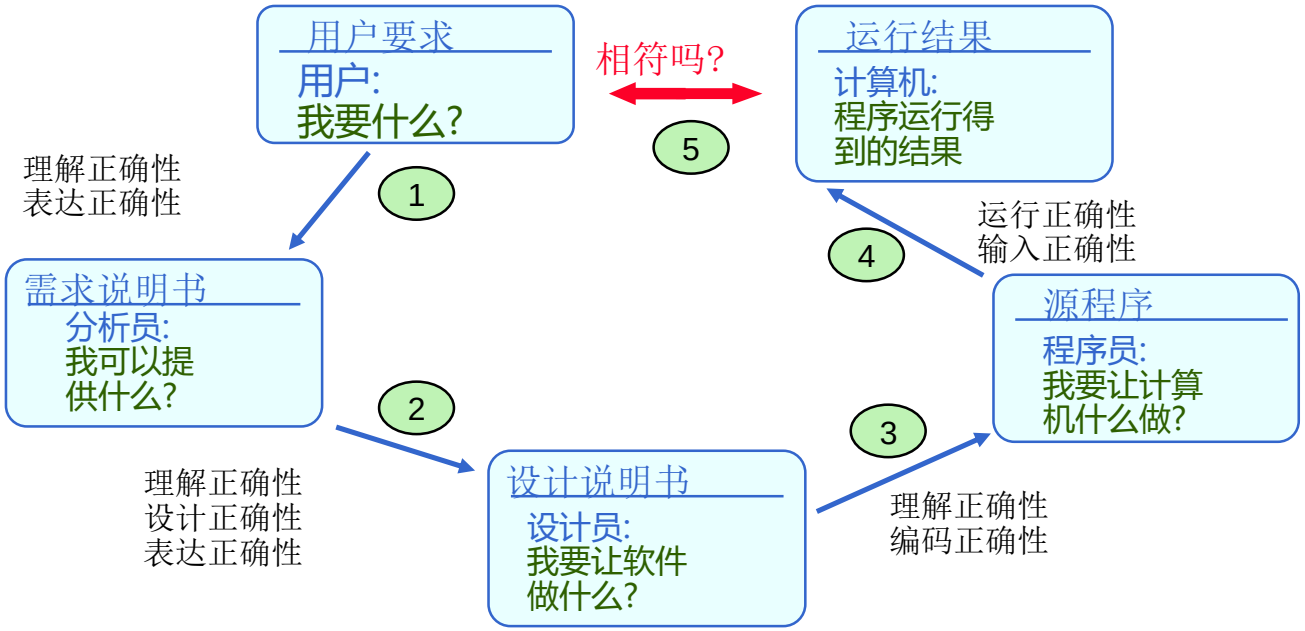
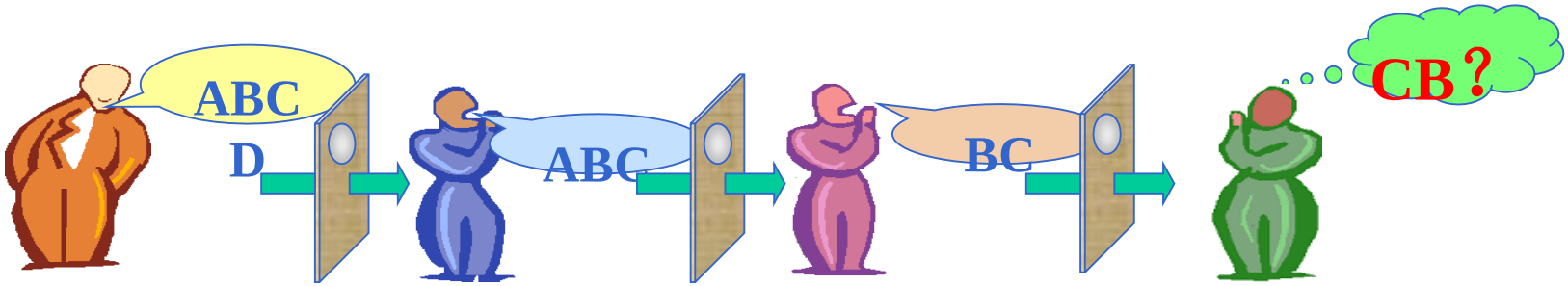
软件测试

软件测试

软件测试的基本概念
软件测试用例设计
软件测试策略
调试
软件可靠性

软件生存期各阶段间需保持的正确性





软件测试的基本概念

- 软件测试的目的和原则
- 软件测试的对象
- 测试信息流
- 白盒测试和黑盒测试
- 软件测试的步骤
- 测试与软件开发各阶段的关系

Myers软件测试目的

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

定义：测试是为了发现程序中的错误而执行程序的过程。

软件测试的原则

1. 应当把“尽早地和不断地进行软件测试”作为软件开发者的座右铭。
2. 测试用例应由测试输入数据和对应的预期输出结果这两部分组成。
3. 程序员应避免测试自己的程序，程序设计组织应避免测试自己的程序。
4. 在设计测试用例时，应包括合理的输入条件和不合理的输入条件。
5. 充分注意测试中的群集现象。
   经验表明，测试后程序中残存的错误数目与该程序中已发现的错误数目成正比。往往80%的错误仅与20%的模块有关。
6. 严格执行测试计划，排除测试的随意性。
7. 应当对每一个测试结果做全面检查。
8. 妥善保存测试计划，测试用例，出错统计和最终分析报告，为维护提供方便。因为在改正错误后或维护后要进行回归测试（regression testing，即全部或部分地重复已做过的测试）。
9. 检查程序是否做了应做的事仅是成功的一半，另一半是检查程序是否做了不该做的事。
10. 在规划测试时不要设想程序中不会查出错误。

软件测试的对象

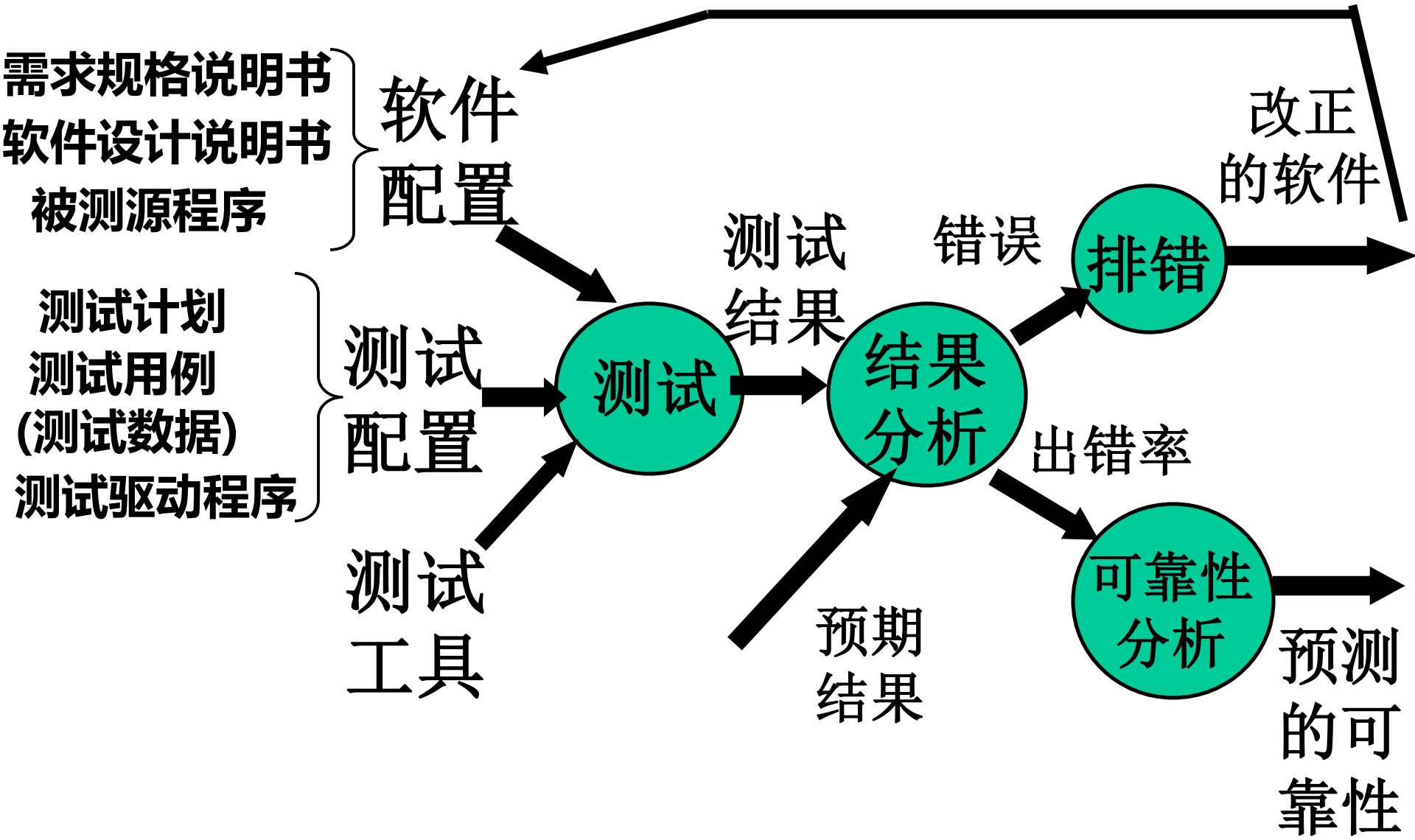
- 软件测试并不等于程序测试。软件测试应贯穿于软件定义与开发的整个期间。
- 需求分析、概要设计、详细设计以及程序编码等各阶段所得到的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

- 验证(Verification)
- 确认(Validation)

Verification:“Are we building the product right?”

Validation :“Are we building the right product?”

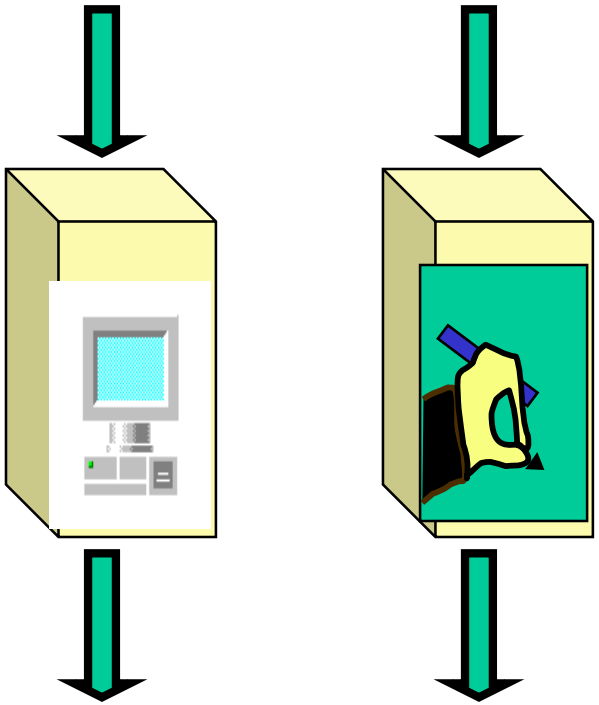
软件测试信息流


需求规格说明书
软件设计说明书
被测源程序
软件配置
测试计划
测试用例
(测试数据)
测试驱动程序
测试配置
测试工具
测试
测试结果
结果分析
错误
排错
改正的软件
出错率
可靠性分析
预测的可靠性
预期结果

