

第十五章 测试战术

王美红



目标

讨论测试用例设计技术

主要内容

- 软件测试基础
- 白盒测试
- 黑盒测试
- 面向对象测试方法
- 类级可应用的测试方法
- 类间测试用例设计

15.1 软件测试基础

- 软件可测试性

- 计算机能够被测试的容易程度

- 受以下方面影响：

- 可操作性 -你所看到的就是你所测的

- 可控制性 -对软件控制的越好，测试越能被自动执行和优化

- 可分解性 -通过控制测试范围，能够很快地分解问题，完成更灵巧的再测试

- 简单性 -需要测试的内容越少，测试的速度越快

- 稳定性 -变更越少，对测试的破坏越小

- 易理解性 -得到的信息越多，进行的测试越灵巧

15.1 软件测试基础（续）

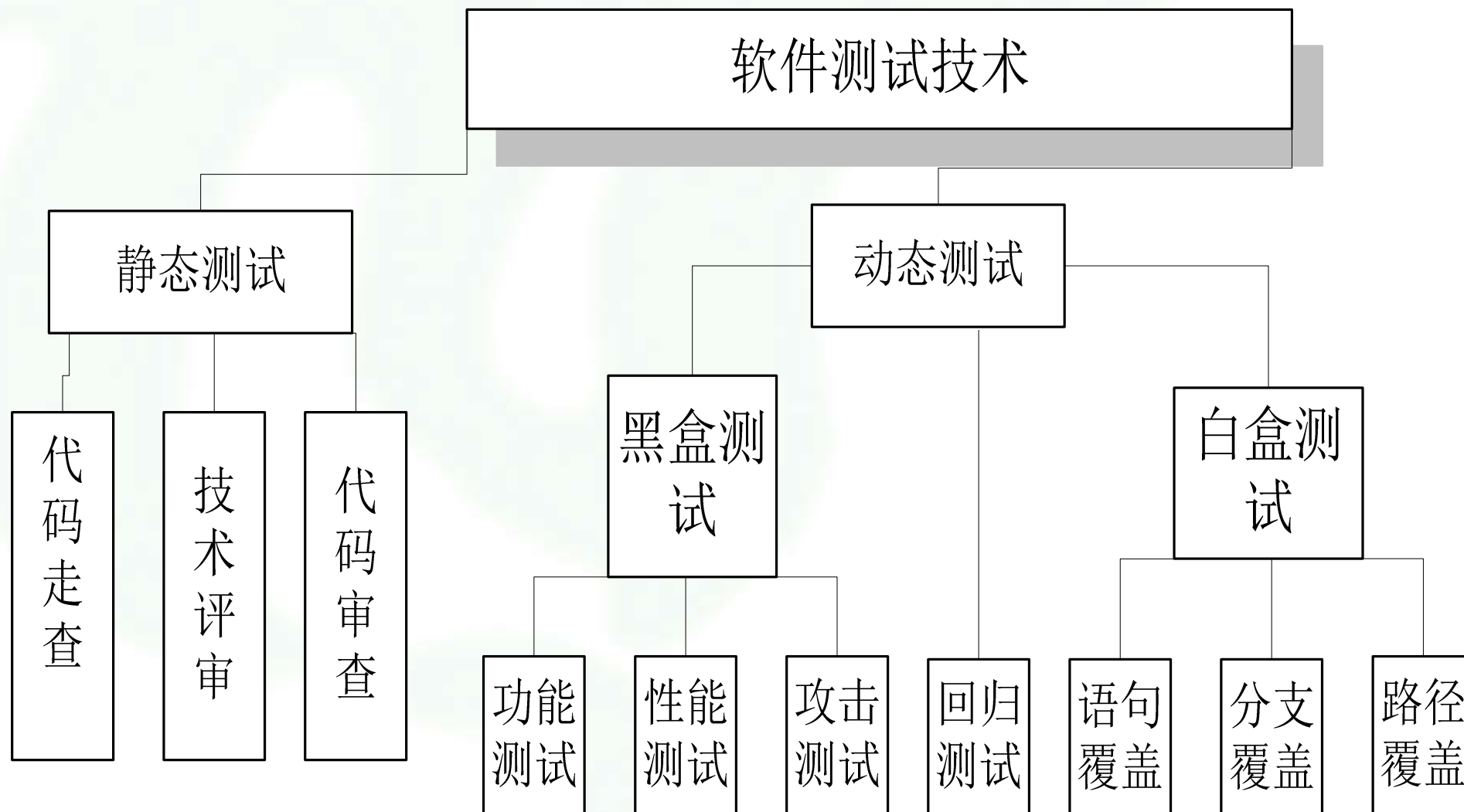
- 好的测试具有以下特征：

1. 高的有效性
2. 不冗余性
3. 最佳性
4. 适当的复杂性

测试的原则

- 原则1: 测试显示缺陷的存在
- 原则2: 穷尽测试是不可能的
- 原则3: 测试的尽早介入
- 原则4: 缺陷的集群性
- 原则5: 杀虫剂悖论
- 原则6: 测试活动依赖于测试内容
- 原则7: 没有失效不代表系统是可用的
- 原则8: 测试的标准是用户的需求
- 原则9: 尽早定义产品的质量标准
- 原则10: 测试贯穿于整个生命周期
- 原则11: 第三方或独立的测试团队

软件测试技术



软件测试技术

- 黑盒测试/白盒测试
 - 从要不要看代码部分来区分
- 动态测试/静态测试
 - 从要不要运行软件来区分

15.2 黑盒测试与白盒测试

- 黑盒测试
 - 在软件接口处进行测试，不需了解内部结构。
- 白盒测试
 - 检查软件的过程细节
 - 将获得“百分之百正确的程序”？ ——穷举测试——不可能

15.3 白盒测试

- 也称**玻璃盒测试**，一种测试用例设计方法
- 主要想对程序模块进行如下的检查：
 - 对程序模块的**所有独立的执行路径**至少被执行一次
 - 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
 - 在上下边界及可操作的范围内执行所有的循环
 - 检验内部数据结构以确保其有效性

白盒测试主要内容

控制结构测试

逻辑覆盖

- 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

基本路径测试

- 利用流图表示控制逻辑
- 确定覆盖测试路径上界的计算（环复杂度计算）
- 根据流图标识独立路径
- 用基本路径法导出测试案例

15.4 逻辑覆盖

- 逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。它属白盒测试。

- 语句覆盖
 - 判定一条条件覆盖
- 判定覆盖
 - 条件组合覆盖
- 条件覆盖
 - 路径覆盖

(1) 语句覆盖 (Statement coverage)

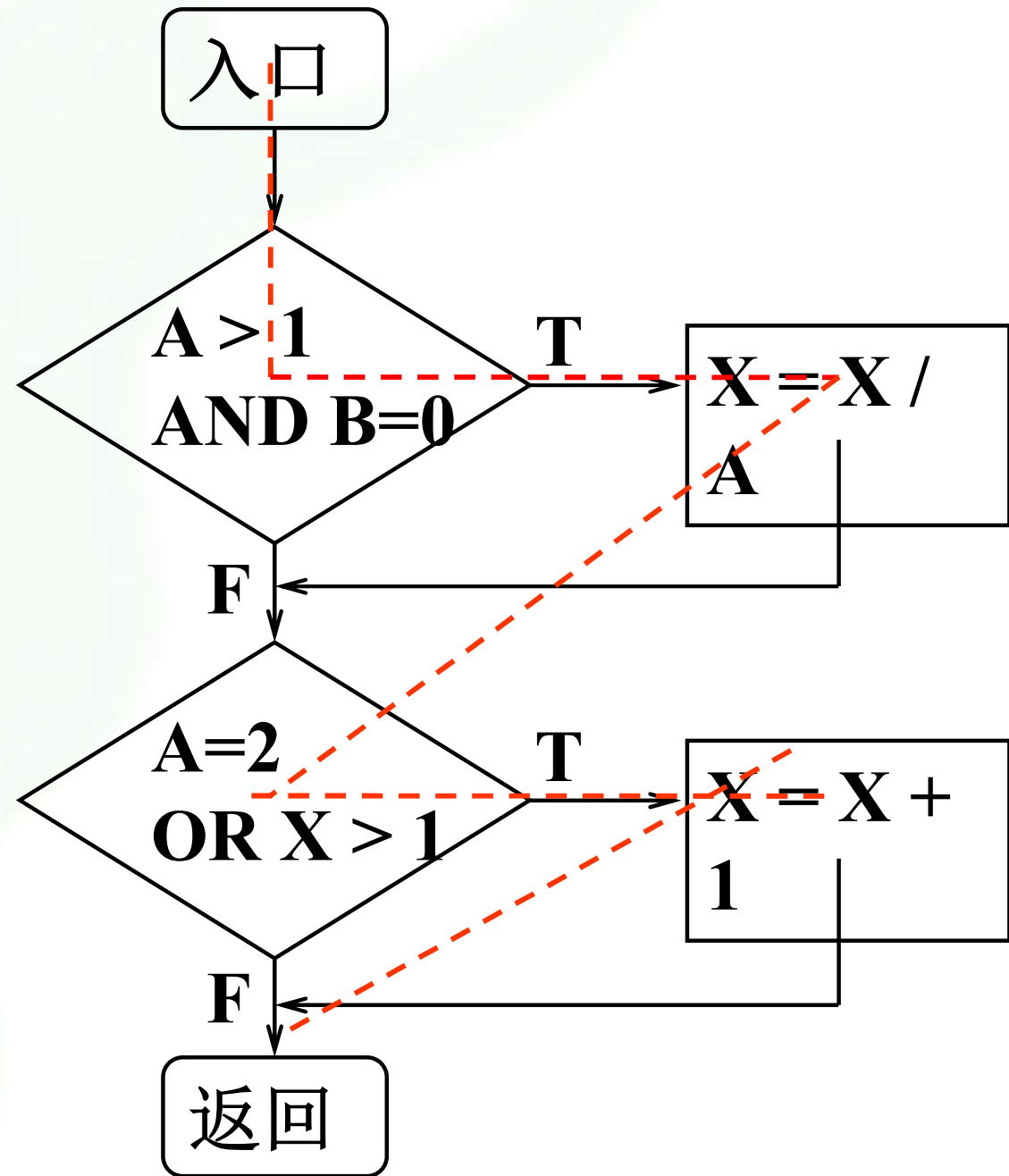
每个语句至少执行一次

例:

Test case :

$A=2$, $B=0$, $X=4$.

