BUPT
TSEG

软件工程 模型与方法
Models & Methods of SE

软件测试

田野: yetian@bupt.edu.cn

本章内容

- 软件测试的目的和原则
  - 软件测试的对象
  - 测试流程
  - 测试与软件开发各阶段的关系
- 软件测试方法
- 软件测试策略
- 软件测试种类
- 软件调试

软件测试的目的

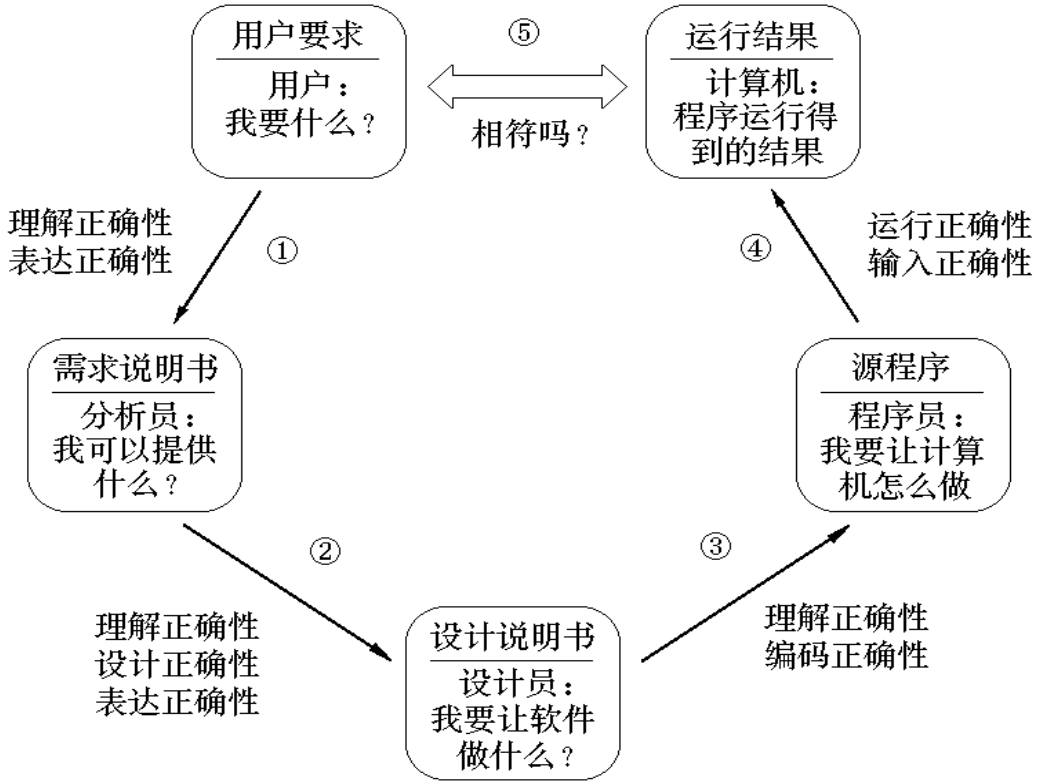
- 从用户的角度：通过软件测试暴露软件中的错误和缺陷，以考虑是否可接受该软件产品。
- 从软件开发者的角度：希望测试成为表明软件产品中不存在错误，验证该软件已正确地实现了用户的要求，确立人们对软件质量的信心。
- Glenford J.Myers：软件测试目的
  - 测试是程序的执行过程，目的在于发现错误；
  - 一个好的测试用例在于能发现至今未发现的错误；
  - 一个成功的测试是发现了至今未发现的错误的测试；
  - 测试不能表明软件中不存在错误，只能说明软件中存在错误。
- 软件测试的定义：
  - 以最少的时间和人力，系统地找出软件中潜在的各种错误和缺陷，并证明软件的功能和性能与需求说明相符合

软件测试的原则

- 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。
- 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。
- 程序员应避免检查自己的程序。
- 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。
- 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。
- 严格执行测试计划，排除测试随意性。应当对每一个测试结果做全面检查。
- 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

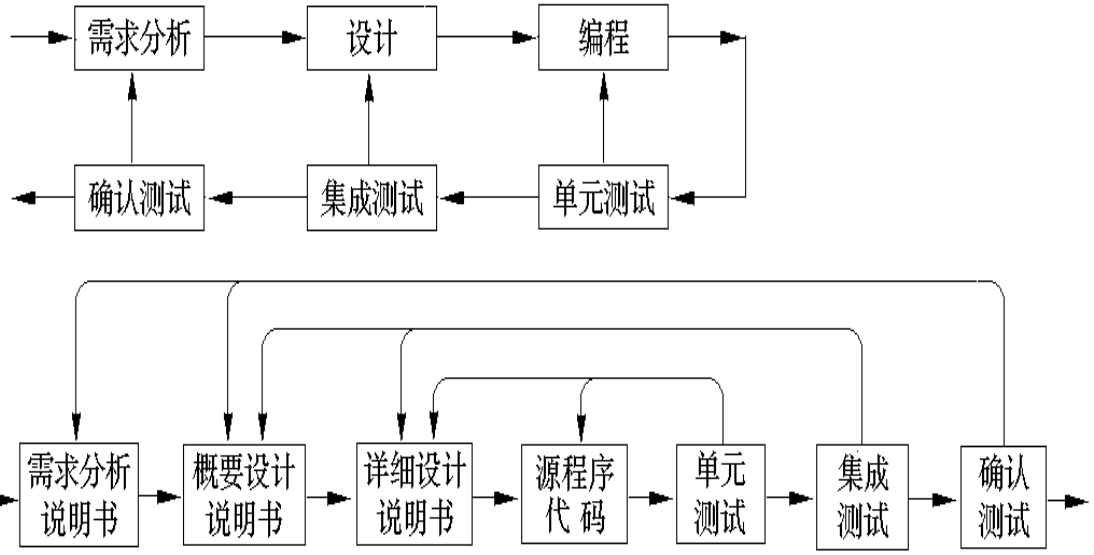
- 软件测试并不等于程序测试。软件测试应贯穿于软件定义与开发的整个周期。
- 软件开发各阶段所得到的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