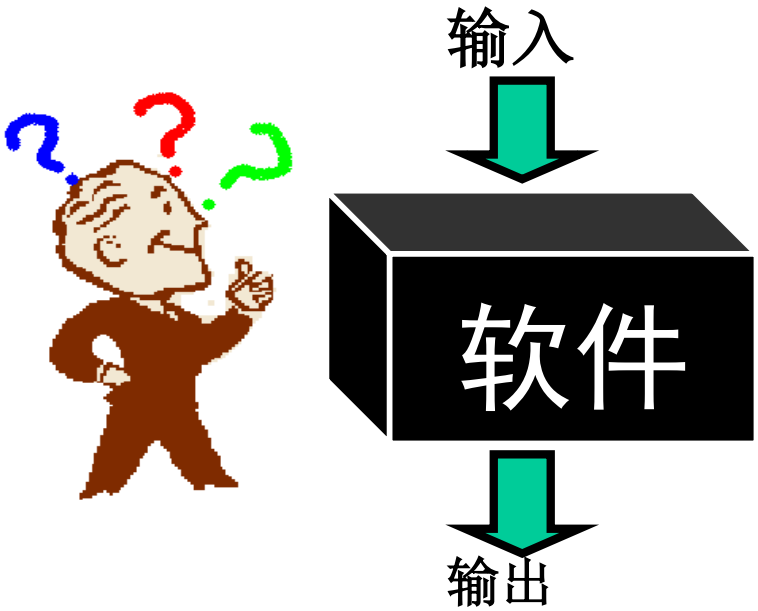
白盒测试与黑盒测试

- 两种常用的测试方法
  - 白盒测试
  - 黑盒测试

白盒测试



黑盒测试



- 白盒测试又称为结构测试
- 黑盒测试又叫做功能测试。

黑盒测试与白盒测试优缺点比较

	黑盒测试	白盒测试
优点	①适用于各阶段测试 ②从产品功能角度测试 ③容易入手生成测试数据	①可构成测试数据使特定程序部分得到测试 ②有一定的充分性度量手段 ③可或较多工具支持
缺点	①某些代码得不到测试 ②如果规格说明有误，则无法发现 ③不易进行充分性测试	①不易生成测试数据(通常) ②无法对未实现规格说明的部分进行测试 ③工作量大，通常只用于单元测试，有应用局限
性质	是一种确认技术，回答“我们在构造一个正确的系统吗？”	是一种验证技术，回答“我们在正确 地构造一个系统吗？”

软件测试步骤

- 单元测试（unit testing）

  也称模块测试，一般在编码阶段进行

  测试对象：模块（功能和内部逻辑）

  测试方法：白盒

  发现的错误：编码、详细设计

- **集成测试（integration testing）**

  根据程序结构图将模块集成为程序进行测试，主要测试模块间的接口和通信

  测试方法：黑盒

  发现的错误：概要设计

- **确认测试（validation testing）**

  根据需求规格说明，检查软件的功能及其它特征是否与用户的需求一致

  测试方法：黑盒

  发现的错误：需求分析

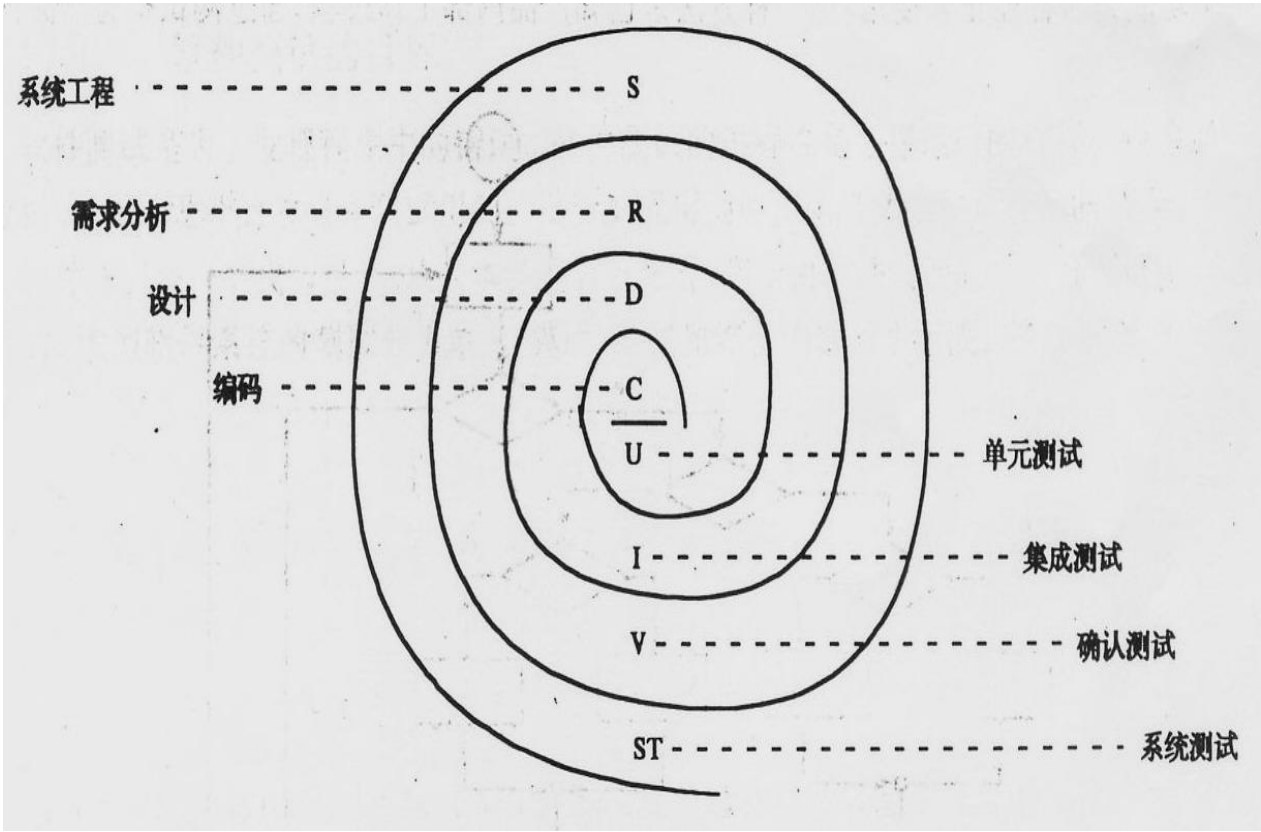
- **系统测试（system testing）**

  将软件、硬件、数据库等集成为计算机系统，检查系统的功能、性能等是否符合计算机系统的要求

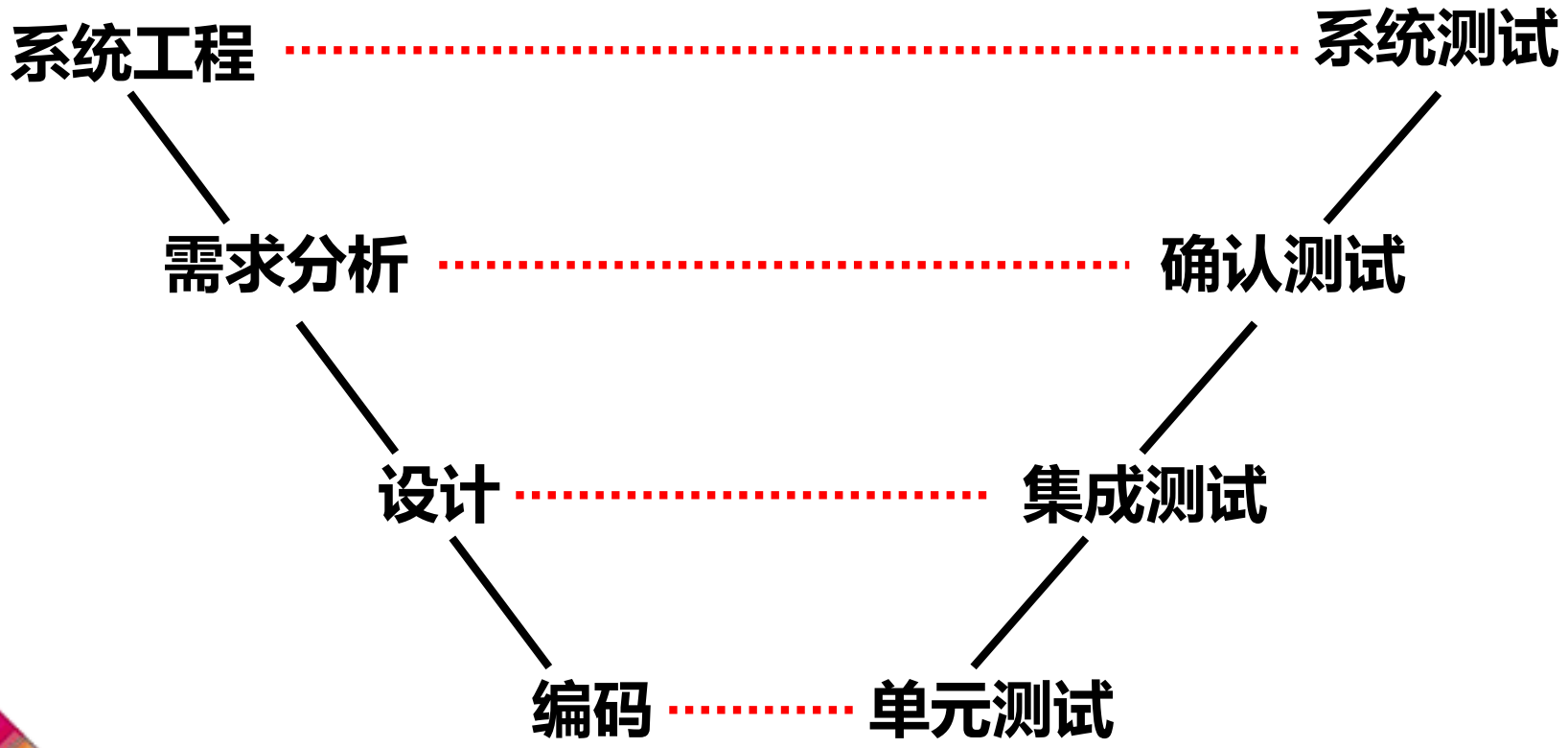
  测试方法：黑盒

  发现的错误：系统工程

测试与软件开发各阶段的关系



软件开发V模型



白盒测试的测试用例设计

主要方法有：

- 逻辑覆盖
- 基本路径覆盖
- 控制结构测试
  - 条件测试
  - 数据流测试
  - 循环测试

逻辑覆盖

逻辑覆盖主要考察使用测试数据运行被测程序时对程序逻辑的覆盖程度。通常希望选择最少的测试用例来满足所需的覆盖标准。主要的覆盖标准有：

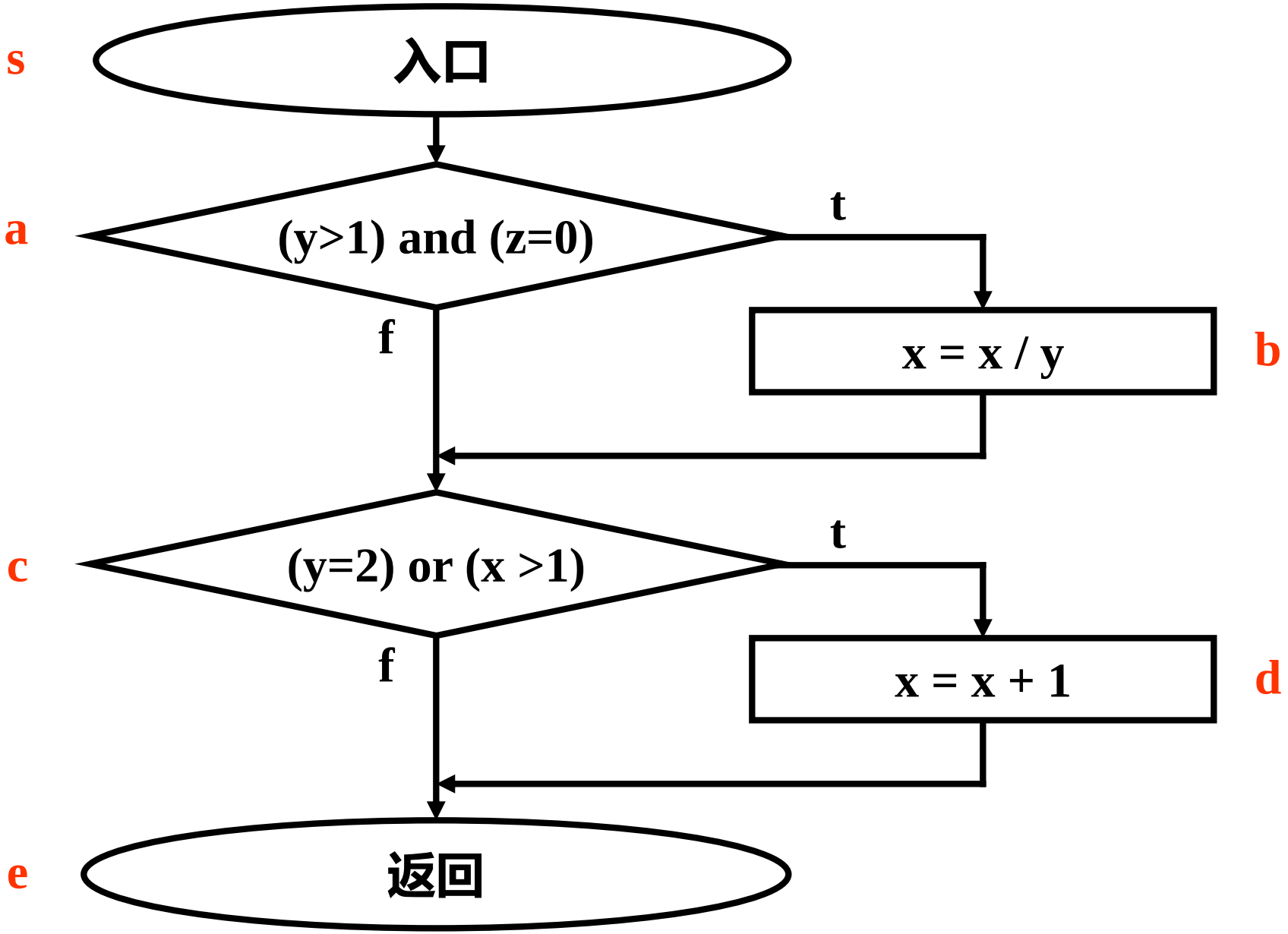
- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定 - 条件覆盖
- 条件组合覆盖
- 路径覆盖