问题: 若 **AND**
错写为 **OR**, 或
 $X>1$ 错写为 **$X<1$** ,
则错误无法由
上例测出。



(2) 判定覆盖 (Branch coverage)

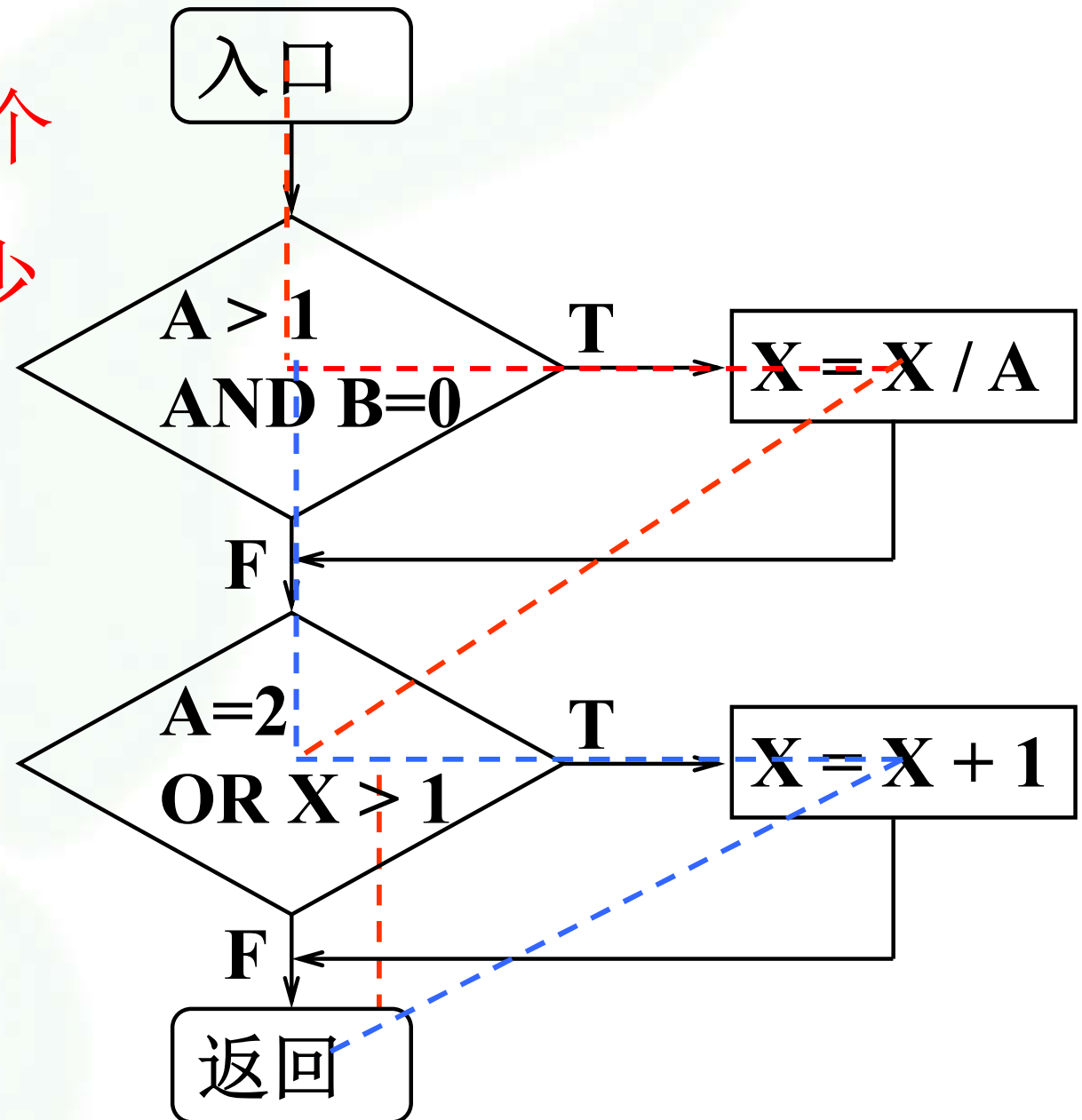
在(1)的基础上，每个判定的每个分支至少执行一次。

Test cases:

① $A=3, B=0, X=3$

② $A=2, B=1, X=1$

问题：若 $X>1$ 错写为 $X<1$ ，仍然无法被测出。

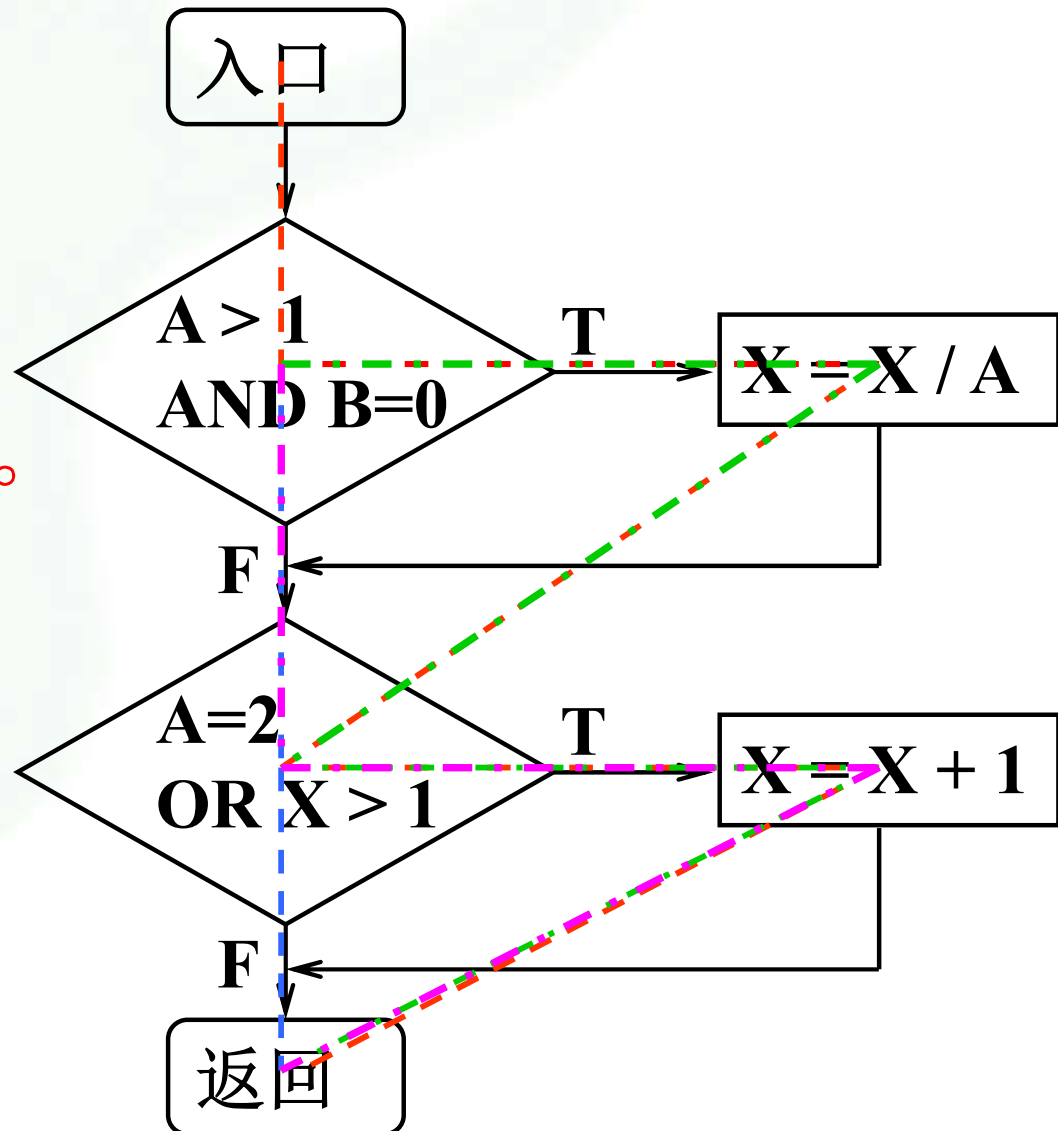


(3) 条件覆盖 (Condition coverage)

在(1)的基础上，使每个判定表达式的每个条件都取到各种可能的结果。

Test cases:

- ① $A=2, B=0, X=4$
(满足 $A>1, B=0; A=2, X>1$)
- ② $A=1, B=1, X=1$
(满足 $A\leq 1, B\neq 0; A\neq 2, X\leq 1$)



