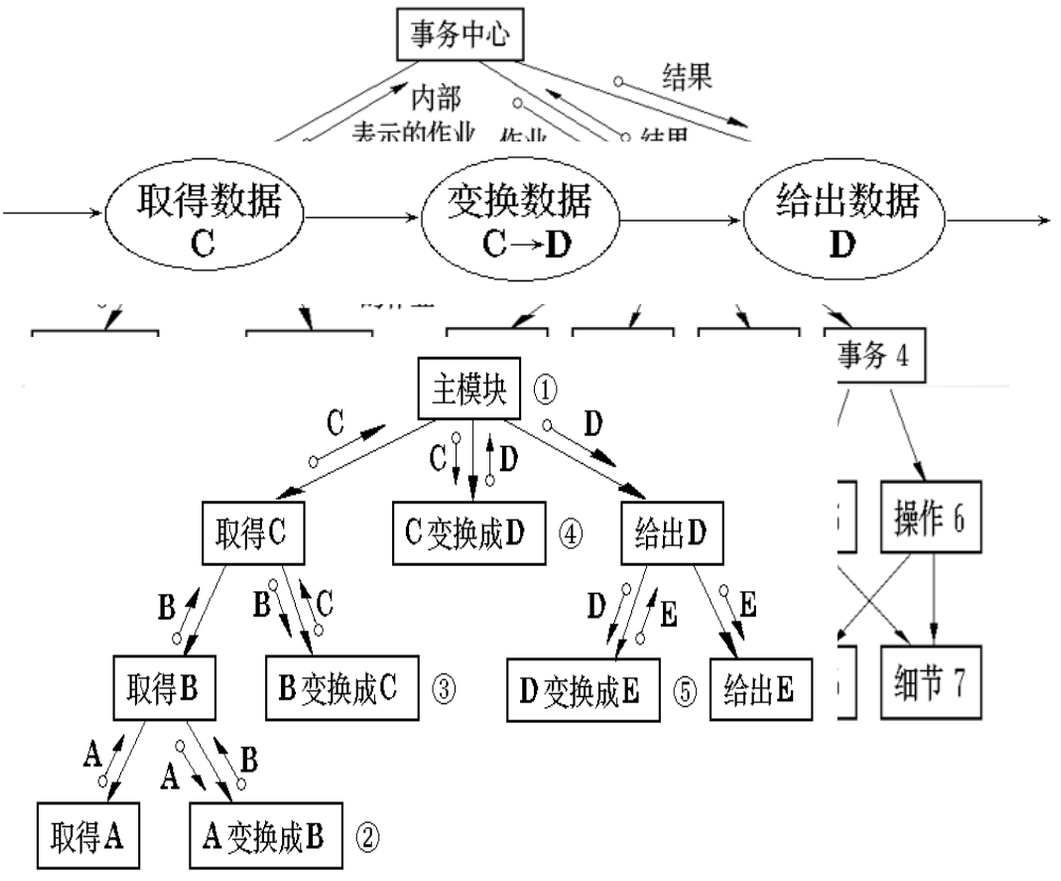
变换型系统结构图（好难啊。。）

-

在变换型DFD中，变换是系统的主加工，变换输入端的数据流称为系统的逻辑输入，输出端的数据流为逻辑输出。而直接从外部设备输入数据称为物理输入，直接从外部设备上的输出数据称为物理输出。