例：对下列子程序进行测试

```
procedure example(y,z:real;var x:real);
begin
    if (y>1) and (z=0) then x:=x/y;
    if (y=2) or (x>1) then x:=x+1;
end;
```

该子程序接受x、y、z的值，并将计算结果x的值返回给调用程序。

与该子程序对应的流程图如下：


s
入口
a
(y>1) and (z=0)
t
f
x = x / y
b
c
(y=2) or (x >1)
t
f
x = x + 1
d
e
返回

该子程序有两个判定：

a: (y>1) and (z=0)

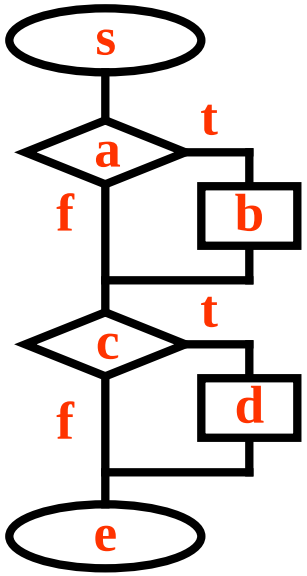
c: (y=2) or (x>1)

判定a中有两个判定条件：

y>1、 z=0

判定c中有两个判定条件：

y=2 、 “x>1”



a: (y>1)
and
(z=0)
c: (y=2)
or
(x>1)

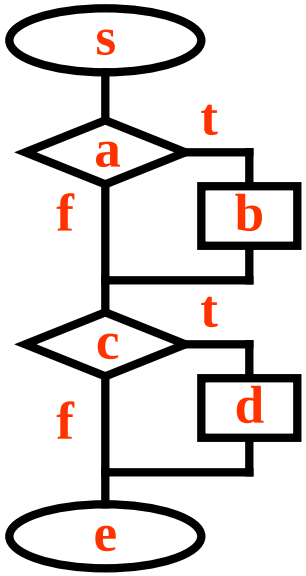
该子程序有四条可执行路径：

L1 (sabcde， a为“t”且c为“t”)

L2 (sace， a为“f”且c为“f”)

L3 (sacde， a为“f”且c为“t”)

L4 (sabce， a为“t”且c为“f”)



a: (y>1)
and
(z=0)
c: (y=2)
or
(x>1)

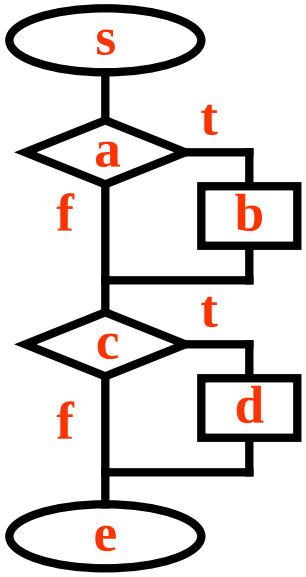
语句覆盖

语句覆盖是指选择足够的测试用例，使得运行这些测试用例时，被测程序的每个可执行语句都至少执行一次

欲使每个语句都执行一次，只需执行路径L1（sabcde）即可。

测试用例如下：

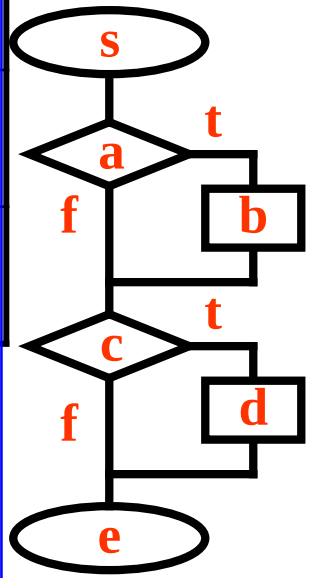
测试数据	预期结果
x=4,y=2,z=0	x=3



a: (y>1) and (z=0)

c: (y=2) or (x>1)

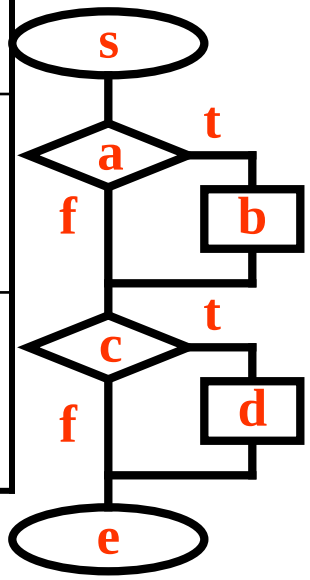
测试数据	预期结果	路径	a	c
x=1,y=2,z=1	x=2	sacde	f	t
x=3,y=3,z=0	x=1	sabce	t	f



判定覆盖将每个判定的所有可能结果都至少执行一次，所以，程序中的所有语句也必定都至少执行一次。因此，满足判定覆盖标准的测试用例也一定满足语句覆盖标准

a: (y>1) and (z=0)
c: (y=2) or (x>1)

测试数据	预期结果	路径	覆盖的条件
x=1,y=2,z=0	x=1.5	sabcde	y>1， z=0， y=2，x ≤ 1
x=2,y=1,z=1	x=3	sacde	y ≤ 1，z ≠ 0， y ≠ 2 ， x>1



- 条件覆盖通常比判定覆盖强，但有时虽然每个条件的所有可能结果都出现过，但判定表达式的某些可能结果并未出现。上面的二个测试用例满足了条件覆盖标准，但判定c为“假”的结果并未出现。

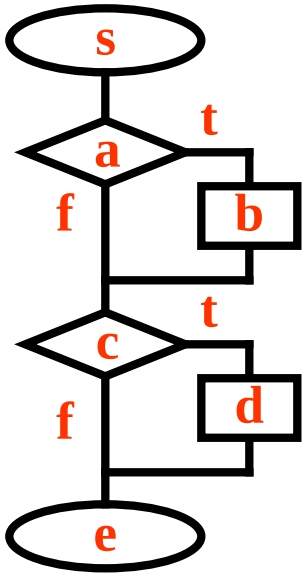
a: (y>1) and (z=0)
c: (y=2) or (x>1)

判定/条件覆盖

判定/条件覆盖是指选择足够的测试用例，使得运行这些测试用例时，被测程序的每个判定的所有可能结果都至少执行一次，并且，每个判定中的每个条件的所有可能结果都至少出现一次

显然，满足判定/条件覆盖标准的测试用例一定也满足判定覆盖、条件覆盖、语句覆盖标准。

a: (y>1)
and
(z=0)
c: (y=2)
or
(x>1)



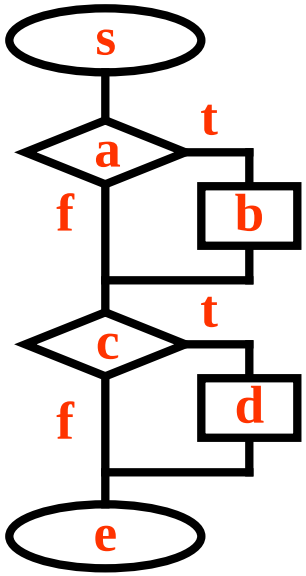
测试数据	预期结果	路径	a	c	覆盖的条件
x=4,y=2,z=0	x=3	sabcde	t	t	y>1， z=0， y=2， x > y
x=1,y=1,z=1	x=1	sace	f	f	y ≤ 1， z ≠ 0， y ≠ 2 ， x ≤1

条件组合覆盖

条件组合覆盖是指选择足够的测试用例，使得运行这些测试用例时，被测程序的*每个判定中*条件结果的所有可能组合都至少出现一次

显然，满足条件组合覆盖标准的测试用例一定也满足判定覆盖、条件覆盖、判定/条件覆盖、语句覆盖标准。

测试数据	预期结果	路径	a	c
x=4,y=2,z=0	x=3	sabcde	t	t
x=3,y=3,z=0	x=1	sabce	t	f
x=2,y=1,z=0	x=3	sacde	f	t
x=1,y=1,z=1	x=1	sace	f	f



路径覆盖实际上考虑了程序中各种判定结果的所有可能组合，但它未必能覆盖判定中条件结果的各种可能情况。因此，它是一种比较强的覆盖标准，但不能替代条件覆盖和条件组合覆盖标准。

a: (y>1)
and
(z=0)
c: (y=2)
or
(x>1)

黑盒测试的测试用例设计

- 黑盒测试主要测试程序是否满足功能、性能等要求。主要诊断以下错误：
  - 不正确或遗漏的功能
  - 接口错误
  - 数据结构或外部数据库访问错误
  - 性能错误
  - 初始化和终止条件错误

黑盒测试的主要方法

- 等价类划分
- 边界值分析
- 错误推测法
- 因果图

等价类划分

- 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

等价类划分设计测试用例的步骤

- **确定等价类**

根据程序的功能说明，对每一个输入条件（通常是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。

可使用如下表格

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) 为每个有效等价类和无效等价类编号；

(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) 为每个无效等价类设计一个新的测试用例。

例：某报表处理系统要求用户输入处理报表的日期，日期限制在2001年1月至2005年12月，即系统只能对该段期间内的报表进行处理，如日期不在此范围内，则显示输入错误信息。

系统日期规定由年、月的6位数字字符组成，前四位代表年，后两位代表月。

如何用等价类划分法设计测试用例，来测试程序的日期检查功能？

第一步：等价类划分

“报表日期”输入条件的等价类表

输入等价类	有效等价类	无效等价类
报表日期的类型及长度	6位数字字符(1)	有非数字字符 (4) 少于6个数字字符 (5) 多于6个数字字符 (6)
年份范围	在2001～2005之间 (2)	小于2001 (7) 大于2005 (8)
月份范围	在1～12之间(3)	小于1 (9) 大于12 (10)

第二步：为有效等价类设计测试用例

对表中编号为1,2,3的3个有效等价类用一个测试用例覆盖：

测试数据	期望结果	覆盖范围
200105	输入有效	等价类(1)(2)(3)

第三步：为每一个无效等价类设至少计一个测试用例

测试数据	期望结果	覆盖范围
001MAY	输入无效	等价类(4)
20015	输入无效	等价类(5)
2001005	输入无效	等价类(6)
200005	输入无效	等价类(7)
200805	输入无效	等价类(8)
200100	输入无效	等价类(9)
200113	输入无效	等价类(10)

