



软件工程与方法

第6讲 软件测试



主要内容

1. 软件测试的概念
2. 软件测试分类
3. 软件测试策略
4. 测试用例设计
5. 软件测试工具



事故案例：阿丽亚娜5火箭事故

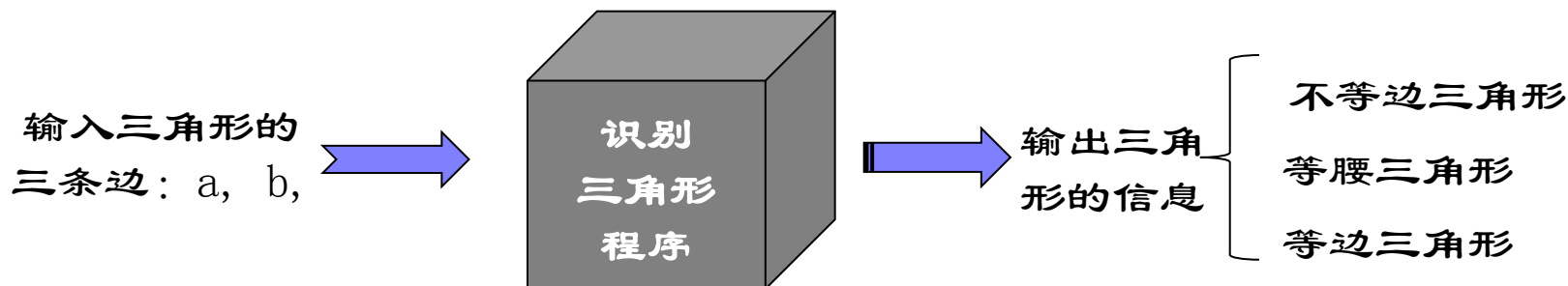


- 现象：1996年法国的阿丽亚娜5火箭首次发射时，发动机点火后37秒火箭爆炸，造成直接经济损失5亿美元，研制计划推迟一所。
- 原因：软件将表示为火箭水平速度的64位浮点数转换成16位有符号整数时，产生了溢出，引擎口偏转到极限状态。
- 工作失误：重用软件使用条件发生了变化，没有重新分析和验证。

课堂练习题

■ 设计测试用例

Myers用以下程序对测试能力进行自我评估：某程序读入3个数值，这3个数值表示三角形的三条边。该程序打印信息表明三角形是不等边三角形、等腰三角形或等边三角形，编写测试用例集测试该程序。

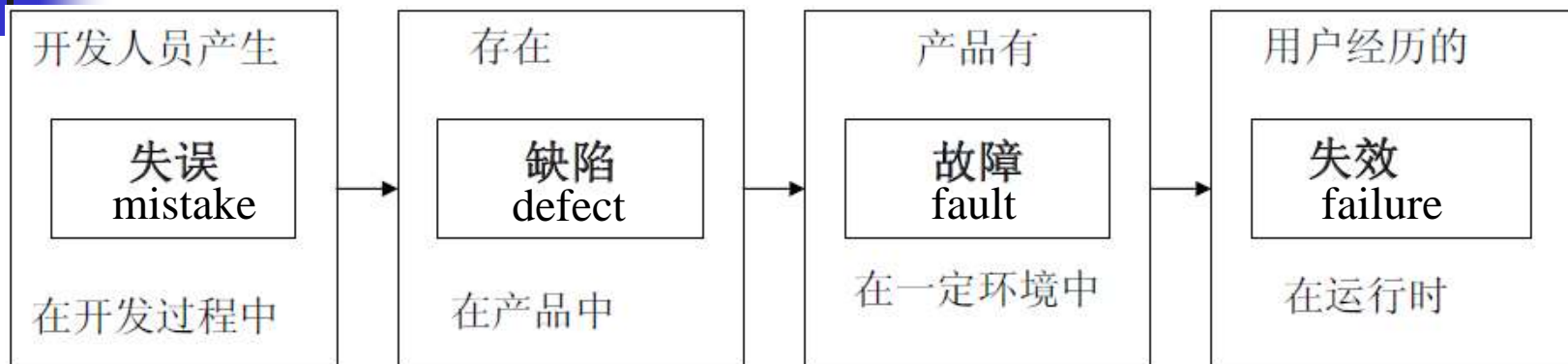




软件测试

1 软件测试的概念

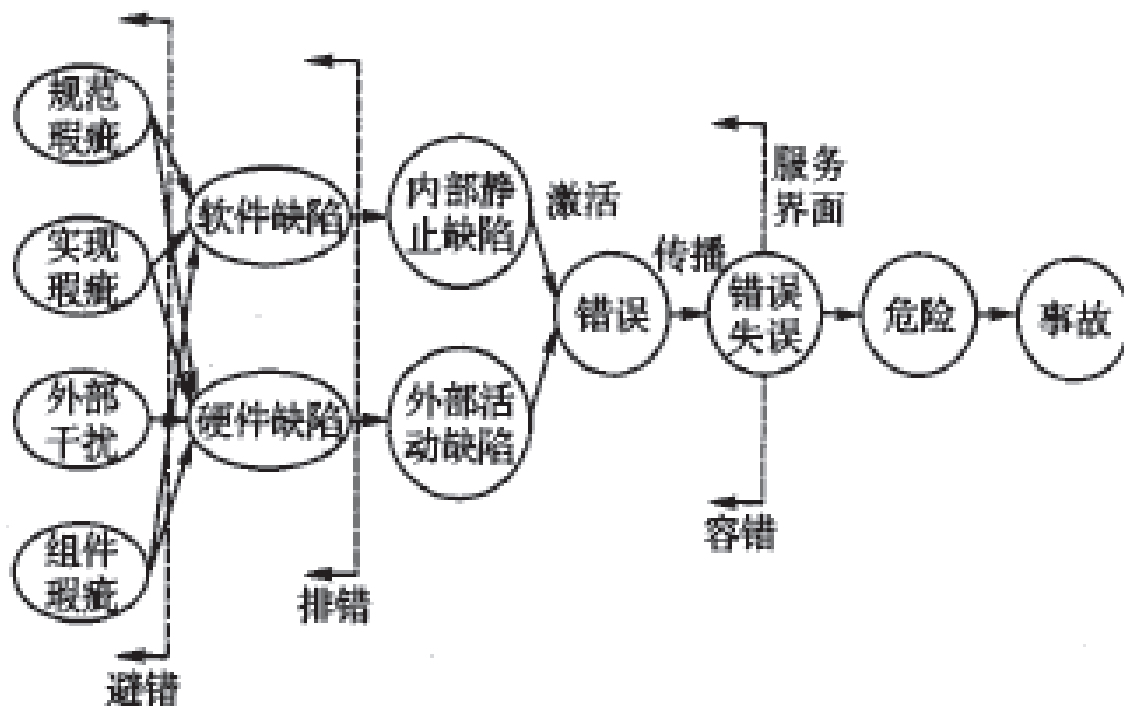
问题



为什么要进行软件测试?

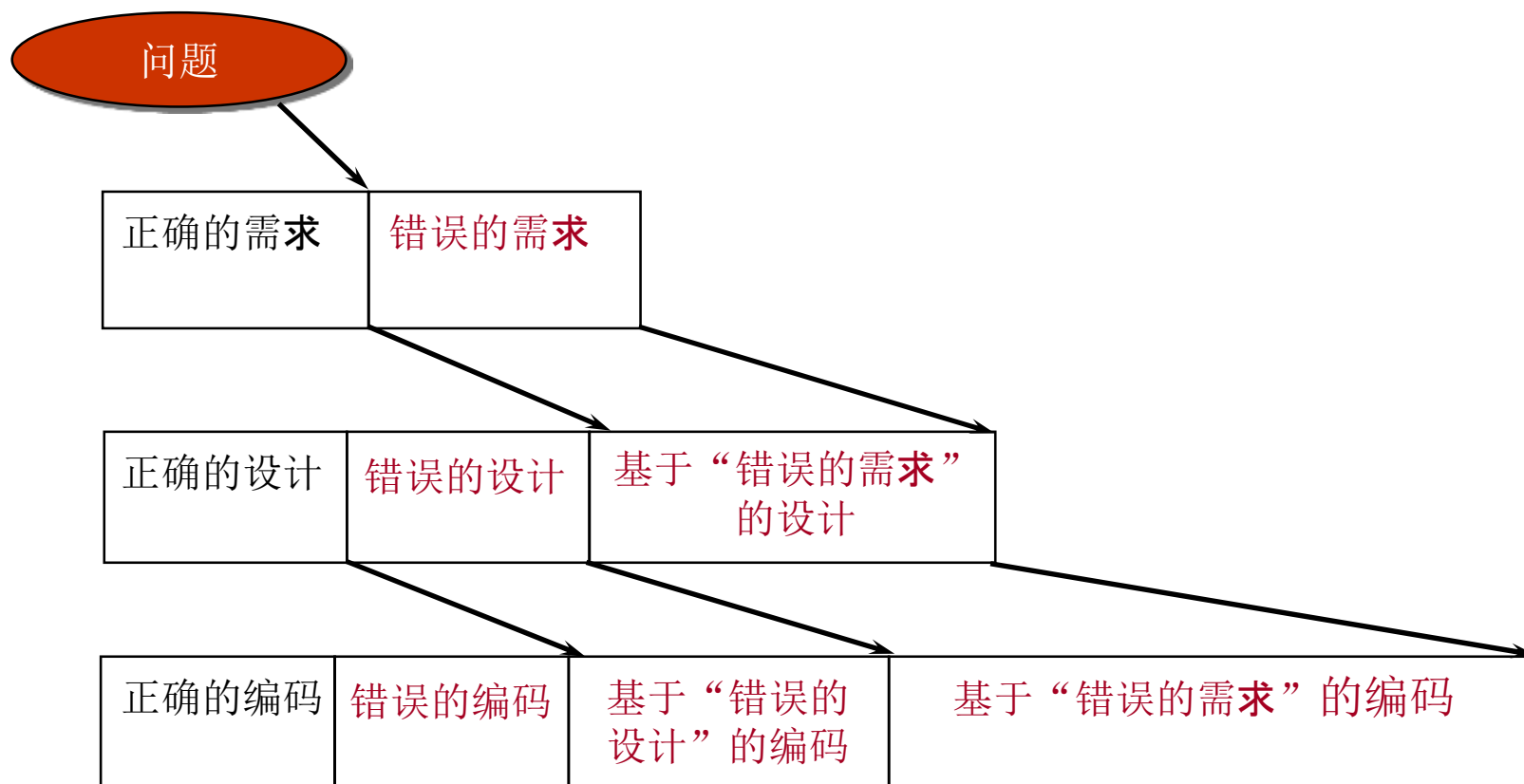
软件测试是为了发现软件产品所存在的软件缺陷（bug），从而纠正（fix）这些软件缺陷，使软件系统更好地满足用户的需求。

缺陷、错误及失效之间的关系



- 缺陷 (Fault)
- 错误 (Error)
- 失效 (Failure)

“错误的需求”的扩散效应



水波效应 (Water wave effect)

- 水波效应：指人们对程序的A处进行修改时引出B处出错误，修改B处时又影响到C处，以此类推形成的一石激起千层浪的连带影响的局面。
- 在软件维护过程中，由一个模块的修改而导致隐含缺陷、错误的放大以及一连串新错误出现，形成“水波效应”。



软件中的Bug数量

- V.R. Basili和B.T. Perricone, 1984年研究成果:
 - Code contains between 6 and 16 bugs per 1,000 lines of executable code
- T.J. Ostrand和E.J. Weyuker, 2002年研究成果:
 - The fault density at two to 75 bugs per 1,000 lines of executable code, depending on module size.
- 保守估计, 按照1000行代码含有6个bug, Linux的kernel约有15,000个bugs, Windows XP的bug数至少两倍于这个数。





测试的定义

- 传统：

- 测试是一种旨在评估一个程序或系统的属性或能力，确定它是否符合其所需结果的活动。

- *Grenford J. Myers*:

- 测试是为了发现错误而执行一个程序或系统的过程。

- *IEEE*:

- 测试是使用人工和自动手段来运行或检测某个系统的过程，其目的在于检验系统是否满足规定的需求或弄清预期结果与实际结果之间的差别。



测试目的

证明程序的可信性

- Grenford J. Myers提出的测试目的
 - 测试是程序的执行过程，目的在于发现错误；
 - 一个好的测试用例在于能够发现至今未发现的错误；
 - 一个成功的测试是发现了至今未发现的错误的测试。

软件测试的一般原则

- 尽早和不断地进行软件测试
- 测试用例合理
- 程序员避免检查自己的程序
- 用例应包括不合理的输入条件
- 注意错误的群集现象(80%错误来自20%的模块)
- 严格执行测试计划
- 对测试结果作全面检查
- 妥善保存测试计划和用例。

Davie测试原则

- 所有测试都应追溯到用户需求
- 应在测试前较早时间就制定测试计划
- 应用Pareto的原则到测试中
- 测试从“小规模”到“大规模”
- 穷举测试是不可能的
- 应该由第三方来构造测试

软件可测试性 (testability)