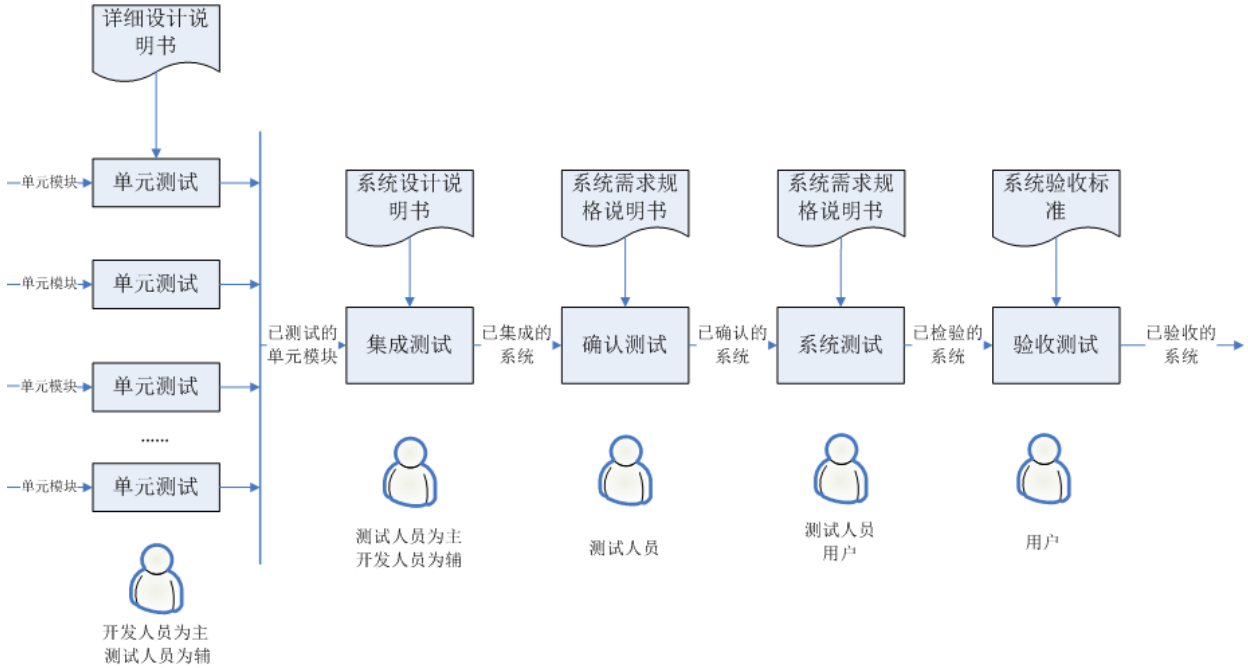


用例编号	XT-YH-0001	用例名称	增加用户-合法输入
测试功能	增加用户	测试目的	在各项用户数据均合法的情况下，系统能够正确添加一个用户。
业务说明	此处给出该系统在增加用户时，各项输入字段的限制要求，以及系统处理的业务规则。		
前置条件	此处说明进行该功能操作时，所需要的数据环境、登录用户所需具有权限等说明		
操作过程	1. 执行“系统管理”下的“用户管理”，出现用户管理页面； 2. 单击“用户管理”页面中的“新增用户”按钮，弹出增加用户窗口； 3. 在“增加用户”窗口中输入用户各项信息，每个字段均符合业务说明中的限制要求； 4. 单击“确定”按钮。		
预期结果	系统提示“增加用户成功”，在返回的“用户管理”页面的用户列表中，显示出新增用户信息。		
其他说明	无。		
用例设计人	张三	用例审核人	李四

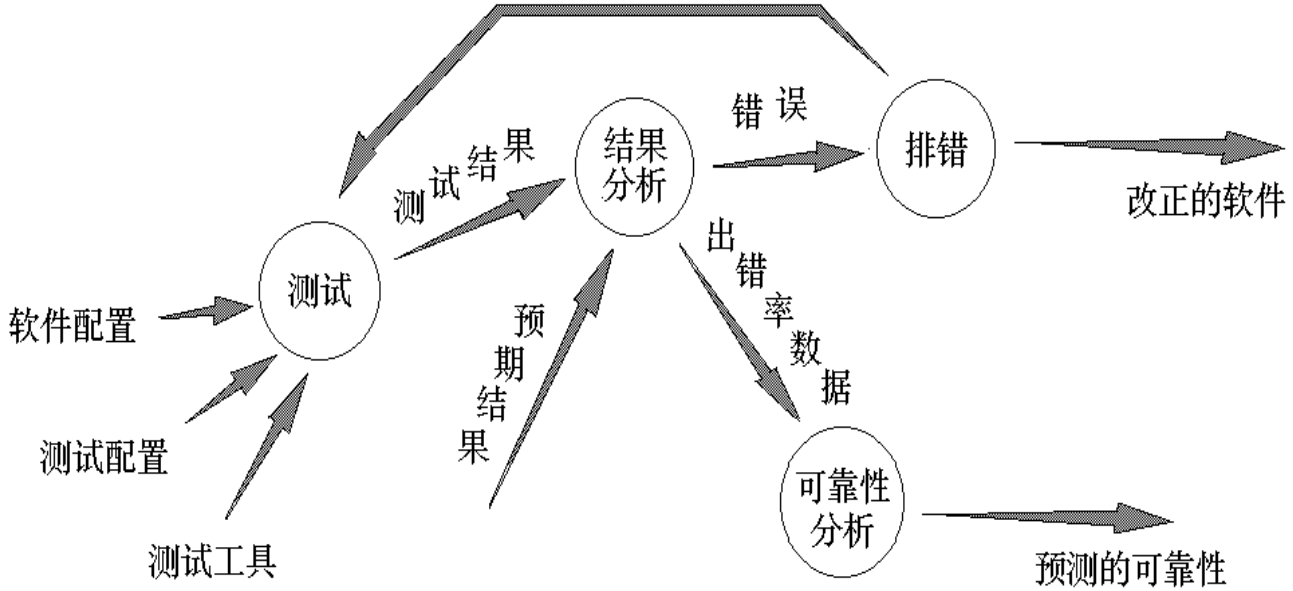
软件测试与软件各阶段的关系

- 软件测试是软件实现之后开始的一系列测试活动；





软件测试流程



- 测试结果分析：比较实测结果与预期结果，评价错误是否发生以及错误的级别和严重性。
- 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。

- 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。

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接受的程度
- 是否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，只能说明测试配置考虑得不够细致充分，错误仍然潜伏在软件中

两种常用的测试方法

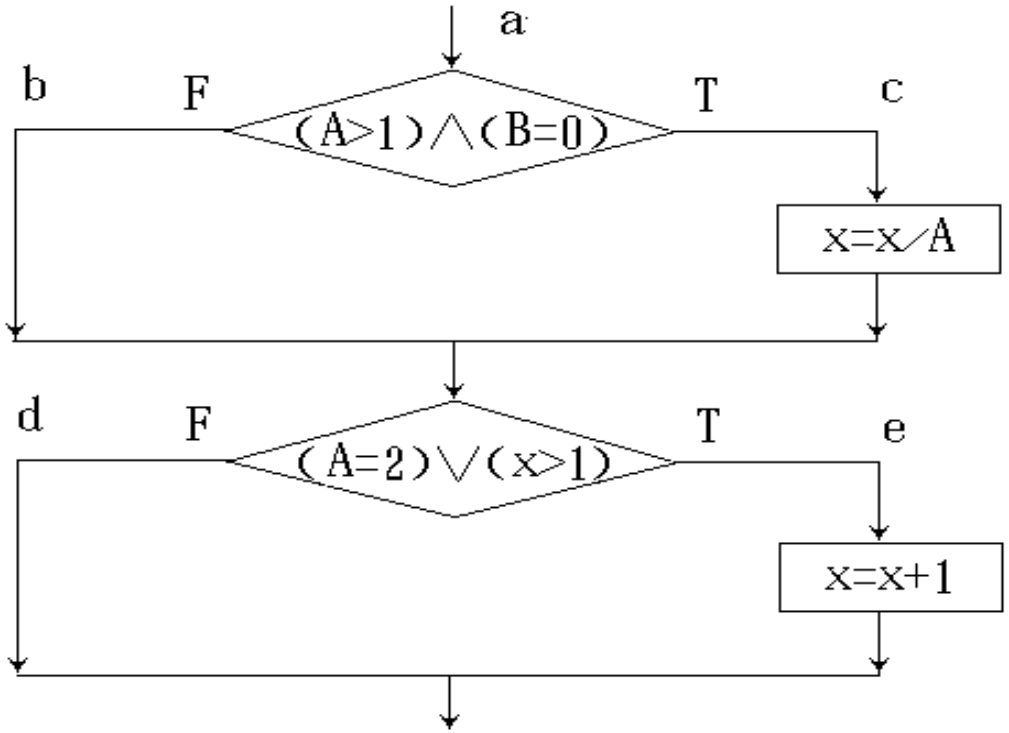
- 白盒测试：将测试对象看做一个透明的盒子，允许利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致，又称为结构测试或逻辑驱动测试。
- 黑盒测试：这种方法完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书和概要设计说明，检查程序的功能是否符合它的功能说明，又称为功能测试或数据驱动测试

