北京交通大学
BEIJING JIAOTONG UNIVERSITY

第三章 动态测试技术

hhli@bjtu.edu.cn

2020年10月

第三章 动态测试技术

3.1 黑盒测试技术

3.2 白盒测试技术

目录

3.2.1 白盒测试方法

3.2.2 程序结构分析

3.2.3 逻辑覆盖测试法

3.2.4 基本路径测试法

3.2.5 最少测试用例数计算

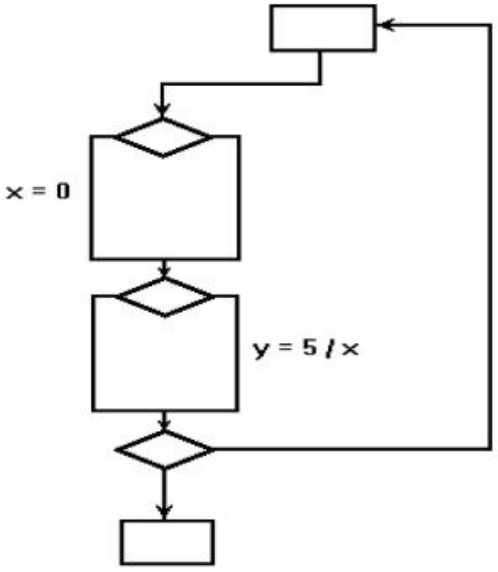
3.2.6 其它白盒测试方法简介

3.2.7 白盒测试方法选择的策略

3.2.1 白盒测试方法

- 为什么要进行白盒测试？

如果所有软件错误的根源都可以追溯到某个唯一原因，那么问题就简单了。然而，事实上一个 bug 常常是由多个因素共同导致的，如下图所示。



假设此时开发工作已结束，程序送交到测试组，没有人知道代码中有一个潜在的被 0 除的错误。若测试组采用的测试用例的执行路径没有同时经过x=0和y=5/x进行测试，显然测试工作似乎非常完善，测试用例覆盖了所有执行语句，也没有被 0 除的错误发生。

白盒测试方法（续）

- 白盒测试也称结构测试或逻辑驱动测试，是针对被测单元内部是如何进行工作的测试。它根据程序的控制结构设计测试用例，主要用于软件或程序验证。
- 白盒测试法检查程序内部逻辑结构，对所有逻辑路径进行测试，是一种穷举路径的测试方法。但即使每条路径都测试过了，仍然可能存在错误。
- 白盒测试的主要特点：主要针对被测程序的源代码，测试者可以完全不考虑程序的功能。
- 白盒测试的测试方法有代码检查法、静态结构分析法、静态质量度量法、逻辑覆盖法、基本路径测试法、域测试、符号测试、Z路径覆盖、程序变异以及程序控制流分析、数据流分析等。其中前三种属于静态测试。

白盒测试方法（续）

- 采用白盒测试方法必须遵循以下几条原则，才能达到测试的目的：
  ① 保证一个模块中的所有独立路径至少被测试一次。
  ② 所有逻辑值均需测试真（true）和假（false）两种情况。
  ③ 检查程序的内部数据结构，保证其结构的有效性。
  ④ 在上下边界及可操作范围内运行所有循环。
- 白盒测试主要是检查程序的*内部结构、逻辑、循环和路径*。
- 常用白盒测试用例设计方法有：
  - 逻辑覆盖法（逻辑驱动测试）
  - 基本路径测试方法