- 可操作性

Operability—it operates cleanly

- 可观察性

Observability—the results of each test case are readily observed

- 可控制性

Controlability—the degree to which testing can be automated and optimized

- 可分解性

Decomposability—testing can be targeted

- 简单性

Simplicity—reduce complex architecture and logic to simplify tests

- 稳定性

Stability—few changes are requested during testing

- 易理解性

Understandability—of the design

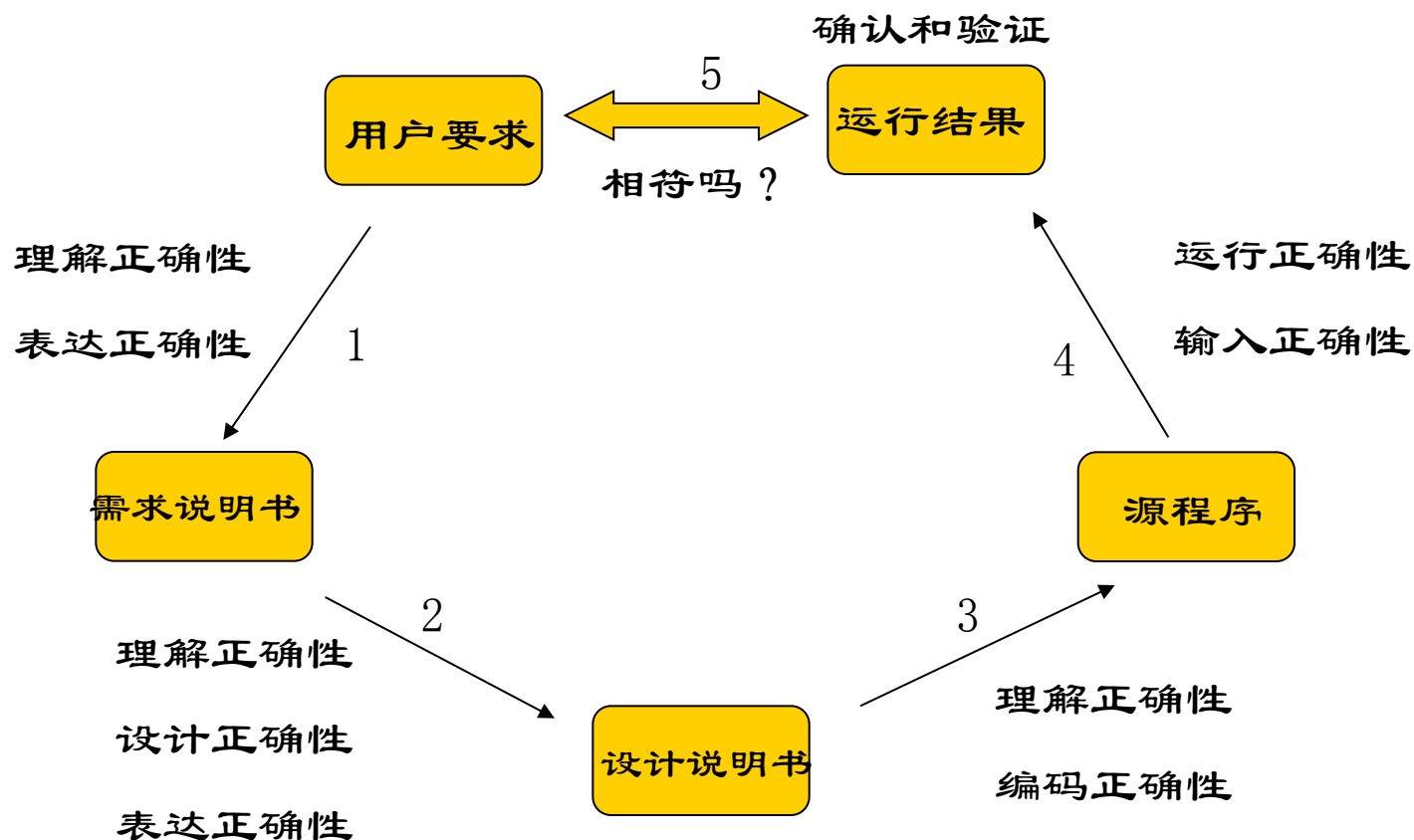
如何提高软件的可测试性？



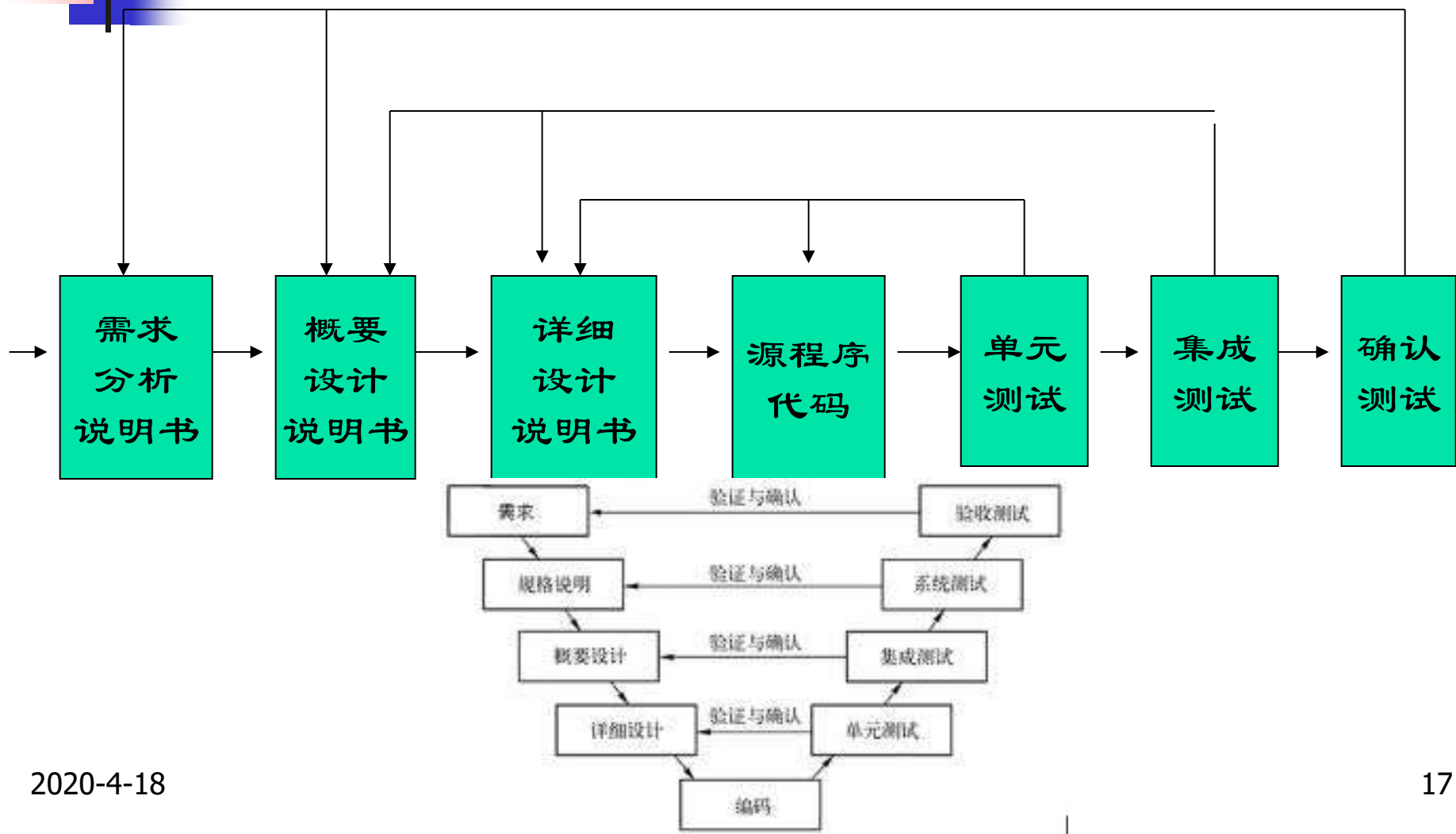
软件测试

2 软件测试活动

软件测试的对象



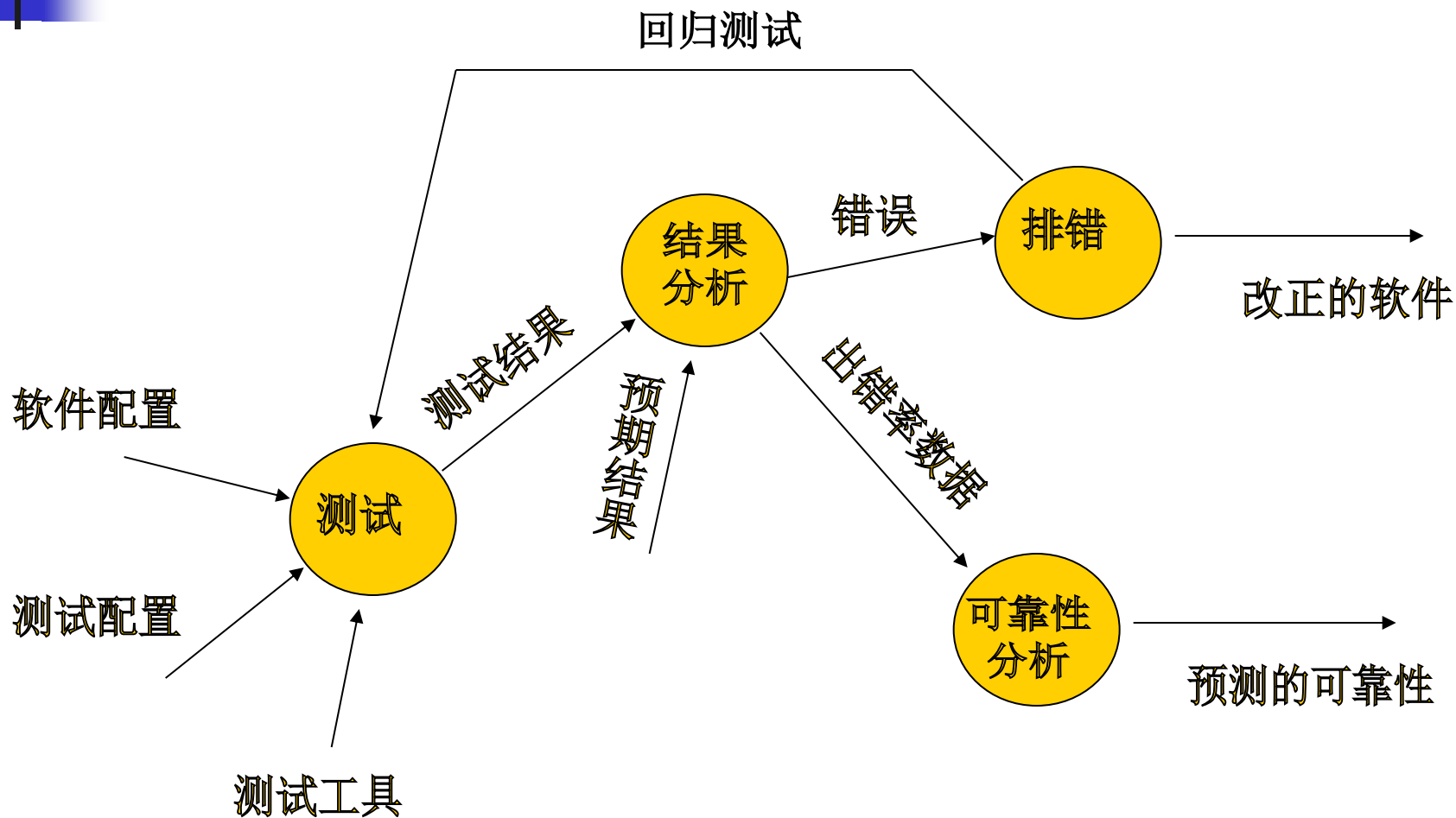
测试与软件开发各阶段的关系



软件测试活动

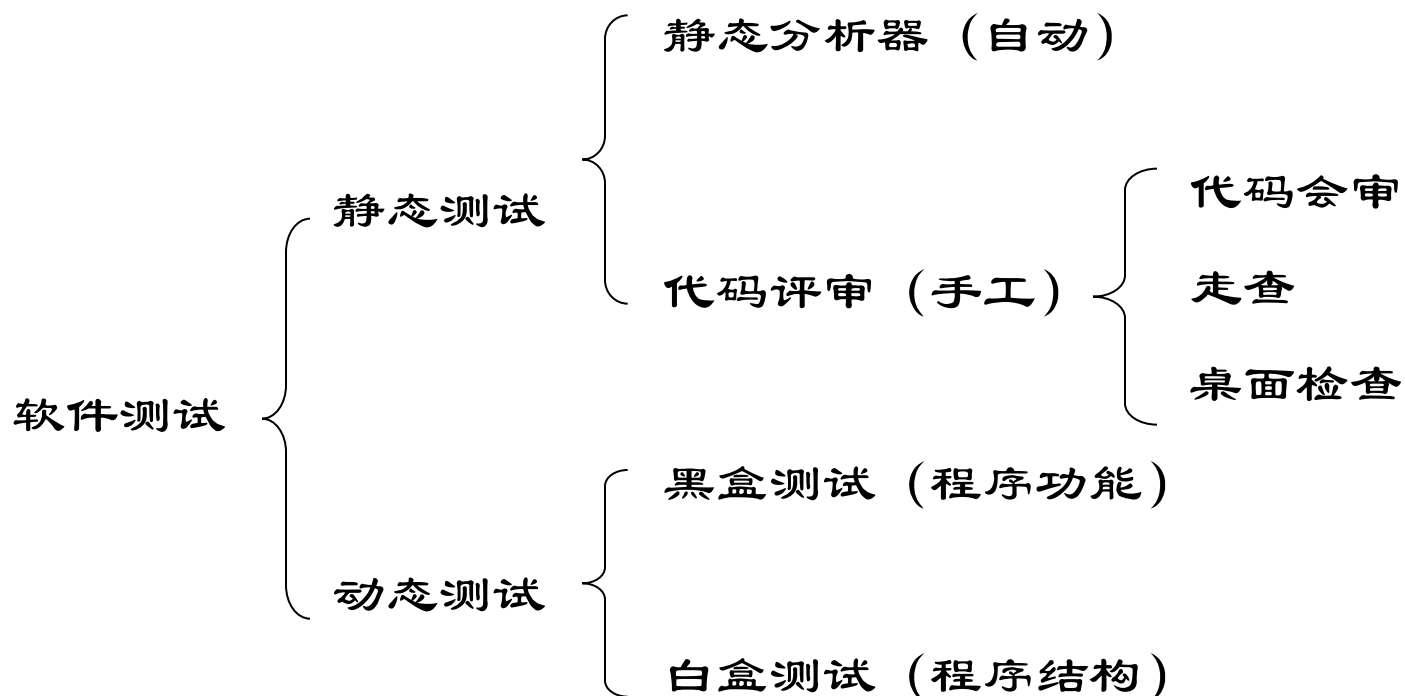
需求阶段	设计阶段	实现阶段	测试阶段	验收阶段
<ul style="list-style-type: none"> 用例情景测试 原型走查 模型走查 需求评审 	<ul style="list-style-type: none"> 模型走查 原型走查 设计评审 	<ul style="list-style-type: none"> 代码走查 接口分析 文档评估 		
<ul style="list-style-type: none"> 制定测试计划 	<ul style="list-style-type: none"> 制定测试计划 测试设计 	<ul style="list-style-type: none"> 编写测试用例 制定测试过程 单元测试 	<ul style="list-style-type: none"> 制定测试过程 集成测试 系统测试 	<ul style="list-style-type: none"> α 测试 β 测试 验收测试
回归测试，质量保证				

软件测试信息流





软件测试类型





软件测试

3 软件测试文档



测试计划 (*Test Plan*)

■ 作用：

- 是测试工作的指导性文档，规定测试活动的范围、方法、资源和进度；
- 明确正在测试的项目、要测试的特性、要执行的测试任务、每个任务的负责人，以及与计划相关的风险。

■ 主要内容：

- 测试目标、测试方法、测试范围、测试资源、测试环境和工具、测试体系结构、测试进度表



测试规范 (*Test Specification*)

■ 作用：

- 是从整体上规定测试案例的运行环境、测试方法、生成步骤、执行步骤以及调试和验证的步骤。

■ 内容：

- 系统运行环境、总体测试方法、测试用例的生成步骤、测试用例的执行步骤、调试和验证。

测试用例 (*Test Case*)

■ 作用：

- 是数据输入和期望结果组成的对；
- 指出使用具体测试案例产生的测试程序的任何限制。

■ 测试用例可以被组织成一个测试系列，用来实现某个特定的测试目。

例如：

- 测试兼容性、安全性、可用性等
- 测试特定功能

测试用例模版

系统名称			版本		
用例标识			模块名		
设计者		时间		测试方法	
用例简介					
测试步骤					
测试输入					
测试输出或预期结果					
测试通过标准					



缺陷报告 (*Bug Report*)

■ 作用：

- 编写在需要调查研究的测试过程期间发生的任何事件。
- 简而言之，就是记录软件缺陷。

■ 主要内容：

- 缺陷编号、题目、状态、提出、解决、所属项目、测试环境、缺陷报告步骤、期待结果、附件

■ 在报告缺陷时，要讲明缺陷的严重性和优先级。

- 严重性表示软件的恶劣程度，反映对产品和用户的影响。
- 优先级表示修复缺陷的重要程度和应该何时修复。



示例：缺陷类型

Defect Name	Description
Documentation	Comments, software unit description
Build, package	Change management, library, version control
Assignment	Declaration, duplicate name, scope, limit
Design Entities Interface	Procedure calls and reference, I/O, user formats
Checking	Error message, inadequate checks
Data	Structure, content
Function	Logic, pointer, loop, recursion, computation, function
System Behavioral	Configuration, timing, memory
Performance	Metric values for measurable attributes of the system, execution time, timing of event, etc,
Quality	Attributes of the system: usability, portability, reliability, maintainability
Constraints	Constraints under which the system must operate usually affected by policies or regulations such as security, safety, or restriction hardware, etc...
Software Interface	Data management systems, operating system, other applications
User Interface	Characteristics for the human/computer interaction: screen format, validation for user input, function availability, page layout, etc...
Norms	Notation for design entities representation



测试报告 (*Test Report*)

■ 作用

- 测试报告是指把测试的过程和结果写成文档，对发现的问题和缺陷进行统计分析。
- 为纠正软件的存在的质量问题提供依据，同时为软件验收和交付打下基础。

■ 内容

- 引言（目的、背景、缩略语、参考文献）
- 测试概要（测试方法、范围、测试环境、工具）
- 测试结果与缺陷分析（功能、性能）
- 测试结论与建议（项目概况、测试时间 测试情况、结论性能汇总）
- 附录（缺陷统计）



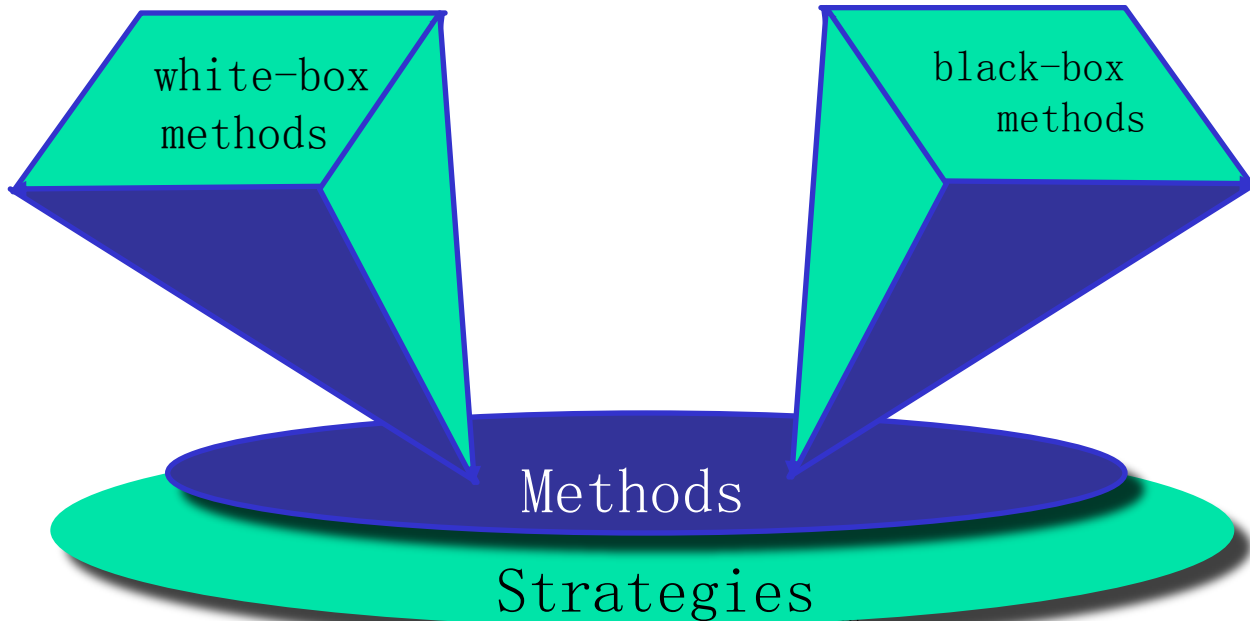
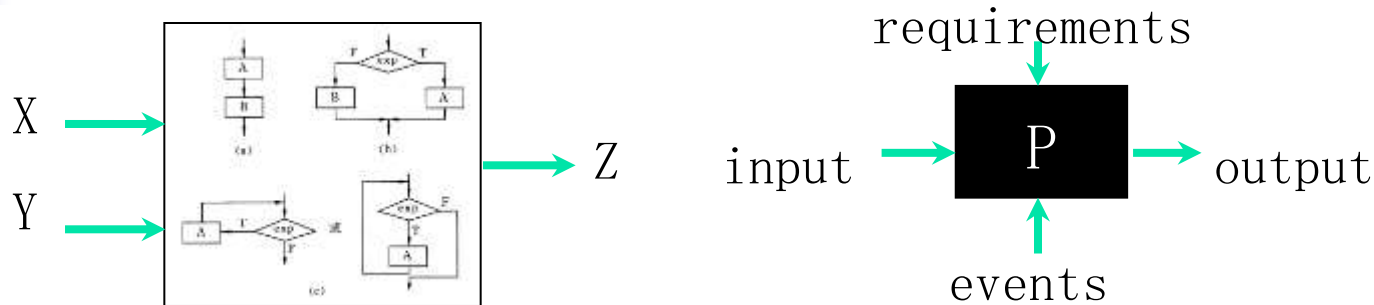
软件测试

4 测试用例设计

软件测试用例设计技术

白盒测试

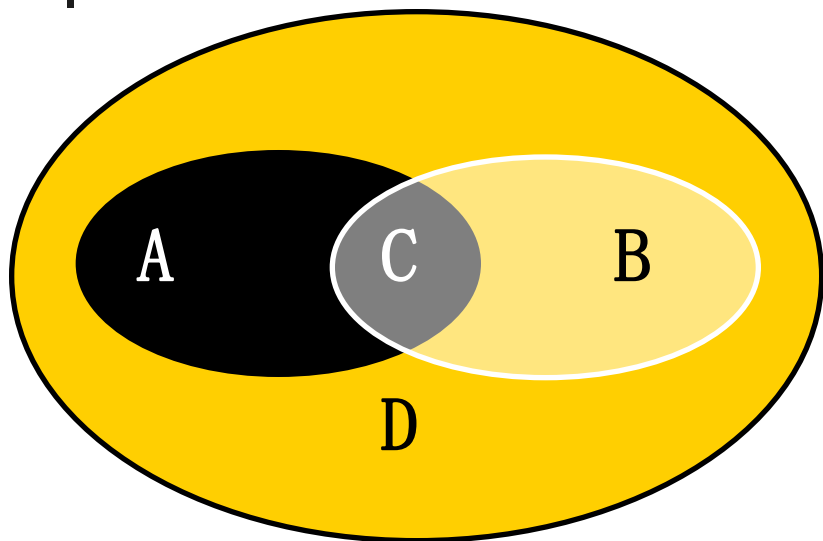
White-Box Testing



黑盒测试

Black-Box Testing

黑盒测试与白盒测试关系



白盒测试比黑盒测试的理论性更强，测试难度更大。

A-只能用黑盒测试发现的错误

B-只能用白盒测试发现的错误

C-两种方法都能发现的错误

D-两种方法都不能发现的错误

黑盒测试是在软件界面上的测试，是为了发现软件错误，但更常用于：

- 证实软件功能的可操作性；
- 证实能很好地接受输入，并正确的产生输出；
- 证实对外部信息完整性的保持。

白盒测试是对程序控制结构的测试，它依赖对程序细节的了解，运用特定条件和与循环集测试用例，对软件逻辑路径进行测试，在不同点检验“程序的状态”以判定预期状态或待验证状态与真实状态是否相符。