白盒测试方法

- 白盒测试主要应用于单元测试，是检查程序逻辑错误的主要方法。
- 使用白盒测试方法，主要对程序模块进行如下的检查：
  - 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。
- 逻辑覆盖：逻辑覆盖是以程序内部的逻辑结构为基础设计的测试用例技术
  - 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

逻辑覆盖

- 确定测试路径的逻辑表达： L1,L2,L



$$L_1(a \to c \to e)$$

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

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

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

$$L_2\ (a \to b \to d)$$

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

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

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

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

$$L_3(a \to b \to e)$$

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

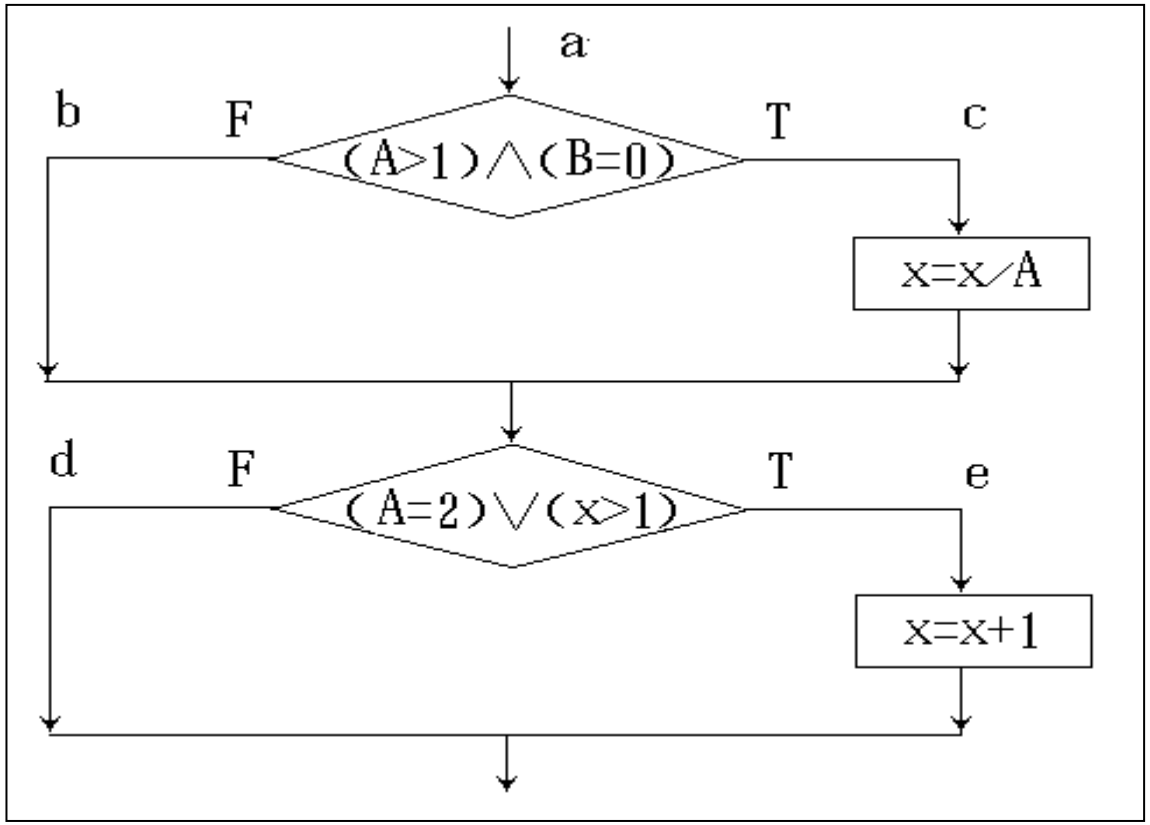
$$= \{\overline{(A>1)}\ or\ \overline{(B=0)}\}\ and\ \{(A=2)\ or\ (X>1)\}$$

$$= \overline{(A>1)}\ \text{and}\ (X>1)\ \text{or}$$

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

$$= (A \le 1)\ and\ (X>1)\ or$$

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



$$L_4(a \to c \to d)$$

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

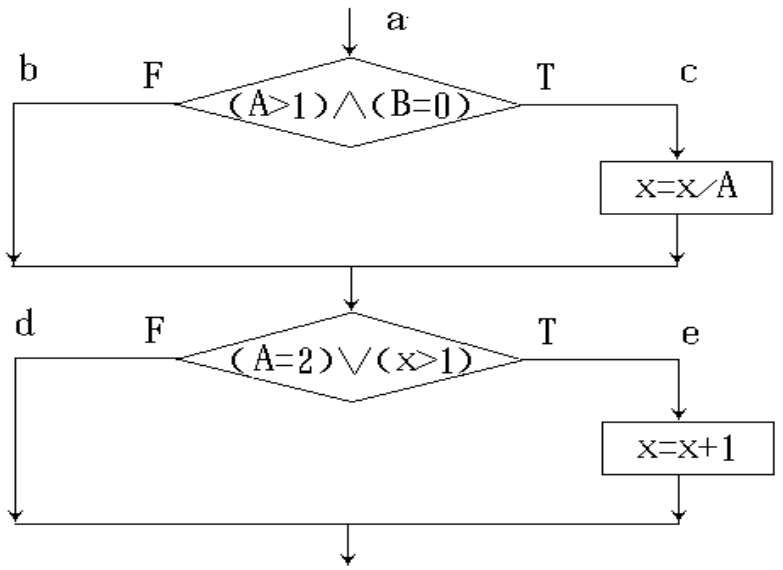
$$= (A>1)\ and\ (B=0)\ and\ (A \ne 2)\ and\ (X/A \le 1)$$

语句覆盖

- 语句覆盖：使得每一个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)**】
- 满足语句覆盖的测试用例是:
  【(2, 0, 4)，(2, 0, 3)】 覆盖 ace【L_1】

- 如果第一个判断的 “and” 错写成 “or” 或者第二个判断的 “or” 错写成 “and”
- 利用上面的测试用例，仍可覆盖所有四个执行语句
- 语句覆盖发现不了判断中逻辑运算中出现的错误，是最弱的逻辑覆盖准则



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

判定覆盖

- 判定覆盖：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，又称为分支覆盖。
- 根据要求，可以选择路径L1和L2，也可以选择L3和L4。