事务型系统结构图

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。



结构化组件设计

组件级设计也成为过程设计，位于数据设计、体系结构设计、接口设计完成之后

- 任何程序总可以用三种结构化的构成元素来设计和实现

顺序：任何算法规约中的核心处理步骤

条件：允许根据逻辑情况选择处理的方式

重复：提供了循环

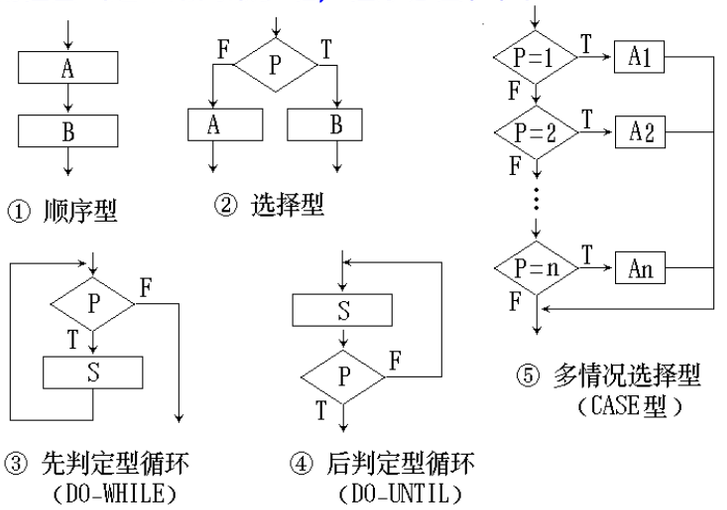
详细设计工具：

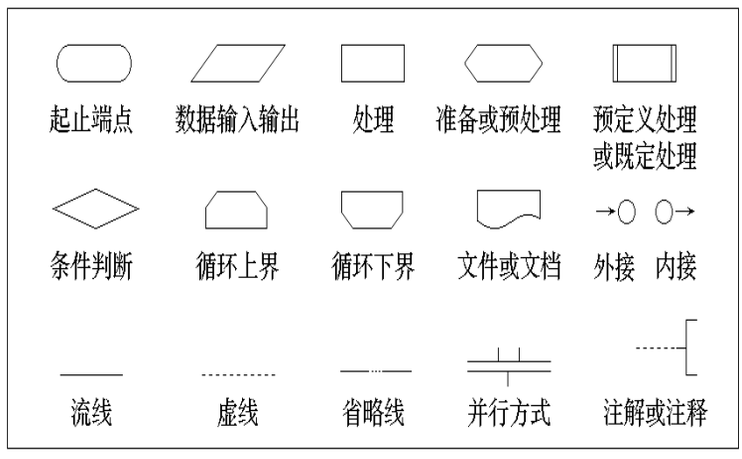
图形设计符号：流程图、盒图等

表格设计符号：决策表等

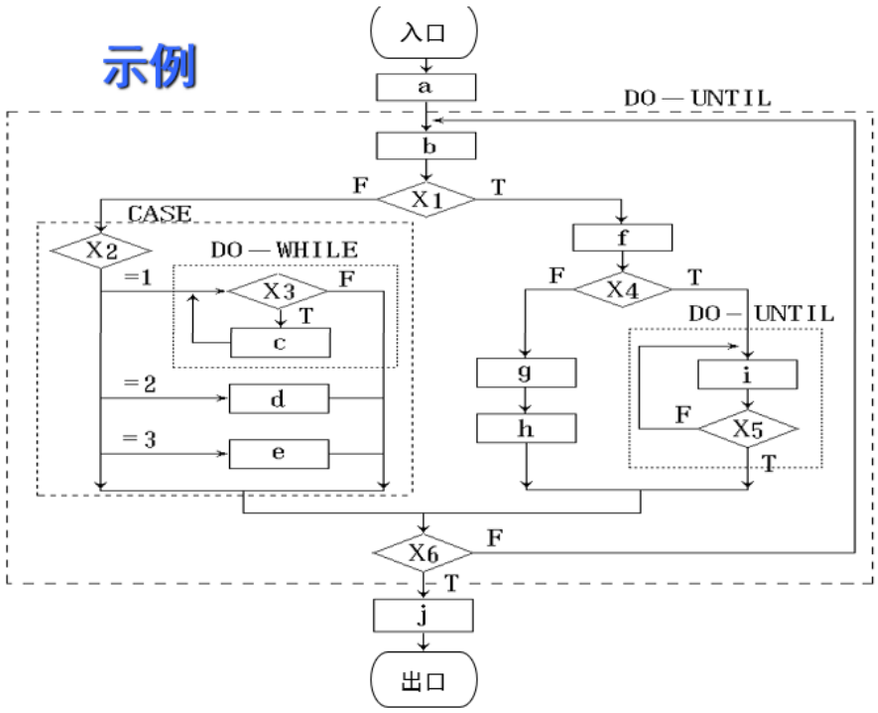
程序设计语言：PDL等

流程图（这个用的最多啦，程序必画。。）

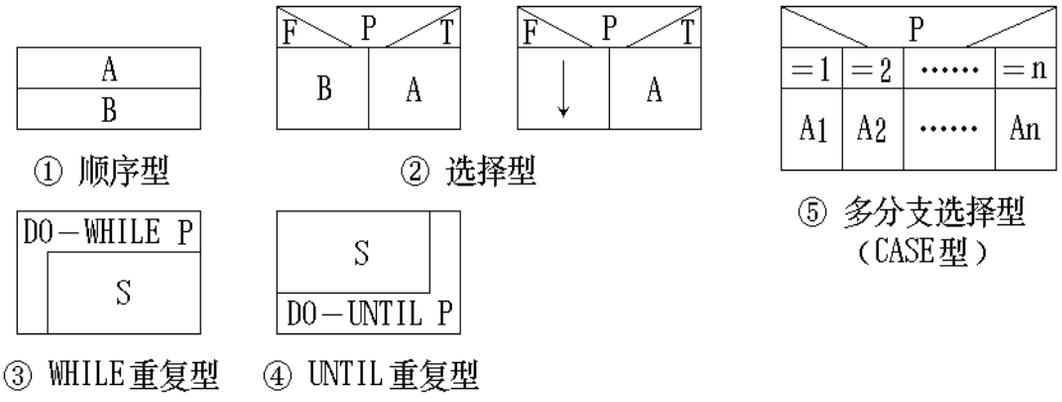


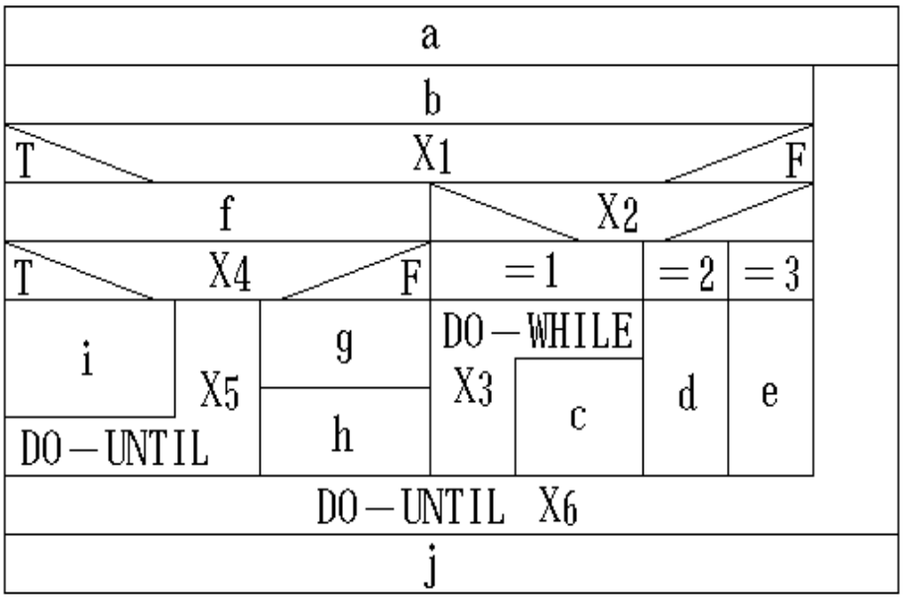


示例


入口
a
DO — UNTIL
b
F
X1
T
CASE
X2
DO — WHILE
=1
X3
F
T
c
=2
d
=3
e
f
F
X4
T
DO — UNTIL
g
h
i
F
X5
T
X6
F
T
j
出口