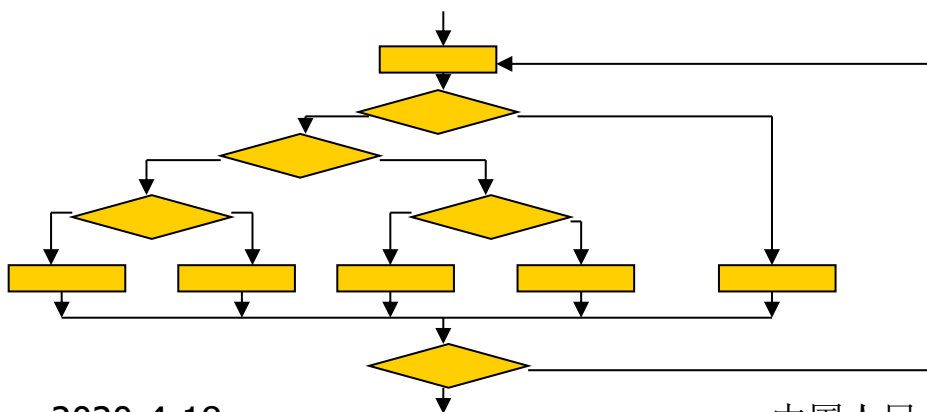


软件测试

5 白盒测试技术

白盒测试 (*White Box Testing*)

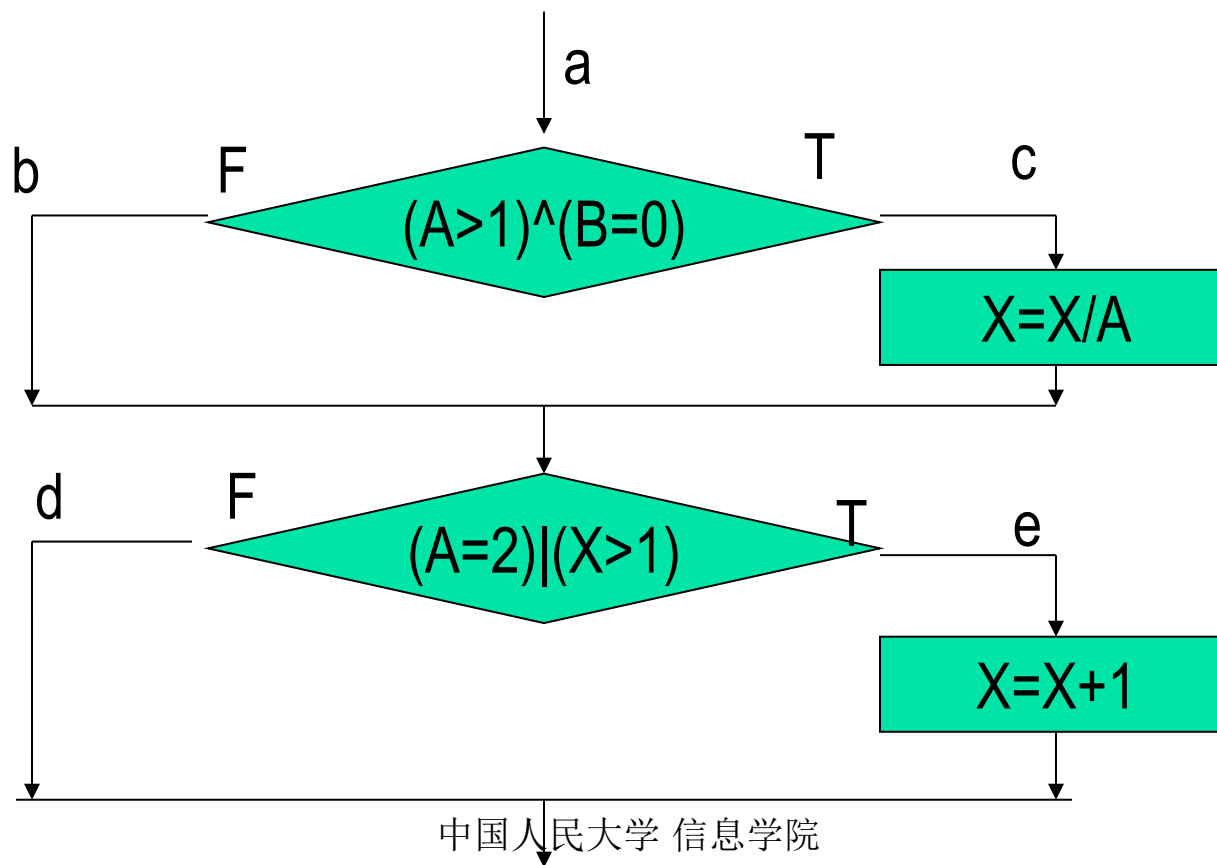
- 又称结构测试，它把测试对象看做一个透明的盒子 (glass box)，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。
- 问题：对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。



考虑20次的循环，有四个 IF-THEN-ELSE 结构，则执行的路径可达 $5^{20} \approx 10^{14}$ 条。
设每个路径测试需1毫秒，一天工作24小时，一年365天，则需3170年

白盒测试用例设计

■ 逻辑覆盖测试方法——解决控制测试问题



路径:

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

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

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

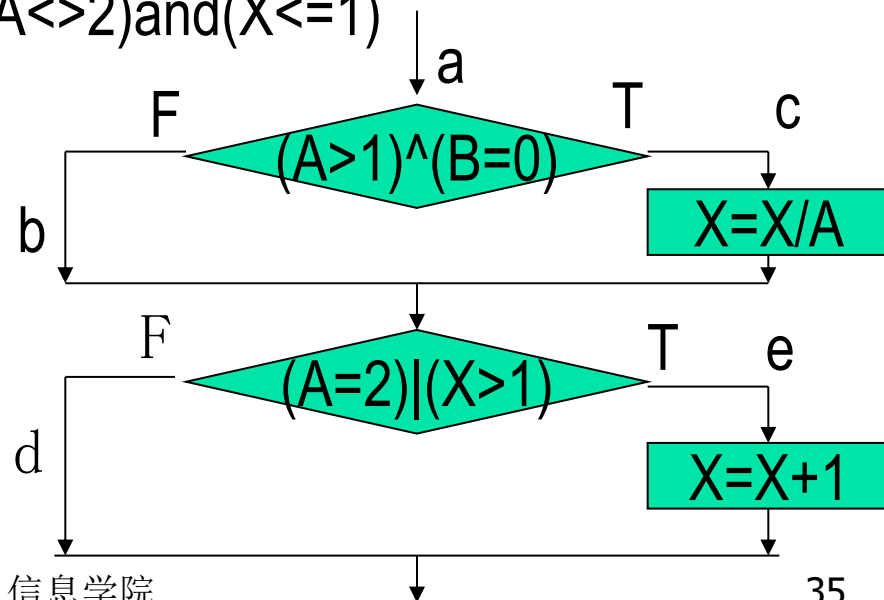
$L2(a\ b\ d) = (A \leq 1) \text{ and } (X \leq 1) \text{ or } (B \neq 0) \text{ and } (A \neq 2) \text{ and } (X \leq 1)$

$L3(a\ b\ e) = (A \leq 1) \text{ and } (X > 1) \text{ or } (B \neq 0)$

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

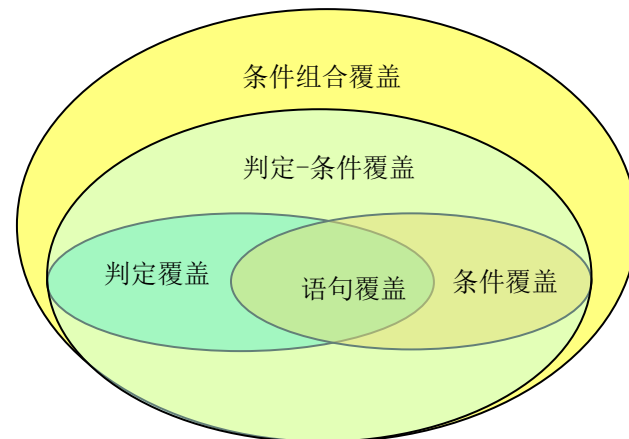
$L4(a\ c\ d) = (A > 1) \text{ and } (B = 0)$

$\text{and } (A \neq 2) \text{ and } (X \leq 1)$



逻辑覆盖测试方法

逻辑覆盖测试是通过对程序逻辑结构的遍历实现程序的覆盖。从覆盖源代码的不同程度可以分为以下六个标准：语句覆盖、判定覆盖（又称为分支覆盖）、条件覆盖、判定-条件覆盖（又称为分支-条件覆盖）、条件组合覆盖和路径覆盖。



覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		$A^B = T$
判定覆盖		$A^B = T$ $A^B = F$

覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		$A = T, A = F,$ $B = T, B = F.$
判定/条件覆盖		$A^B = T, A^B = F.$ $A = T, A = F,$ $B = T, B = F.$
条件组合覆盖		$A = T, B = T,$ $A = T, B = F,$ $A = F, B = T,$ $A = F, B = F.$

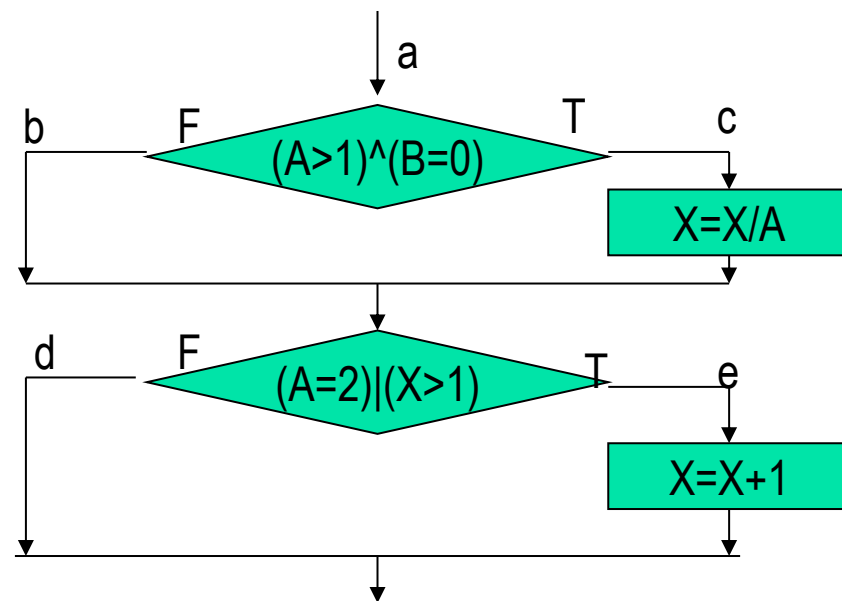
语句覆盖用例

语句覆盖就是设计若干个测试用例，运行所测程序，使得每一可执行语句至少执行一次。上例中可执行语句都在路径L1上，选择L1路径就可实现语句覆盖。

$[(A, B, X), (A, B, X)]$

例： $[(2, 0, 4), (2, 0, 3)]$

问题：如果运算符错误，则测不出来。



例：语句覆盖

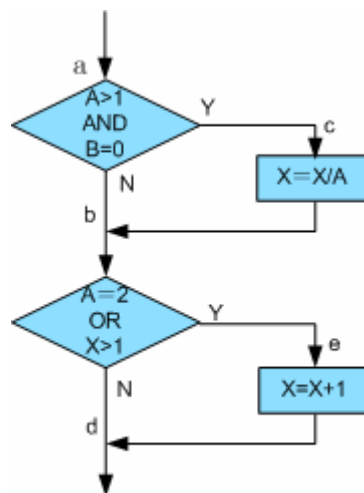


图6.4

设计1个设计用例，即可覆盖全部的语句。

判定覆盖用例

设计若干用例运行所测程序，使程序中每个判断的取真和取假分支至少经历一次。上例中选取L1和L2即可满足要求。

例：

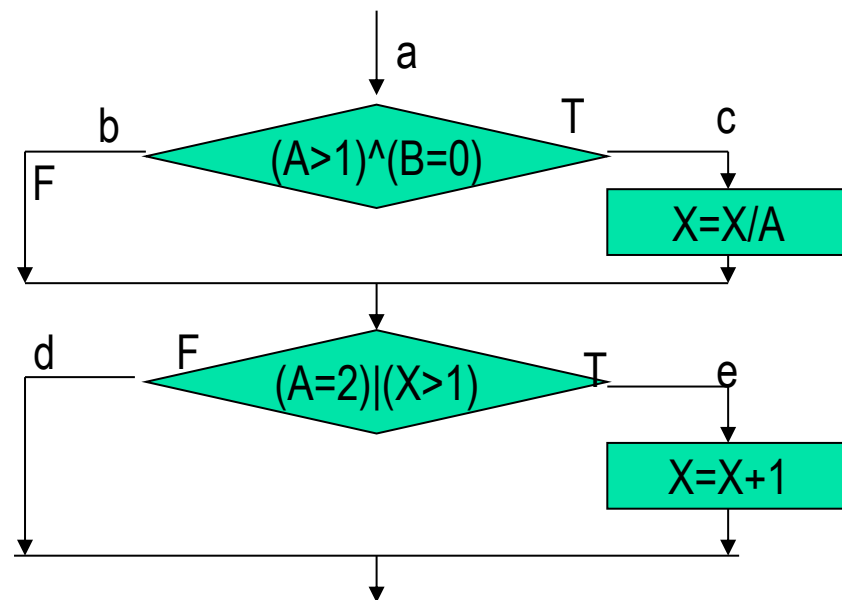
[(2, 0, 4) , (2, 0, 3)] 覆盖 (ace)

[(1, 1, 1) , (1, 1, 1)] 覆盖 (abd)

选择路径L3, L4还可得另一组测试用例。

[(2, 1, 1) , (2, 1, 2)] 覆盖 (abe)

[(3, 0, 3) , (3, 1, 1)] 覆盖 (acd)



问题：测不出某个变量的错误。

例：判定覆盖

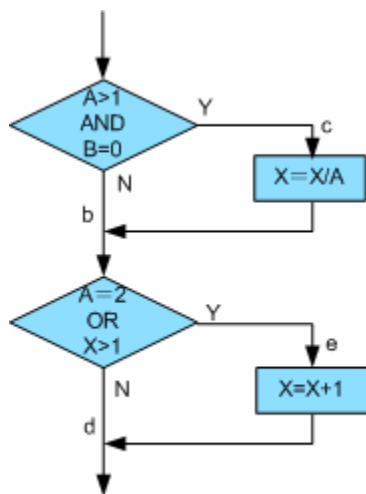


图6.4

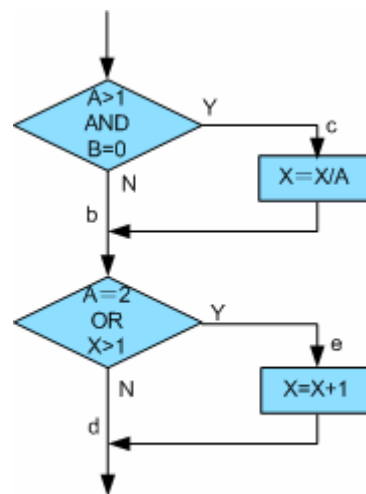


图6.4

设计2个测试用例，运行所测程序，使程序中每个判断的取真和取假分支至少经历一次。

条件覆盖用例

设计若干用例，使程序中每个判断的每个条件的可能取值至少执行一次。

例：上例中对第一个判断有： $A > 1$, $B = 0$

对第二个判断有： $A = 2$, $X > 1$

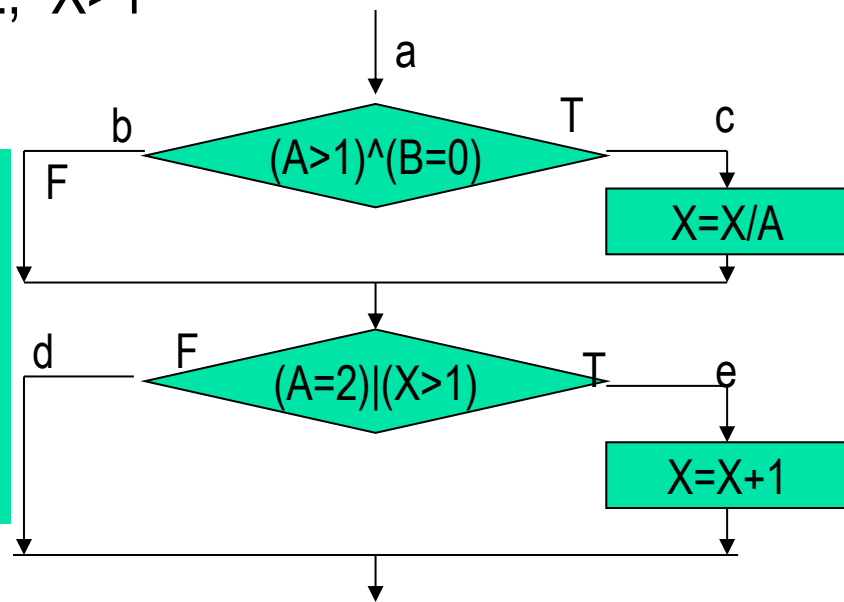
用例设计：

$[(2,0,4),(2,0,3)]$ a c e 或： $[(1,0,3),(1,0,4)]$ a b e

$[(1,0,1),(1,0,1)]$ a b d $[(2,1,1),(2,1,2)]$ a b e

$[(2,1,1),(2,1,2)]$ a b e

未考虑判定覆盖



例：条件覆盖

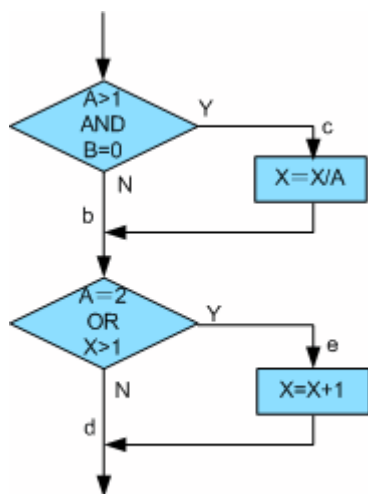


图6.4

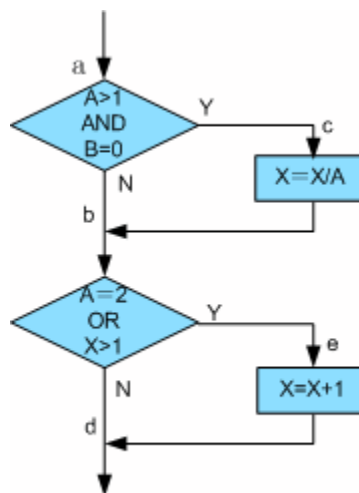


图6.4

设计2个测试用例，确保程序中每个判断的真、假取值至少执行一次。

判定-条件覆盖用例

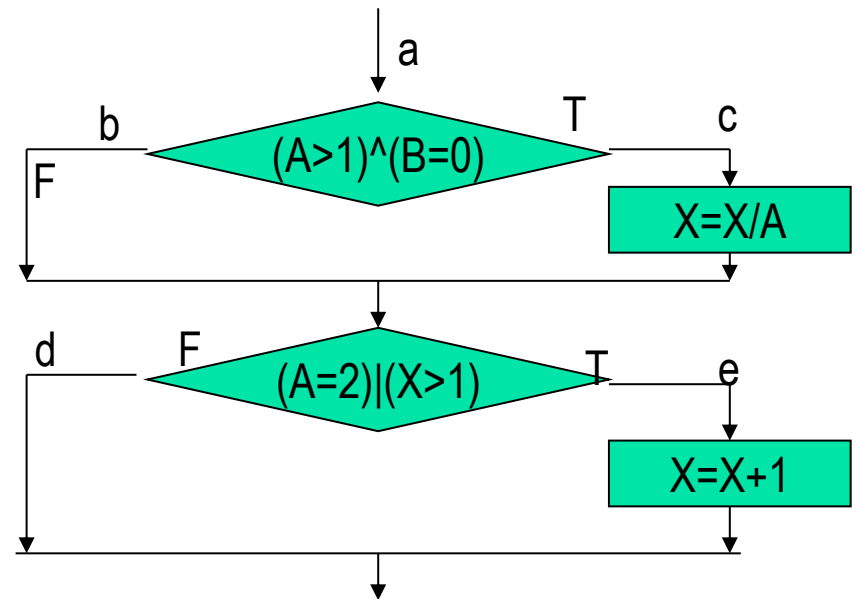
设计足够测试用例，使判断中每个条件的所有可能取值至少执行一次，同时每个判断的所有结果至少执行一次。