不能出现相同的测试用例

本例的10个等价类至少需要8个测试用例

例：考察一个把数字串转变成整数的函数。用二进制补码表示整数，机器字长16位，即整数范围最小为- 32768，最大为32767。函数及参数的PASCAL说明如下：

```
function StrToInt (dstr : shortstr) : integer;
type shortstr = array [1..6] of char;
```

要求被处理的数字串是右对齐的，即在少于6个字符的串左边补空格。负号在最高位数字左边一位。

试用等价划分法设计测试方案。

解：首先根据规格说明划分等价类。考虑到PASCAL编译器的固有检错功能，测试时不需要使用长度不等于6的数组，也不需要用非字符数组类型的参数。

有效输入类：

①1~6个数字字符组成的数字串（最高位非0）；

②最高位为0的数字串； ③最高位左邻负号的数字串；

无效输入类：

④空字符串（6位空格）； ⑤左边补位的既非0亦非空格；

⑥最高位右边含有空格；

⑦最高位右边含有其它非数字字符；

⑧负号与最高位间有空格；

有效输出类：

⑨ 在合法范围内的负整数；

⑩ 0；

⑪ 在合法范围内的正整数；

无效输出类：

⑫ 小于 - 32768的负整数；

⑬ 大于 32767正整数。

下面根据等价划分，设计出一套测试方案：

①1~6个数字字符组成的数字串，最高位非0；输出 为合法正整数。

输入：

					1

预期输出：1

②最高位为0的数字串，输出为合法正整数。

输入：

0	0	0	0	0	1

预期输出：1

③负号与最高位数字相临；输出合法负整数。

输入：

-	0	0	0	0	1

预期输出：-1

④最高位为0；输出0。

输入：

0	0	0	0	0	0

预期输出：0

⑤太小的负整数。

输入：

-	3	2	7	6	9

预期输出：“错误，无效输入”

⑥太大的正整数。

输入：

	3	2	7	6	8

预期输出：“错误，无效输入”

⑦空字符串。

输入：| | | | | | | 预期输出：“错误：没有数字”

⑧左边补位的非0也非空格。

输入：| a | a | a | a | a | 1 | 预期输出：“错误：非法填充”

⑨最高位右边也含空格。

输入：| | | 1 | | | 2 | 预期输出：“错误：无效输入”

⑩最高位右边含其它非数字字符。

输入：| 0 | 0 | 1 | x | x | 2 | 预期输出：“错误：无效输入”

⑪ 负号与最高位间有空格。

输入：| | | - | | 1 | 2 | 预期输出：“错误：负号位置非法”

边界值分析

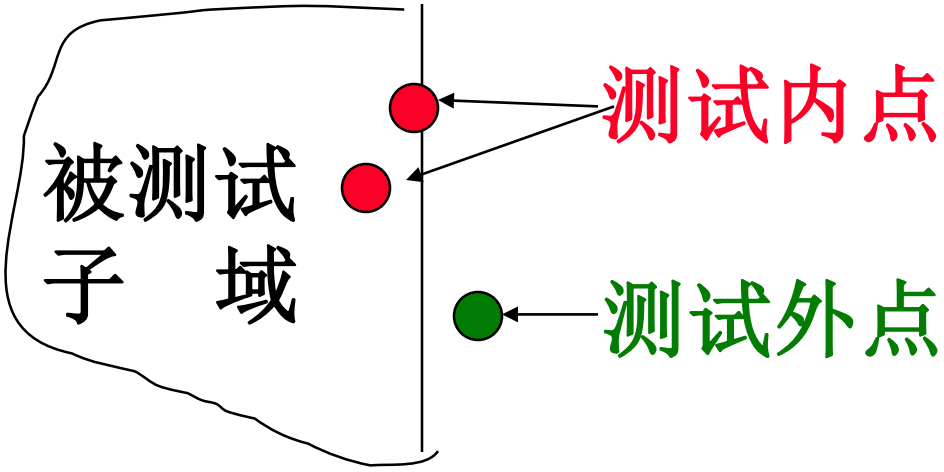
- 边界值分析也是一种黑盒测试方法，是对等价类划分方法的补充。
- 人们从长期的测试工作经验得知，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。因此针对各种边界情况设计测试用例，其揭露程序中错误的可能性就更大。

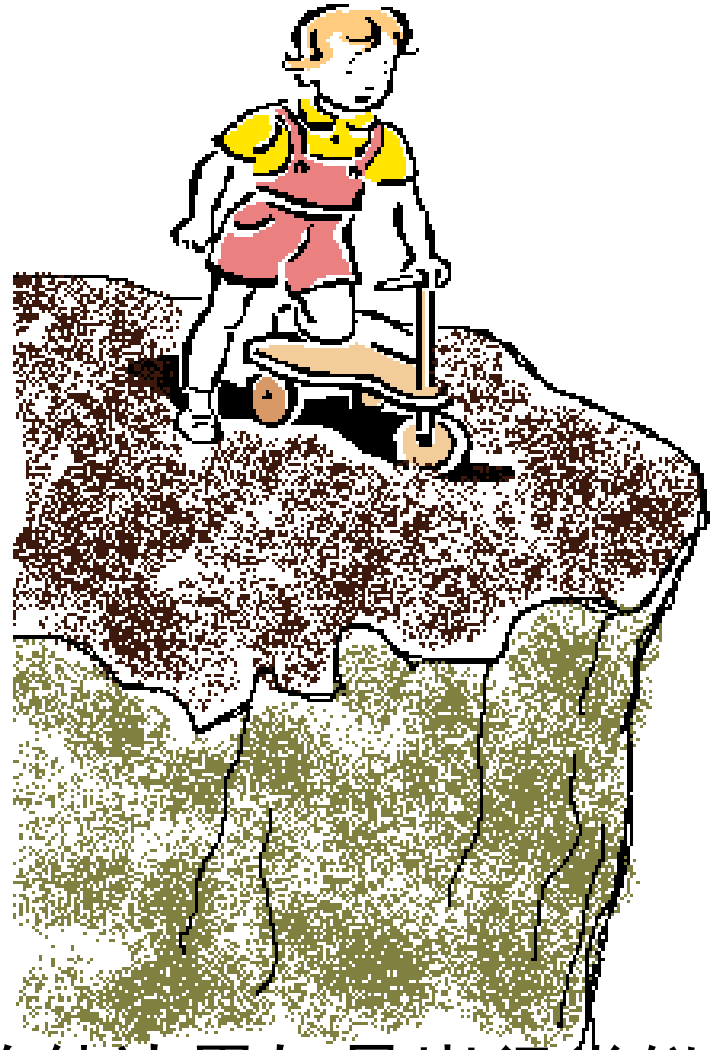
边界值分析法

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





软件边界与悬崖很类似

测试边界线

- 测试临近边界的合法数据,以及刚超过边界的非法数据.

- 越界测试通常简单地加1或很小的数
(对于最大值)和减1或很小的数(对于最小值).

“报表日期”边界值分析法测试用例

输入条件	测试用例说明	测试数据	期望结果	选取理由
报表日期的类型及长度	1个数字字符	5	显示出错	仅有1个合法字符
	5个数字字符	20015	显示出错	比有效长度少1
	7个数字字符	2001005	显示出错	比有效长度多1
	有1个非数字字符	2001.5	显示出错	只有1个非法字符
	全部是非数字字符	MAY---	显示出错	6个非法字符
	6个数字字符	200105	输入有效	类型及长度均有效
日期范围	在有效范围边界上选取数据	200101	输入有效	最小日期
		200512	输入有效	最大日期
		200100	显示出错	刚好小于最小日期
		200513	显示出错	刚好大于最大日期
月份范围	月份为1月	200101	输入有效	最小月份
	月份为12月	200112	输入有效	最大月份
	月份<1	200100	显示出错	刚好小于最小月份
	月份>12	200113	显示出错	刚好大于最大月份

错误推测法

- 错误推测方法是一种凭直觉和经验推测程序中可能存在的各种错误，从而针对这些可能存在的错误设计测试用例的方法。
- 这种方法没有机械的执行步骤，主要依靠直觉和经验。
- 错误推测法的基本想法是：列举出程序中所有可能有的错误和容易发生错误的特殊情况，根据它们设计测试用例。

- 例如，测试一个排序子程序，可考虑如下情况：
  - 输入表为空；
  - 输入表只有一个元素；
  - 输入表的所有元素都相同；
  - 输入表已排序。
- 又如，测试二分法检索子程序，可考虑如下情况：
  - 表中只有一个元素；
  - 表长为2^n；
  - 表长为2^n-1；
  - 表长为2^n+1

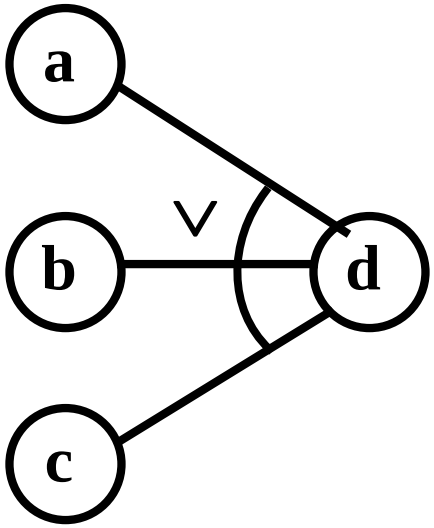
因果图

- 边值分析和等价类划分方法都没有考虑输入条件的各种组合，但输入条件组合的数目相当大，因此应该用某种方法来选择输入条件的子集，再考虑它们的组合。因果图就是一种帮助人们系统地选择一组高效测试用例的方法。

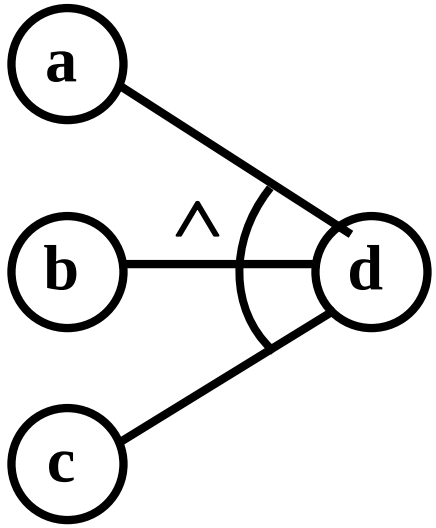
- **因果图方法的特点是：**
  - **考虑输入条件的组合关系；**
  - **考虑输出条件对输入条件的依赖关系，即因果关系；**
  - **测试用例发现错误的效率高；**
  - **能检查出功能说明中的某些不一致或遗漏。**