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

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

【(2, 0, 4)，(2, 0, 3)】覆盖 ace【L_1】
【(1, 1, 1)，(1, 1, 1)】覆盖 abd【L_2】

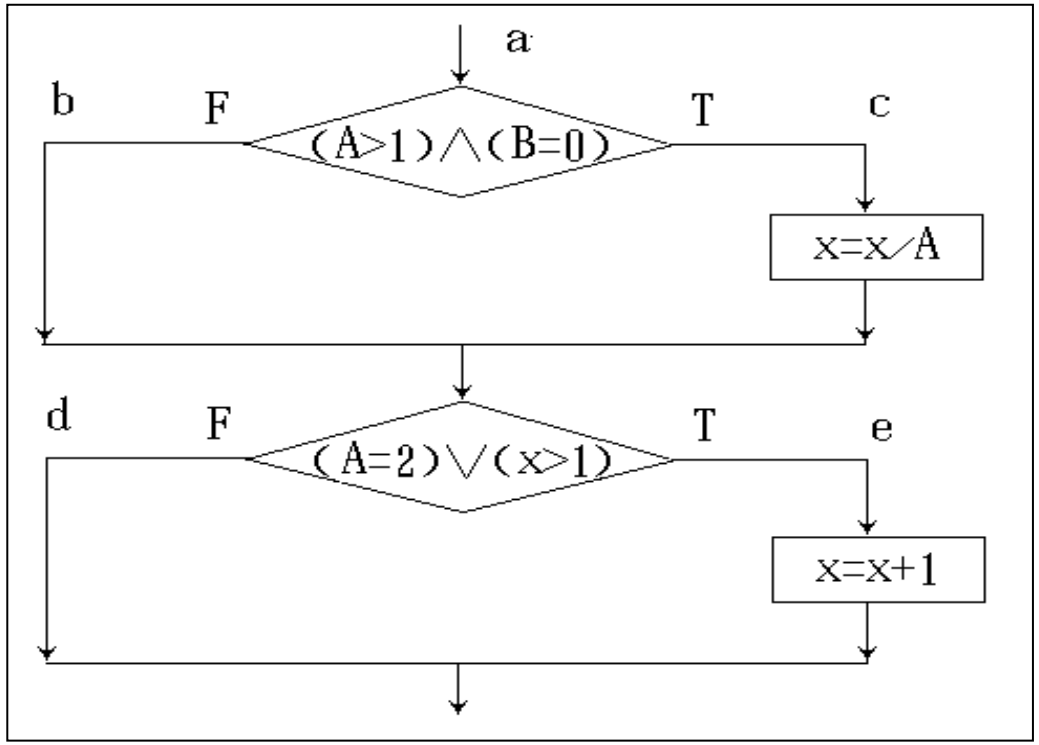
$$(A\leq 1)\ and\ (X>1)\ or\ (B\neq 0)\ and\ (A=2)\ or\ (B\neq 0)\ and\ (X>1)$$

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

如果选择路径L3和L4，有测试用例:
【(2, 1, 1)，(2, 1, 2)】覆盖 abe【L_3】
【(3, 0, 3)，(3, 0, 1)】覆盖 acd【L_4】

判定覆盖的问题

- 如果把第二个判断中的条件X>1错写成X<1
- 利用上面两组测试用例，仍能得到同样的结果
- 判定覆盖还不能保证一定查出在判断的条件中存在的错误，因此需要更强的逻辑覆盖准则检验判断内部条件



条件覆盖

- 条件覆盖：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。
- 第一判定表达式：
  - 条件 A>1 取真 记为T_1，取假 记为 $\overline{T_1}$
  - 条件 B=0 取真 记为T_2，取假 记为 $\overline{T_2}$

如果选择路径L1和L2，有测试用例：
【(2, 0, 4)，(2, 0, 3)】覆盖 ace【L_1】 $T_1\ T_2\ T_3\ T_4$
【(1, 0, 1)，(1, 0, 2)】覆盖 abd【L_2】 $\overline{T_1}\ T_2\ \overline{T_3}\ T_4$

- 第二判定表达式：
  - 条件 A=2 取真 记为T_3，取假 记为 $\overline{T_3}$
  - 条件 X>1 取真 记为T_4，取假 记为 $\overline{T_4}$

如果选择路径L2和L3，有测试用例：
【(1, 0, 3)，(1, 0, 4)】覆盖 abd【L_2】 $\overline{T_1}\ T_2\ \overline{T_3}\ T_4$
【(2, 1, 1)，(2, 1, 1)】覆盖 abe【L_3】 $T_1\ \overline{T_2}\ T_3\ \overline{T_4}$

- 后一组用例虽满足了条件覆盖，但只覆盖了第一个判断的取假分支和第二个判断的取真分支，不满足判定覆盖的要求，因此，需要对条件和分支兼顾

判定-条件覆盖

- 判定－条件覆盖：使得判断中每个条件的所有可能取值至少执行一次，同时每个判断的所有可能判断结果取值至少执行一次。
- 基于上述条件，该例子可以选择L1和L2

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

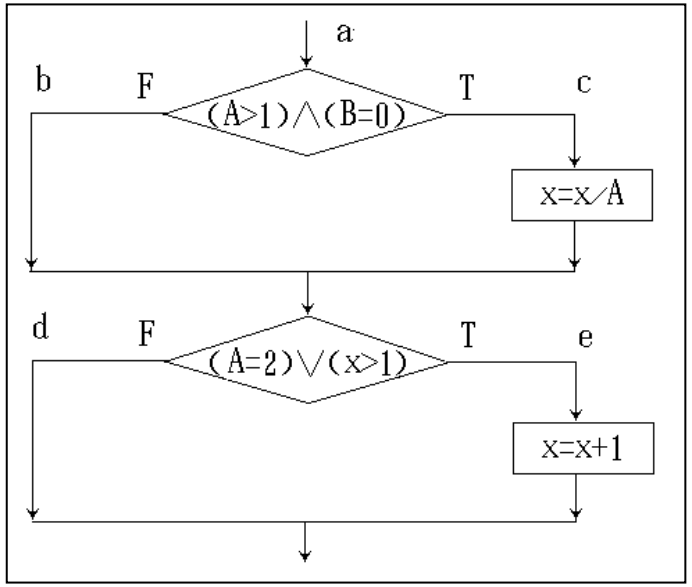
$$(A\le 1)\ and\ (X\le 1)\ or\ (B\ne 0)\ and\ (A\ne 2)\ and\ (X\le 1)$$

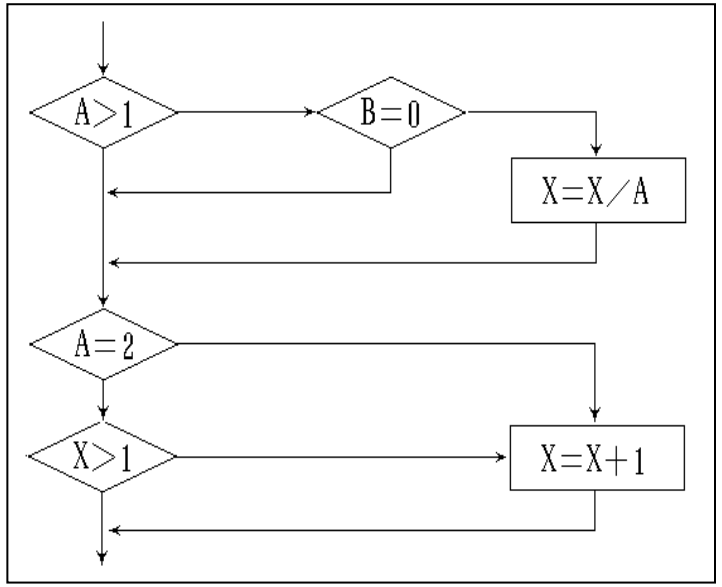
如果选择路径L1和L2，有测试用例:
【(2, 0, 4)，(2, 0, 3)】覆盖 ace【L_1】 $T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】覆盖 abd【L_2】 $T_1T_2T_3T_4$

- 对于条件表达式(A>1)&(B=0)
- 如果(A>1)的测试结果为真，则还要测试(B=0)，才能决定表达式的值
- 如果(A>1)的结果为假，则立刻确立表达式的结果为假，往往就不再测试(B=0)的取值了
- 因此采用判定-条件覆盖，逻辑表达式中的错误不一定能够查得出来

条件组合覆盖

- 条件组合覆盖：使得每个判断的所有可能的条件取值组合至少执行一次。
  - 需要将多重条件判断分解成有多个基本判断组成的流程图。





1.A > 1,B=0 T_1T_2
2.A > 1,B≠0 $T_1\overline{T_2}$
3.A≯1, B=0 $\overline{T_1}T_2$
4.A≯1, B≠0 $\overline{T_1}\overline{T_2}$
5.A=2, X > 1 T_3T_4
6.A=2, X≯1 $T_3\overline{T_4}$
7.A≠2, X > 1 $\overline{T_3}T_4$
8.A≠2, X≯1 $\overline{T_3}\overline{T_4}$