上例中取各值为真和假即可覆盖8个条件取值和4个判断分支。

例：

$[(2, 0, 4), (2, 0, 3)]$ ace

$[(1, 1, 1), (1, 1, 1)]$ abd



例：判定-条件覆盖

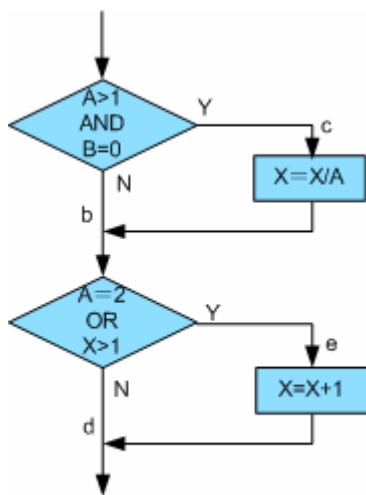


图6.4

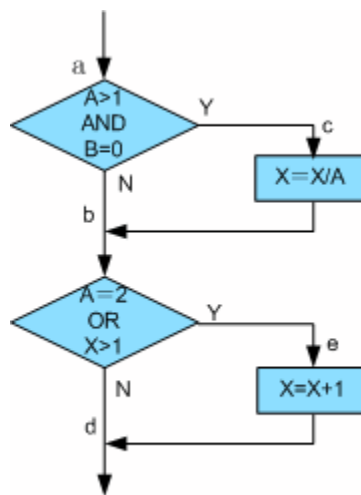


图6.4

设计2个测试用例，使判断中每个条件的所有可能取值至少执行一次，同时每个判断的所有结果至少执行一次。

条件组合覆盖用例

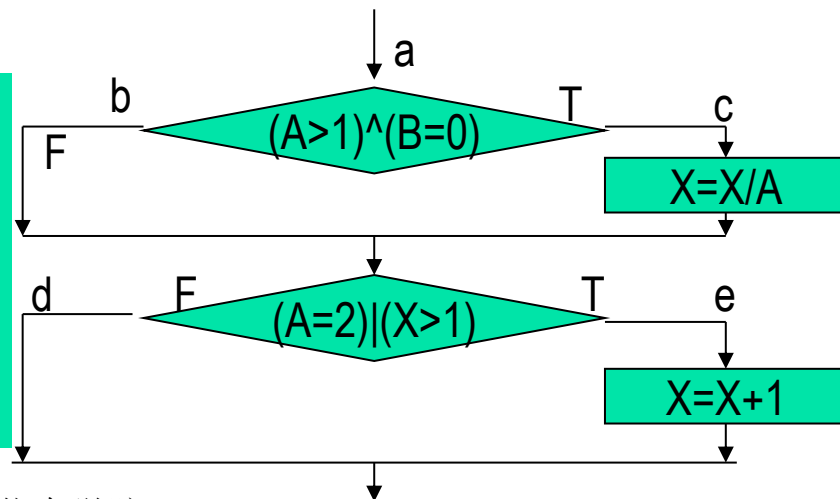
设计足够的测试用例，使每个判断的所有可能的条件取值组合至少执行一次。

上例中可能的组合为：

第一个判断： $A > 1, B = 0$ $A > 1, B \neq 0$ $A \leq 1, B = 0$ $A \leq 1, B \neq 0$

第二个判断： $A = 2, X > 1$ $A = 2, X \leq 1$ $A \neq 2, X > 1$ $A \neq 2, X \leq 1$

测试 [(2, 0, 4) , (2, 0, 3)] ace
用例： [(2, 1, 1) , (2, 1, 2)] abe
 [(1, 0, 3) , (1, 0, 4)] abe
 [(1, 1, 1) , (1, 1, 1)] abd



例：条件组合覆盖

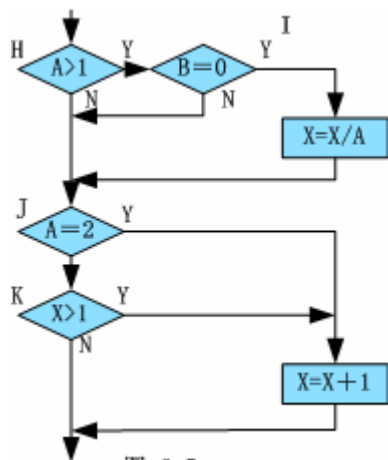


图 6.5

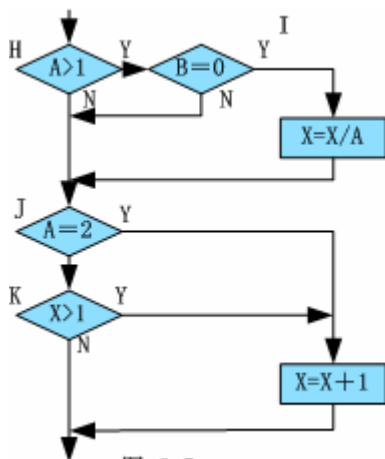


图 6.5

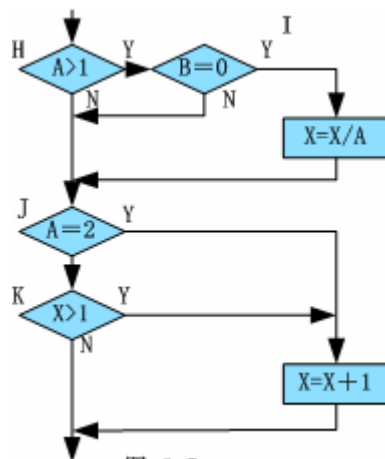


图 6.5

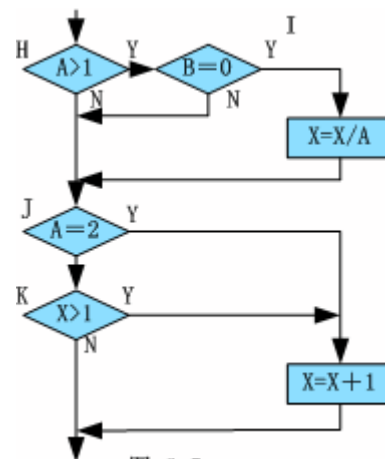


图 6.5

设计4个测试用例，使每个判断的所有可能的条件取值组合至少执行一次。

路径测试用例

设计足够的测试用例，覆盖程序中所有可能的路径。

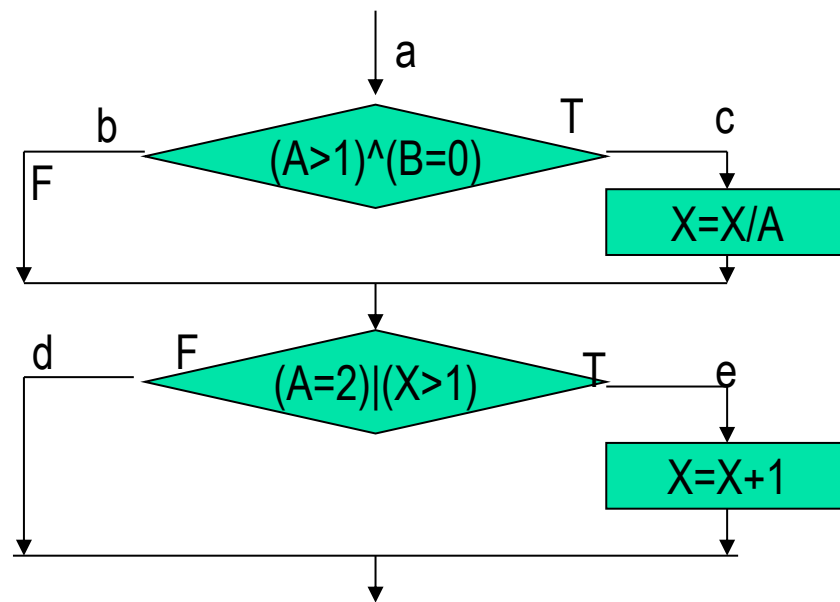
用例：

[(2, 0, 4) , (2, 0, 3)] ace

[(1, 1, 1) , (0, 1, 1)] abd

[(1, 1, 2) , (1, 1, 3)] abe

[(3, 0, 3) , (3, 0, 1)] acd





基本路径测试方法

实际程序中路径是个很庞大的数字，所有路径覆盖是不现实的，测试中把路径数压缩在一定范围内，称为**基本路径测试**。

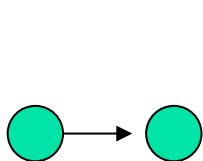
它是在程序控制流图的基础上，通过分析控制构造的环路复杂性测度，导出基本可执行路径的基本集合，从而设计测试用例。该基本集导出的测试用例保证程序的每一个可执行语句至

基本路径测试用例设计（解决循环问题，计算复杂度）：

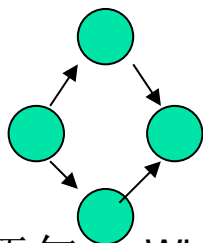
- **程序的控制流图**
- **程序环路复杂性**
- **导出测试用例**
- **图矩阵**

控制流图

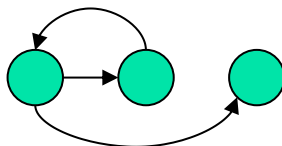
流图的符号：每一个结构化构成元素有一个相应的流图符号



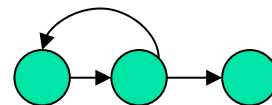
顺序语句



IF语句

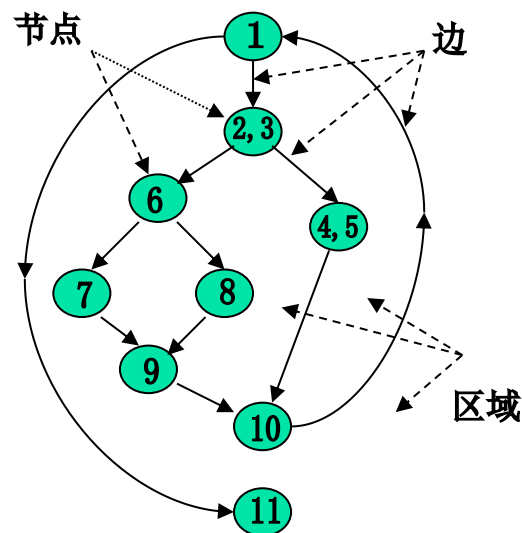
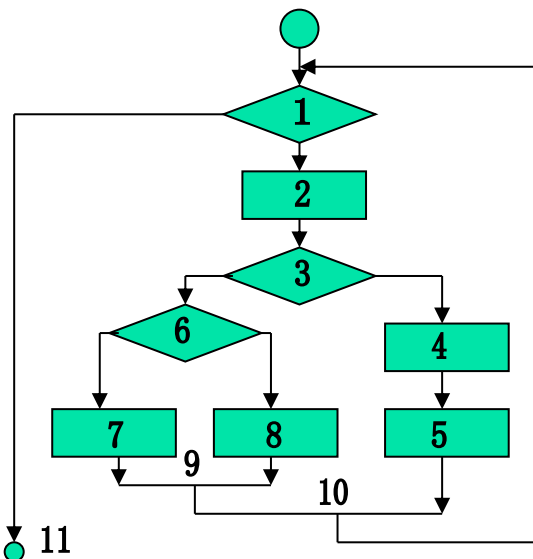
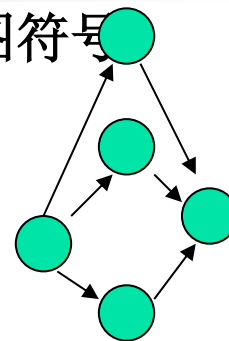


While语句



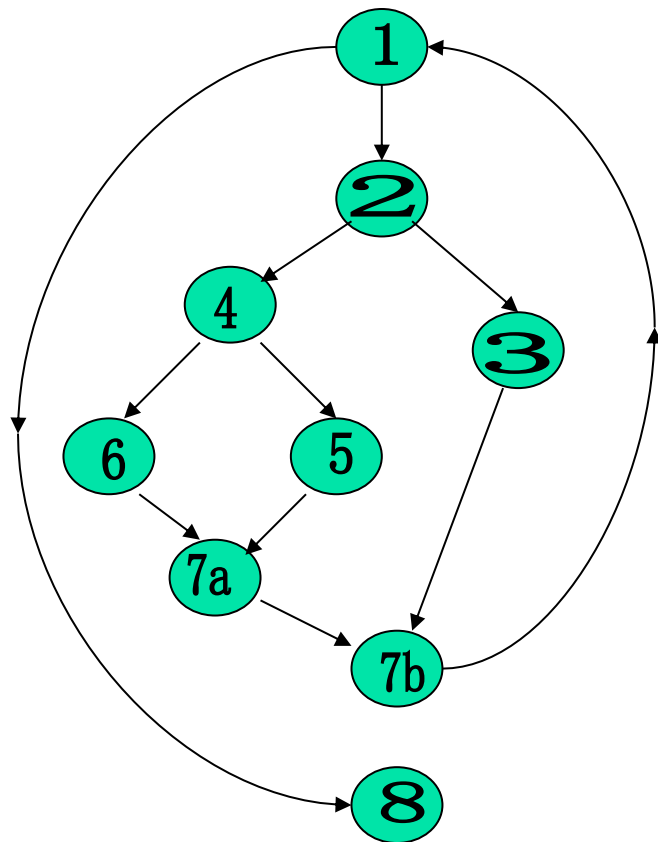
Until语句

Case语句



(flow chart到flow graph)

PDL翻译成流图

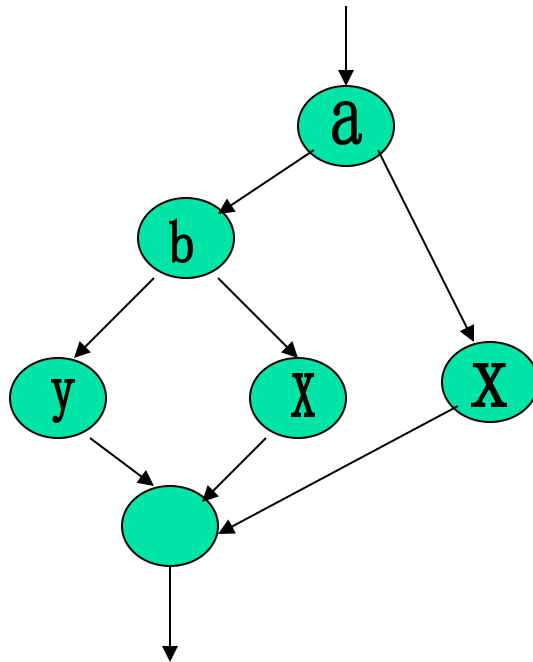


PDL

Procedure: sort

```
1:  do while records remain
    read record;
2:  if record field 1 = 0
3:  then process record;
    store in buffer;
    increment counter;
4:  else if record field 2 = 0
5:  then reset counter;
6:  else process record;
    store in file;
7a: end if
    end if
7b: end do
8: end
```

复合逻辑的流图



.....

IF a OR b

THEN procedure x

ELSE procedure y

ENDIF

.....