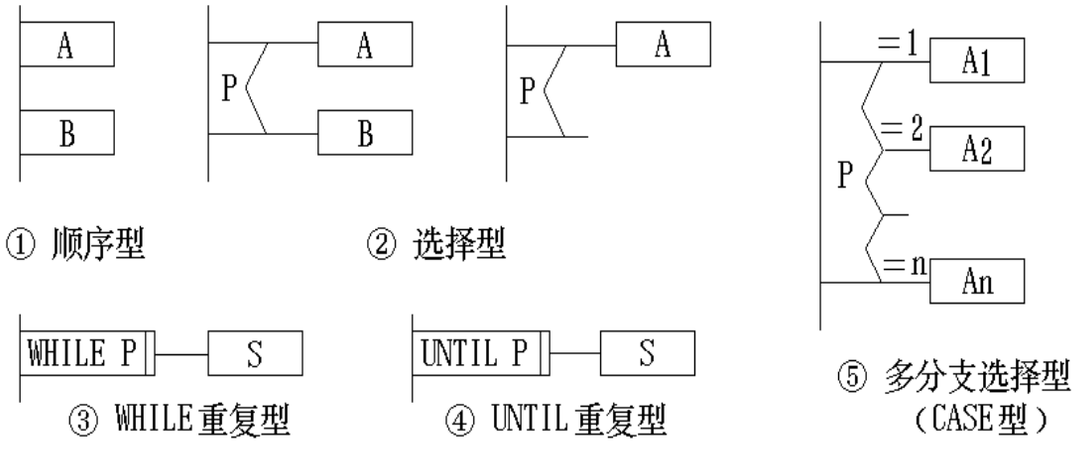
N-S图


A
B
① 顺序型
F P T
B A
F P T
A
② 选择型
P
=1 =2 …… =n
A1 A2 …… An
⑤ 多分支选择型
（CASE型）
DO－WHILE P
S
③ WHILE重复型
S
DO－UNTIL P
④ UNTIL重复型



感脚这个不会考啦

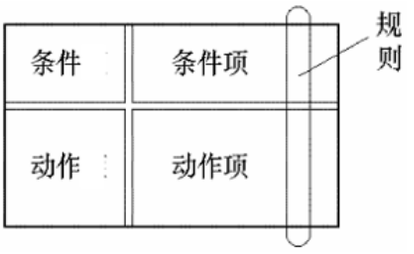
问题分析图(PAD)。。不会考啦。。



判定表（我觉得这个不会考啊，不想贴上去占空间了，你知道有它就行了）

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

PDL 就是伪代码啦!

面向对象的系统设计

类内聚——

设计类的原则是一个类的属性和操作全部都是完成某个任务所必须的，其中不包括无用的属性和操作

- 交互耦合——
  如果对象之间的耦合是通过消息连接来实现的，则这种耦合就是交互耦合。在设计时应该尽量减少对象之间发送的消息数和消息中的参数个数，降低消息连接的复杂程度。
- 继承耦合——
  继承耦合是一般化类与特殊化类之间的一种关联形式，设计时应该适当使用这种耦合。在设计时要特别认真分析一般化类与特殊化类之间继承关系，如果抽象层次不合理，可能会造成对特殊化类的修改影响到一般化类，使得系统的稳定性降低。另外，在设计时特殊化类应该尽可能多地继承和使用一般化类的属性和服务，充分利用继承的优势。

强内聚+弱耦合

构架设计——组件图，部署图

通常按以下3个步骤进行

第1步 构造系统的物理模型

第2步 设计子系统

第3步 非功能需求设计

第一步： UML的部署图（配置图）描述物理模型。
第二步：UML组件图 按功能划分、按物理布局划分、按软件层次划分
解决子系统关系过于密切的方法：重新划分子系统、定义子系统的接口

用例设计（进一步细化）：
组件级设计——
详细设计类，对类的操作等进行说明（细化活动图、时序图、状态图等）

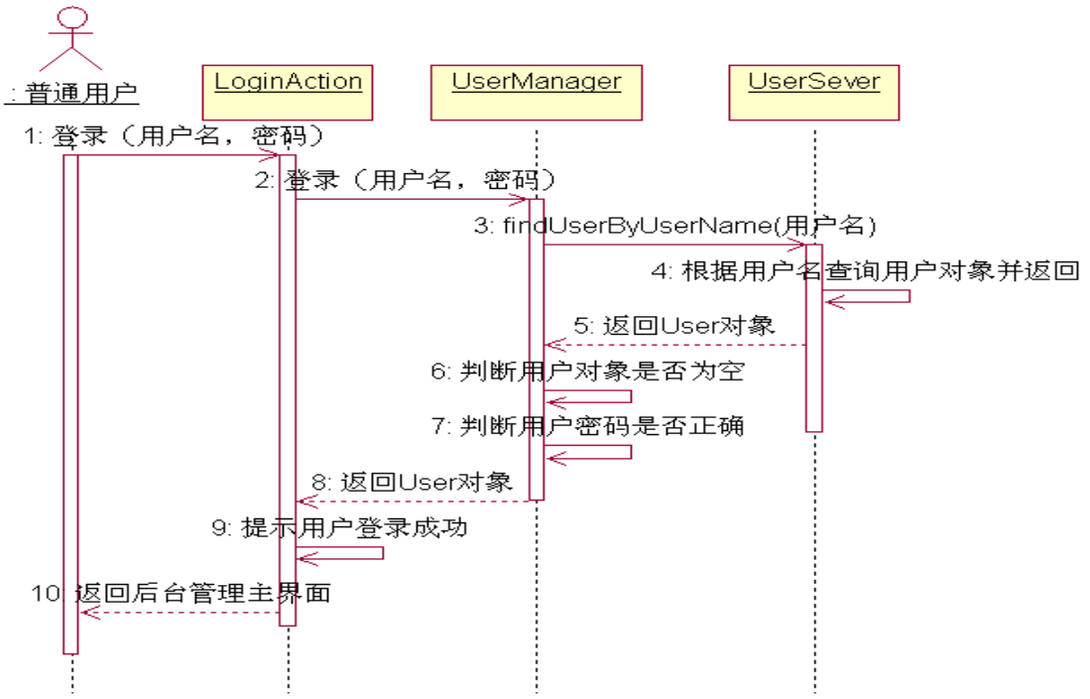
每个控制类还可以分别画出子顺序图

顺序图是强调消息时间顺序的交互图。
顺序图描述了对象之间传送消息的时间顺序
，用来表示用例中的行为顺序。

顺序图将交互关系表示为一个二维图。即在图形上，顺序图是一张表，其中显示的对象沿横轴排列，从左到右分布在图的顶部；而消息则沿纵轴按时间顺序排序。创建顺序图时，以能够使图尽量简洁为依据布局。