(4) 判定/条件覆盖

问：条件覆盖 \Rightarrow 判定覆盖

答：不一定。

反例：①A=2, B=0, X=1

②A=1, B=1, X=2

- 判定/条件覆盖即判定覆盖 \wedge 条件覆盖

(5) 条件组合覆盖

每个判定表达式中条件的各种可能组合都至少出现一次。

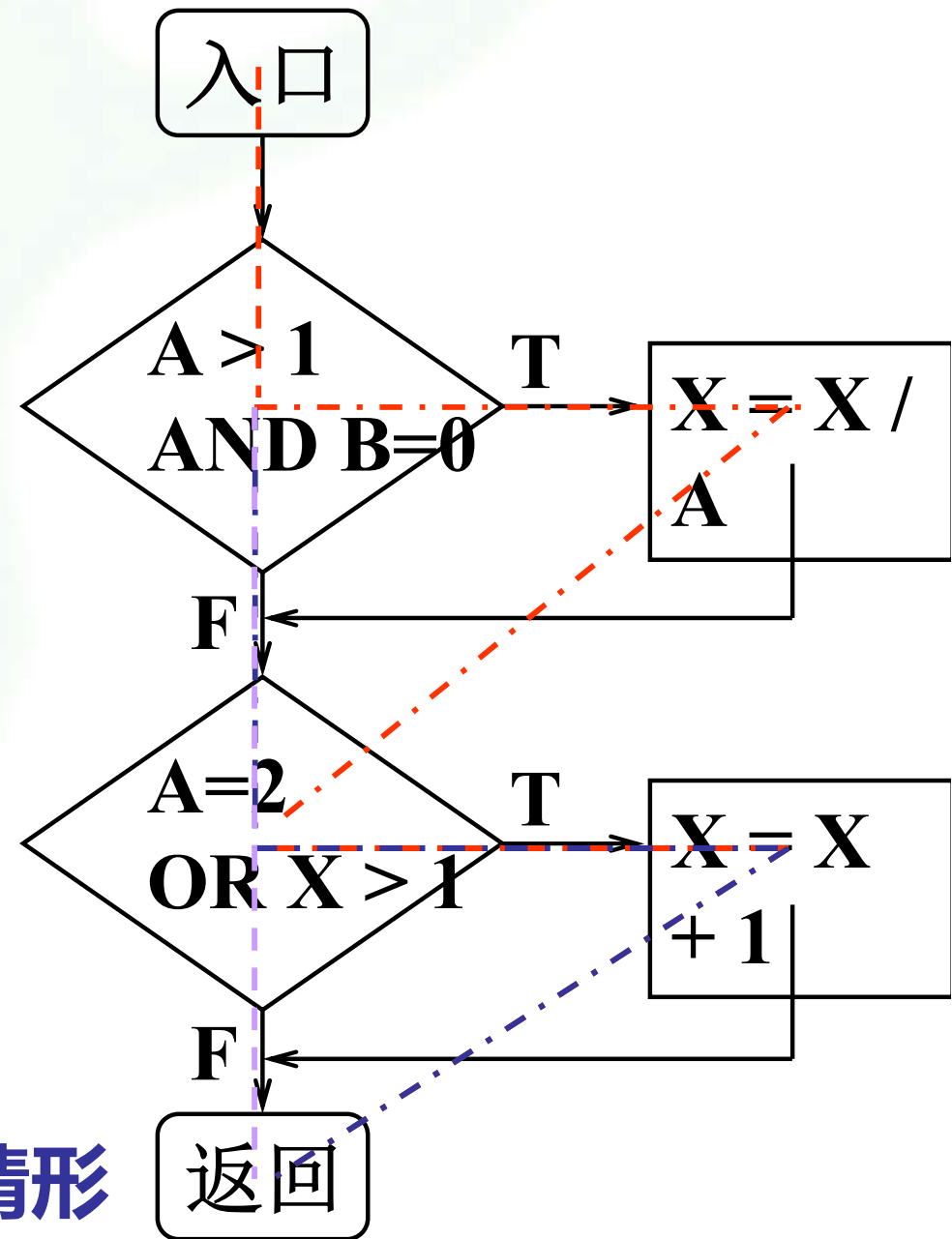
全部可能的条件组合为：

- | | |
|---------------------|------------------------|
| ① $A > 1, B = 0$ | ② $A > 1, B \neq 0$ |
| ③ $A \leq 1, B = 0$ | ④ $A \leq 1, B \neq 0$ |
| ⑤ $A = 2, X > 1$ | ⑥ $A = 2, X \leq 1$ |
| ⑦ $A \neq 2, X > 1$ | ⑧ $A \neq 2, X \leq 1$ |

Test cases:

- | | |
|-------------------------|-------|
| ① $A = 2, B = 0, X = 4$ | (T T) |
| ② $A = 2, B = 1, X = 1$ | (F T) |
| ③ $A = 1, B = 0, X = 2$ | (F T) |
| ④ $A = 1, B = 1, X = 1$ | (F F) |

问题：没有测试到 (T F) 的情形



考察control flow graph 的角度，还可考虑下述覆盖^{软件工程}：

(6) 点覆盖 = 语句覆盖

(7) 边覆盖 = 判定覆盖

(8) 路径覆盖(Path coverage):
每条可能的路径都至少执行一次，若图中有环，则每个环至少经过一次。

Test cases:

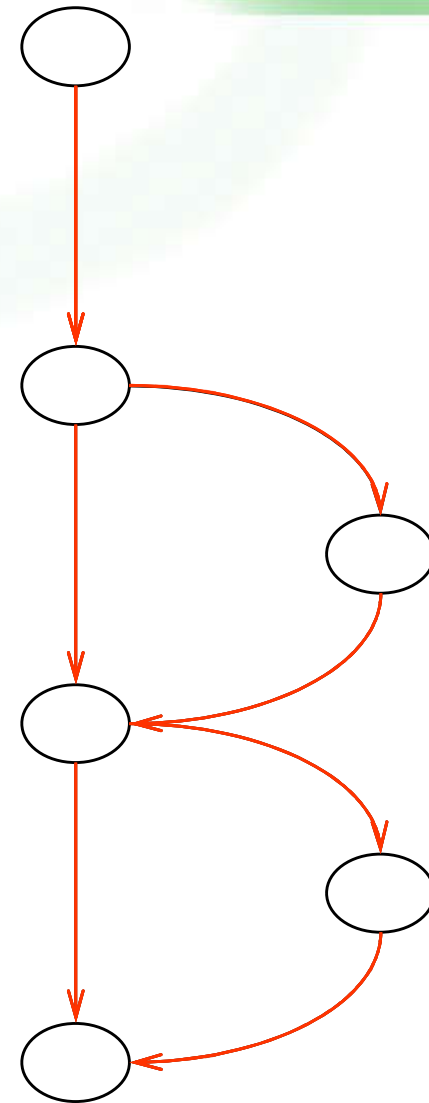
① A=1 , B=1 , X=1

② A=1 , B=1 , X=2

③ A=3 , B=0 , X=1

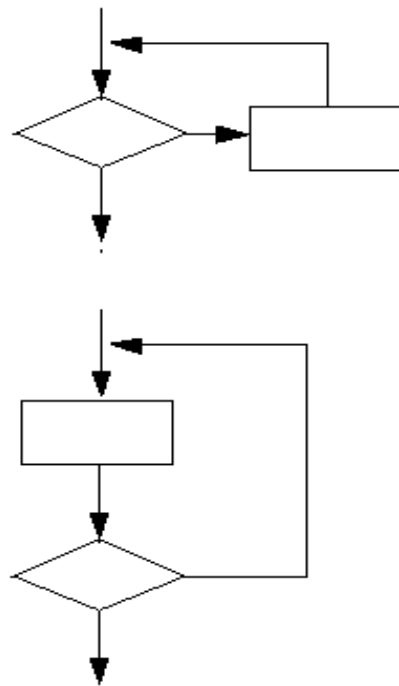
④ A=2 , B=0 , X=4

(9) 路径覆盖 \wedge 条件组合覆盖

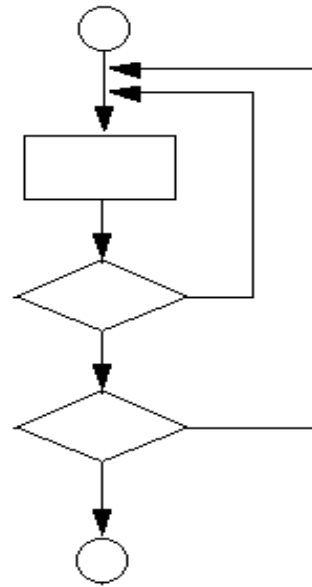


(6) 循环测试路径选择

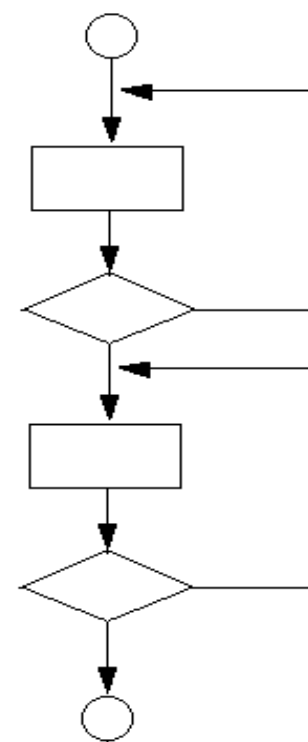
- 循环分为4种不同类型：简单循环、连锁循环、嵌套循环和非结构循环。



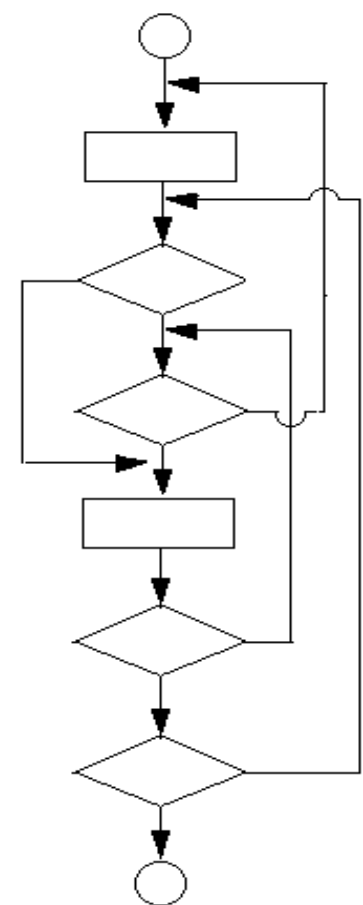
简单循环



嵌套循环



连锁循环



非结构循环

简单循环

- 零次循环：从循环入口到出口
- 一次循环：检查循环初始值
- 二次循环：检查多次循环
- m 次循环：检查在多次循环
- 最大次数循环、比最大次数多一次、少一次的循环。