转化成简单分支形式

程序环路复杂性

环路复杂性是一种为程序逻辑复杂性提供定量测度的软件度量，计算所得的值定义了程序基本集的独立路径数量，提供了确保所有语句至少执行一次的测试数量上界。

独立路径是指程序中至少引进一个新的处理语句集合或一个新条件的任意路径，即至少包括一条在定义路径前没有用到的边。

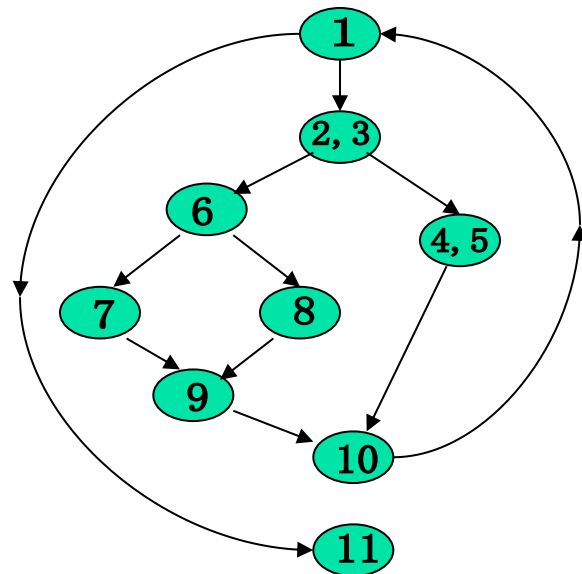
上例中独立路径为：

路径一： 1-11

路径二： 1-2-3-4-5-10-1-11

路径三： 1-2-3-6-8-9-10-1-11

路径四： 1-2-3-6-7-9-10-1-11



环路复杂性计算

环路复杂性计算以图论为基础可用三种方法计算

- 流图中区域数量对应与环路的复杂性

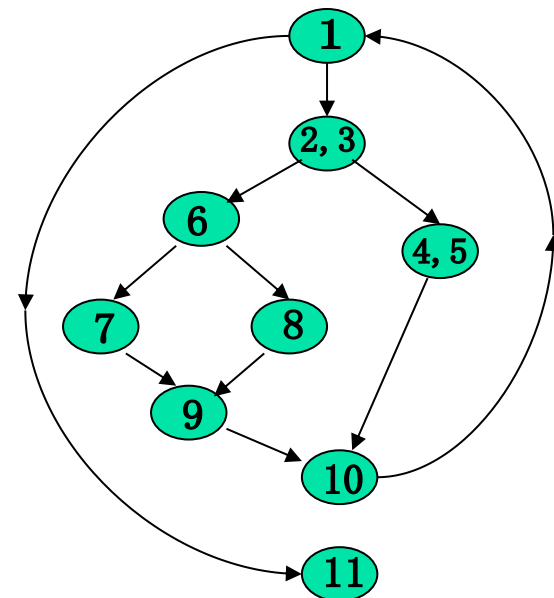
- 给定流图**G**的环路复杂性计算:

$V(G)=E-N+2$ **E**边的数量, **N**节点数

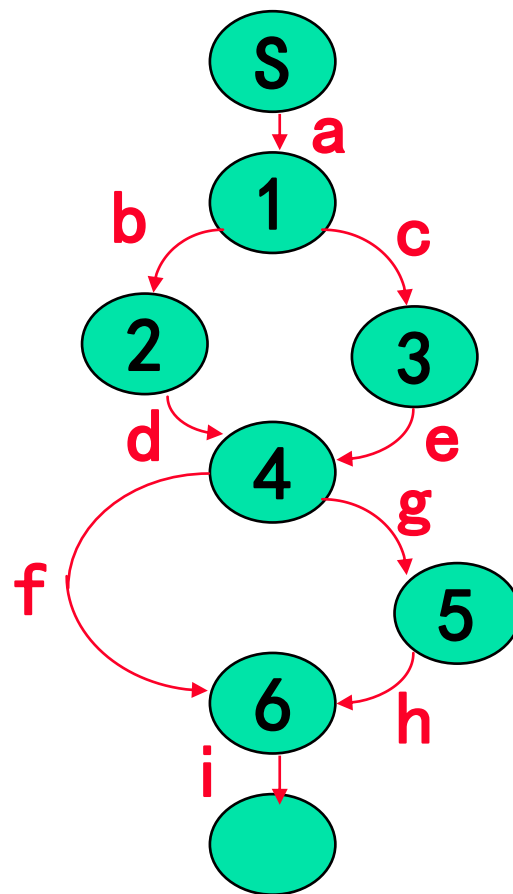
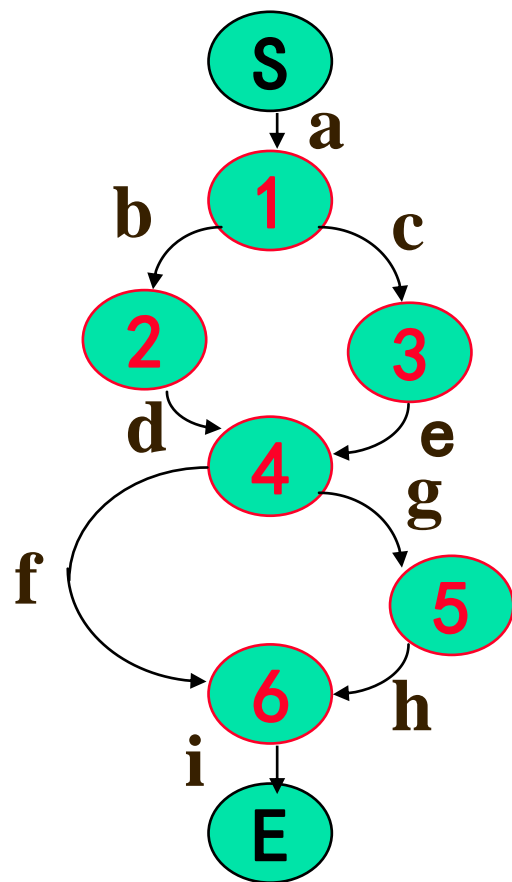
- 给定流图**G**的环路复杂性计算:

$V(G)=P+1$ **P**是流图**G**中判定节点的数量

1. 流图中有四个区域
2. $V(G)=11\text{条边}-9\text{个节点}+2=4$
3. $V(G)=3\text{个判定节点}+1=4$



点覆盖和边覆盖



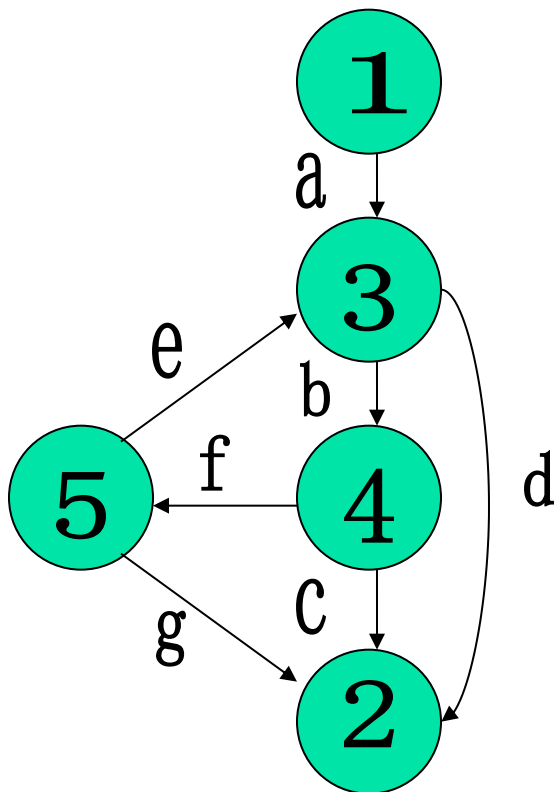


导出测试用例

- 从PDL导出流图
- 确定流图的复杂性
- 确定线性独立的路径的一个基本集
- 准备测试用例

图矩阵

用图矩阵实现确定基本测试路径自动化



所连接的节点						
节点		1	2	3	4	5
1				a		
2						
3			d		b	
4			c			f
5			h	e		

图矩阵例

最简单的情况下，
连接权值是1或0，但是，
连接权值可以赋予其它
属性：

- 执行连接边的概率
- 穿越连接的处理时间
- 穿越连接时所需内存
- 穿越连接时所需资源

所连接的节点						
节点	1	2	3	4	5	
1			1			
2						
3		1		1		
4		1			1	
5		1	1			

计算：

$$1-1=0$$

$$2-1=1$$

$$2-1=1$$

$$2-1=1$$

$$3+1=4$$



McCabe度量法

- 它是一种基于程序控制流的复杂性度量方法，定义的度量值又称环路复杂度。
- 它是基于一个程序模块的程序图中环路的个数。如果把程序流程图中每个处理符号都退化成一个结点，原来联结不同处理符号的流线变成连接不同结点的有向弧，这样得到的有向图就叫做程序图。
- 计算有向图 G 的环路复杂性的公式：

$$V(G)=m-n+2$$

其中， $V(G)$ 是有向图 G 中的环路个数， m 是图 G 中有向弧个数， n 是图 G 中结点个数。

- **McCabe** 环路复杂度度量值实际上是为软件测试的难易程度提供了一个定量度量的方法，同时也间接地表示了软件的可靠性。
- 实验表明，源程序中存在的错误数以及为了诊断和纠正这些错误所需的时间与**McCabe** 环路复杂度度量值有明显的关系。



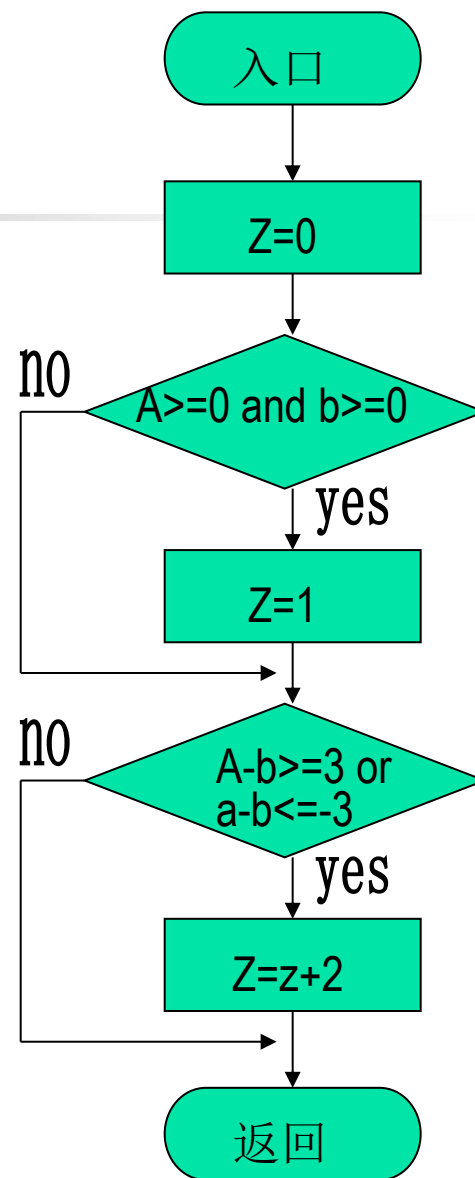
利用McCabe环路复杂度度量的几点说明：

- 环路复杂度取决于程序控制结构的复杂度。当程序的分支数目或循环数目增加时其复杂度也增加。环路复杂度与程序中覆盖的路径条数有关。
- 环路复杂度是可加的。例如，模块A的复杂度为3，模块B的复杂度为4，则模块A与模块B的复杂度是7。
- McCabe建议，对于复杂度超过10的程序，应分成几个小程序，以减少程序中的错误。Walsh用实例证实了这个建议的正确性。他发现，在McCabe复杂度为10的附近，存在出错率的间断跃变。
- McCabe环路复杂度隐含的前提是：错误与程序的判定加上例行子程序的调用数目成正比。而加工复杂性、数据结构、录入与打乱输入卡片的错误可以忽略不计。

课堂练习题

本流程描述了某子程序的处理流程，现要求用白盒测试法对子程序进行测试。

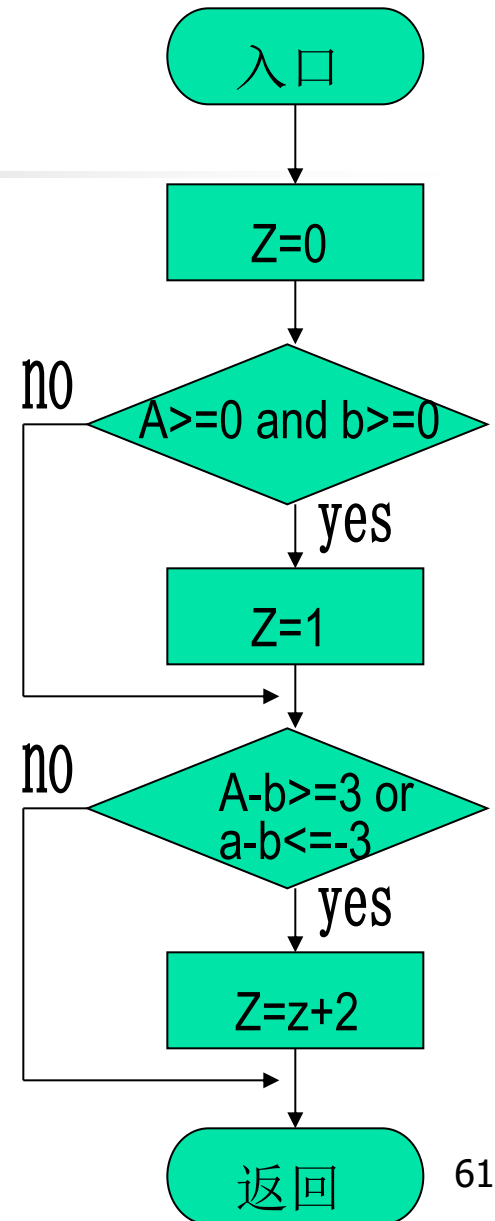
问题：根据判定覆盖、条件覆盖、判定条件覆盖、条件组合覆盖、路径覆盖等五种覆盖标准，从供选择的答案中分别找出满足相应覆盖标准的最小的测试数组。



习题答案：④，③，⑥，⑧，⑦

课堂练习题续

- ① $a=5, b=1$
- ② $a=5, b=-1$
- ③ $a=5, b=1; a=-5, b=-1$
- ④ $a=5, b=1; a=0, b=-1$
- ⑤ $a=5, b=-1; a=-5, b=1; a=-5, b=-1$
- ⑥ $a=5, b=1; a=0, b=0; a=-5, b=-1$
- ⑦ $a=5, b=1; a=0, b=1; a=0, b=-1; a=-5, b=1$
- ⑧ $a=5, b=1; a=0, b=-1; a=-5, b=1; a=-5, b=-1$
- ⑨ $a=5, b=1; a=0, b=-1; a=0, b=1; a=-5, b=1; a=-5, b=-1$
- ⑩ $a=5, b=1; a=5, b=0; a=5, b=-1; a=0, b=1; a=0, b=0; a=0, b=-1; a=-5, b=1; a=-5, b=0; a=-5, b=-1$





控制结构测试

基本路径测试简单高效但不够充分，人们提出一些变种的测试方法：

- 条件测试
- 数据流测试
- 循环测试
 - 简单循环
 - 嵌套循环
 - 串接循环
 - 不规则循环



条件测试

- 条件测试是检查程序模块中所包含逻辑条件的测试用例设计方法。

- 关系表达式的形式：

$E1 <relation-operator> E2$

- 其中：

- $E1$ 和 $E2$ 是算术表达式。
- $<relation-operator>$ 是下列之一：<，≤，≠，(¬ =)，>，≥
- 布尔运算符 OR |，AND &，NOT ¬

- 如果条件不正确，则至少有一个条件成份不正确，条件错误类型如下：

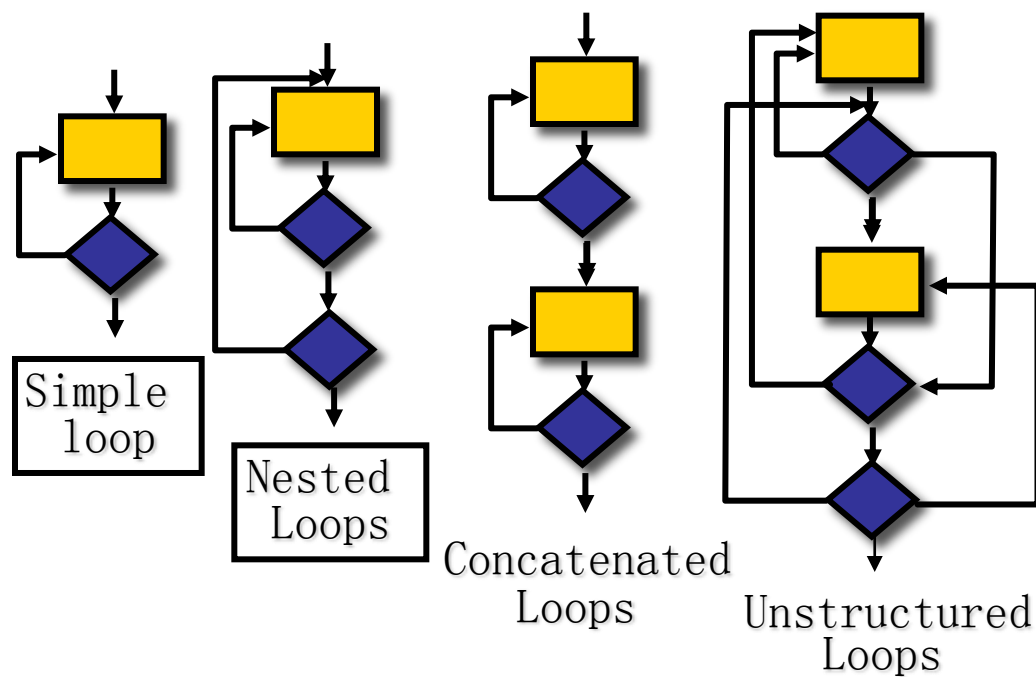
- 布尔操作符错误（遗漏布尔操作符，布尔操作符多余或布尔操作符不正确）
- 布尔变量错误
- 布尔括号错误
- 关系运算符错误
- 算法表达式错误



数据流测试

- 数据流测试按照程序中的变量定义和使用的位置来选择程序的测试路径。
- 测试策略是要求覆盖每个DU (*definition-use*) 链至少一次——DU测试策略。
- 已经证明DU测试不能保证覆盖程序的所有分支，但是，不覆盖某个分支仅仅在于如下情况：*if-then-else*中的*then*没有定义变量，而且不存在*else*部分。
- 假设程序的每条语句赋予了唯一的语句编号，每个函数都不改变参数和全局变量。对于语句号为S的语句，
$$\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$$
$$\text{USE}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$$
- 定义X的DU链式如[X, S, S']，其中S和S'是语句号，X在DEF(S)和USE(S)中，而且语句S定义的X语句S'有效。

循环测试



■ 简单循环（ n 是允许通过循环的最大次数）：

- ① 跳过整个循环
- ② 只有一次通过循环
- ③ 两次通过循环
- ④ m 次通过循环，其中 $m < n$
- ⑤ $n-1$, n , $n+1$ 次通过循环



(续)

- 嵌套循环：如果要将简单循环的测试方法用于嵌套循环，可能的测试数就会随嵌套层数成几何级增加，减少测试数的方法：
 - ① 从最内层循环开始，将其他循环设置为最小值。
 - ② 对最内层循环使用简单循环测试，而使外层循环的迭代参数最小，而为范围外或排除的值增加其他测试。
 - ③ 由内向外构造下一个循环的测试，但其他的外层循环为最小值，并使其他的嵌套循环为典型值。
 - ④ 继续直到测试完所有的循环。
- 串接循环：如果串接循环的循环都彼此立，可以使用简单循环的策略测试，但是，如果循环不独立，则使用嵌套循环测试策略。
- 不规则循环：尽可能将这类循环重新设计为结构化的程序结构。