将对象置于顺序图的顶部意味着在交互开始的时候对象就已经存在了，如果对象的位置不在顶部，那么表示对象是在交互的过程中被创建的。
结构化程序设计中，模块间传递信息的方式主要是过程（或函数）调用。
顺序图中消息编号可显示，也可不显示。协作图中必须显示。
进行顺序图建模时，所用到的消息主要包括以下几种类型：

- 简单消息（Simple Message）
- 同步消息（Synchronous Message）
- 异步消息（Asynchronous Message）
- 反身消息（Message to Self）
- 返回消息 虚线（Return Message）



第五章（啦啦啦~这一章我没听，觉得不重要的样子……）

程序设计风格：

1）基本要求

程序结构清晰且简单易懂，单个函数的行数一般不要超过100行

算法设计应该简单，代码要精简

尽量使用标准库函数（类方法）和公共函数（类方法）

最好在表达式中使用括号以避免二义性

2）可读性要求

注释：程序头，函数头说明；借口说明；子程序清单；模块位置；开发历史；主要变量；不要对每条语句注释，保持注释和代码完全一致

格式：程序格式清晰，利用缩进，统一为4个字节

程序本身：语句力求简单、清晰；尽量使用三种基本控制结构编写程序；少使用含有“否定”运算符的条件语句；避免空ELSE语句和IF THEN
IF语句；避免过多循环嵌套和条件嵌套；数据结构利于程序简化；模块功能尽可能单一化。利用信息隐蔽确保每一个模块的独立性
数据说明：数据说明先后次序规范化

简单变量类型说明、数组说明、公用数据块说明、文件说明
整型量说明、实型量说明、字符量说明、逻辑量说明

3）正确性与容错性要求

4）可移植性要求

尽量使用语言的标准部分，避免使用第三方接口；对数据库操作尽量使用标准SQL数据类型和SQL语句，符合语言规范的标准接口类 JDBC

5）输入和输出要求

6）重用性要求

7）面向对象的程序设计风格

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。对效率无重要改善，不能牺牲程序的简单性、可读性和正确性。

程序设计语言基本成分分类：数据成分、运算成分、控制成分、传输成分。

发展历程：机器语言、汇编语言、高级程序设计语言、4GL

结构化程序设计原则：自顶向下、逐步细化、模块化

影响程序效率的因素：算法、存储、输入/输出。
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。

编码策略：自顶向下、自底向上、复合模式、线程模式

编码规范：一个团队、企业给出的内部开发最佳做法的建议、最好方式的推荐和经验总结。

编码规范的意义：
使开发人员有据可依、代码易读、易于测试和维护、易于定位错误和变更管理。

第六章（这个很重要啦~特别是白盒）

- 软件需求是软件质量测量的基础
- 缺乏规定的一致性就是缺乏软件的质量
- 制定的标准会定义软件工程发展的标准，它引导着软件工程经理

质量保证——
系统地监测和评估一个工程的各个方面，以最大限度地提高正在由生产过程中实现的质量的最低标准，
原则：

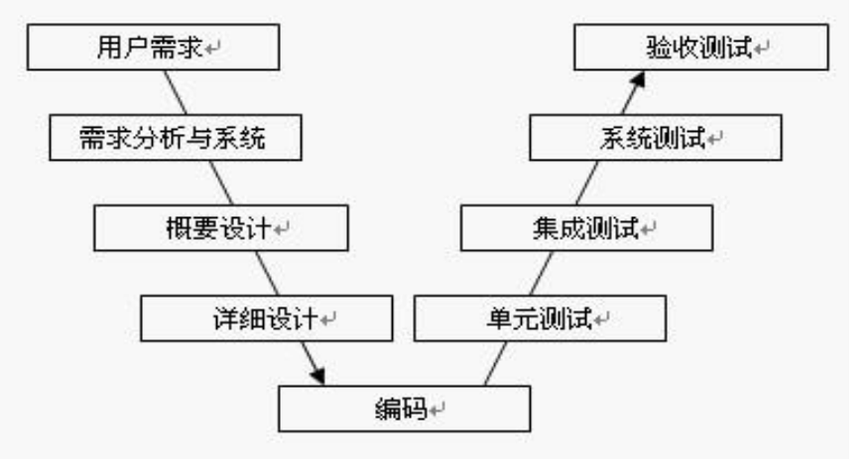
- “适合用途”：该产品应符合预期的目的
- “一次成功”：错误应该被淘汰

软件质量保证（SQA）——一个监控软件工程以确保软件质量的过程
SQA活动：审查、监督、审核

软件可靠性：

- 是指在给定时间内、特定环境下软件无错运行的概率

软件测试策略
过程模型
软件测试策略V模型



单元测试：保证每个模块作为一个单元是否能够按详细设计的规格说明正确运行
集成测试：检查多个模块间是否按概要设计说明的方式协同工作。重点在于测试模块间的接口。
系统测试：验证整个系统是否满足需求规格说明
验收测试：有用户参加的系统测试，验证是否满足用户需要

各阶段：
1单元测试=模块接口+局部数据结构+边界条件+独立路径+出错处理
针对软件设计的最小单位——程序模块。
主要依据：详细设计。
主要内容
:模块接口测试、局部数据结构测试、路径测试、错误处理测试、边界测试

2集成测试——自顶向下的集成方法、自底向上的集成方法、SMOKE方法

自顶向下（广度优先、深度优先）——
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集成。从属于主控模块的按深度优先方式（纵向）或者广度优先方式（横向）集成到结构中去。
优点——自顶向下的集成方式在测试过程中
较早地验证了主要的控制和判断点。
选用按深度方向集成的方式，可以首先实现和验证一个完整的软件功能。
缺点——桩模块的开发量较大