例：求最小值

 $k = i;$

①

 $\text{for } (j = i+1; j \leq n; j++)$

②

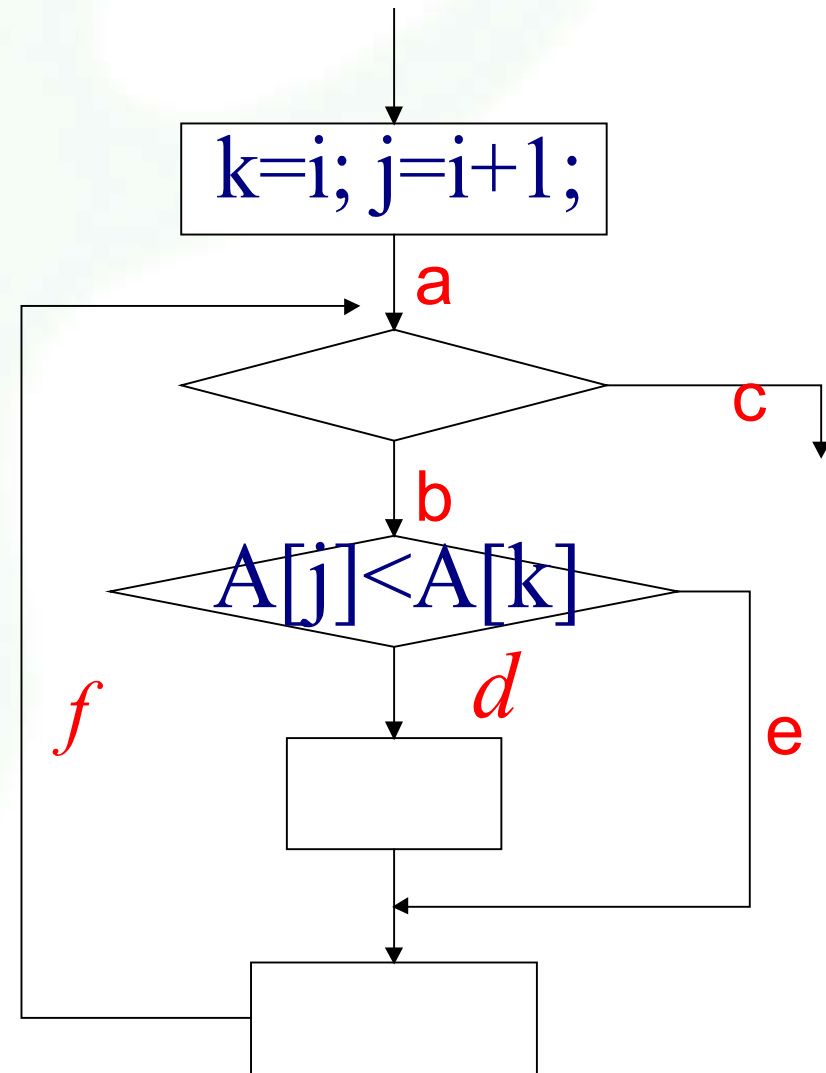
③

⑥

 $\text{if } (A[j] < A[k]) \text{ then } k = j;$

④

⑤



测试用例设计

循环	i	n	A[i]	A[i+1]	A[i+2]	k	路 径
0	1	1				i	ac
1	1	2	1	2		i	ab e fc
			2	1		i+1	ab d fc
2	1	3	1	2	3	i	ab e f e fc
			2	3	1	i+2	ab e f d fc
			3	2	1	i+2	ab d f d fc
			3	1	2	i+1	ab d f e fc

~~d~~ 改 ~~k~~ 的值, ~~e~~ 不改 ~~k~~ 的值

嵌套循环

- 对最内层循环做简单循环的全部测试。所有其它层的循环变量置为最小值；
- 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值
- 反复进行，直到所有各层循环测试完毕
- 对全部各层循环同时取最小循环次数，或者同时取最大循环次数

连锁循环

- 如果各个循环**互相独立**，则可以用与简单循环相同的方法进行测试。但如果几个循环**不是互相独立**的，则需要使用测试嵌套循环的办法来处理。

非结构循环

- 这一类循环应该使用结构化程序设计方法
重新设计测试用例

(7) 数据流测试

- **数据流测试方法**就是根据变量的**定义和使用位置**来选择程序测试路径的测试方法。
- 变量X的**定义-使用链（或称DU链）**：
 - 形式为[X,S,S']，其中S和S'为语句号，X在DEF(S)和USE(S')中，且在语句S中定义的X在语句S'中有效。
- 一个简单的数据流测试策略要求每个DU链至少覆盖一次。

15.5 基本路径测试

- 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。
- 它是在程序控制流图的基础上，分析控制构造的环路复杂性，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。

1. 画出流图

- 流图（或程序图）

- 一种简单的控制流表示方法

- **圆**：流图结点，表示一个或多个过程语句

- 流程图中的处理框序列和一个菱形判定框可以映射为单个结点。

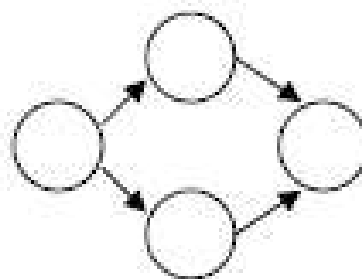
- **箭头**：边或连接，表示控制流，一条边必须终于一个节点

- 由边和节点限定的区间称为**域**，计算区域时不要忘记区域外的部分

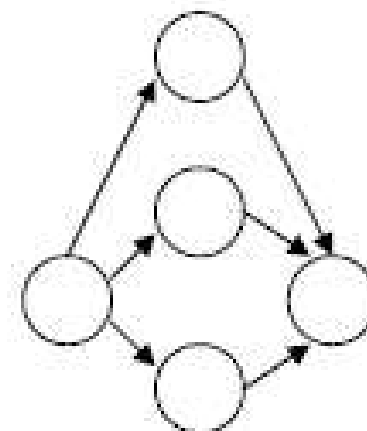
利用流图(flow graph)表示控制逻辑



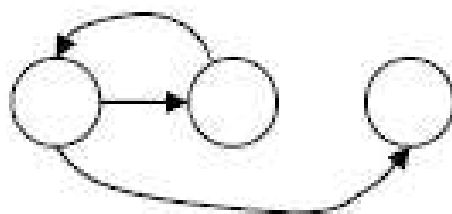
顺序结构



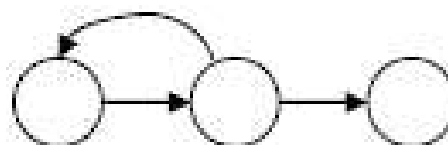
if 结构



Case 结构

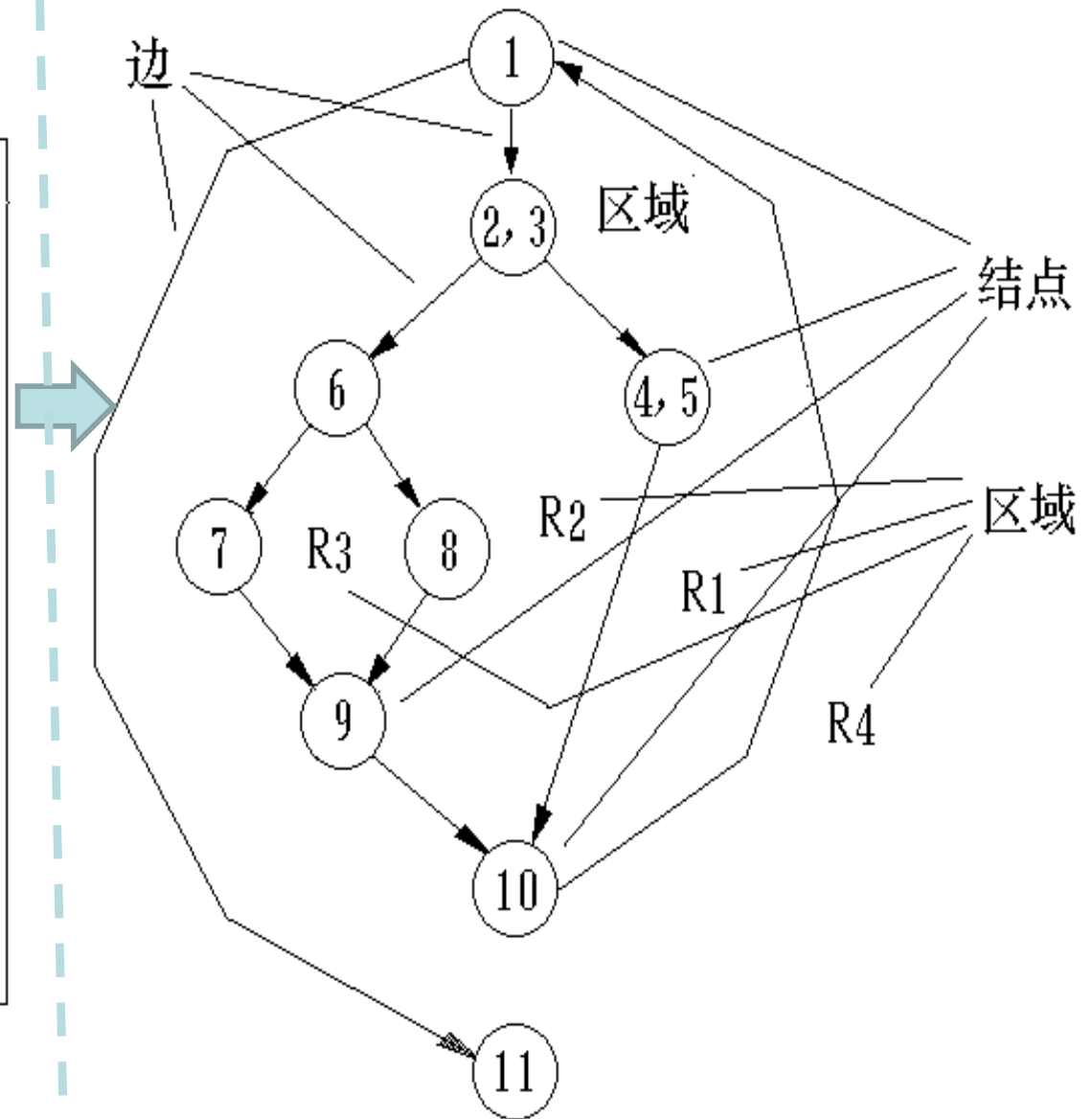
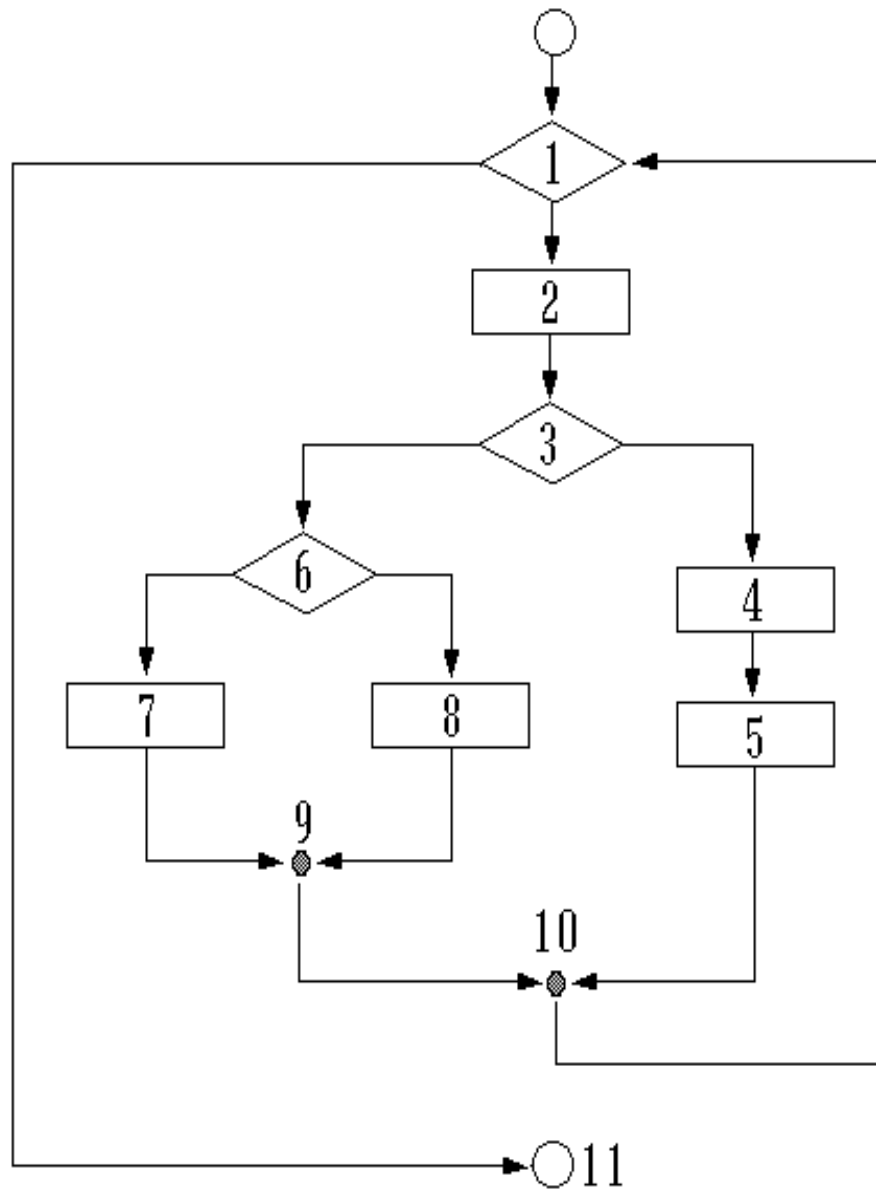