测试用例	覆盖组合
【(2, 0, 4)，(2, 0, 3)】L1	1,5
【(2, 1, 1)，(2, 1, 2)】L3	2,6
【(1, 0, 3)，(1, 0, 4)】L3	3,7
【(1, 1, 1)，(1, 1, 1)】L2	4,8

- 这组测试用例覆盖了所有条件的可能取值的组合
- 覆盖了所有判断的可取分支
- L4 缺失，测试不完整

路径测试

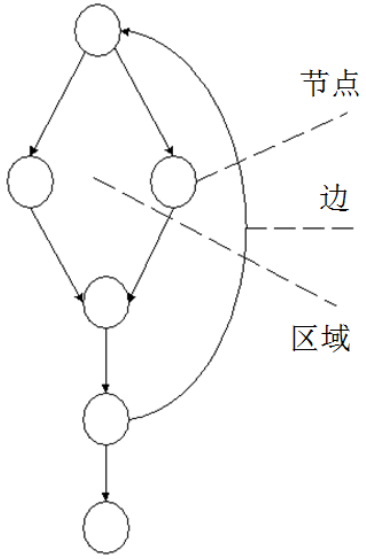
- 路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。
- 需要选择能够覆盖所有路径的测试用例：

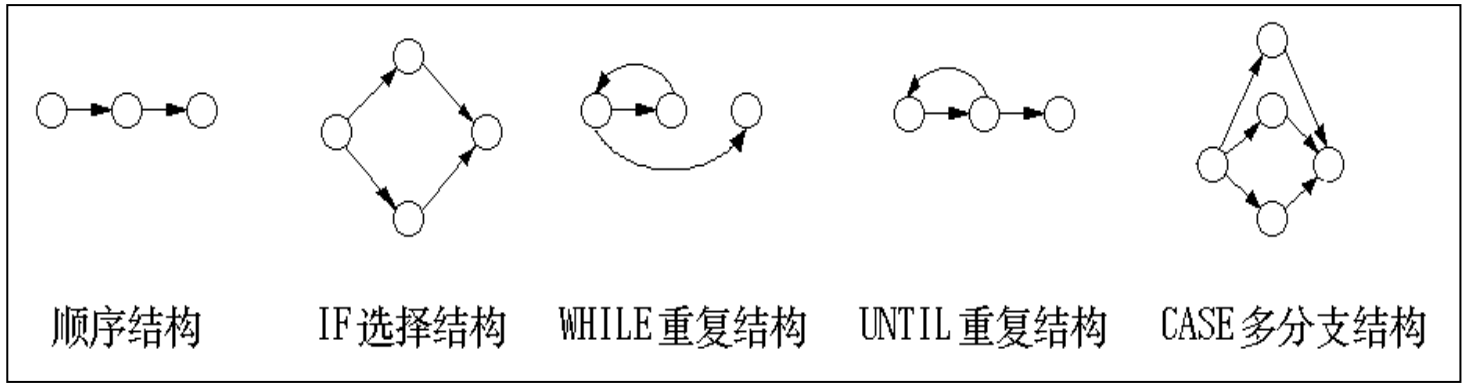
测试用例	覆盖条件
【(2, 0, 4)，(2, 0, 3)】L1 (ace)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】L2 (abd)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$
【(1, 1, 2)，(1, 1, 3)】L3 (abe)	$\overline{T_1}\overline{T_2}\overline{T_3}T_4$
【(3, 0, 3)，(3, 0, 1)】L4 (acd)	$T_1T_2\overline{T_3}\overline{T_4}$

- 这种测试仍然无法满足条件组合覆盖某些条件，也并非是完善的测试方法。
- 在实际的测试用例设计过程中需要综合以上6种测试方法去设计测试用例。
- 以路径覆盖为主，辅助其他5种方法。

基本路径测试

- 对于具有循环结构的程序而言，其路径数有可能很多，要求做到路径覆盖有难度。基本路径测试方法力图把覆盖的路径数压缩到一定限度内，使得程序中的循环体最多只执行一次。
- 这个方法需引入程序控制流图：基于程序流程图进行简化，得到程序的控制结构。
- 进而分析控制结构的环路复杂性，导出基本可执行路径集合，设计测试用例的方法。设计出的测试用例要保证在测试中，程序的每一个可执行语句至少要执行一次。



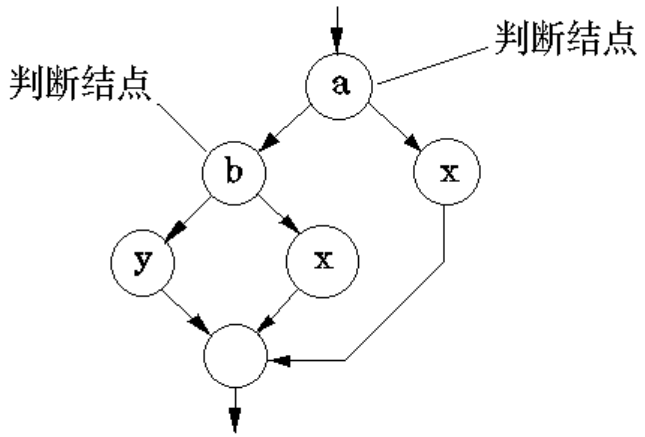


控制流图的转换

- 顺序结构的多个结点可以合并为一个结点。
- 在选择或多分支结构中，分支的汇聚处应有一个虚拟汇聚结点。
- 边和结点圈定的范围叫做区域，当对区域计数时，图形外的范围也应记为一个区域。
- 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)

控制流图的环路复杂度计算

- 控制流图的环路复杂度（也称为McCabe复杂度）确定了程序中独立路径的上界，以此为依据可以找出程序中的全部独立路径。
- 环路复杂度有三种计算方法：
  - 等于控制流图中的区域数，包括封闭区域和开放区域；
  - 设E为控制流图的边数，N为图的结点数，则定义环路复杂性为 V(G) = E − N + 2；
  - 若设P为控制流图中的判定结点数，则有 V(G) = P + 1。
- 基本路径集：指程序的控制流图中，从入口到出口的路径，该路径至少经历一个从未走过的边。
  - 基本路径集不是唯一的，对于给定的控制流图，可以得到不同的基本路径集。
  - 最大的基本路径条数就是环路复杂度。

导出测试用例

- 根据控制流图的基本路径导出测试用例，确保基本路径集中每一条路径的执行。
- 根据判断结点给出的条件，选择适当的数据以保证每一条路径可以被测试到，考虑使用逻辑覆盖方法。
- 每个测试用例执行之后，与预期结果进行比较。
- 如果所有测试用例都执行完毕，则可以确信程序中所有的可执行语句至少被执行了一次。

控制流图转换举例

```
void Func(int iRecordNum,int iType)
1{
2    int x=0;
3    int y=0;
4    while (iRecordNum > 0)
5     {
6        if(0= =iType)
7          {x=y+2; break;}
8        else
9            if (1= =iType)
10               {x=y+10; iRecordNum--;}
11           else
12               {x=y+20; iRecordNum--;}
13    }
14 }
```