自底向上——
自底向上集成方法是从软件结构最底层的模块开始，按照接口依赖关系逐层向上集成以进行测试
优点——每个模块调用其他底层模块都已经测试，不需要桩模块
缺点——每个模块都必须编写驱动模块；缺陷的隔离和定位不如自顶向下。

Smoke方法——
将已经转换为代码的软件构件集成为构造（build）。一个构造包括所有的数据文件、库、可复用的模块以及实现一个或多个产品功能所需的工程化构件。

- 设计一系列测试以暴露影响构造正确地完成其功能的错误。其目的是为了发现极有可能造成项目延迟的业务阻塞（show stopper）错误。
- 每天将该构造与其他构造，以及整个软件产品集成起来进行冒烟测试。这种集成方法可以是自顶向下，也可以自底向上。

3系统测试（软件质量保证的最重要环节）——
功能性测试、性能测试、压力测试、恢复测试、安全测试，其他的系统测试还包括配置测试、兼容性测试、本地化测试、文档测试、易用性测试等

- 系统测试基本上使用黑盒测试方法
- 系统测试的依据主要是软件需求规格说明

- 功能性测试：在规定的一段时间内运行软件系统的所有功能，以验证有无严重错误
- 性能测试：检查系统是否满足需求规格说明书中的性能，常与压力测试结合
- 压力测试：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。敏感性测试
- 恢复测试：克服硬件故障后，系统能否继续正常工作。
- 安全测试：检测系统的安全性、保密性措施是否发挥作用，有无漏洞

4.验收测试——以用户为主的测试。——α 测试和β 测试

α测试是由一个用户在开发环境下进行的测试。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。
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。开发这不在现场。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。Alpha测试达到一定的可靠程度时开始。

回归测试（范围：缺陷再测试，功能改变的测试，新功能测试，完全回归测试）
回归测试——指有选择地重新测试系统或其组件，以验证对软件的修改没有导致不希望出现的影响，以及系统或组件仍然符合其指定的需求。

软件缺陷：
软件未实现产品说明书要求的功能。
软件出现了产品说明书指明不能出现的错误。
软件实现了产品说明书未提到的功能。
软件未实现产品说明书虽未明确提及但应该实现的目标。
软件难以理解、不易使用、运行缓慢或者——从测试员的角度看——最终用户会认为不好。

验证（正确地构造了产品吗）+确认（构造了正确的产品吗）
软件测试：找出软件缺陷，并确保修复。
软件质量保证：创建、执行改进开发过程并防止缺陷发生的标准和方法。

- 测试的目标是发现软件缺陷的存在
- 调试的目标是定位与修复缺陷

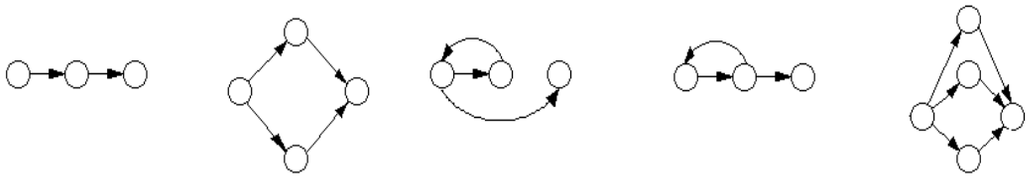
软件测试的评估准则
覆盖率，故障插入，变异分值
覆盖率——给定一个测试需求集合TR和一个测试集合T，覆盖率可以定义为T 满足的测试需求占TR 总数的比例。
故障插入——用于评价遗留在一个程序中的故障的数量和种类。
编译分值——
该指标与变异测试密切相关。变异测试：程序进行两个或更多个变异，然后用同样的测试用例执行测试，可以评估这些测试用例探测程序变异间的差异的能力

软件测试的主要方法：黑盒测试+白盒测试+灰盒测试

黑盒测试——忽略系统或组件的内部机制
，仅关注于那些响应所选择的输入及相应执行条件的输出的测试形式，也称
功能性测试或数据驱动测试。
白盒测试——考虑系统或组件的内部机制
的测试形式（如分支测试、路径测试、语句测试等），也称
结构性测试或逻辑驱动测试。
灰盒测试——
介于白盒测试和黑盒测试之间的一种测试行驶，多用于集成测试阶段，不仅关注输出、输入的正确性，同时也关注程序内部的情况。

白盒测试
逻辑覆盖——以程序内部的逻辑结构为基础
的设计测试用例的技术。属白盒测试。（语句覆盖、分支覆盖、条件覆盖、条件组合覆盖）
(看看例题啦！！！一定会考的，很重要啦)
1.语句覆盖——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。
2.分支覆盖——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
3.条件覆盖——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。
4.条件组合覆盖——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

控制流图覆盖测试——将代码转变为控制流图
（CFG），基于其进行测试的技术。它属白盒测试。

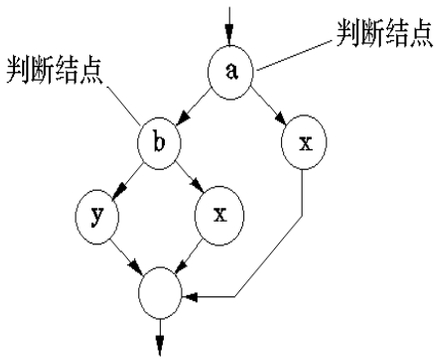


顺序结构　IF选择结构　WHILE重复结构　UNTIL重复结构　CASE多分支结构

- 如果判断中的条件表达式是由一个或多个逻辑运算符（OR，AND，NAND，NOR）连接的复合条件表达式，则需要改为一系列只有单个条件的嵌套的判断。

```
      .
      .
      .
if   a OR b
  then  procedure  x
  else  procedure  y;
      .
      .
      .
```

(a)



(b)