while 结构

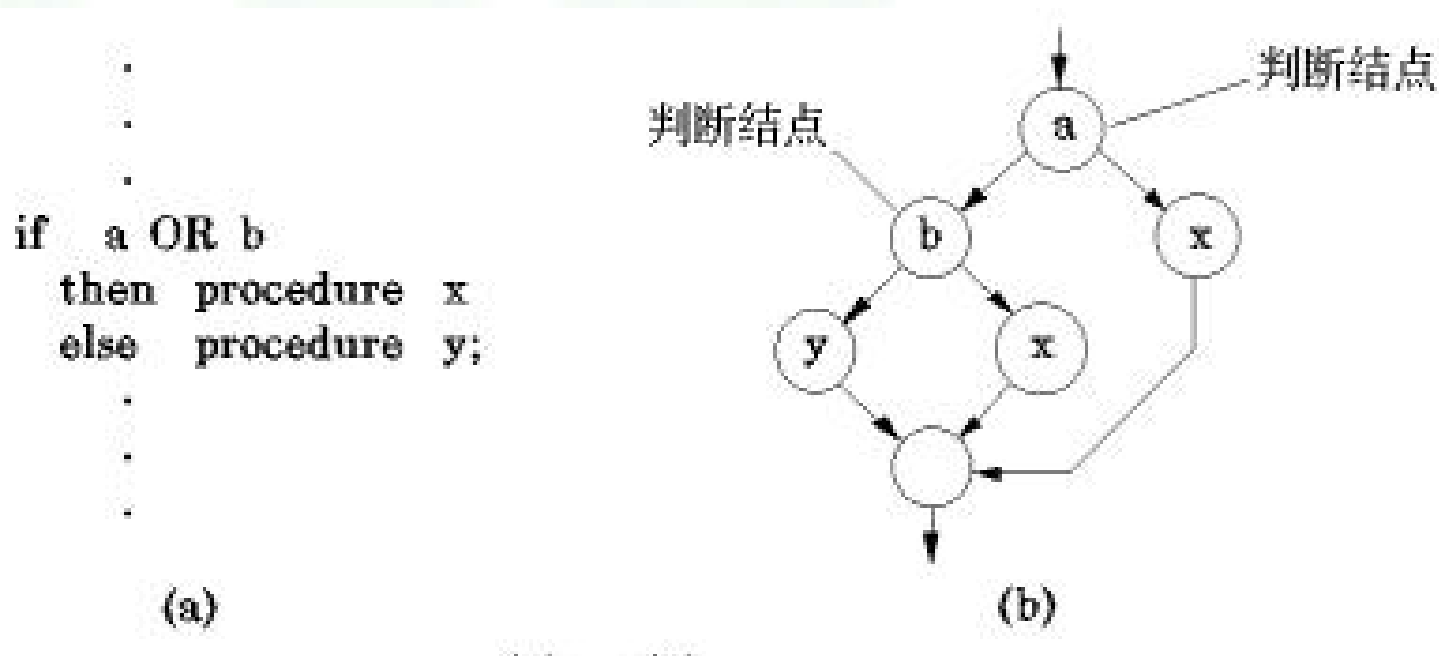


until 结构

1. 画出流图



- 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条件的嵌套的判断。



2. 计算程序环路复杂性

- 程序的环路复杂性给出了程序基本路径集中的独立路径条数，这是确保程序中每个可执行语句至少执行一次所必需的测试用例数目的上界。
- 从流图来看，一条独立路径是至少包含有一条在其它独立路径中从未有过的边的路径。

2. 计算程序环路复杂性(续)

- 环形复杂度计算方法:

1. 流图的区域数量应该对应于环路复杂度

2. 给定流图G的环路复杂度V(G)定义为:

$$V(G)=E-N+2$$

- 其中: E为流图中的边数量, N为流图中的节点数量

— 给定流图G的环路复杂度V(G)也可以定义为:

$$V(G)=P+1$$

- 其中: P为流图中的判断节点数量

独立路径

- 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。

3. 导出测试用例

- 导出测试用例，确保基本路径集中的每一条路径的执行。
- 根据判断结点给出的条件，选择适当的数据以保证某一条路径可以被测试到 — 用逻辑覆盖方法。

15.5 基本路径测试（续）

- 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。
- 必须注意，一些独立的路径（如例中的路径1），往往不是完全孤立的，有时它是程序正常的控制流的一部分，这时，这些路径的测试可以是另一条路径测试的一部分。

15.6 黑盒测试

- 黑盒测试又叫做功能测试、数据驱动测试或行为测试。
- 这种方法是把测试对象看做一个黑盒子
- 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。

黑盒测试所发现的错误类型：

- 功能不正确或遗漏
- 接口错误
- 数据结构或外部数据库访问错误
- 行为和性能错误
- 初始化和终止错误

黑盒测试的测试用例设计

- 等价类划分
- 边界值分析
- 错误推测法
- 因果图

1. 等价类划分

- 等价类划分是一种典型的黑盒测试方法，使用这一方法时，完全不考虑程序的内部结构，只依据程序的规格说明来设计测试用例。
- 等价类划分方法把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。

等价类划分

- 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）和选取测试用例两步。

1. 划分等价类

等价类是指某个输入域的子集合。在该子集合中，各个输入数据对于揭露程序中的错误都是等效的。测试某等价类的代表值就等价于对这一类其它值的测试。

等价类划分

- 等价类的划分有两种不同的情况：
 - ① **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。
 - ② **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。
- 在设计测试用例时，要同时考虑有效等价类和无效等价类的设计。

等价类划分

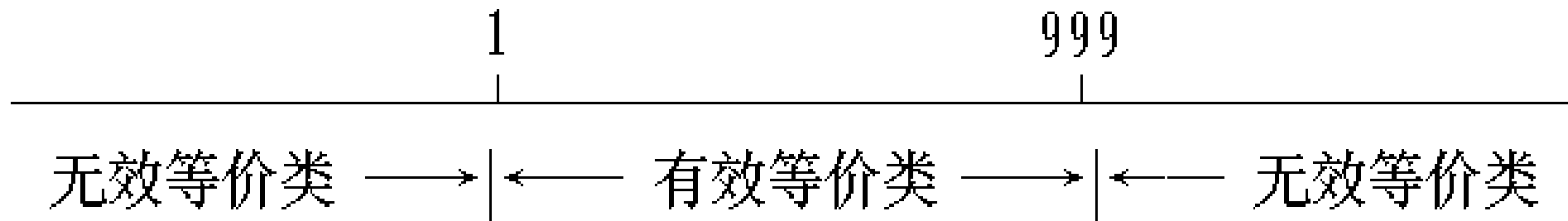
- 划分等价类的原则。
 - (1) 如果输入条件规定了取值范围，或值的个数，则可以确立一个有效等价类和两个无效等价类。

等价类划分

- 例如，在程序的规格说明中，对输入条件有一句话：

“…… 项数可以从1到999 ……”