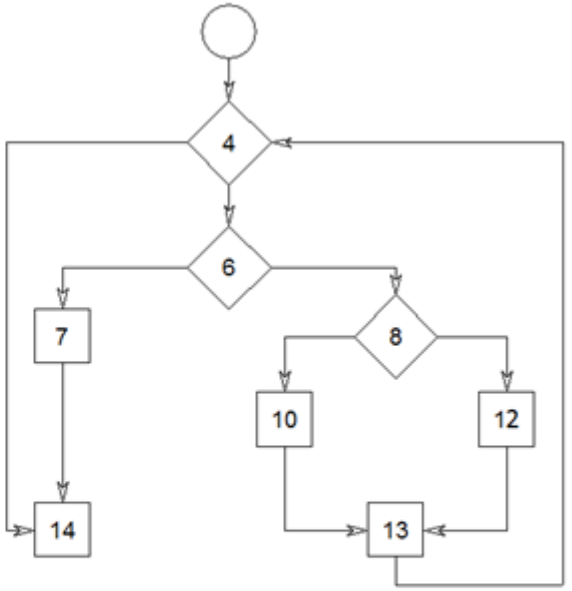
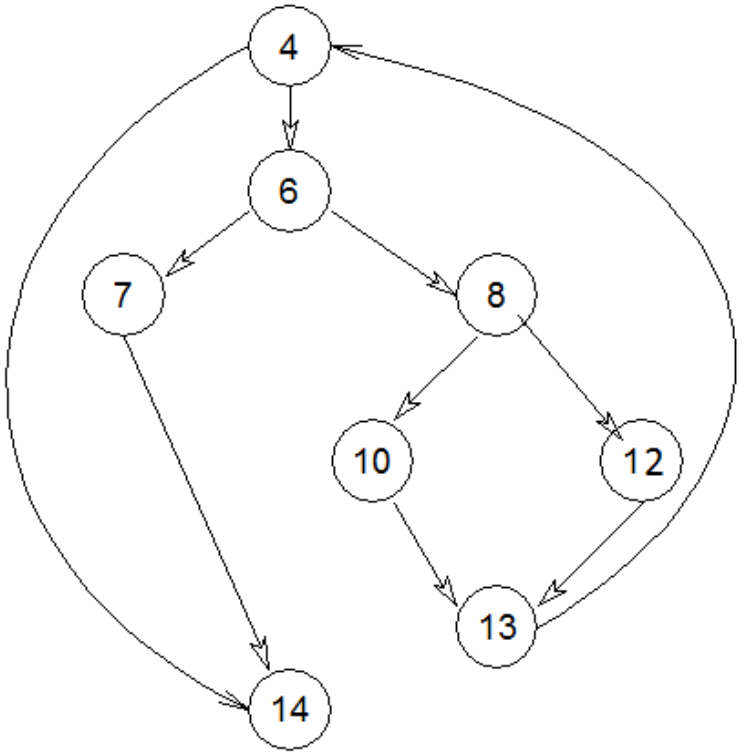


图 10-18 程序流程图



- 路径1：4-14　　输入数据：iRecordNum = 0；预期结果：x = 0；
- 路径2：4-6-7-14　　输入数据：iRecordNum = 1,iType = 0；预期结果：x = 2；
- 路径3：4-6-8-10-13-4-14　　输入数据：iRecordNum = 1,iType = 1；预期结果：x = 10；
- 路径4：4-6-8-12-13-4-14　　输入数据：iRecordNum = 1,iType = 2；预期结果：x = 20。

黑盒测试方法

- 相对于白盒测试，黑盒测试是在不需要了解程序结构的基础上，根据概要设计或者需求分析的结果进行测试用例的设计。常用的方法有：
  - 等价类划分；
  - 边界值分析；
  - 因果图；
- 黑盒测试方法一般用于集成测试、系统测试和验收测试，某些特殊情况也会用到单元测试。
- 黑盒测试方法用于测试程序接口，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？

等价类划分

- 黑盒测试方法不能选用穷举方式，为此通过寻找具有代表意义的数据进行替代其它同类型的数据，称为等价类。
- 并合理地假定：测试某等价类的代表值就等价于对这一类其它值的测试。
- 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
  - 划分等价类：根据输入域的要求和数据类型定义寻找等价类，确定等价类表结构。
  - 等价类的划分有两种不同的情况：
    - 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。
    - 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。

划分等价类原则

- 按区间划分：如果某个字段的输入条件属于一个取值范围[x,y]，则可以确立
  - 一个有效等价类;
  - 两个无效等价类。
- 按数值集合划分：如果输入条件规定了输入数据的集合，则可划分
  - 一个有效等价类：所有符合输入条件的数据集合，
  - 一个无效等价类：所有不允许输入的数据集合。
- 如果输入条件是一个布尔量，则可以确定
  - 一个有效等价类，即取真;
  - 一个无效等价类，即取假。
- 按数值划分：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，这时可以为
  - 每一个输入值确立一个有效等价类;
  - 一个无效等价类，包含所有不允许输入的数值
- 按限制条件或规则划分：如果规定了输入数据必须遵守的规则或限制条件，则可以确立
  - 一个有效等价类，即各方面均符合规则要求;
  - 若干个无效等价类，每个无效等价类从不同角度违反输入规则。

确定测试用例

- 为每一个等价类规定一个唯一编号;
- 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;
- 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。

等价类划分举例-1

- 需求分析要求：在某高校教师管理系统中，增加一个教师用户时，需要输入以下字段：工作证号、姓名、密码、参加工作时间等字段，其中的字段有如下要求：
  - 工作证号必须是整数，范围区间为[1,5000]，不能为空；
  - 姓名必须是中文字符，不能超过20个中文汉字，不能为空；
  - 密码必须大于等于6位，必须包括数字和字母；
  - 参加工作时间必须是8位数字，格式为YYYYMMDD，例如20130525。
- 等价类表结构
  - 输入条件：表示等价类的种类；
  - 有效等价类
  - 无效等价类
  - 编号

输入条件	有效等价类	编号	无效等价类	编号
工作证号	整数	1	非整数	11
	[1,5000]	2	<1	12
			>5000	13
姓名	中文字符	3	包含非中文字符	14
	不超过 20 个汉字	4	超过 20	15
	不能为空	5	空	16
密码	长度大于等于 6	6	长度为 6 位以下	17
	必须包括数字和字母	7	只包含数字	18
			只包含字母	19
			不包含数字和字母	20
参加工作时间	8 位	8	不是 8 位	21
	数字	9	出现非数字	22
	YYYYMMDD	10	YYYY<1	23
			MM>12	24
			MM<1	25
			DD>31	26
			DD<1	27

等价类划分举例-1

用例序号	测试用例（工作证号，姓名，密码，参加工作时间）	覆盖的等价类
1	2799，张三，ABC12345，20060912	1，2，3，4，5，6，7，8，9，10
2	5.6，张三，ABC12345，20060912	11
3	0，张三，ABC12345，20060912	12
4	6000，张三，ABC12345，20060912	13
5	4567，赵 A，ABC12345，20060912	14
6	4567，我是一个名字特别长的人我是一个名字特别长的人，45678909，20060912	15
8	4567，　，ABC12345，20060912	16
9	4567，张三，4567，20060912	17
10	4567，张三，4567777，20060912	18
11	4567，张三，ABCDEFG，20060912	19
12	4567，张三，@#￥%……&*，20060912	20

等价类划分举例-2

- 在某计算机语言版本中规定：
  - 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。

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）

第二步：选取测试用例覆盖所有等价类

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)

⑥ VAR T$：CHAR； (10)

⑦ VAR GOTO：INTEGER； (11)

⑧ VAR 2T：REAL； (13)