1.单条件嵌套

1.节点覆盖——即对于图G

中每个语法上可达的节点，测试用例所执行的测试路径的集合中至少存在一条测试路径访问该节点（节点覆盖和语句覆盖是等价的）。

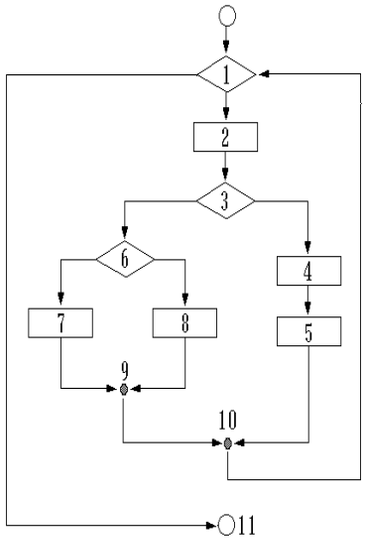
2.边覆盖——即对于图G　　中每一个可到达的长度小于等于1的路径，测试用例所执行的测试路径的集合中至少存在一条测试路径游历该路径（包含节点覆盖，也可以实现分支覆盖）。

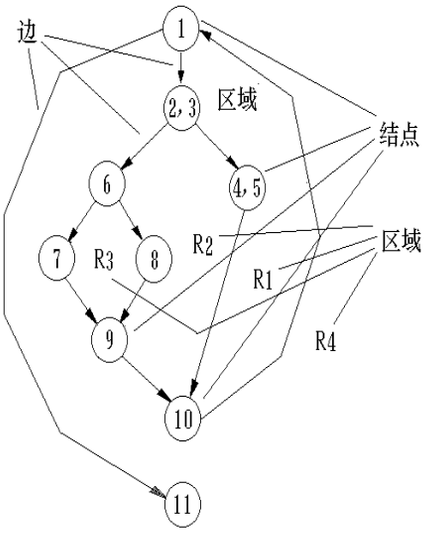
3.路径覆盖测试——设计足够的测试用例，覆盖程序中所有可能的路径。

4.基本路径测试方法——把覆盖的路径数压缩到一定限度内，程序中的循环体最多只执行一次。

环路的复杂性：V（G）=e-n+2

基本路径数等于复杂度





- 例如，在图示的控制流图中，一组独立的路径是
  path1：1-11
  path2：1-2-3-4-5-10-1-11
  path3：1-2-3-6-8-9-10-1-11
  path4：1-2-3-6-7-9-10-1-11
- 路径 path1，path2，path3，path4组成了控制流图的一个基本路径集。

黑盒测试——等价类划分、边界值分析、状态测试、静态分析法

- 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。

等价类划分方法——
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（等价类：某个输入域的子集合，在该子集合中，各个输入数据对于揭示程序中的错误是等价的）。
等价类划分原则：
(1)
如果输入条件规定了取值范围，或值的个数（范围），则可以确立一个有效等价类和两个无效等价类。
(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。
(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。
确立测试用例：
(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。

边界值分析法——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。

用例	A	B	C	预期输出
1	100	100	1	等腰三角形
2	100	100	2	等腰三角形
3	100	100	100	等边三角形
4	100	100	199	等腰三角形
5	100	100	200	非三角形
6	100	1	100	等腰三角形
7	100	2	100	等腰三角形
8	100	100	100	等边三角形
9	100	199	100	等腰三角形
10	100	200	100	非三角形
11	1	100	100	等腰三角形
12	2	100	100	等腰三角形
13	100	100	100	等边三角形
14	199	100	100	等腰三角形
15	200	100	100	非三角形

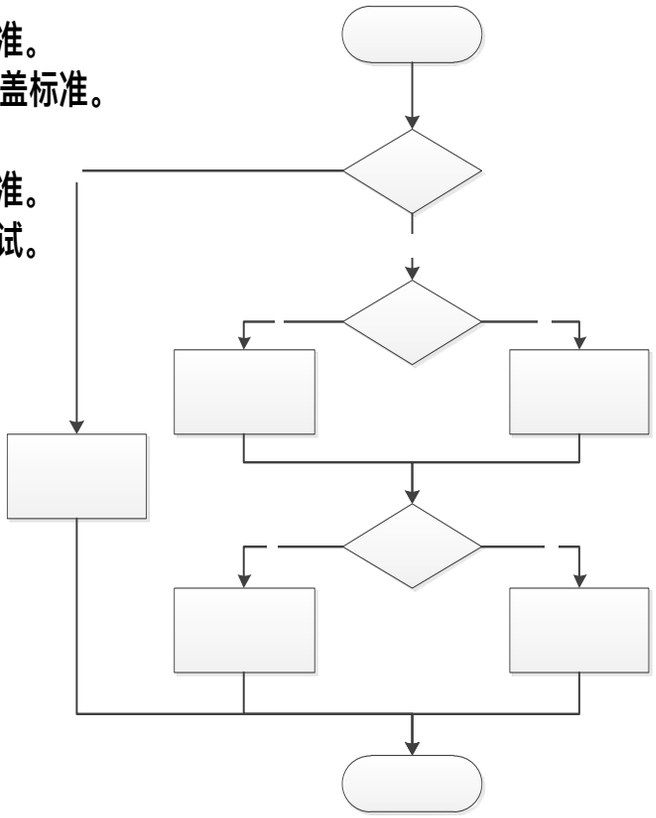
状态测试——建立状态转换图
静态分析方法——
不实际运行程序，通过检查和阅读等手段来发现错误并评估代码质量的软件测试技术。
检查需求、检查设计、检查代码

例题！！！：阅读下面的程序，并按要求进行解答。

```
    public String foo(int x, int y){
    boolean z;
    if(y>10)
    return "C";
if(x<y)
    z=true;
```

```
else
    z=false;
if(z&&x+y==10)
    return "A";
else
    return "B";
}
```

（1）设计测试用例，达到分支覆盖标准。
（2）设计测试用例，达到条件组合覆盖标准。
（3）绘出代码对应的CFG 图。
（4）设计测试用例，达到节点覆盖标准。
（5）设计测试用例，进行基本路径测试。