则有效等价类是 “ $1 \leq \text{项数} \leq 999$ ”



等价类划分

(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。

- 例如，在Pascal语言中对变量标识符规定为“以字母打头的……串”。那么所有以字母打头的构成有效等价类，而不在集合内（不以字母打头）的归于无效等价类。

等价类划分

(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。

等价类划分

- 例如，在教师上岗方案中规定对教授、副教授、讲师和助教分别计算分数，做相应的处理。因此可以确定4个有效等价类为教授、副教授、讲师和助教，一个无效等价类，它是所有不符合以上身份的人员的输入值的集合。

等价类划分

(5) 如果规定了输入数据必须遵守的规则，则可以确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

等价类划分

2. 测试用例设计步骤

- 设计一个新方案以尽可能多地覆盖尚未被覆盖的有效等价类；重复这一步骤直到所有有效类都被覆盖为止
- 设计一个新方案以覆盖一个且仅一个尚未被覆盖的无效等价类；重复这一步骤直到所有无效类都被覆盖为止。（通常程序执行一个错误后即不继续检测其它错误，故每次只测一个无效类）

等价类划分———举例1

- 例1：用等价类划分法设计测试用例的实例

在某一PASCAL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。”

等价类划分———举例1

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)

等价类划分———举例2

例2：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。

等价类划分———举例2

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间；

等价类划分——举例2

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:

① 1~6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

等价类划分——举例2

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”

等价类划分——举例2

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”

2. 边界值分析 (Boundary Value Analysis)

- 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，可以查出更多的错误。

注意： ① 程序最容易在边界发生错误；

② 通常与等价划分结合进行。

边界值分析

- 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。

但如果把六个不等式中的任何一个大于号“ $>$ ”错写成大于等于号“ \geq ”，那就不能构成三角形。问题恰出现在容易被疏忽的边界附近。

3. 错误推测法

- 可以靠经验和直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的例子。这就是错误推测法（如常见的错误）。
- 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，根据它们选择测试用例。

3. 错误推测法

思路：① 列出可能有的错误；

② 列出容易发生错误的特殊情况。以此为基础设计测试方案。

根据：直觉、经验；

工具：常见错误清单、判定表

实用策略：黑盒设计 ⊕ 白盒补充

1. 在任何情况下都应该使用边界值分析的方法；
2. 必要时用等价划分法补充；
3. 必要时再用错误推测法补充；
4. 对照程序逻辑，检查测试方案。可根据对程序可靠性的要求采用不同的逻辑覆盖标准，必要时补充一些测试方案。

注：即使用上述综合策略设计测试方案，仍不能保证发现一切错误。例如Lucent公司经过包括逐行检查源代码在内的多方面测试之后，其软件能达标运行的成功率为：80%

4. 因果图

- 如果在测试时必须考虑输入条件的各种组合，可使用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来设计测试用例，这就需要利用因果图。
- 因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

用因果图生成测试用例的基本步骤

1. 分析软件规格说明描述中，哪些是**原因**（即输入条件或输入条件的等价类），哪些是**结果**（即输出条件），并给每个原因和结果赋予一个标识符。
2. 分析软件规格说明描述中的语义，找出原因与结果之间，原因与原因之间对应的是有什么关系？根据这些关系，**画出因果图**。

用因果图生成测试用例的基本步骤

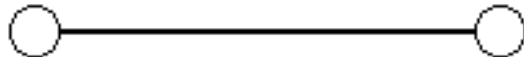
3. 由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出现。为表明这些特殊情况，在因果图上用一些记号标明约束或限制条件。
4. 把因果图转换成判定表。
5. 把判定表的每一列拿出来作为依据，设计测试用例。



因果图

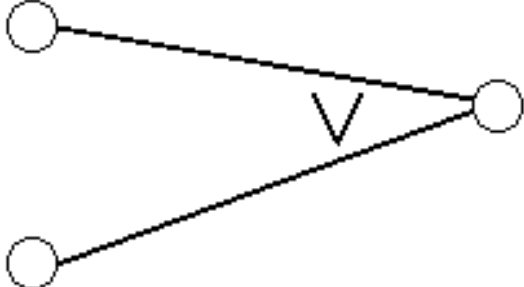
- 在因果图中出现的基本符号


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，可取值“0”或“1”。“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：

(a) 恒等  \bigcirc ————— \bigcirc **E1**

(b) 非 $C1$  \bigcirc ———  ——— \bigcirc **E1**

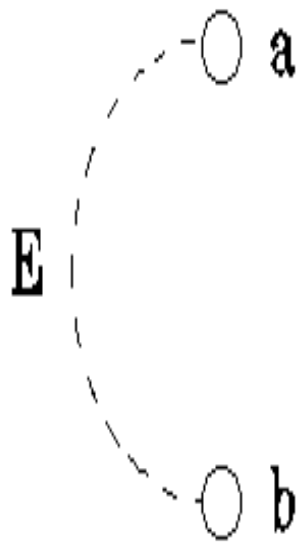
(c) 或  \bigcirc ——— \bigvee ——— \bigcirc **E1**
 \bigcirc ——— \bigvee ——— \bigcirc **E1**

(d) 与 $C1$  \bigcirc ——— \bigwedge ——— \bigcirc **E1**
 $C2$ \bigcirc ——— \bigwedge ——— \bigcirc **E1**

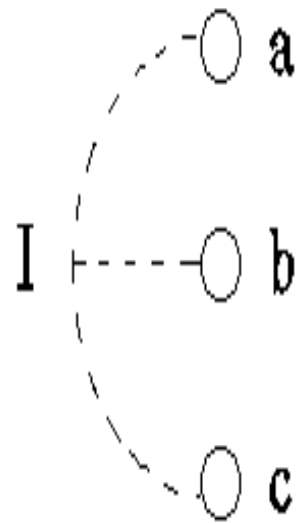
因果图

- 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。



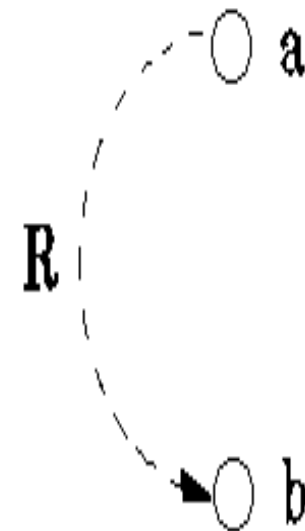
(1) E (互斥·排他)



(2) I (包含·或)



(3) O (唯一)



(4) R (要求)



(5) M (屏蔽)

因果图

- 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：

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。”

因果图

(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

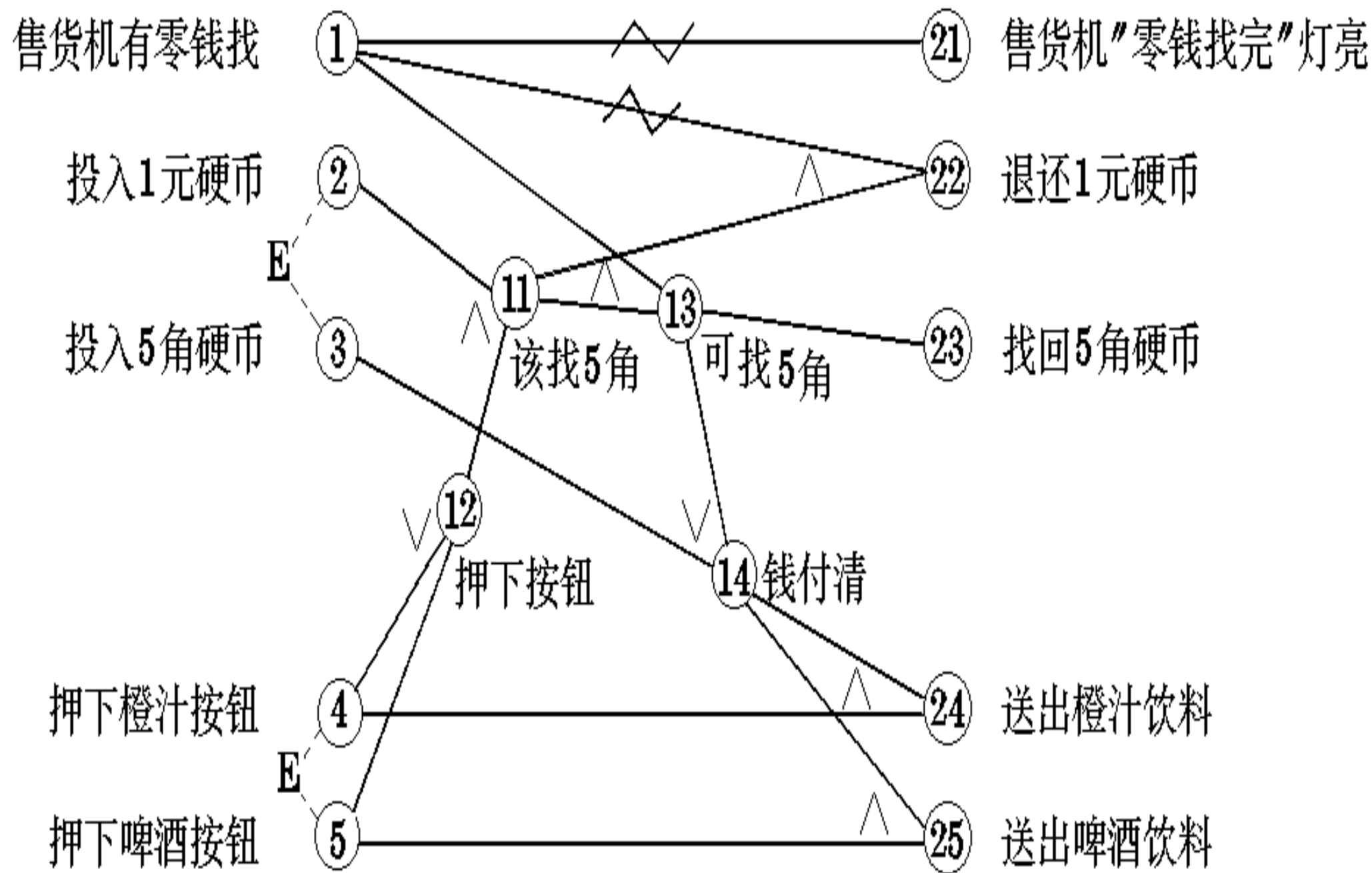
11. 投入1元硬币且押下饮料按钮
15. 押下〔橙汁〕或〔啤酒〕的按钮
15. 应当找5角零钱并且售货机有零钱找
14. 钱已付清