软件测试

6 黑盒测试技术

黑盒测试

- 黑盒测试注重测试软件功能性需求。
- 黑盒测试用于发现：
 - 功能不对或遗漏
 - 界面错误
 - 数据结构或外部数据库访问错误
 - 性能错误
 - 初始化和终止错误



■ 黑盒测试方法

- 等价划分 (Equivalence Classes)
- 边界值分析 (Boundary Value Analysis)
- 错误推测法 (error guessing methods)
- 因果图 (cause effect graphing)
- 决策表技术 (decision table techniques)
- 基于图的测试 (系统模型)



黑盒测试用例

- 黑盒测试用于测试的后期，它不考虑控制结构，而是注重信息域。
- 设计黑盒测试用例要考虑：
 - 如何测试功能的有效性？
 - 何种类型的输入会产生好的测试用例？
 - 系统是否对特定的输入值尤其敏感？
 - 如何分隔数据类的边界？
 - 系统能够承受何种数据率和数据量？
 - 特定类型的数据组合会对系统产生何种影响？

等价划分

- 如果对象由具有对称性、传递性或自反性的关系连接，就存在等价类。
- 将程序的输入域划分为数据类，以便导出测试用例。
- 等价类划分试图定义一个测试用例，以发现同一等价类中的错误，从而减少开发测试用例数。

等价划分原则：

- 有效等价类；
- 无效等价类；
- 按照数值集合划分；
- 按照限制条件或规则划分；
- 细分等价类





如何划分等价类？

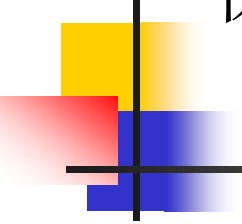
- ① 如果输入条件代表一个范围，可以定义一个有效等价类和两个无效等价类；
- ② 如果输入条件需要特定的值，可以定义一个有效等价类和两个无效等价类；
- ③ 如果输入条件代表集合的某个元素，可以定义一个有效等价类和一个无效等价类；
- ④ 如果输入条件是布尔式，可以定义一个有效等价类和一个无效等价类；
- ⑤ 如果规定输入条件必须遵守的规则，则可划分一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则)；
- ⑥ 如已划分的等价类各元素在程序中的处理方式不同，则应将此等价类进一步划分成更小的等价类。

用等价类划分法设计测试用例步骤

- ① 形成等价类表，每一等价类规定一个唯一的编号；
- ② 设计一测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类，重复这一步骤，直到所有有效等价类均被测试用例所覆盖；
- ③ 设计一新测试用例，使其只覆盖一个无效等价类，重复这一步骤直到所有无效等价类均被覆盖。

■ 例：划分等价类法

输入条件	有效等价类	无效等价类
区号：三位数，但不是0和1开始	定义在200和999之间的数值	1) 小于200 2) 大于999



例1：用等价类划分法设计电话号码的测试用例。

某城市电话号码由三部分组成：

地区码：空白或3位数字

前缀：非‘0’或‘1’开头的三位数字

后缀：4位数字

第一步：电话号码等价类划分

输入条件	有效等价类	无效等价类
地区码	空白 (1) 3位数字 (2)	有非数字字符 (5) 少于3位数字 (6) 多于3位数字 (7)
前缀	从200到999之间的3位数字(3)	有非数字字符 (8) 起始位为 ‘0’ (9) 起始位为 ‘1’ (10) 少于3位数字 (11) 多于3位数字 (12)
后缀	4位数字(4)	有非数字字符 (13) 少于4位数字 (14) 多于4位数字 (15)



第二步：确定测试用例

对表中4个有效等价类可公用

下面两个测试用例：

测试数据	测试范围	期望结果
() 276-2345	等价类 (1) (3) (4)	有效
(635) 805-9321	等价类 (2) (3) (4)	有效



第三步：确定无效结果测试用例

对表中11个无效等价类应选择11个测试用例

测试数据	测试范围	期望结果
(20A) 123-4567	无效等价类 (5)	无效
(33) 234-5678	无效等价类 (6)	无效
(7777) 345-6789	无效等价类 (7)	无效
.	.	.
.	.	.
.	(15)	无效

本例的15个等价类至少需要13个测试用例

例2:对招干考试系统“输入学生成绩”子模块设计测试用例

招干考试分三个专业，准考证号第一位为专业代号，如：

- 1-行政专业,
- 2-法律专业,
- 3-财经专业.

行政专业准考证号码为：110001~111215

法律专业准考证号码为：210001~212006

财经专业准考证号码为：310001~314015

例：准考证号码的等价类：

•有效等价类：

(1) 110001 ~ 111215

(2) 210001 ~ 212006

(3) 310001 ~ 314015

•无效等价类：

(4) $-\infty$ ~ 110000

(5) 111216 ~ 210000

(6) 212007 ~ 31000

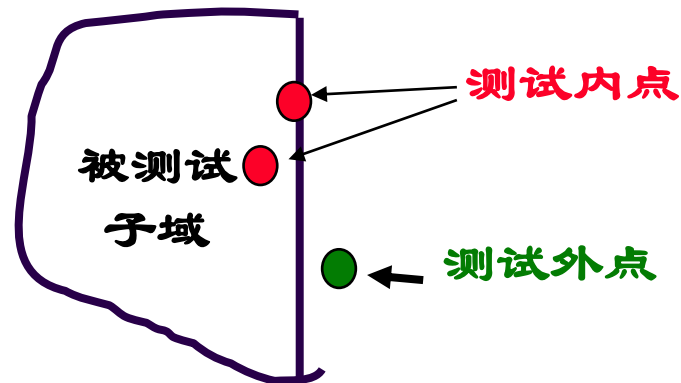
(7) 314016 ~ $+\infty$

边界值分析

- 实践表明，输入域的边界比中间更容易发生错误，作为等价类划分法的补充边界值分析 (*boundary value analysis, BVA*) 可作为一种测试方法。

- 边界值分析法与等价类划分法区别：

- (1) 边界值分析不是从某等价类中随便挑一个作为代表，而是使这个等价类的每个边界都要作为测试条件。
- (2) 边界值分析不仅考虑输入条件，还要考虑输出空间产生的测试情况





边界值分析设计测试用例原则

1. 如输入条件代表以a和b为边界的范围，测试用例应包含a、b、略大于a和略小于b的值。
2. 如输入条件代表一组值，测试用例应当执行其中的最大值和最小值，还应测试略大于最大值和略小于最小值的值。

■ 例：

- 邮件收费规定 1~5 kg收费2元，则应对：
0.9, 1, 5, 5.1 kg 或 0.99, 1, 5, 5.01 kg设计测试用例。
- 一个输入文件可有1~255个记录则可分别设计有：
1个、255个、0个、256个记录的输入文件



(续)

3. 如规格说明中提出输入输出的有序集（顺序文件、有序表等），取有序集的第一个和最后一个元素做测试用例。
 4. 如程序数据结构有预定的边界，应测试其边界的数据项。
-
- 例：程序中定义一数组，其元素下标的下界是0，上界是100，则应选择达到这个数组下标边界的值，如0与100作为测试用例。



例：取边界上下浮动值做测试用例

- 每日保险扣除额(输出项)在0~1165.25 元，则应设计测试用例使其恰好产生0元和1165.25元的结果，此外还应考虑设计结果为负值或 >1165.25 元的测试用例，如：
-0.01元和1165.26元。
- 由于输入值的边界不与输出值的边界相对应，所以要检查输出值的边界及产生超出输出值值域的结果不一定可能。



错误推测法

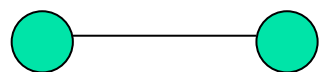
- 根据经验来设计测试用例的方法，例举出程序中可能有的错误和容易发生错误的特殊情况，根据它们选择测试方案。



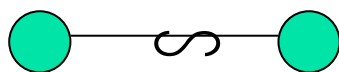
因果图

- 因果图是根据分析时对各种条件可能产生的结果的一种对照分析法。
- 其测试用例的基本依据可以是分析阶段的决策表
 - 根据因果关系建立因果关系图
 - 建立决策表获得用例

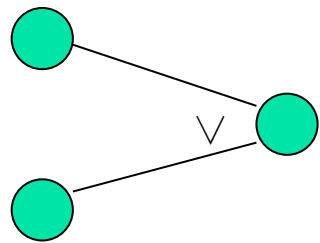
因果图图例：



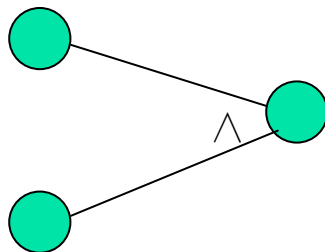
恒等



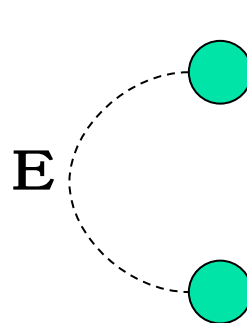
非



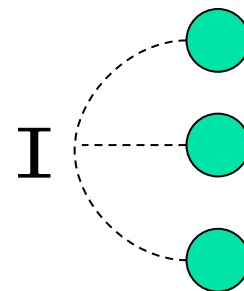
或



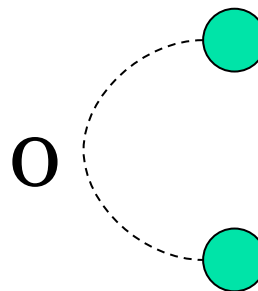
与



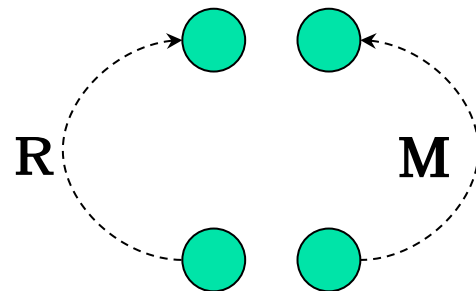
E



I



O



R

M

5种约束：互斥 (E)，包含 (I)，唯一 (O)，要求 (R)，屏蔽 (M)



例1：自动饮料机软件的黑盒测试

■ 问题：

- 饮料单价1.5元。
- 投入1.5元，按下“可乐”、“雪碧”或“红茶”按钮，测送出饮料。
- 若投入2元，送出饮料时还退还5角。

■ 解决方法：

- 建立软件因果图
- 设计测试用例

因果图例（续）

输入条件

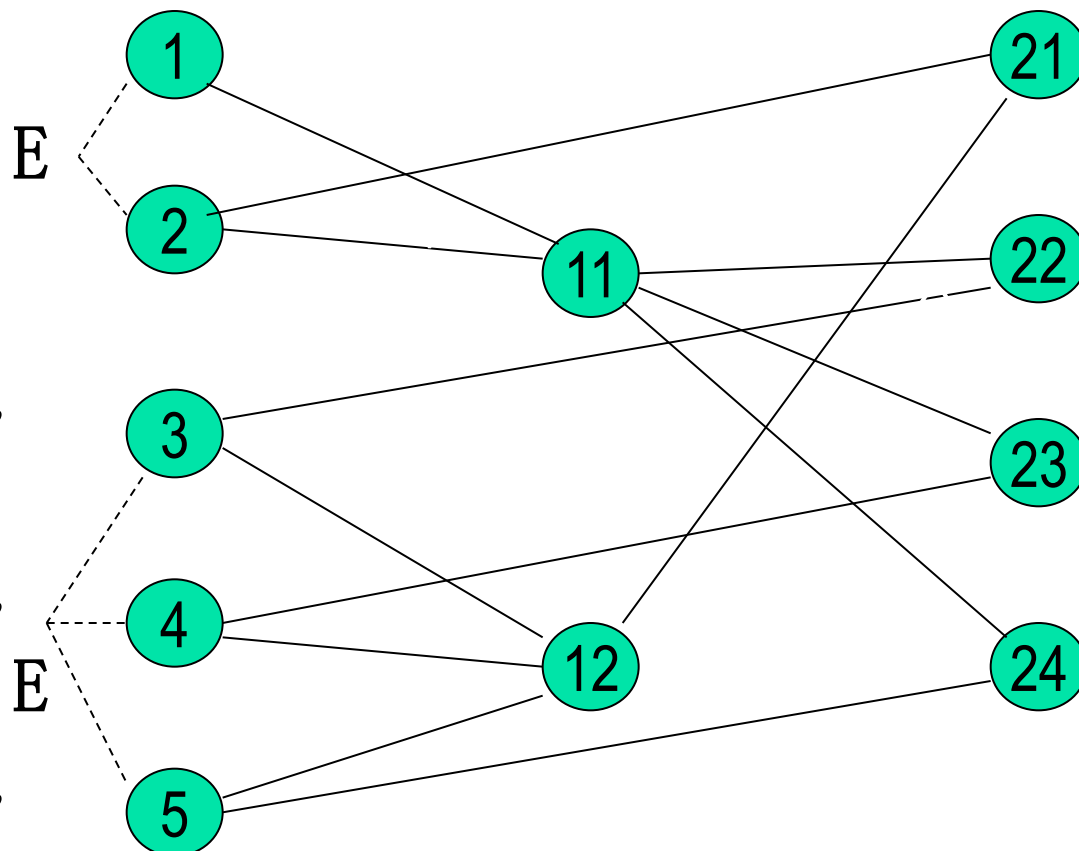
投1.5

投2元

按“可乐”

按“雪碧”

按“红茶”



输出条件:

退5角

送出“可乐”

送出“雪碧”

送出“红茶”

因果图例（续）

			1	2	3	4	5	6	7	8	9	10	11
输入节点	投1.5元 (1)		1	1	1	1	0	0	0	0	0	0	0
	投2元 (2)		0	0	0	0	1	1	1	1	0	0	0
	按“可乐” (3)		1	0	0	0	1	0	0	0	1	0	0
	按“雪碧” (4)		0	1	0	0	0	1	0	0	0	1	0
	按“红茶” (5)		0	0	1	0	0	0	1	0	0	0	1
中间节点	已投币 (11)		1	1	1	1	1	1	1	1	0	0	0
	已按钮 (12)		1	1	1	0	1	1	1	0	1	1	1
输出	退5角 (21)		0	0	0	0	1	1	1	0	0	0	0
	送“可乐” (22)		1	0	0	0	1	0	0	0	0	0	0
	送“雪碧” (23)		0	1	0	0	0	1	0	0	0	0	0
	送“红茶” (24)		0	0	1	0	0	0	1	0	0	0	0



因果图方法——例2:

- 某软件规格说明中规定：
 - 输入两个字符，当第一个字符是A或B，且第二个字符是数字时修改文件；
 - 若第一个字符不是 A，也不是B 时，输出错误信息S；
 - 若第二个字符不是数字时，输出错误信息P。



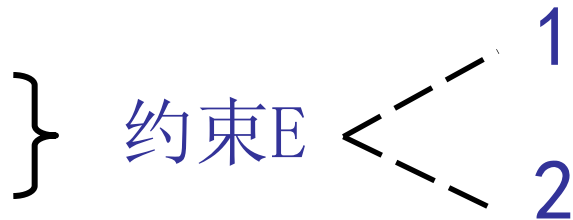
分析规格说明并编号：

原因是：

1—第1个字符是A

2—第1个字符是B

3—第2个字符是数字



(1, 2不能同时成立)

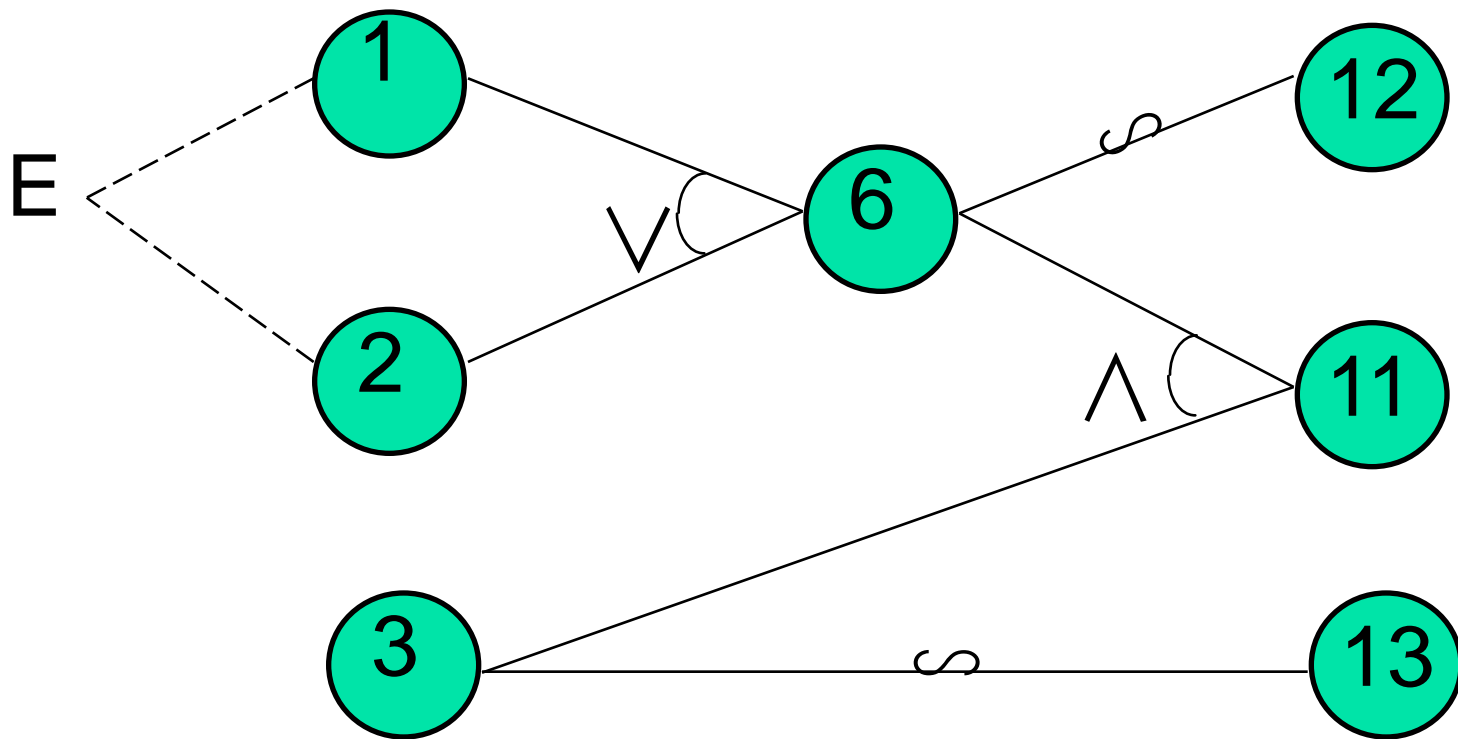
结果是：

11—修改文件。

12—输出错误信息S。

13—输出错误信息P。

因果图的实例



因果图转换的判定表

组合条件		1	2	3	4	5	6	7	8
条件 原因	1	1	1	1	1	0	0	0	0
	2	1	1	0	0	1	1	0	0
	3	1	0	1	0	1	0	1	0
动作 结果	11			1	0	1	0	0	0
	12			0	0	0	0	1	1
	13			0	1	0	1	0	1
测试用例				A3	A*	B8	BB	X6	CC