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

边界值分析

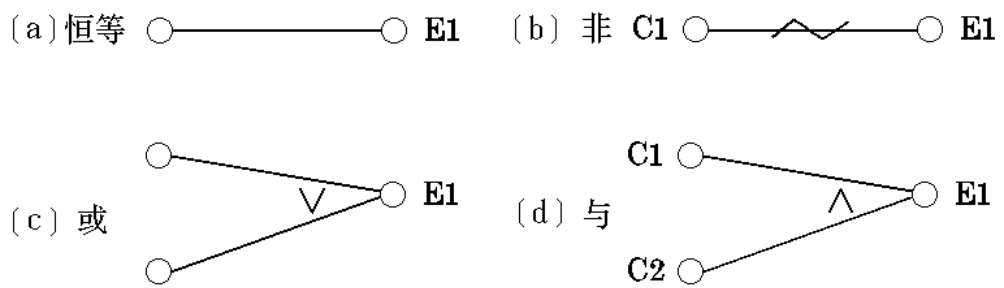
- 边界值分析方法是对等价类划分方法的补充。
- 从长期的测试工作经验得知，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。
- 这里所说的边界是指，相当于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。
- 使用边界值分析方法设计测试用例，首先应确定边界情况。应当选取正好等于，刚刚大于，或刚刚小于边界的值做为测试数据，而不是选取等价类中的典型值或任意值做为测试数据。
- 比如，在做三角形计算时，要输入三角形的三个边长：A、B和C。我们应注意到这三个数值应当满足：
  - A > 0、B > 0、C > 0、A + B > C、A + C > B、B + C > A，才能构成三角形。在程序中如果把不等式中的任何一个“ > ”错写成“≥”，那就不能构成三角形。

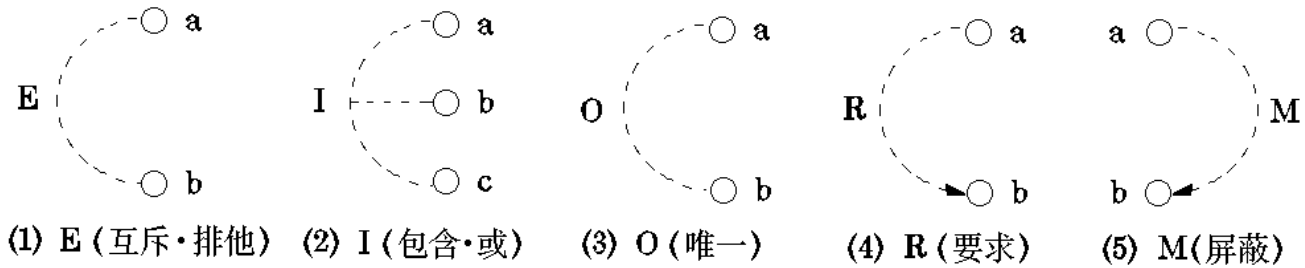
因果图

- 在测试时必须考虑输入条件的各种组合，以及相应动作的形式来设计测试用例，这就需要利用因果图。因果图方法需要使用判定表。
  - 分析（需求）软件规格说明描述中，哪些是原因（输入或状态），哪些是结果（输出或动作），并给每个原因和结果赋予一个唯一的标识符。
  - 分析（需求）软件规格说明中的语义，找出原因与原因之间，原因与结果之间的关系，根据这些关系，画出因果图。
  - 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。
  - 把因果图转换成判定表，并根据因果图中的制约关系对判定表进行化简，去掉不可能存在的组合情况。
  - 简化后的判定表中的每一列就是一种有效的条件组合，对应一个测试用例。

因果图中的符号

- 通常在因果图中用C_i表示原因，用E_i表示结果，各结点表示状态。
  - “0”表示某状态不出现，
  - “1”表示某状态出现。
- 主要的原因和结果之间的关系有：
- 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。





自动饮料售货机实例

- 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：
  - 可投入5角钱或1元钱的硬币，按下〖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。此时在投入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。”
- 分析步骤：
  - 建立原因
  - 建立对应的结果
  - 建立中间结果
  - 绘制因果图并添加约束条件
  - 建立判定表

建立原因结果及中间结点

步骤一：列出原因

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

步骤二：给出结果

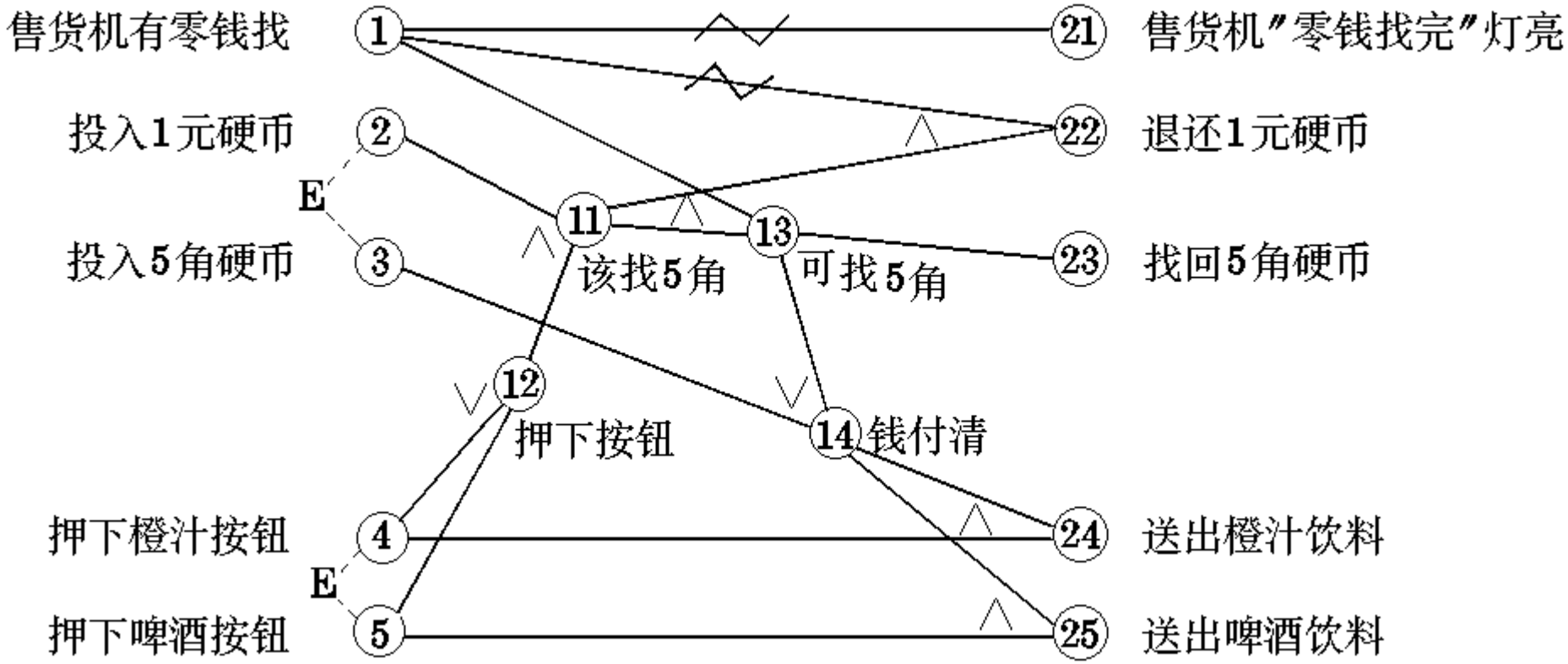
21. 售货机〖零钱找完〗灯亮
22. 退还1元硬币
23. 找回5角硬币
24. 送出橙汁饮料
25. 送出啤酒饮料

步骤三：建立中间结点

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

绘制因果图

- 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。
  - 由于 2 与 3，4 与 5 不能同时发生，分别加上约束条件E。



转换判定表

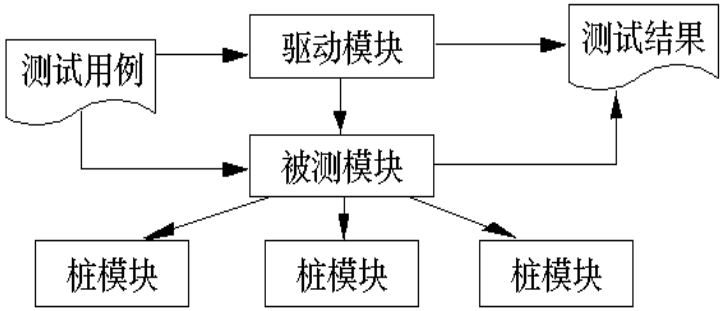
序号		1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1	2
条件	①	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	②	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
	③	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
	④	1	1	0	0	1	1	0	0	1	1	0	0	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	