恒等

非



或



与

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

(1) 分析这一段说明，列出原因和结果

原因: 1. 售货机有零钱找
2. 投入1元硬币
3. 投入5角硬币
4. 押下橙汁按钮
5. 押下啤酒按钮

建立中间结点，表示处理中间状态

11. 投入1元硬币且押下饮料按钮
12. 押下〖橙汁〗或〖啤酒〗的按钮
13. 应当找5角零钱并且售货机有零钱找
14. 钱已付清

结果： 21. 售货机〖零钱找完〗灯亮

22. 退还1元硬币

23. 退还5角硬币

24. 送出橙汁饮料

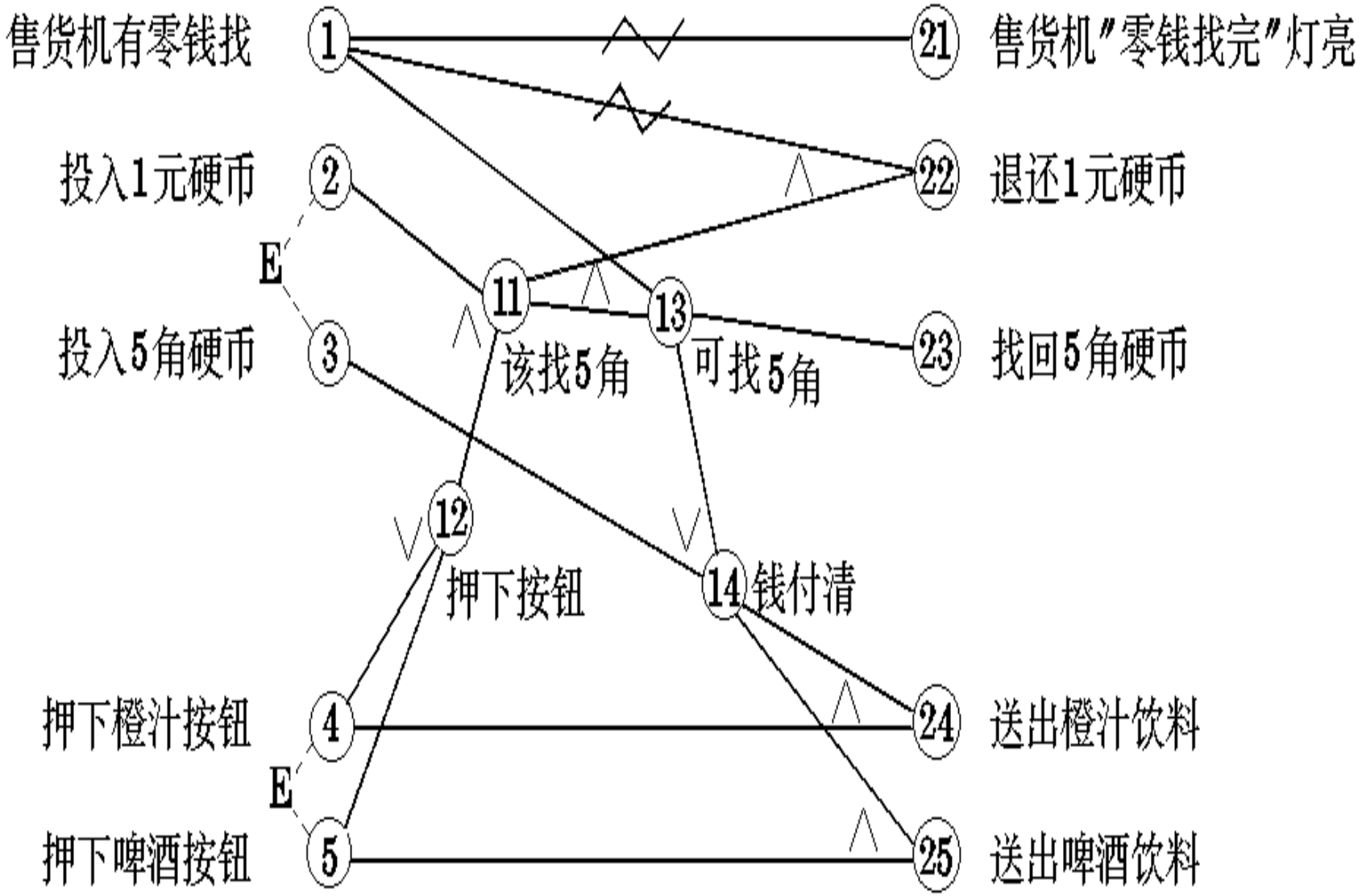
25. 送出啤酒饮料

(2) 画出因果图。所有原因结点列在左边，所有结果结点列在右边。

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

(4) 根据因果图画出判定表

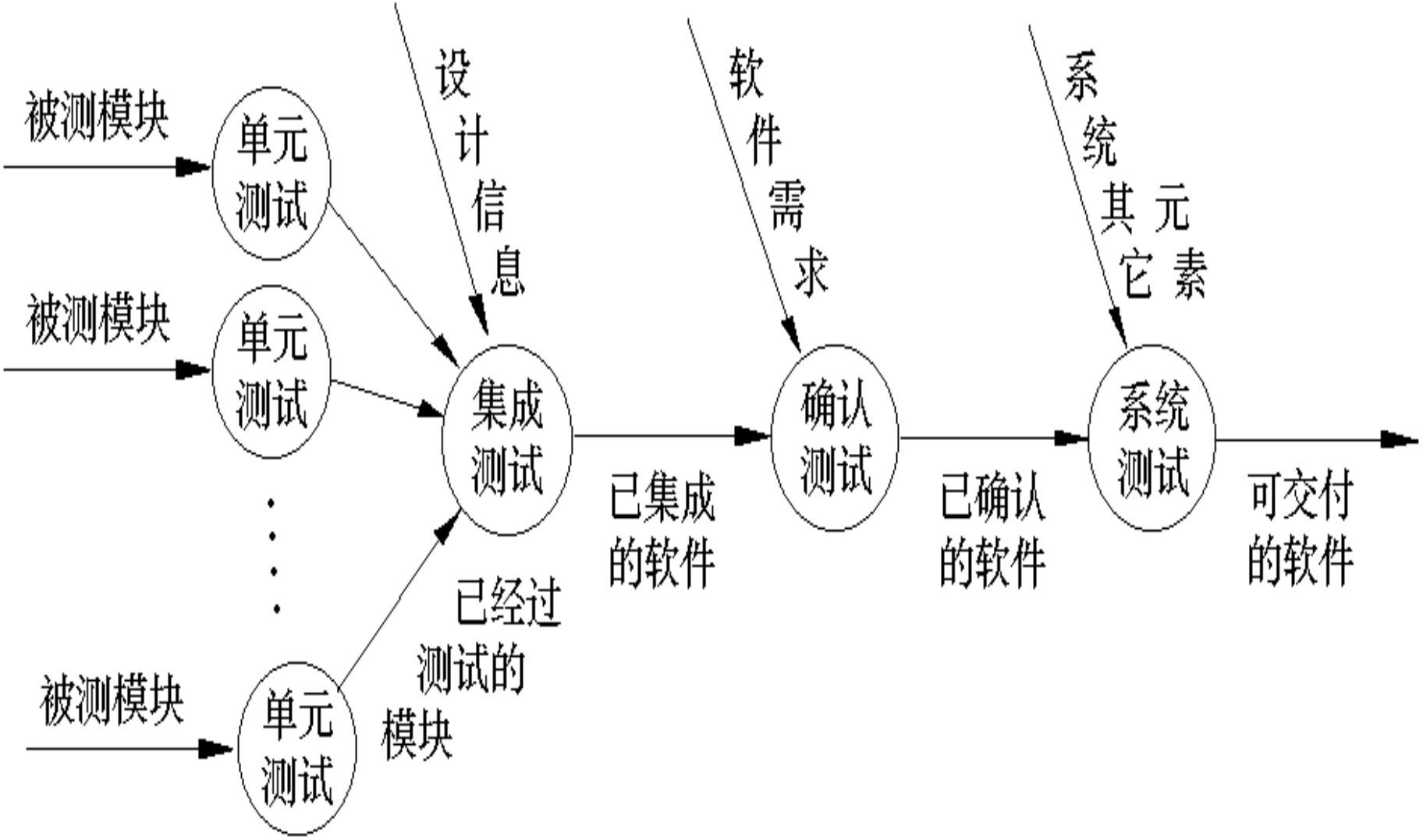
(5) 根据判定表设计测试用例


售货机有零钱找
1
21
售货机"零钱找完"灯亮
投入1元硬币
2
22
退还1元硬币
E
11
13
投入5角硬币
3
该找5角
可找5角
23
找回5角硬币
12
押下按钮
14
钱付清
押下橙汁按钮
4
24
送出橙汁饮料
E
押下啤酒按钮
5
25
送出啤酒饮料

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	0	0	0	0	0	0	0	0
	③	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	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	

软件测试的策略

- 测试过程通常按4个步骤进行，即单元测试、集成测试、确认测试和系统测试。


被测模块
单元测试
被测模块
单元测试
被测模块
单元测试
设计信息
已经过测试的模块
集成测试
已集成的软件
软件需求
确认测试
已确认的软件
系统其它元素
系统测试
可交付的软件

单元测试 (Unit Testing)

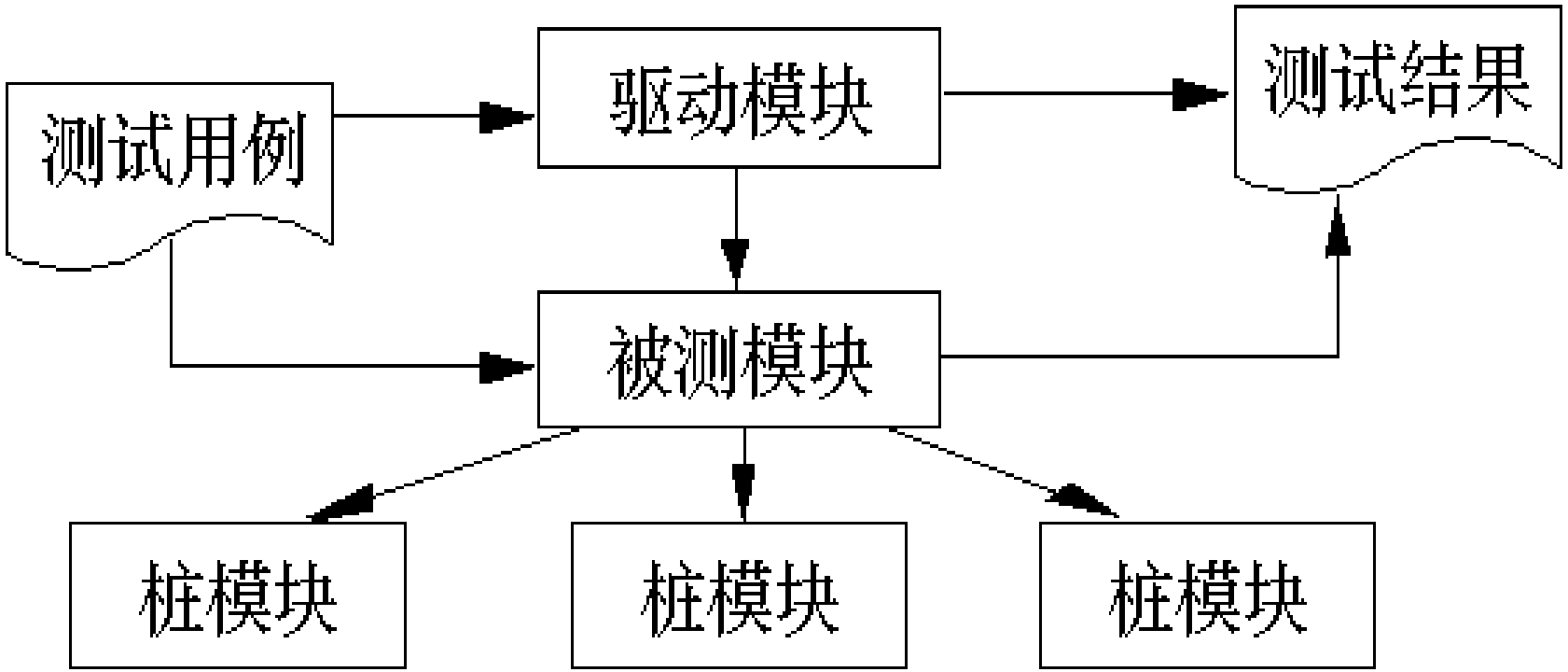
- 单元测试又称模块测试，是针对软件设计的最小单位 — 程序模块，进行检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。通常采用白盒测试。
- 单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试。

1. 单元测试的内容

- 模块接口：保证测试时进出程序单元的信息是正确流动的；
- 局部数据结构：保证临时存储的数据在算法执行的整个过程中都能维持其完整性；
- 边界条件：保证模块在极限或严格的情况下仍然能正确执行；
- 所有独立路径：保证模块中的所有语句都至少执行一次；
- 所有错误处理路径。

2. 单元测试的步骤

- 单元测试通常与编码工作结合起来进行。



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：
  - 数据可能在穿越模块接口时丢失；
  - 一个模块可能对另一个模块产生无法预料的副作用；
  - 当子功能被组合起来时，可能不能达到预期的父功能；
  - 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。