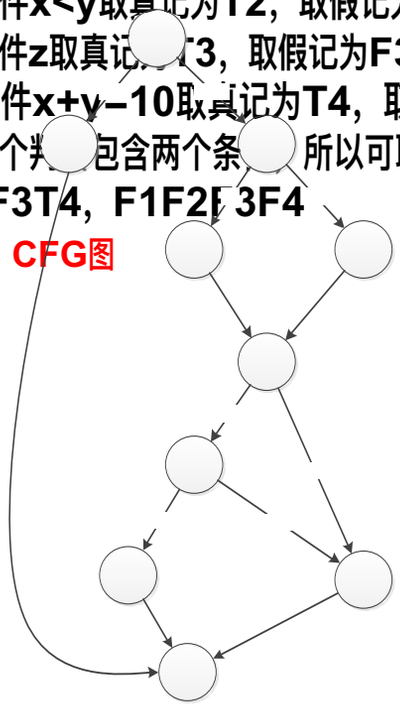
答:

（1）分支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

(2)

将所有条件编号：
对条件y>10取真记为T1，取假记为F1；
对条件x<y取真记为T2，取假记为F2；
对条件z取真记为T3，取假记为F3；
对条件x+y–10取真记为T4，取假记为F4。
第三个判断包含两个条件，所以可取四种组合：T1T2T3T4，F1T2T3F4，F1F2F3T4，F1F2F3F4

（3）CFG图



(4)

节点覆盖与语句覆盖等价，在该例中又与分支覆盖等价，故可用（1）中测试用例。

(5)

首先算环路复杂度V(G)=P+1=5；
然后确定线性独立路径的基本集合

基线路径L1(1-2-10)
L1回溯到1，得到L2(1-3-4-6-7-8-10)
L2回溯到7，得到L3(1-3-4-6-7-9-10)
L2回溯到6，得到L4(1-3-4-6-9-10)
L2回溯到3，得到L5(1-3-5-6-9-10)

，

第七章 软件维护

软件维护（耗钱最多）——
软件维护是指由于软件产品出现问题或需要改进而对代码及相关文档的修改，其目的是对现有软件产品进行修改的同时保持其完整性。

四种基本类型：纠错性维护+适应性维护+完善性维护+预防性维护（哪四种，背）
1.
纠错性维护（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）——
为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的误用，应当进行的诊断和改正错误的过程就叫做纠错性维护。
2.适应性维护——为使软件适应外部环境（新的硬、软件配置
），数据环境等变化，而去修改软件的过程
3.完善性维护（第一）——
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。
4.预防性维护——
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。

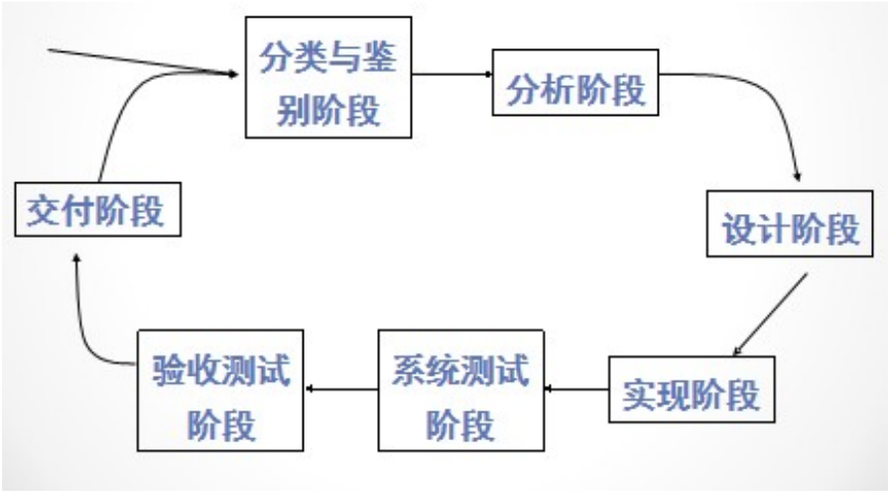
完善性维护占全部维护活动的50%～66%，纠错性维护占17%～21%，适应性维护占18%～25%，预防性维护活动只占4%左右（记住谁最多谁最少就OK了啦~）

软件维护的困难性——
配置管理工作不到位、许多软件可读性差、人员变动造成影响、任务紧时间急的情况下处理维护请求。

软件维护中应注意的问题：

软件可维护性的主要因素——
可理解性、可测试性、可修改性、可移植性、可重用性
影响软件可维护性的环境因素——软件维护的文档
、软件的运行环境、软件的维护组织、软件维护质量。

软件维护过程模型（IEEE维护模型图）——7个阶段



软件维护技术（程序的理解、软件再工程、软件逆向工程）
程序理解的任务：以软件维护、升级和再工程为目的，在不同的抽象级别上建立基本软件的概念模型，包括从代码本身的模型到基本应用领域的模型，即建立从问题/应用域到程序设计/实现域的映射集
软件再工程：指对现有软件进行仔细审查和改造，对其进行重新构造，使之成为一个新的形式，同时包括随之产生的对新形式的实现。（
库存目录分析+文档重构+逆向工程+代码重构+数据重构+正向工程）

软件逆向工程——
分析目标系统，识别系统的构件及其交互关系，并且通过高层抽象或其他形式来展现目标系统的过程
逆向工程的主要内容——处理的逆向工程、用户界面的逆向工程、数据的逆向工程。