因果图

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



(5) 转换成判定表

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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	②	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	0	
	③	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	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	0	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	1
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	⑳						0	0	0		0	0	0		0	0	0							1	1	1		1	1	1		1	1	1
	㉑						0	0	0		0	0	0		0	0	0							1	1	0		0	0	0		0	0	0
	㉒						1	1	0		0	0	0		0	0	0							0	0	0		0	0	0		0	0	0
	㉓						1	0	0		1	0	0		0	0	0							0	0	0		1	0	0		0	0	0
	㉔						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	

15.7 面向对象测试方法

- 以设计一系列检验类操作的“小型测试”以及当一个类与其他类进行协作时是否出现错误开始
- 对子系统测试：
 - 结合基于故障的方法，运用基于使用的测试对相互协作的类进行完全检查
- 最后利用用例发现软件确认层的错误。

1. 面向对象概念的测试用例设计的含义：

- 封装 – 为测试带来了麻烦
- 继承 – 为测试提出了额外的挑战，每个新的使用环境也需要重新测试。

2. 传统测试用例设计方法的可用性:

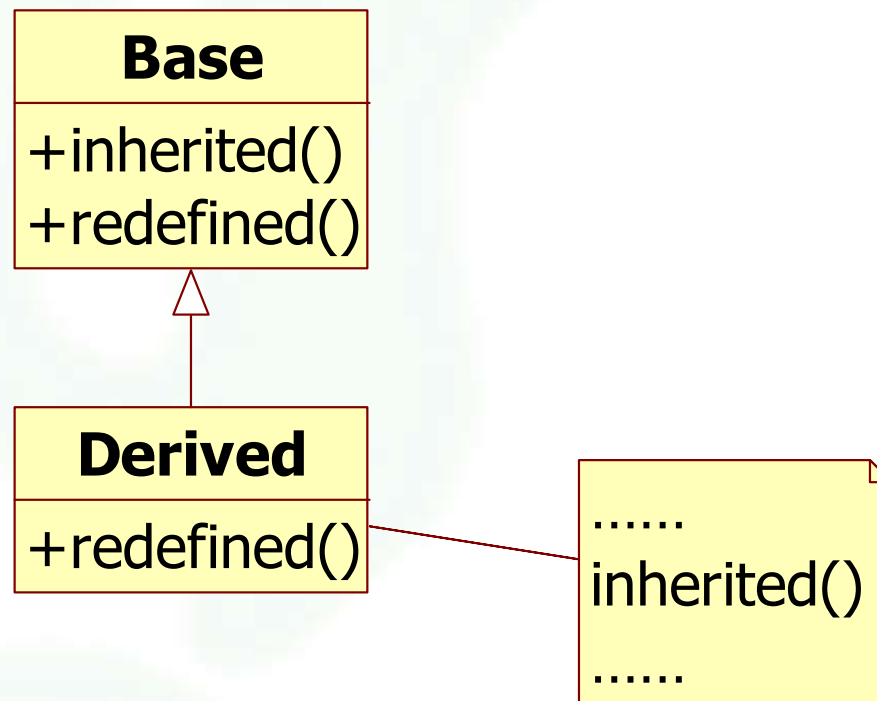
- 前面所述**白盒测试方法**可以用于类中定义的操作
- **黑盒测试**也适用于面向对象系统
 - 用例可以作为黑盒测试和基于状态的测试设计提供有用的输入

3. 基于故障的测试

- 基于故障的**目标**是设计最有可能发现似乎可能故障的测试
- 依赖于测试人员**如何感知**似然故障
 - 若分析和设计模型可以洞察有可能出错的事务，则基于故障的测试可以花费少时间而发现大量的错误。
- **集成测试**寻找操作调用或信息连接时的似然错误
 - 可以发现：非预期的错误、错误的操作/消息使用以及不正确的调用

4. 测试用例与类层次

- 继承并不能排除对所有派生类进行全面测试的需要



Derived::inherited需要被重新测试吗？

需要测试在子类中会被执行或调用的部分代码。

5. 基于场景的测试

- 基于场景的测试关心用户做什么，而不是产品做什么
- 场景可以发现交互错误
- 倾向于用单一测试检查多个子系统

5. 基于场景的测试举例

- 打印“最终”草稿

1. 打印整个文档
2. 在文档中移动，修改某些页
3. 当修改各页时，打印它
4. 有时打印多页。

- 打印一个新副本

1. 打开文档
2. 打印文档
3. 关闭文档

两个之间有联系吗？

现代编辑器中，文档记住最后一次打印时的情况！！！！

5. 基于场景的测试举例

- 用例：打印一个新副本

1. 打开文档

2. 在菜单中选择“打印”

3. 检查是否将连续打印若干页，如果是，点击
以打印整个文档

4. 按“打印”按钮

5. 关闭文档

15.8 类级可应用的测试方法

- OO中，“小型”测试侧重于单个类及该类封装的方法
- 随机测试和分割是可用于检查类的方法。

15.8.1 面向对象的随机测试

Account

+open()
+setup()
+deposit()
+withdraw()
+balance()
+summarize()
+creditLimit()
+close()

即使有一些限制，仍存在很多操作排列：
最小序列：

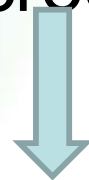
open.setup.deposit.withdraw.close

可能序列：

open.setup.deposit.

[deposit|withdraw|balance|

summarize|creditLimit]ⁿ.withdraw.close



测试用例r1: open.setup.deposit.deposit.
balance.summarize.withdraw.close

测试用例r2: open.setup.deposit.withdraw.deposit.
balance.creditLimit.withdraw.close

15.8.2 类级的划分测试

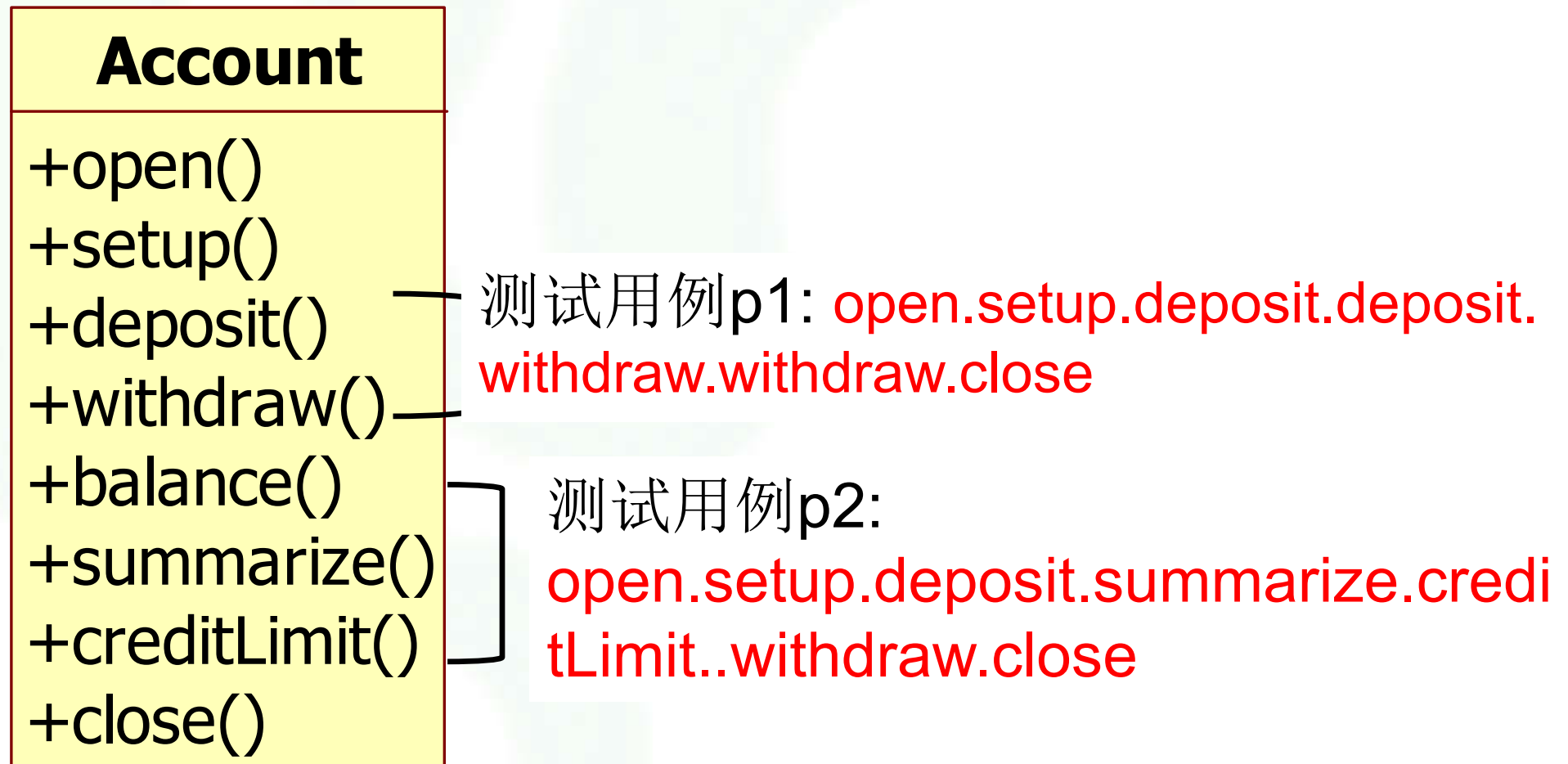
- **划分测试**减小测试特定类所需的测试用例数量。**对输入和输出进行分类**，设计测试用例以检查每个类。