五个软件测试的基本类型

- 单元测试：编码阶段运用白盒测试方法，对已实现的最小单位代码进行正确性检查；
- 集成测试：编码阶段在单元测试的基础上，运用黑盒测试方法检查被测单元的接口问题，并检查代码集成后各功能的完整性；
- 确认测试：开发后期，针对系统级的软件验证所实现的功能和性能是否与用户的要求一致；
- 系统测试：在开发环境或实际运行环境中，以系统需求分析规格说明书作为验收标准，对软硬件系统进行的一系列集成和确认测试；
- 验收测试：在实际运行环境中，试运行一段时间后所进行的测试活动，确认系统功能和性能符合生产要求。验收通过后交付给用户使用。

单元测试

- 单元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。
- 单元测试采用白盒测试方法设计测试用例。单元测试包括一些内容：
  - 路径测试
  - 接口测试
  - 边界条件测试
  - 局部数据结构测试
  - 错误处理测试



- 被测模块并不是一个独立的程序，还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

集成测试

- 通常在单元测试的基础上，需要将所有模块按照设计要求集成成为系统。需要考虑的问题是：
  - 把各模块集成之后，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？
- 在单元测试的同时可进行集成测试，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。
- 尽早对核心模块进行测试
  - 基于用例中的核心功能；
  - 基于功能结构图中的变换中心。

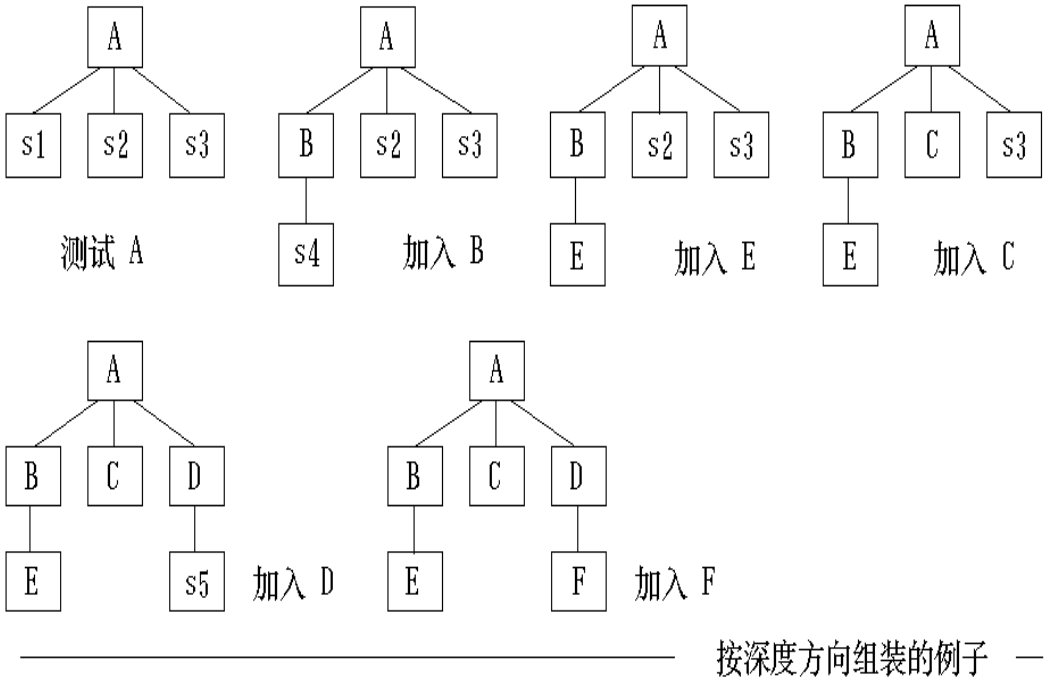
两种集成测试方法

- 一次性集成方式：首先对所有模块进行单元测试，然后再把所有模块集成在一起进行测试，最终得到要求的软件系统。

- 增殖式集成方式：首先对每个模块进行模块测试，然后将这些模块逐步集成成较大的系统；在集成的过程中边连接边测试；最后逐步集成成为要求的软件系统。
  - 自顶向下的增殖方式
  - 自底向上的增殖方式
  - 混合增殖式

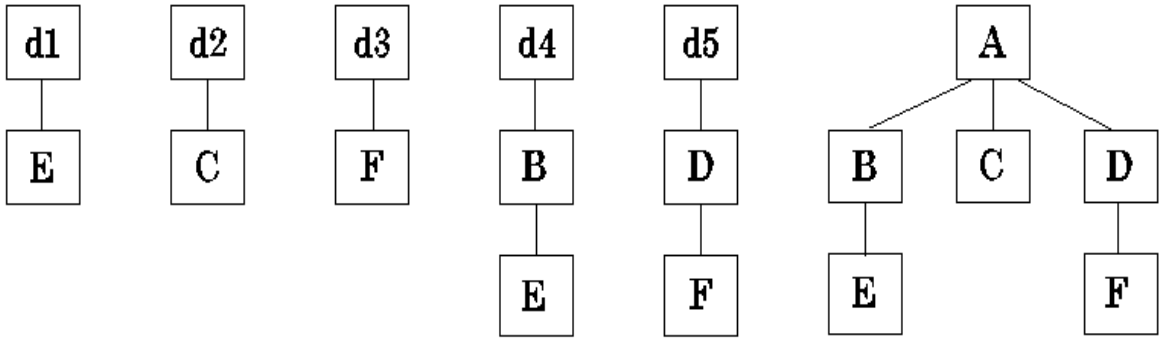
自顶向下的增值方式

- 这种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。
- 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。选用按深度方向集成的方式，可以首先实现和验证一个完整的软件功能。
  - 采用深度优先的策略
  - 进行回归测试
  - 判断是否所有模块已集成



自底向上的增值方式

- 这种集成的方式是从程序模块结构的最底层的模块开始集成和测试。
- 因为模块是自底向上进行集成，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）已经集成并测试完成，所以不再需要桩模块。



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;
  - 再自底向上集成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集成和测试;
  - 然后对含写操作的子系统做自顶向下的集成与测试。
- 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

- 确认测试又称有效性测试，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，以及软件配置是否完整和正确。
- 确认测试一般是由开发组织中专门的测试人员在开发环境中完成的系统级测试活动，采用黑盒方法验证被测软件是否满足需求规格说明书列出的需求。
- 对于需求规格说明中定义的非功能性需求，在不受运行环境影响的情况下，也可以在确认测试环节进行相应的测试，如系统可移植性、兼容性、出错自动恢复、可维护性等。

系统测试

- 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。
- 系统测试包括一系列测试内容：
  - 功能/性能测试；
  - 压力/强度测试；
  - 大数据量测试；
  - 安全性/易用性测试；
  - …… 文档测试。

验收测试

- 系统正式投入试运行一段时间后，需要对系统进行最终的评估，确认是否可以将系统移交给最终用户，这个测试称为验收测试。
- 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。
- 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。

软件测试专业术语

- 软件错误（Error）：即常说的bug，是人们在进行软件的需求分析、设计和开发中人为产生的错误，可存在于文档或代码中，是一种面向开发的概念;
- 软件缺陷（Defect）：指存在于软件（文档、数据、程序）中的那些不希望或不可接受的偏差。
- 软件故障（Fault）：指软件运行过程中产生的一种不可接受的系统内部状态，是一种系统动态行为。
- 软件失效（Failure）：指软件运行过程中产生的一种不可接受的系统外部状态。