因果图转换的判定表分析

判定表所有条件组合数： $2^3 = 8$ ，第1、2列条件组合不可能出现，针对第3～8列设计测试用例如下：

条件组合	测试用例 (输入数据)	预期结果 (输出动作)
3列	A3	修改文件
4列	A*	
5列	B8	
6列	BB	信息P
7列	X6	信息S
8列	CC	信息S, P



因果图方法——例3:

某电力公司有A、B、C、D四类收费标准，并规定：

- 居民用电

- <100 度/月，按A类收费
- ≥ 100 度/月，按B类收费

- 动力用电

- <10000 度/月，非高峰，B类收费
- ≥ 10000 度/月，非高峰，C类收费
- <10000 度/月，高峰，C类收费
- ≥ 10000 度/月，高峰，D类收费



列出输入条件和输出动作并编号

输入条件(原因) 输出动作(结果)

1—居民用电 A- A类计费

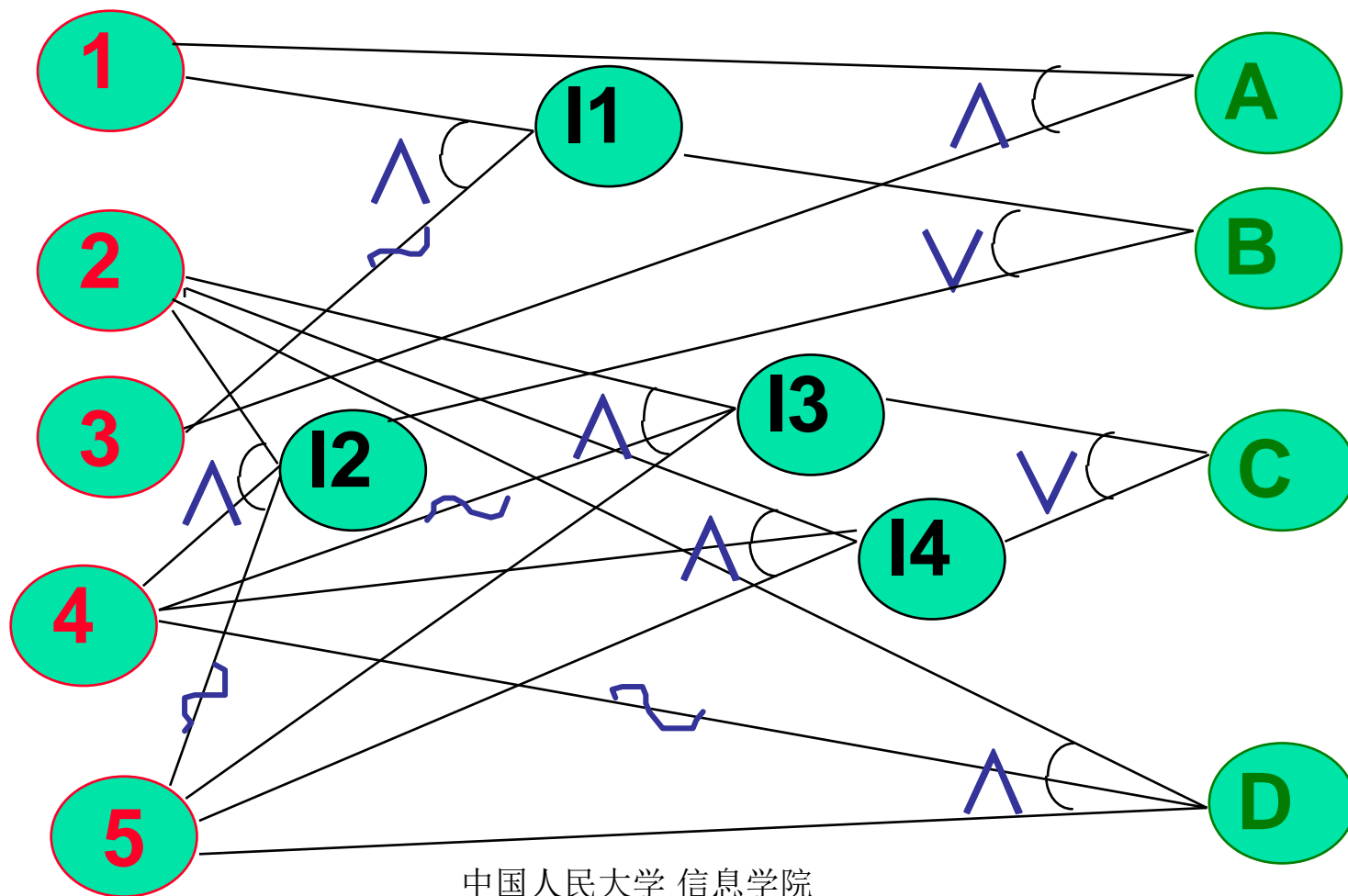
2—动力用电 B- B类计费

3— <100 度/月 C- C类计费

4— <10000 度/月 D- D类计费

5—高峰用电

输入和输出间的逻辑关系



把因果图转换为判定表

组合条件		1	2	3	4	5	6
条件 原因	1	1	1	0	0	0	0
	2	0	0	1	1	1	1
	3	1	0				
	4			1	0	1	0
	5			0	0	1	1
动作 结果	A	1	0	0	0	0	0
	B	0	1	1	0	0	0
	C	0	0	0	1	1	0
	D	0	0	0	0	0	1
测试用例							



为判定表每一列设计一个测试用例

条件 组合	测试用例 (输入数据)	预期结果 (输出动作)
----------	----------------	----------------

1列	居民电, 90度/月	A
2列	居民电, 110度/月	B
3列	动力电, 非高峰, 8000度/月	B
4列	动力电, 非高峰, 1.2万度/月	C
5列	动力电, 高峰, 0.9万度/月	C
6列	动力电, 高峰, 1.1万度/月	D



课堂练习题

为求平方根的程序设计测试用例：

$$\text{SQRT}((x - 3) / (5 - x))$$

采用黑盒测试，要求用两种方法设计测试用例：

- (1) 等价划分法；
- (2) 边界值法。



软件测试

7 针对环境和应用的测试



针对环境和应用的测试

- GUI测试
- Client/Server体系结构测试
- 文档测试
- 实时系统
- Web测试



GUI测试

常见GUI测试指南：

- 对于窗口
- 对于菜单和鼠标操作
- 对于数据项



C/S体系结构的测试

- **整体C/S测试策略（三个不同层次）：**
 - **客户端应以“分离的”模式被测试（不考虑服务器和底层网络的运行）**
 - **客户端软件和关联的服务器端应用被一起测试（网络运行不被明显考虑）**
 - **完整的C/S体系结构（包括网络运行和性能）被测试**



C/S常用测试方法

- 客户端应用功能测试
- 服务器测试（协调和数据管理功能、性能）
- 数据库测试
- 事务测试
- 网络通信测试



软件测试

8 软件测试的策略



软件测试的策略

- 验证 (verification) 和确认 (validation)
- 软件测试的组织
- 软件测试步骤：从下而上
- 测试完成的标准？

- 永远也不可能完成测试：开发者转向用户
- 时间不够或资金不够时
- Musa和Ackerman:1000小时CPU时间内不出错的概率大于0.995，则有95%信心说进行了足够的测试。
- 概率论模型 $f(t)=(1/p)\ln(l_0pt+1)$ ，求导得故障率 $l(t)$



软件测试的策略

■ 单元测试

单元测试 (unit testing)，是指对软件中的最小可测试单元进行检查和验证。

■ 集成测试

集成测试 (integration testing)，也叫组装测试或联合测试。在单元测试的基础上，将所有模块按照设计要求（如根据结构图）组装成为子系统或系统，进行集成测试。

■ 系统测试

系统测试 (system testing)，是对整个系统的测试，将硬件、软件、操作人员看作一个整体，检验它是否有不符合系统说明书的地方。

■ 确认测试

确认测试 (confirmation testing)，又称有效性测试，是对通过集成测试的软件在模拟的环境下进行的有效性测试，目的是要表明软件是可以工作的，并且符合“软件需求说明书”中规定的全部功能和性能要求。

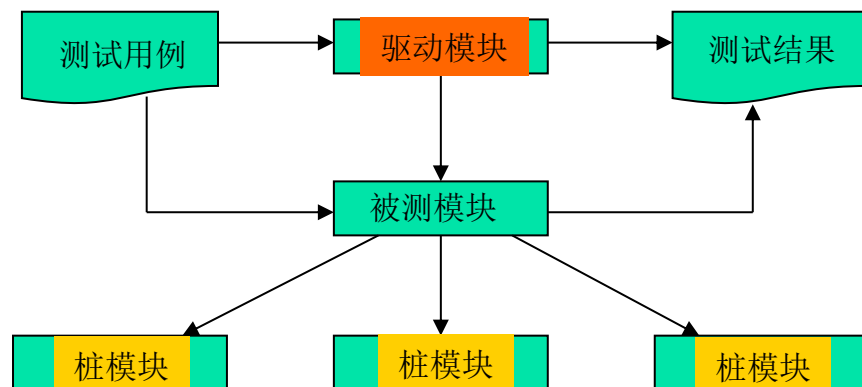
单元测试

■ 内容

- 接口测试
- 数据结构测试
- 路径测试
- 错误处理（非正常）测试
- 边界测试

■ 步骤

- 驱动模块——辅助模块
- 桩模块——辅助模块



驱动模块用来模拟被测模块的上层调用模块，功能要比真正的上级模块简单，仅仅是接受测试数据，并向被测模块传送数据，启动被测测试模块，回收并输出测试结果。

桩模块用来模拟被测模块在执行过程中所要调用的模块。它接受被测试输出的数据并完成它所指派的任务。



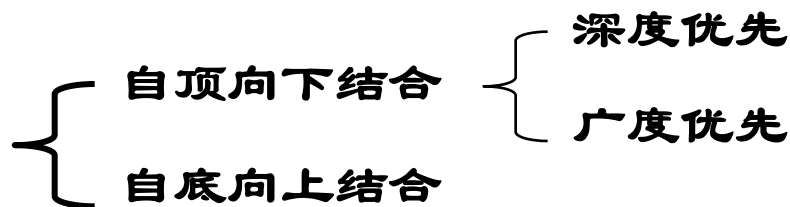
集成测试

- 模块连接时，穿越模块的数据是否会丢失；
- 是否会对另外模块产生不利影响；
- 各子模块组合能否达到预期的父模块功能；
- 全局数据结构是否有问题；
- 各模块误差的累积是否到了无法接受。
- 集成测试的策略
 - 一次性集成方式
 - 增殖式集成方式
 - 集成测试的组织和实施
 - 集成测试完成的标志



集成测试方法

- 通常采用黑盒测试技术
- 实施策略：
 - 非渐增式测试：一次就把所有通过了单元测试的模块组合在一起进行全程序的测试。
 - 缺点：发现错误难以诊断定位，又称“莽撞测试”。
 - 渐增式测试：从一个模块开始，测一次添加一个模块，边组装边测试，以发现与接口相联系的问题。





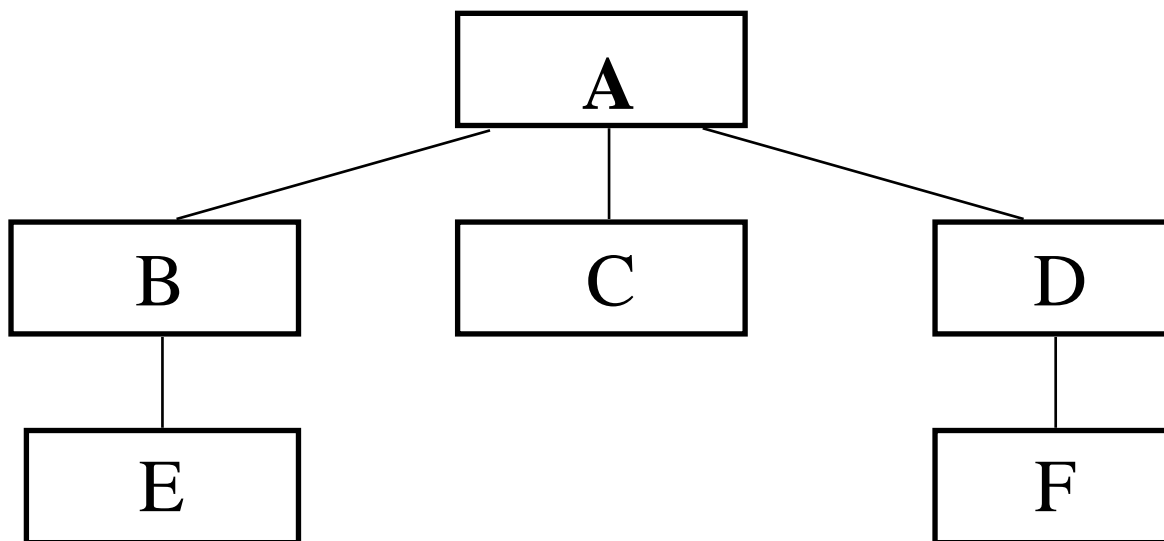
自顶向下集成

步骤：

- (1) 主控模块为驱动模块，所有直属主模块的下属模块全用桩模块代替，测试主模块。
- (2) 根据所选结合方法（先深度或先广度），每次用一实际模块替换相应桩模块。
- (3) 模块结合一个，测试一个。
- (4) 完成一组测试后，用实际模块替换另一个桩模块。
- (5) 为保证不引入新错误，进行回归测试。



自顶向下集成方式

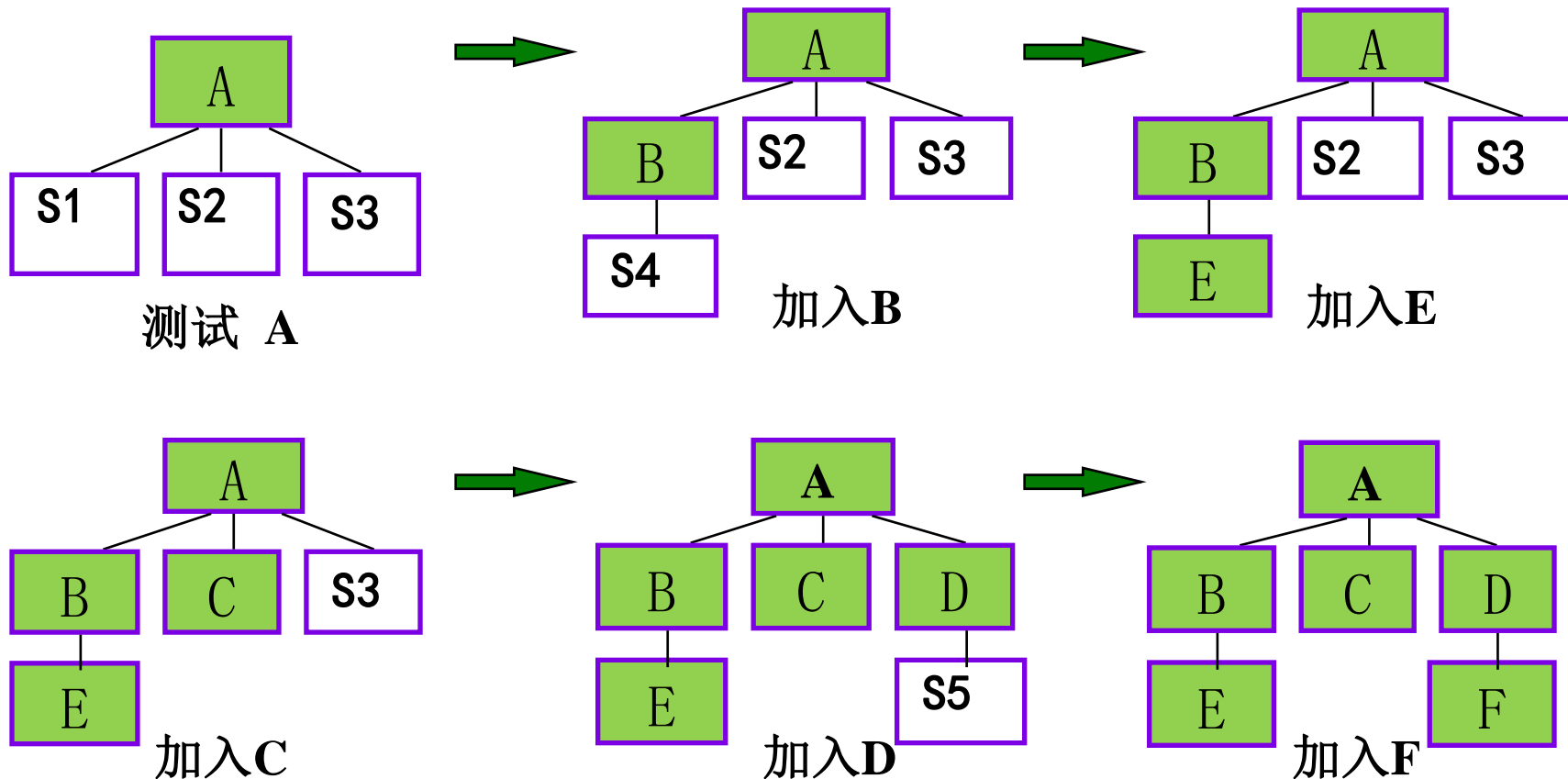


模块测试结合顺序

深度优先:A、B、E、C、D、F

广度优先:A、B、C、D、E、F

自顶向下集成方式(深度优先)





自顶向下集成的逻辑问题

- 测试初期，如上层模块对下层模块有很大的依赖性（要求下层模块返回的信息量大，种类多），而桩模块替换下层模块后，不能向高层模块传送足够量的信息。
- 解决办法：
 - 把一些测试推迟到用实际模块替换桩模块后进行。
 - 使桩模块能模拟实际模块功能。
 - 自底向上集成测试。