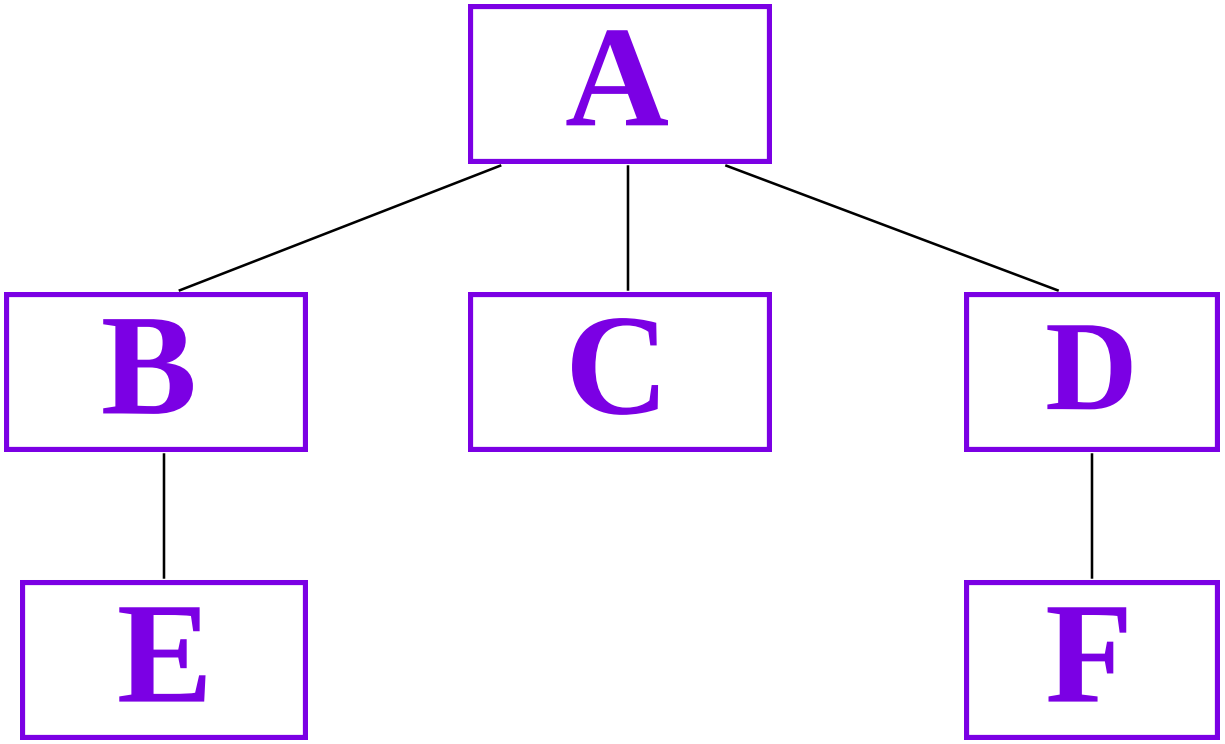
集成测试方法

通常采用黑盒测试技术

实施策略:

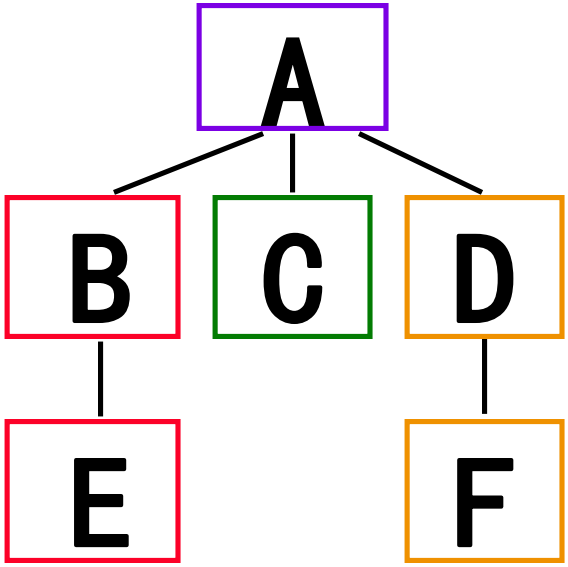
- 非渐增式测试
- 渐增式测试
  - 自顶向下结合
    - 深度优先
    - 广度优先
  - 自底向上结合

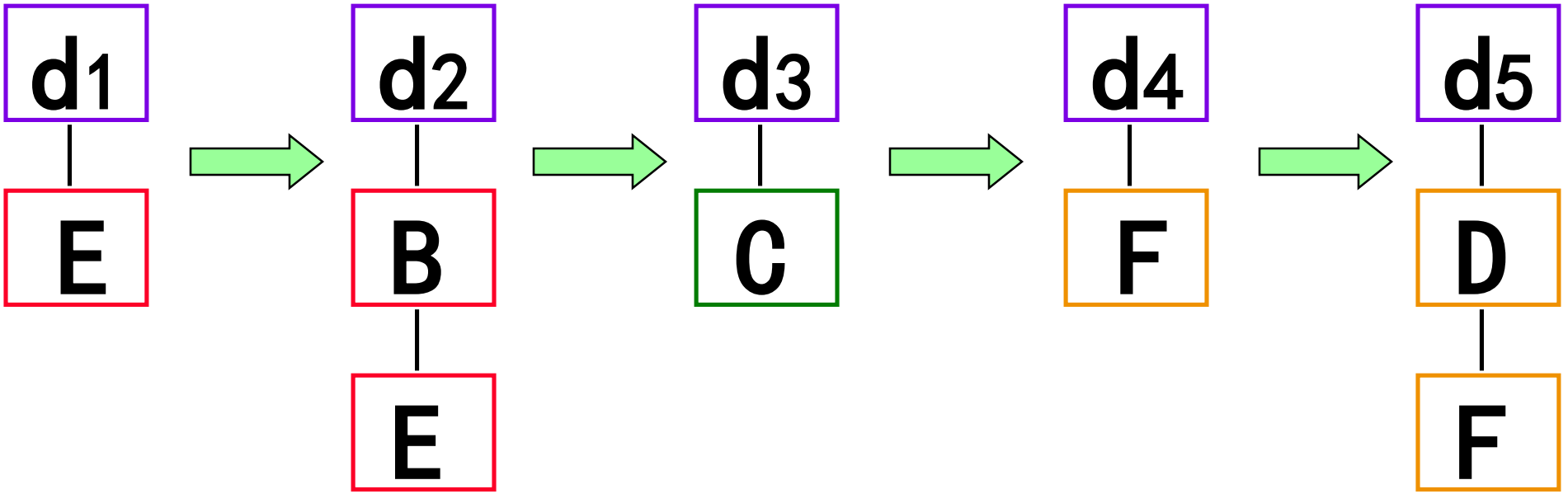
自顶向下结合方式举例:



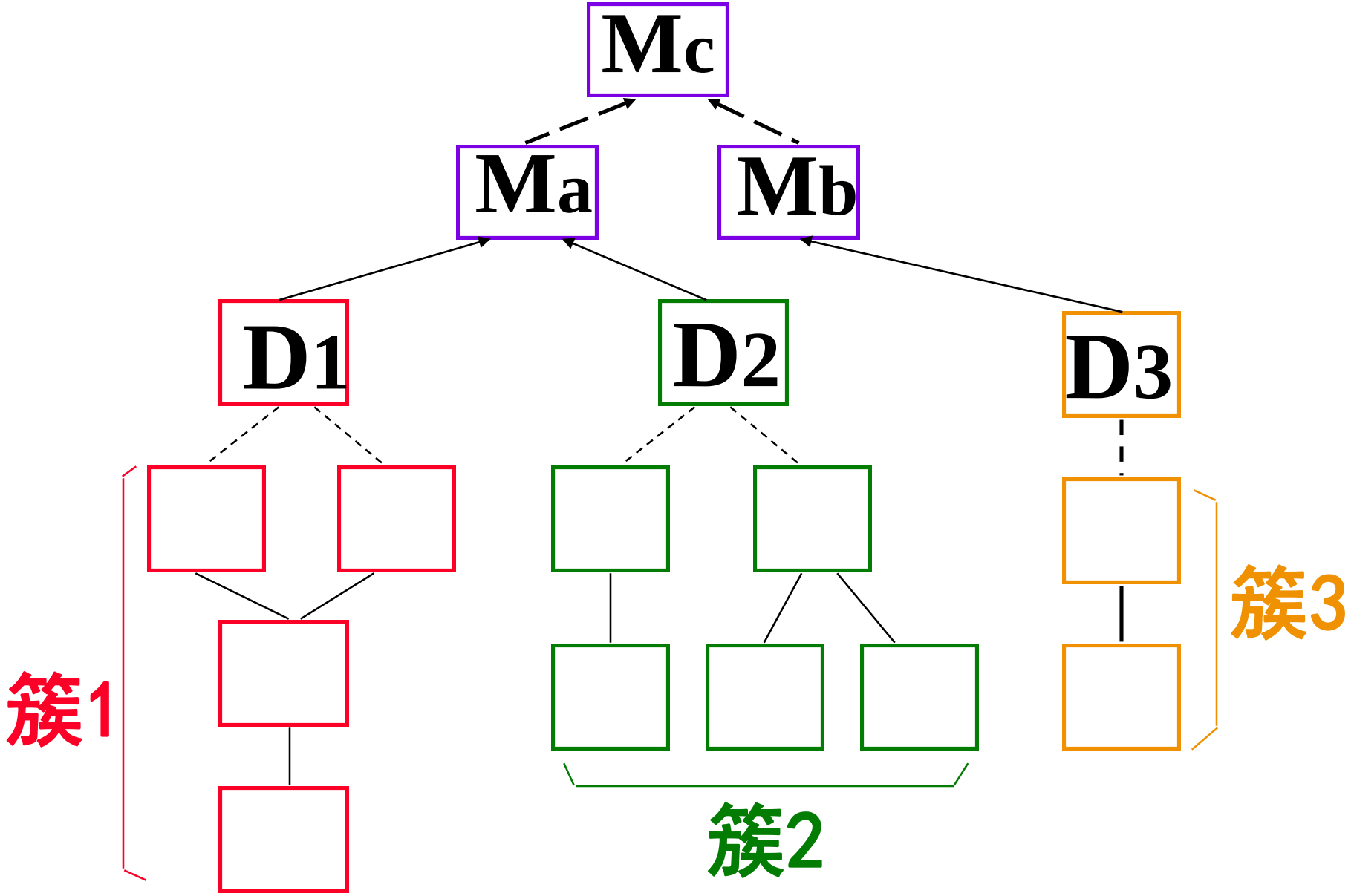
模块测试结合顺序
- 深度优先:A、B、E、C、D、F
- 广度优先:A、B、C、D、E、F

自底向上结合方式举例:





自底向上结合方式举例:



	自顶向下	自底向上
优点	可在测试早期实现并验证系统主要功能	设计测试用例容易
缺点	不需驱动模块 需桩模块	不需桩模块 只有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;

回归测试（Regression Testing）

在集成测试时，每当增加一个新模块时，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操作可能也会出现，还可能激活新的控制逻辑。这些改变可能使原本正常工作的功能产生错误。

在软件维护时，当软件修改后，也可能使原本正常工作的功能产生错误。

回归测试是对某些已经进行过的测试的子集的重新执行，以保证软件的改变不会传播不可预料的副作用或附加的错误。

回归测试集（已经过测试的子集）包括三种不同类型的测试用例：

- 能测试软件所有功能的代表性测试用例
- 专门针对可能会被修改影响的软件功能的附加测试
- 注重于修改过的软件模块的测试

确认测试（Validation Testing）

确认测试标准

确认测试以软件需求规格说明书为依据，检查软件的功能和性能及其它特性是否与用户的需求一致，包括合同规定的全部功能和性能、文档资料（正确且合理）、其它需求（如可移植性、兼容性、错误恢复能力、可维护性等）。

确认测试的结果可分为两类：

- 满足需求规格说明要求的功能或性能特性，用户可以接受。
- 发现与需求规格说明不一致的功能或性能特征，须列出提问单。

α测试和β测试

- 如果软件是为一个客户开发的，那么，最后由客户进行验收（Acceptance Testing）测试，以便客户确认该软件是他所需要的。
- 如果软件是给许多客户使用的，那么不可能让每个客户做验收测试，大多数软件厂商使用一个称为α测试和β测试的过程，来发现那些似乎只有最终用户才能发现的错误。