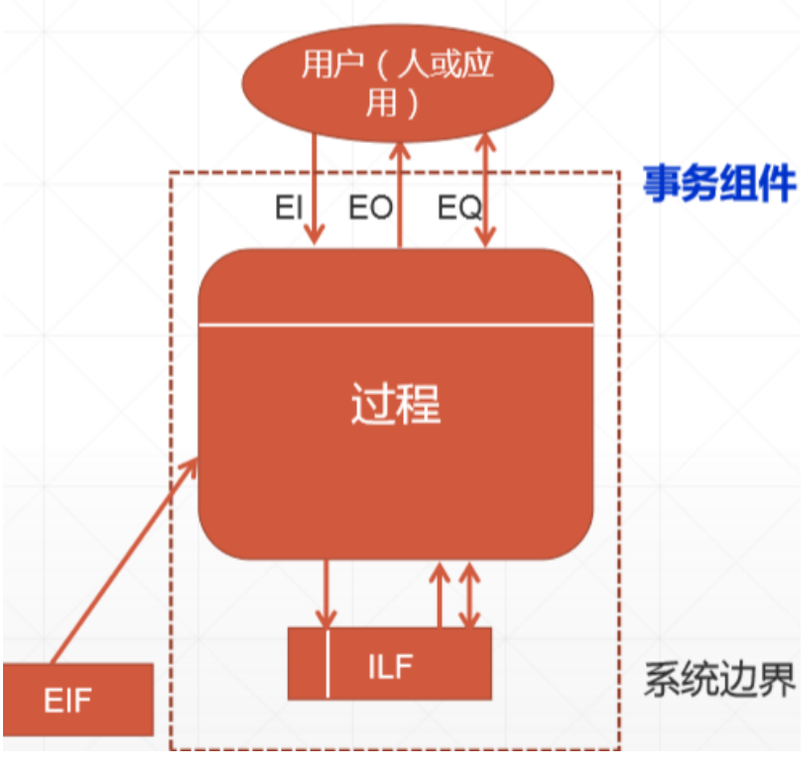
还有什么可以当考点的么。。我不知道了啦。。都是个人看法，没划准不要打我啦！！
再在网上找了点题，做一做找感觉吧（答案有的是错的样。。。。。）

第八章（感脚这一章有点酱油，只有计算可能出个题啦~）

软件项目管理的四大要素：
人员(People)、产品(Product)、过程(Process)、项目(Project)
软件度量（生产率度量、质量度量）：
面向规模的度量（直接测量）
生产率测量（基于规模KLOC）
每KLOC（千行代码）的错误数，即总错误数除以总KLOC
每KLOC（千行代码）的缺陷数，即总缺陷数除以总KLOC
每KLOC（千行代码）的文档页数，即总文档页数除以总KLOC
基于代码行：
优点
简单易行，自然直观
缺点
依赖于程序设计语言的表达能力和功能
软件开发初期很难估算出最终软件的代码行数
对精巧的软件项目不合适
例如，项目计划人员难于在分析和设计完成之前估算LOC

面向功能的度量（功能点）（间接测量）
项目开发初期就可估算

UFC相关的五类组件：内部逻辑文件ILF、外部接口文件EIF、外部输入EI、外部输出EO、用户查询EQ。



FP：
$FP = UFC \times TCF = UFC \times (0.65 + 0.01 \times \sum Fi)$
UFC：未调整功能点计数；TCF：技术复杂因子

（total_counts：总计数。。Fi (i取1到14)：复杂性调整值）
每FP的错误数，即总的错误数除以总的FP数。
每FP的缺陷数，即总的缺陷数除以总的FP数。
每FP的文档页数，即总的文档页数除以总的FP数。
每人月的FP数，即总的FP数除以总的人月数

优点：与程序设计语言无关，在开发前就可以估算出软件项目的规模
缺点：
没有直接涉及算法的复杂度，不适合算法比较复杂的软件系统
功能点计算主要靠经验公式，主观因素比较多

软件估算
策略
项目度量方法为项目估算提供了依据与有效输入
尽量把估算推迟到项目的后期进行
根据已经完成的项目进行估算

基于分解技术的项目估算方法

三点期望法
估计期望值=(最大值+4×最可能值+最小值) / 6

基于经验的项目估算方法

算法成本模型——
在软件过程的推进过程中，可利用的信息越多，估算的精确度越高
COCOMO(构造性成本模型)是一个经验模型

- COCOMO模型的层次 - 支持不同的阶段
  - 基本COCOMO模型
    - 系统开发的初期，估算整个系统的工作量(包括维护)和软件开发和维护所需的时间
  - 中间COCOMO模型
    - 估算各个子系统的工作量和开发时间
  - 详细COCOMO模型
    - 估算独立的软构件，如各个子系统的各个模块的工作量和开发时间

- 基本COCOMO模型
  - $E = a * (KLOC)^b$；E是工作量(人月)，a和b是经验常数
  - $D = c * E^d$；D是开发时间(月)，c和d是经验常数
  - 其中，a,b,c,d为经验常数，其取值见下表

软件类型	a	b	c	d	适用范围
组织型	2.4	1.05	2.5	0.38	各类应用程序
半独立型	3.0	1.12	2.5	0.35	各类编译程序等
嵌入型	3.6	1.20	2.5	0.32	实时软件、OS等

- 中间COCOMO模型
  - $E = a * (KLOC)^b * EAF$
  - 其中，E表示工作量(人月)，EAF表示工作量调节因子，a,b为经验常数，其取值见下表

软件类型	a	b
组织型	3.2	1.05
半独立型	3.0	1.12
嵌入型	2.8	1.20

项目进度计划概念与可视化

项目进度计划的价值

- 有序、可控制地对软件项目进行管理
- 确保员工保持高生产率
- 及时交付软件产品
- 降低软件开发成本
- 提高客户满意度
- 及时发布产品新版本

工作分解结构（WBS）

工作分解结构（Work Breakdown Structure, WBS）是将项目按照功能或过程进行逐层分解，直到划分为若干内容单一、便于组织管理的单项工作，最终形成的树形结构示意图。

基于功能分解

基于过程分解

WBS构建应该注意的原则：

- 一个任务只应该在WBS中的一个地方出现
- WBS中某项任务的内容是其下所有WBS项的总和
- 一个WBS项只能由一个人责任，其他人只能是参与者
- WBS必须与实际工作中的执行方式一致
- 应让项目团队成员积极参与创建WBS，以确保WBS的一致性
- 每个WBS项都必须文档化，以确保准确理解已包括和未包括的工作范围
- WBS可以根据需求进行必要变更维护

任务网络图

任务网络图是项目所有任务（活动）及其之间逻辑关系（依赖关系）的一个图解表示，并从左到右来表示项目的时间顺序。

关键路径的意义：

- 关键路径上任何任务（活动）的延长都会导致整个项目周期的延长
- 如果想缩短项目周期，就必须缩短关键路径的长度
- 项目经理应该随时关注关键路径上任务（活动）的完成情况以及关键路径是否发生了变化
- 对WBS中任务的串行与并行安排方式有指导意义