15.8.2 类级的划分测试（续）

- 划分类如何得到？
 - 基于状态划分
 - 基于属性划分
 - 基于类别划分

1. 基于状态划分

- 根据他们改变类状态的能力对类操作进行分类



2. 基于属性划分

- 根据所使用的属性进行划分类操作
 - Account具有属性balance和creditLimit。可将操作划分成三类：
 - 使用creditLimit的类
 - 修改creditLimit的类
 - 既不使用也不修改creditLimit的类

3.基于类别划分

- 根据每个操作所完成的一半功能进行划分
类操作

- Account中:

- 初始化操作: `open()`, `setup()`
- 计算操作: `deposit()`, `withdraw()`
- 查询操作: `balance()`, `summarize()`, `creditLimit()`
- 中止操作: `close()`

15.9 类间测试用例设计

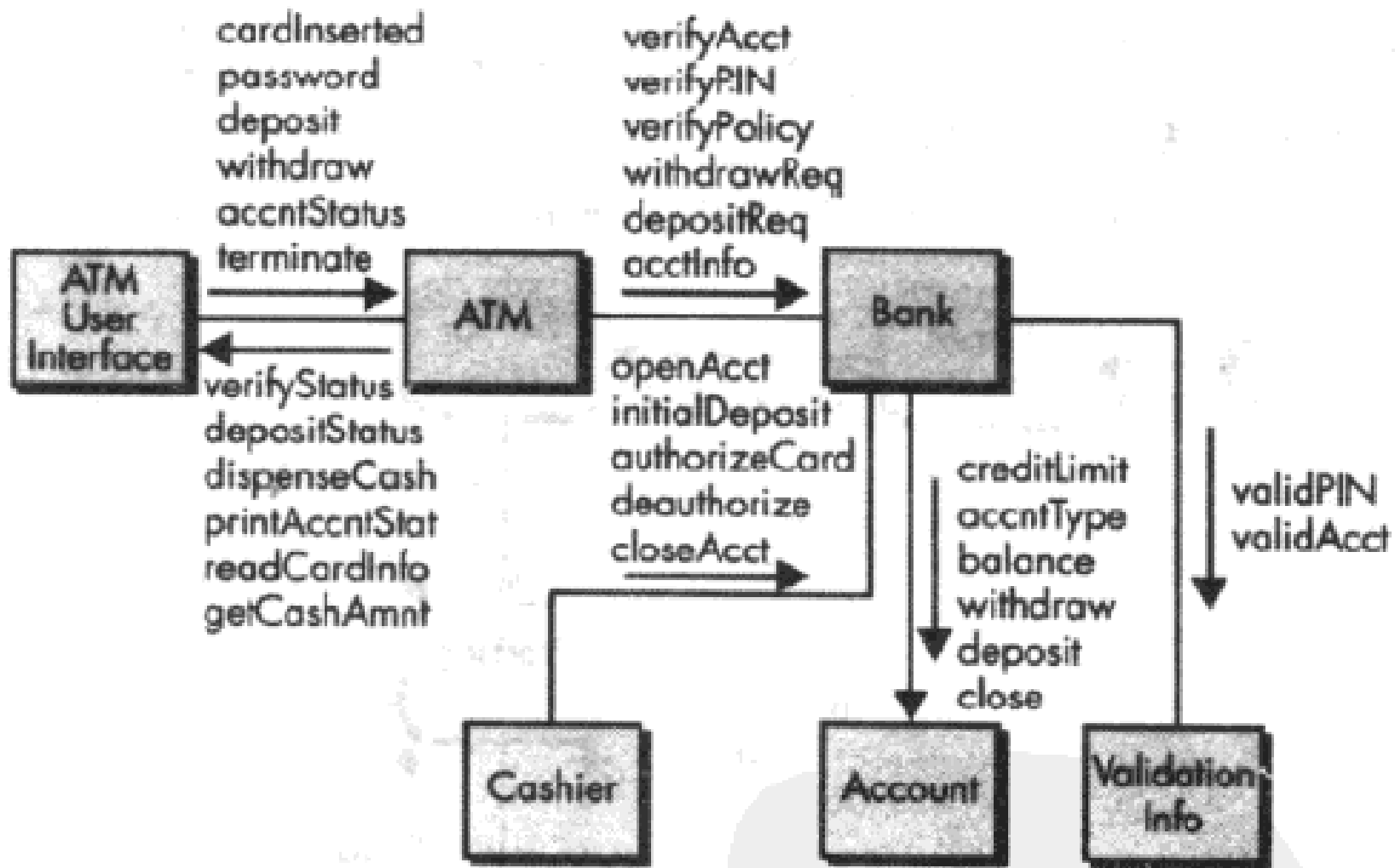


图 银行应用的类协作图 (摘自[KIR94])

1. 多类测试

- 生成多类测试用例的方法：
 1. 对每个用户类，使用类操作列表来生成一系列的随即测试序列。这些操作将向其他服务类发送消息
 2. 对每个生成的消息，确定协作类和服务对象中的相应操作
 3. 对服务对象中的每个操作（已被用户对象发送的消息调用），确定它所发送的消息
 4. 对每个消息，确定下一层被调用的操作并将其引入到测试序列中。

1. 多类测试

- 考虑类Bank和类ATM的操作序列

`verifyAcct • verifyPIN • [[verifyPolicy • withdrawReq] | depositReq | acctInfoREQ]n`

类Bank的一个随机测试序列:

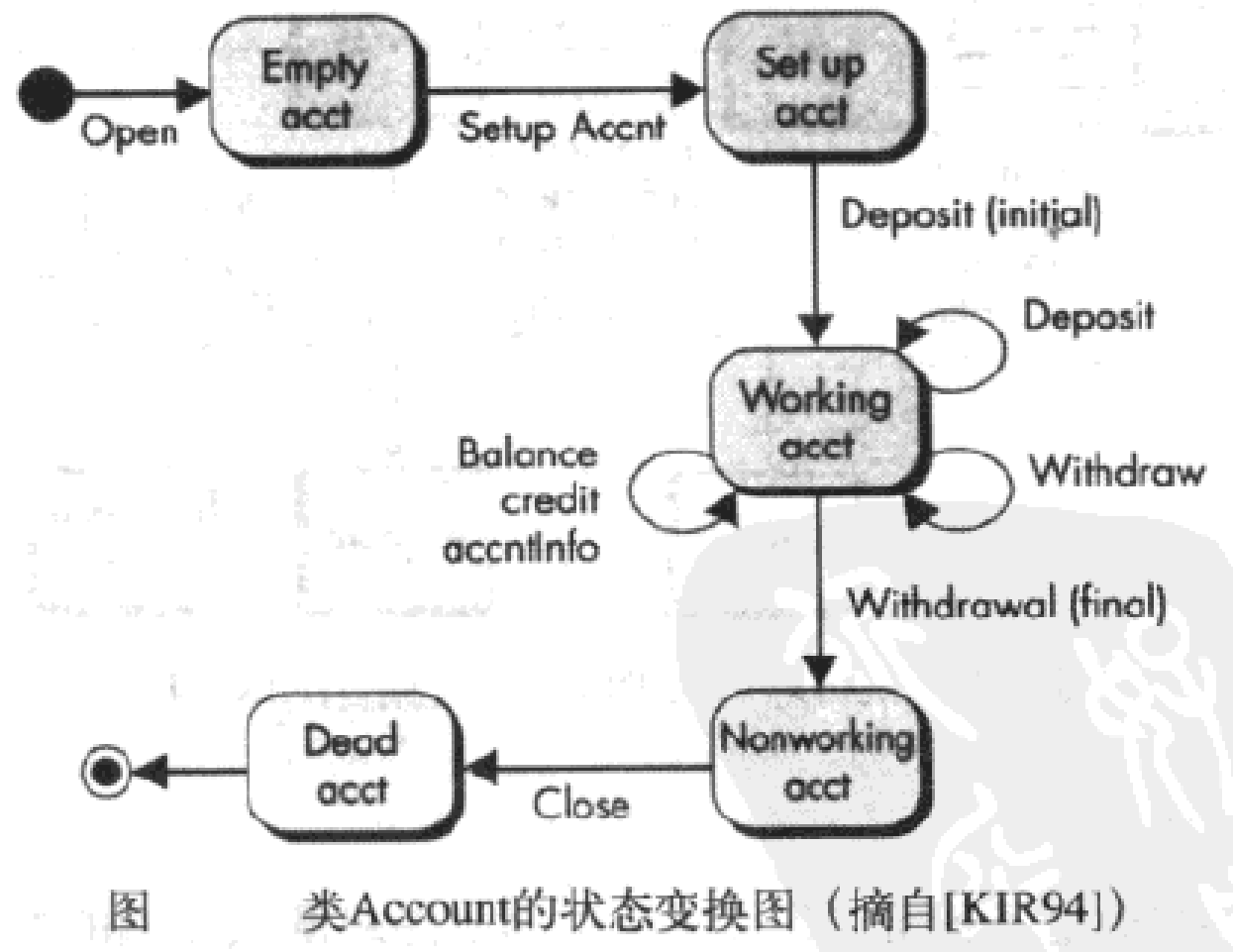
测试用例 $r_3 = \text{verifyAcct} \cdot \text{verifyPIN} \cdot \text{depositReq}$

考虑与测试用例 r_3 中提到的操作相关的信息

测试用例 $r_4 = \text{verifyAcctBank}[\text{validAcctValidationInfo}] \cdot \text{verifyPINBank} \cdot [\text{validPinValidationInfo}] \cdot \text{depositReq} \cdot [\text{depositaccount}]$

2.从行为模型导出的测试

- 类的状态图
可以辅助生成检查类
(以及与该类的协作类)
的动态行为
的测试序列



2. 从行为模型导出的测试

- 设计的测试应该覆盖所有的状态

测试用例 S_1 : open • setupAcct • deposit (initial) • withdraw (final) • close

测试用例 S_2 : open • setupAcct • deposit(initial) • deposit • balance • credit • withdraw (final) • close

测试用例 S_3 : open • setupAcct • deposit(initial) • deposit • withdraw • acctInfo • withdraw (final) • close

举例

- 测试计划模板
- Web软件测试计划
- 测试分析计划模板