- **α测试**是由一个用户在开发者的场所来进行的，软件在开发者对用户的“指导”下进行测试，开发者负责记录错误和使用中出现的问题，所以，α测试是在一个受控的环境中进行的。
- **β测试**是由软件的最终用户在一个或多个用户场所来进行的，开发者通常不在测试现场，因此β测试是软件在一个开发者不能控制的环境中的“活的” 应用。用户记录下所有在β测试中遇到的（真正的或是想象中的）问题，并定期把这些问题报告给开发者，在接到β测试的问题报告后，开发者对系统进行最后的修改，然后开始准备向所有的用户发布最终的软件产品。

系统测试（System Testing）

系统测试，是将通过确认测试的软件，作为整个基于计算机系统的一个元素，与其它系统成分（如硬件、外设、某些支持软件、数据和人员等）集成起来，在实际运行环境下，对计算机系统进行一系列的集成测试和确认测试。

这些测试不属于软件工程过程的研究范围，而且也不只是由软件开发人员来进行的。

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。

系统测试的类型

- 恢复测试
- 安全测试
- 强度测试
- 性能测试

面向对象测试策略

- **单元测试（类测试）**
  - 在面向对象环境下，最小的可测试的单元是封装了的类或对象，而不是程序模块。
  - 面向对象软件的类测试等价于传统软件开发方法中的单元测试。但它是由类中封装的操作和类的状态行为驱动的。
  - 完全孤立地测试类的各个操作是不行的。

- 考虑一个类的层次。在基类中我们定义了一个操作X。
- 每一个派生类都使用操作X，它是在各个类所定义的私有属性和操作的环境中使用的。因使用操作X的环境变化太大，所以必须在每一个派生类的环境下都测试操作X。
- 在面向对象开发环境下，把操作完全孤立起来进行测试，其收效是很小的。

- **集成测试**
  - 因为面向对象软件没有一个层次的控制结构，所以传统的自顶向下和自底向上的组装策略意义不大。
  - 每次将一个操作组装到类中（像传统的增殖式组装那样）常常行不通，因为在构成类的各个部件之间存在各种直接的和非直接的交互。
  - 对于面向对象系统的组装测试，存在两种不同的测试策略。

- **基于线程测试 (Thread-based Test)**
  - 它把为响应某一系统输入或事件所需的一组类组装在一起。每一条线索将分别测试和组装。
- **基于使用的测试 (Use-based Test)**
  - 它着眼于系统结构，首先测试独立类，这些类只使用很少的服务器类。再测试那些使用了独立类的相关类。一系列测试各层相关类的活动继续下去，直到整个系统构造完成。

- **确认测试**
  - 在进行确认测试和系统测试时，不关心类之间连接的细节。着眼于用户的要求和用户能够认可的系统输出。
  - 为了帮助确认测试的执行，测试者需要回到分析模型，根据那里提供的事件序列（脚本）进行测试。
  - 可以利用黑盒测试的方法来驱动确认测试。

测试用例

- 测试类的方法
  - 随机测试
  - 划分测试
    - **基于状态的划分**
    - **基于属性的划分**
    - **基于功能的划分**
  - 基于故障的测试

调 试(Debugging)

测试 —— 发现错误

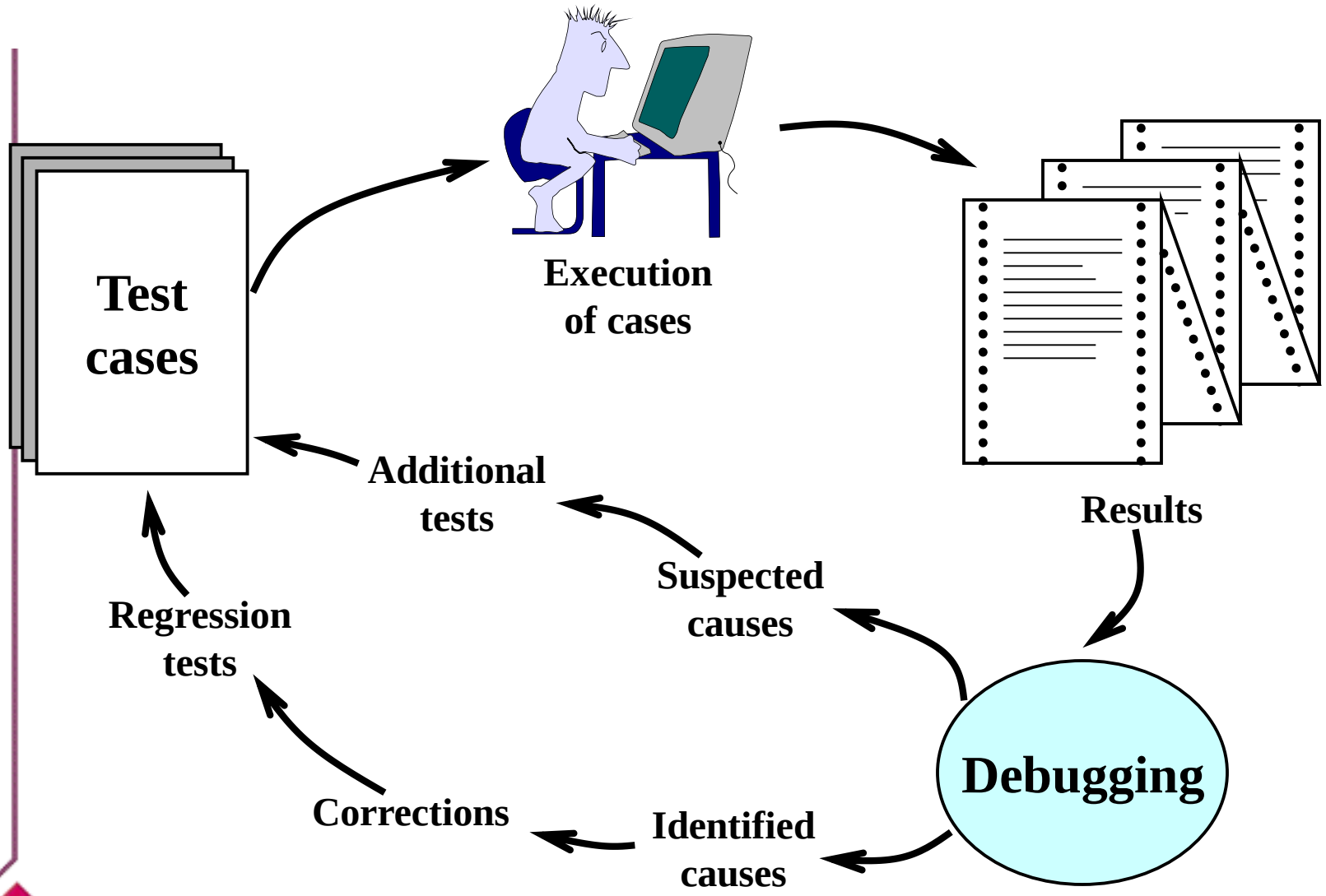
调试 —— 改正错误

第1步：确定错误的位置(95%工作量)；

第2步：改正错误。

Failure（外错误）is the departure of a system from its required behavior.

Fault（故障、内错误、error、bug）is resulted by human error in some software product.


Test cases
Execution of cases
Results
Additional tests
Suspected causes
Regression tests
Debugging
Corrections
Identified causes

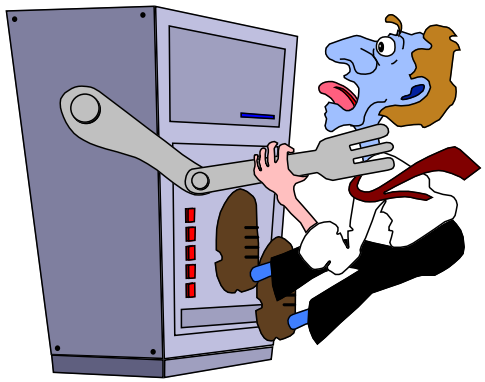
Debugging

1、调试技术

① 输出存储器内容(memory dump):
以八进制或十六进制的形式印出存储器的内容。

缺点:

- 输出信息量极大，
不易解读且大多无用；
- 输出的是程序在某一
时刻的静态情况，且
往往不是出错时的状态。



② 插入“watch points”(或称“spy points”)