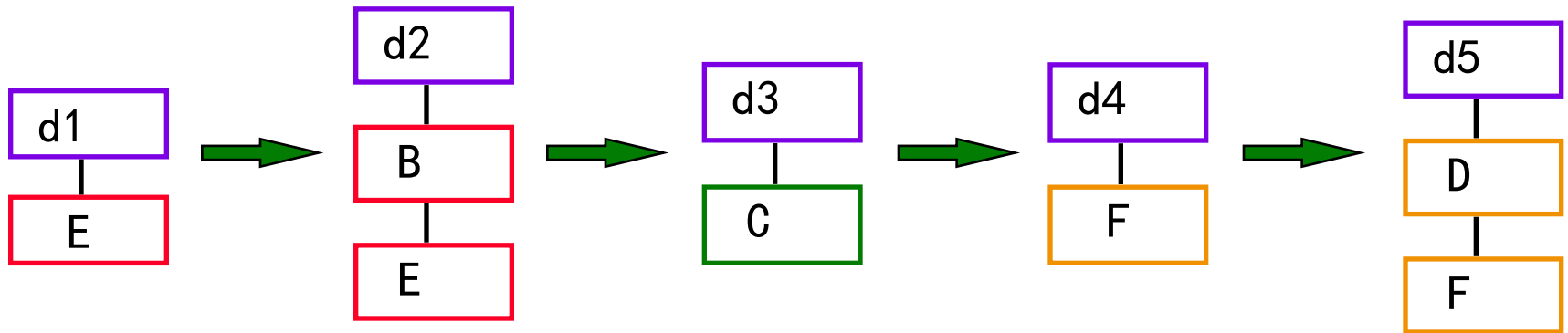
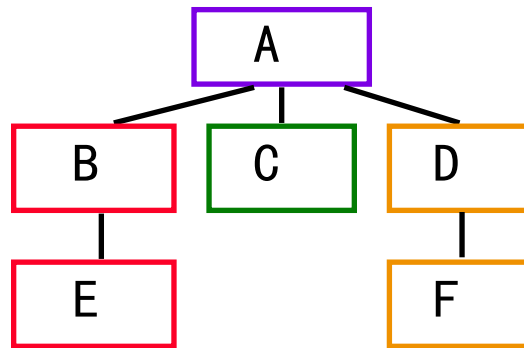


自底向上集成

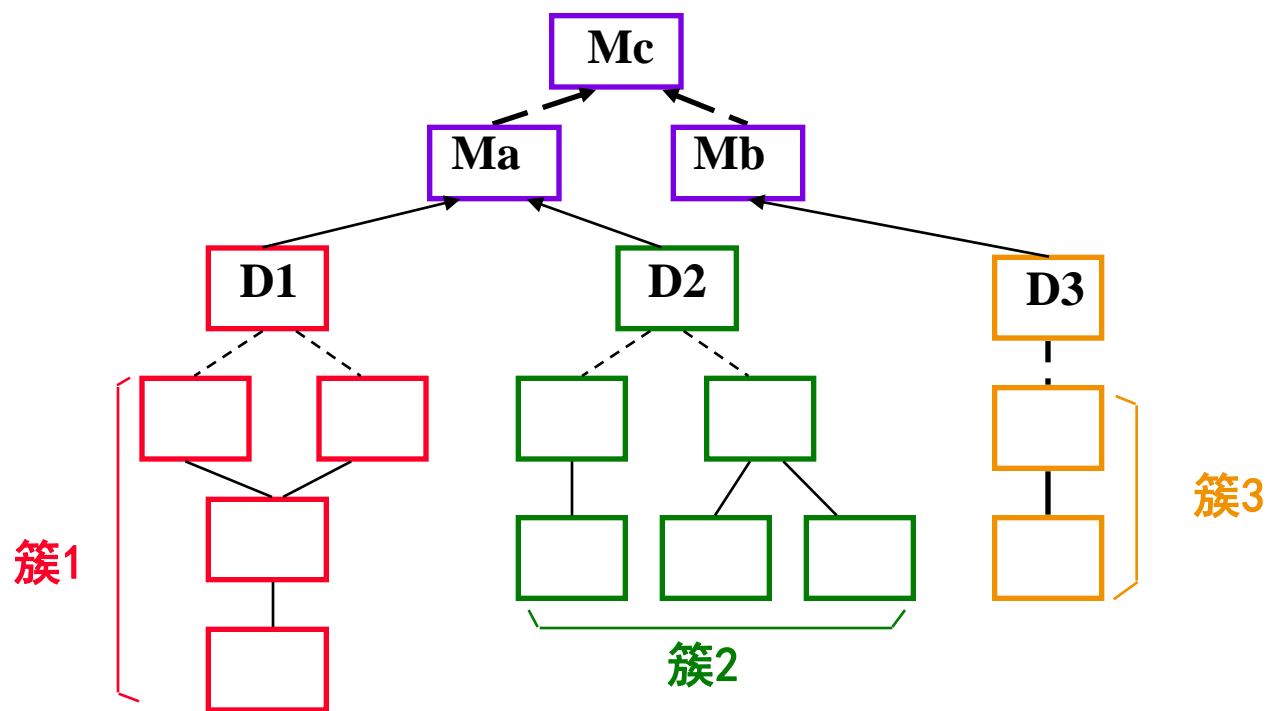
步骤：

- (1) 对叶模块配以驱动模块对其测试，也可把最底层模块组合成实现某一特定软件功能的簇，由驱动模块对它测试。**
- (2) 用实际模块代替驱动模块，与它已测试的直属模块组装成子系统。**
- (3) 为子系统配备驱动模块，进行新的测试。**
- (3) 判断是否已组装到达主模块 是则结束，否则执行(2)**

自底向上集成方式



自底向上集成方式





两种方法比较

- **自顶向下：**可在测试早期实现并验证系统主要功能不需驱动模块，但需要桩模块。
- **自底向上：**设计测试用例容易，但要到最后程序才能作为一个整体。
- **混合集成测试方法**
 - ◆ 一般对软件结构的上层使用自顶向下结合的方法；
 - ◆ 对下层使用自底向上结合的方法。



系统测试

■ 系统测试 (*System Testing*)

- 系统测试是将已经集成好的软件系统作为一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其他元素结合在一起，在实际运行环境下进行的一系列测试。

■ 系统测试方法

- 功能测试、协议一致性测试
- 性能测试、压力测试、容量测试、安全性测试、恢复测试
- 备份测试、GUI测试、健壮性测试、兼容性测试、可用性测试
- 安装测试、文档测试、在线帮助测试、数据转换测试



系统测试

■ 功能测试 (*Functional Testing*)

- 功能测试是系统测试中最基本的测试，它不管软件内部的实现逻辑，主要根据软件需求规格说明和测试需求列表，验证产品的功能实现是否符合需求规格。
- 功能测试主要发现以下错误：
 - 是否有不正确或遗漏的功能？
 - 功能实现是否满足用户需求和系统设计的隐藏需求？
 - 能否正确地接受输入？能否正确地输出结果？
- 常用的测试技术
 - 黑盒测试方法：等价类划分、边界值测试



系统测试

■ 压力测试 (*Press Testing*)

- 压力测试是检查系统在资源超负荷情况下的表现，特别是对系统的处理时间有什么影响。
- 压力测试的例子
 - 对于一个固定输入速率的单词处理响应时间
 - 在一个非常短的时间内引入超负荷的数据容量
 - 成千上万的用户在同一时间从网上登录到系统
 - 引入需要大量内存资源的操作
- 压力测试采用边界值和错误猜测方法，且需要工具的支持。



系统测试

- 安全性测试 (*Security Testing*)
 - 安全性测试检查系统对非法侵入的防范能力。
 - 安全性测试期间，测试人员假扮非法入侵者，采用各种办法试图突破防线。
 - 安全性测试的例子
 - 想方设法截取或破译口令
 - 专门定做软件破坏系统的保护机制
 - 故意导致系统失败，企图趁恢复之机非法进入
 - 试图通过浏览非保密数据，推导所需信息



系统测试

■ 恢复测试 (*Recovery Testing*)

- 恢复测试是检验系统从软件或者硬件失败中恢复的能力，即采用各种人工干预方式使软件出错，而不能正常工作，从而检验系统的恢复能力。
- 恢复性测试的例子
 - 当供电出现问题时的恢复
 - 恢复程序的执行
 - 对选择的文件和数据进行恢复
 - 恢复处理日志方面的能力
 - 通过切换到一个并行系统来进行恢复



系统测试

- GUI测试 (*Graphic User Interface Testing*)
 - GUI 测试一是检查用户界面实现与设计的符合情况，二是确认用户界面处理的正确性。
 - GUI 测试提倡界面与功能的设计分离，其重点关注在界面层和界面与功能接口层上。
 - GUI自动化测试工具
 - WinRunner, QARun, QARobot, Visual Test
 - 常用的测试技术
 - 等价类划分、边界值分析、基于状态图方法、错误猜测法



系统测试

■ 安装测试 (*Installation Testing*)

- 系统验收之后，需要在目标环境中进行安装，其目的是保证应用程序能够被成功地安装。

■ 安装测试应考虑

- 应用程序是否可以成功地安装在以前从未安装过的环境中？
- 应用程序是否可以成功地安装在以前已有的环境中？
- 配置信息定义正确吗？
- 考虑到以前的配置信息吗？
- 在线文档安装正确吗？
- 安装应用程序是否会影响其他的应用程序吗？
- 安装程序是否可以检测到资源的情况并做出适当的反应？



确认测试

■ 验收测试 (*Acceptance Testing*)

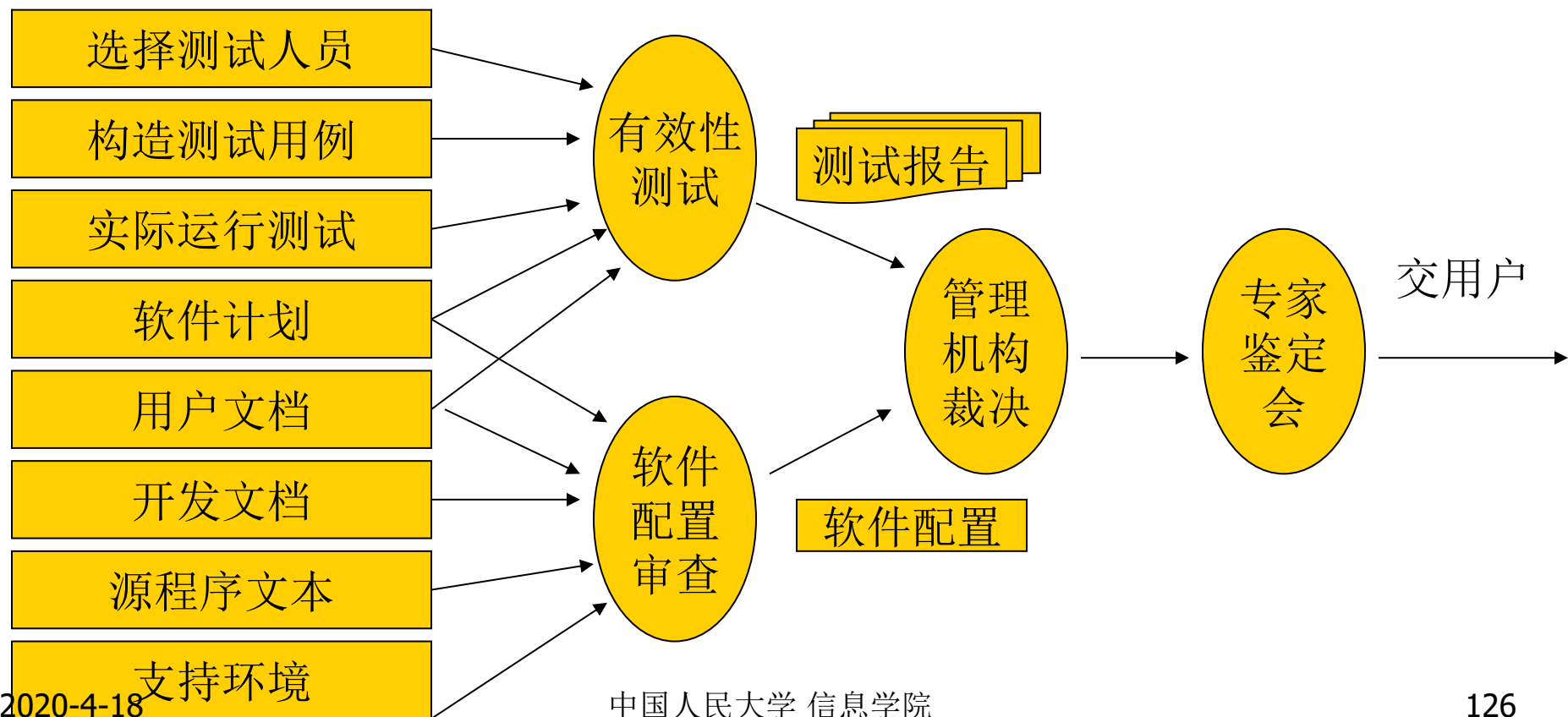
- 验收测试是以用户为主的测试，一般使用用户环境中的实际数据进行测试。
- 在测试过程中，除了考虑软件的功能和性能外，还应
对软件的兼容性、可维护性、错误的恢复功能等进行
确认。

■ α 测试与 β 测试

- α 测试与 β 测试是产品在正式发布前经常进行的两种测试；
 - α 测试是由用户在开发环境下进行的测试；
 - 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。

确认测试

验证软件的功能和性能及其他特征是否与用户的要求一致。





程序的静态分析方法

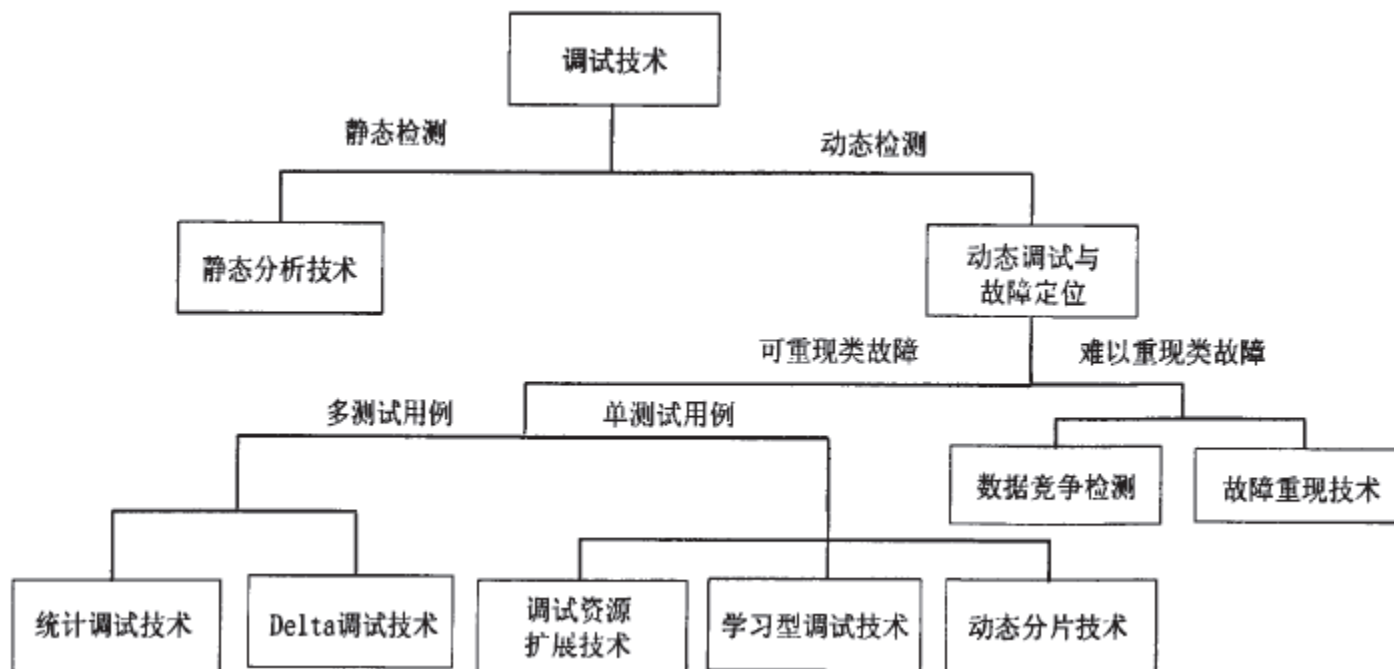
- 对程序的静态分析
 - 生成各种引用表
 - 静态错误分析
- 人工测试
 - 桌前检查
 - 代码评审
 - 走查



软件测试

9 软件测试工具

软件调试技术分类





软件测试工具主要应用在下列方面:

- ☐ 静态分析 (Static analyzer)
- ☐ 代码审计 (Code Auditors)
- ☐ 断言处理 (Assertion processors)
- ☐ 测试文件生成 (Test file generators)
- ☐ 测试数据生成 (Test Data Generators)
- ☐ 测试验证 (Test Verifiers)
- ☐ 输出比较 (Output comparators)



思考题

- ① 保证绝对正确的程序可行吗？说明理由。
- ② 如何计算复杂度
- ③ 测试与软件开发周期的关系
- ④ 四种测试策略的内容

习题

☀ 习题二：A: (2) B: (5) C: (1) D: (1) E: (4)

在设计测试用例时，A 使用的最多的一种黑盒方法。黑盒测试中，等价类划分方法设计测试用例步骤是：

1. 根据输入条件把输入数据分成若干有效等价类和无效等价类。
2. 设计一个用例，使其覆盖B 尚未被覆盖的有效等价类，重复这一步，直到所有有效等价类均被覆盖。
3. 设计一个用例，使其覆盖C 尚未被覆盖的无效等价类，重复这一步，直到所有无效等价类均被覆盖。

因果图方法是根据D 之间的因果关系来设计用例的。

在实际中，纠正了程序中的错误后，还应测试原先测试过的测试用例，对修改过的程序重新测试，这种测试称为E。

A: (1) 等价类划分 (2) 边值分析 (3) 因果图 (4) 判定表

B、C: (1) 1个 (2) 7个左右 (3) 一半 (4) 尽可能少 (5) 尽可能多 (6) 全部

D: (1) 输入与输出 (2) 设计和实现 (3) 条件与结果 (4) 主程序与子程序

2020-4-18 E: (1) 验收测试 (2) 强度测试 (3) 系统测试 (4) 回归测试



习题

☀ 习题三：A: (4) B: (3) C: (1) D: (2) E: (3)

 A 在实现阶段进行，它所依据的模块功能描述和内部细节以及测试方案应在 B 阶段完成，目的是发现编程错误。

 C 所依据的模块说明书和测试方案应在 D 阶段完成，它能发现设计错误。

 E 应在模拟的环境中进行强度测试的基础上进行，测试计划应在软件需求分析阶段完成。

- | | | | |
|---------------|------------|------------|------------|
| A: (1) 用户界面测试 | (2) 输入输出测试 | (3) 集成测试 | (4) 单元测试 |
| B: (1) 需求分析 | (2) 概要设计 | (3) 详细设计 | (4) 结构设计 |
| C: (1) 集成测试 | (2) 可靠性测试 | (3) 系统性能测试 | (4) 强度测试 |
| D: (1) 编程 | (2) 概要设计 | (3) 维护 | (4) 详细说明 |
| E: (1) 过程测试 | (2) 函数测试 | (3) 确认测试 | (4) 逻辑路径测试 |