- 人工插入打印

  缺点:
  - 改动源代码，增加了出错机会;
  - 打印信息可能太多。
- 自动调试工具 —— 无须打印额外信息，且不改动源代码

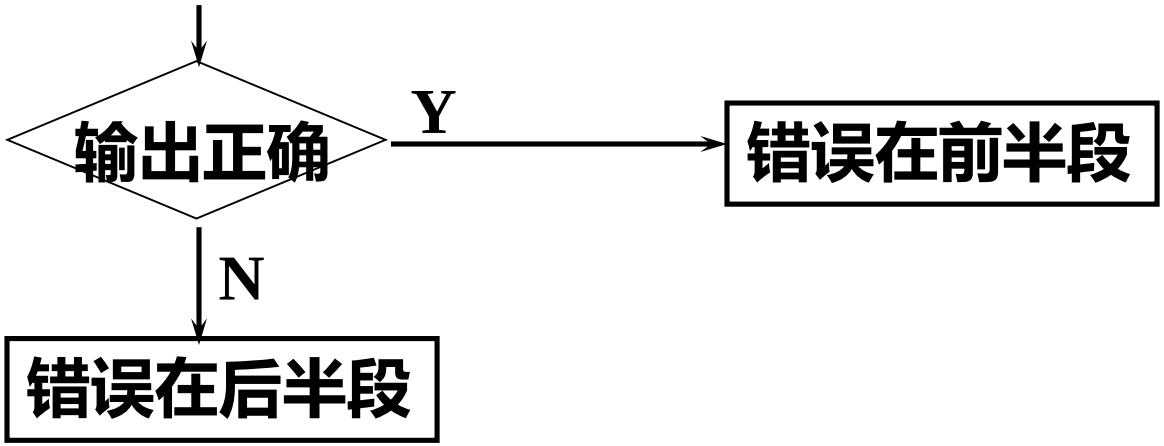
2、调试策略

调试过程的关键不是调试技术，而是用来推断错误原因的基本策略。主要有：

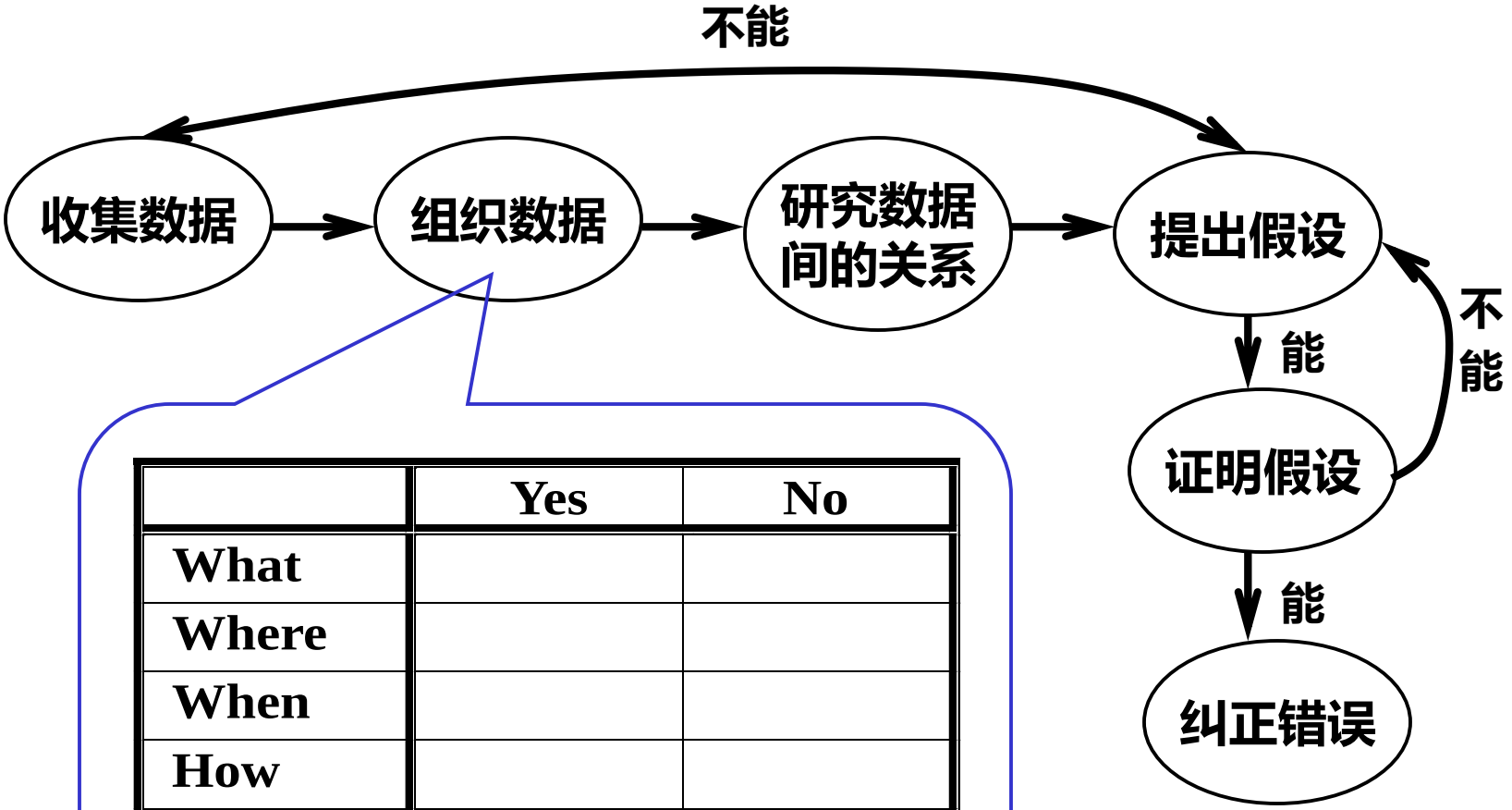
① 试探法，凭经验猜测。

② 回溯法：由症状(symptom)最先出现的地方，沿control flow向回检查。适用于小型程序。

③ 对分法：在关键点插入变量的正确值，则：



④ 归纳法：从错误症状中找出规律，推断根源。

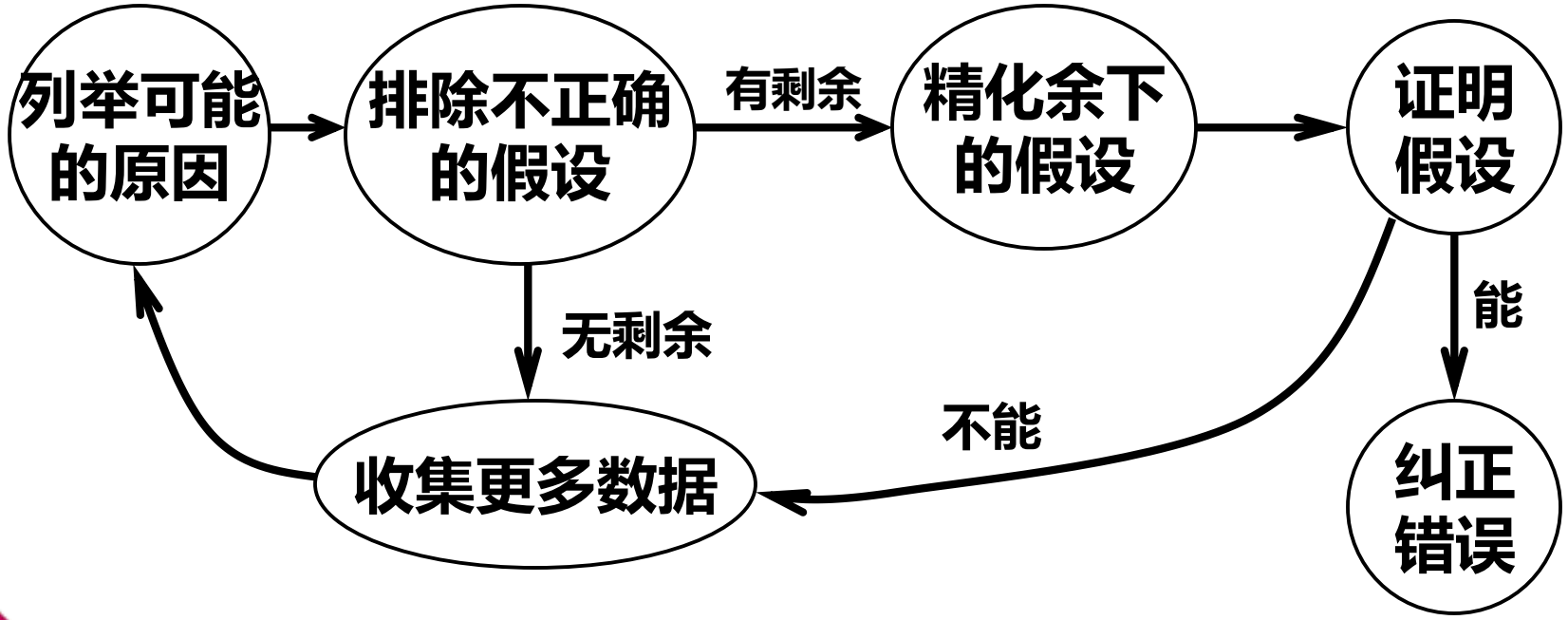


	Yes	No
What		
Where		
When		
How		

3W1H Table

⑤ 演绎法：普通 → 特殊

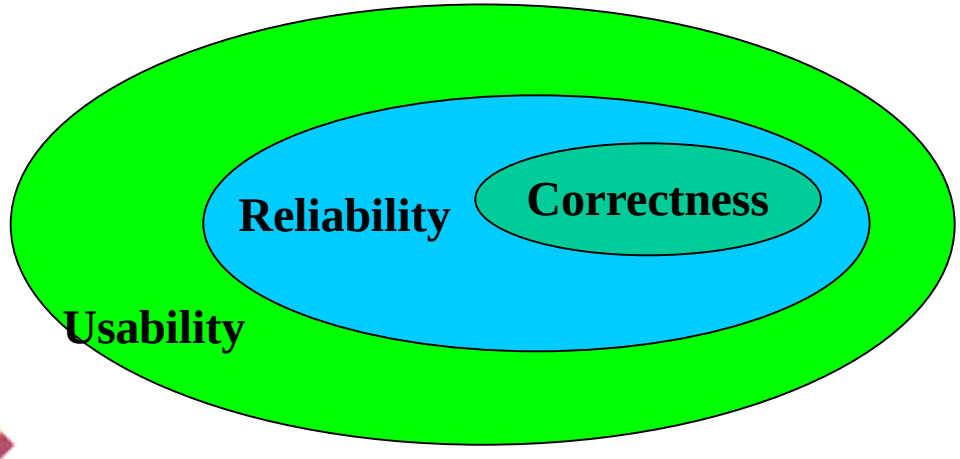
从假设中逐步排除、精化，从而导出错误根源。



软件可靠性(Reliability)

1、基本概念

- 可靠性(Reliability)：程序在给定的时间间隔内，按照说明书的规定，成功地运行的概率。
- 可用性(Usability)：程序在给定的时间点，按照说明书的规定，成功地运行的概率。
- 正确性(Correctness)：程序的功能正确。

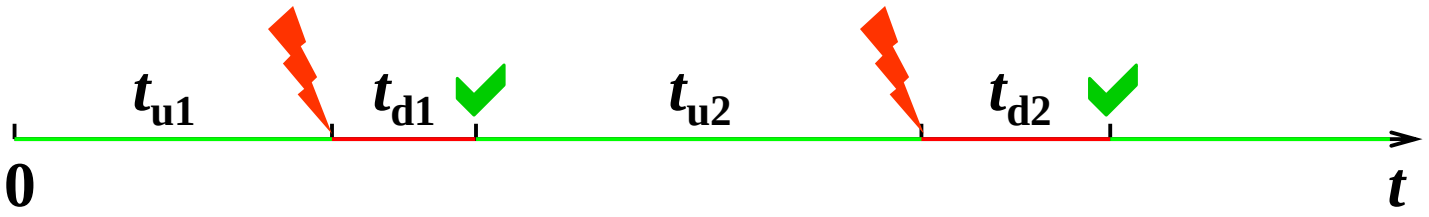


设系统故障停机时间为$t_{d1}, t_{d2}, \ldots$; 正常运行时间为$t_{u1}, t_{u2}, \ldots$; 则系统的“稳态可用性”为

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \quad \text{(Shooman, 1983)}$$

其中 MTTF = Mean Time To Failure = $\frac{1}{n}\sum_{i=1}^{n} t_{ui}$

MTTR = Mean Time To Repair = $\frac{1}{n}\sum_{i=1}^{n} t_{di}$



2、估算 MTTF：

$$MTTF = \frac{1}{K(E_T/I_T - E_C(\tau)/I_T)}$$

其中：K为经验常数（典型值约在200左右）；

E_T为测试前故障总数；

I_T为程序长度（机器指令总数）；

τ为测试（包括调试）时间；

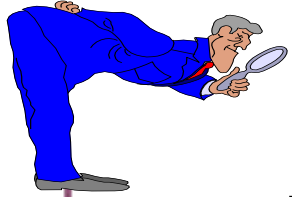
$E_C(\tau)$为时间从0至τ期间改正的错误数。

前提假设：

① $E_T / I_T \approx$ Constant （通常为0.5 ~ 2%）

② 调试中没有引入新故障（即E_T与时间τ无关）

③MTTF与剩余故障成反比



换个角度看问题

$$E_C(\tau) = E_T - \frac{I_T}{K \cdot MTTF}$$

意义：可根据对软件平稳运行时间的要求，估算需改正多少个错误后才能结束测试。

还有一个问题 —— $E_T = ?$

估算方法：

① 植入故障法：

人为植入N_S个故障，测后发现n_s个植入故障和n个原有故障，则设 $\frac{n_S}{N_S} = \frac{n}{\hat{N}} \Rightarrow \hat{N} = \frac{n}{n_S} \approx E_T$

② Hyman 分别测试法：

二人（组）分别独立测试同一程序，甲测得故障总数为B_1，乙测得为B_2，其中有b_c是相同的，设以甲的测试结果为基准（即相当于①中的植入故障），则设

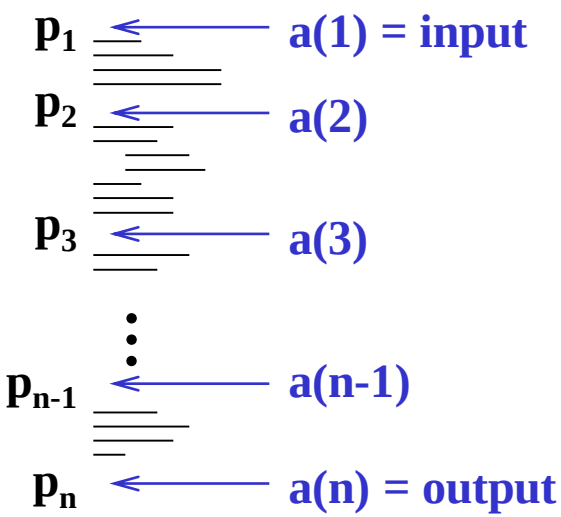
$$\frac{b_c}{B_1} = \frac{B_2}{\hat{B}} \Rightarrow \hat{B} = \frac{B_2}{b_c} B_1 \approx E_T$$

一般多测几个 $\hat{B}$ 取平均。

3、正确性说明

目标：研究能证明程序正确性的自动系统。

原理：



a(i):断言(assertion)

证明：对任一个i，执行了p_i到p_{i+1}之间的语句后，可将a(i)变为a(i+1)，则证明由a(1)可导出a(n)。若a(1)和a(n)正确，且程序确定可终止，则证明程序正确。

但是：即使有了正确性证明程序，软件测试也仍然是需要的。

因为：①正确性证明只证明程序功能正确，但不能验证动态特性；

②证明过程本身也可能发生错误。

小结

- 软件测试的目的和原则
- 白盒测试和黑盒测试
- 软件测试策略
- 调试
- 软件可靠性