3.2.2 程序结构分析

1 控制流图

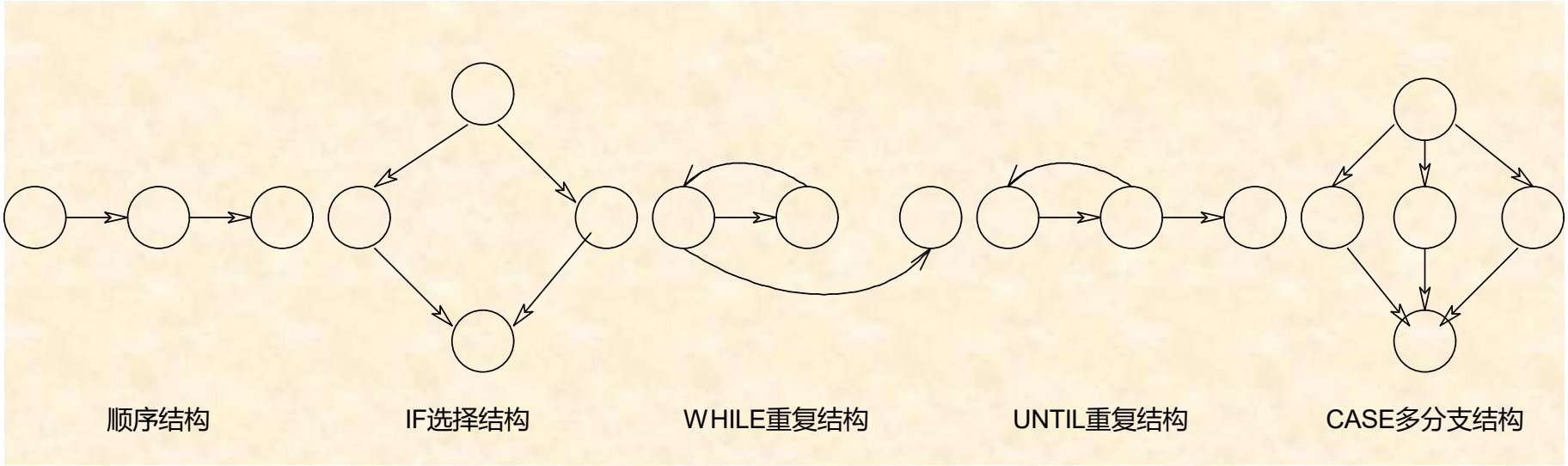
2 环形复杂度

3、图矩阵

1、 控制流图

- 控制流图（可简称流图）是对程序流程图进行简化后得到的，它可以更加突出的表示程序控制流的结构。
- 控制流图中包括两种图形符号：节点和控制流线。
  - 节点由带标号的圆圈表示，可代表一个或多个语句、一个处理框序列和一个条件判定框（假设不包含复合条件）。
  - 控制流线由带箭头的弧或线表示，可称为边。它代表程序中的控制流。
  - 对于复合条件，则可将其分解为多个单个条件，并映射成控制流图。

常见结构的控制流图



其中，包含条件的节点被称为判定节点（也叫谓词节点），由判定节点发出的边必须终止于某一个节点，由边和节点所限定的范围被称为区域。

2 环形复杂度

- 环形复杂度也称为圈复杂度，它是一种为程序逻辑复杂度提供定量尺度的软件度量。
- 环形复杂度的应用——可以将环形复杂度用于基本路径方法，它可以提供：程序基本集的独立路径数量，确保所有语句至少执行一次的测试数量的上界。
- 独立路径是指程序中至少引入了一个新的处理语句集合或一个新条件的程序通路。采用流图的术语，即独立路径必须至少包含一条在本次定义路径之前不曾用过的边。
- 测试可以被设计为基本路径集的执行过程，但基本路径集通常并不唯一。

计算环形复杂度的方法

- 环形复杂度以图论为基础，为我们提供了非常有用的软件度量。可用如下三种方法之一来计算环形复杂度：

1）控制流图中区域的数量对应于环形复杂度。

2）给定控制流图G的环形复杂度—V(G)，定义为

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

其中，E是控制流图中边的数量，N是控制流图中的节点数量。

3）给定控制流图G的环形复杂度—V(G)，也可定义为

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

其中，P是控制流图G中判定节点的数量。

3 图矩阵

- 图矩阵是控制流图的矩阵表示形式。
- 图矩阵是一个方形矩阵，其维数等于控制流图的节点数。矩阵中的每列和每行都对应于标识的节点，矩阵元素对应于节点间的边。
- 通常，控制流图中的结点用数字标识，边则用字母标识。如果在控制流图中从第 i 个结点到第 j 个结点有一个标识为 x 的边相连接，则在对应图矩阵的第 i 行第 j 列有一个非空的元素 x 。

示例

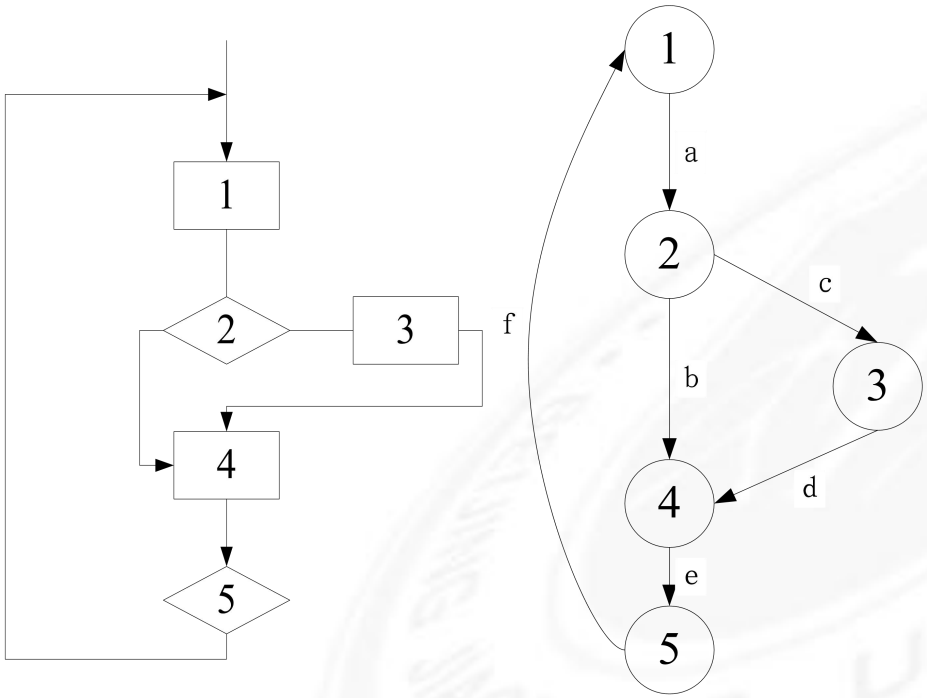


图4-2-2

- 在控制流图中只有两种图形符号，它们是：

1. 节点：以标有编号的圆圈表示。它代表了程序流程图中矩形框表示的处理、菱形表示的两个到多个出口判断以及两条到多条流线相交的汇合点。
2. 控制流线或弧：以箭头表示。它与程序流程图中的流线是一致的，表明了控制的顺序。为了方便讨论，控制流线通常标有名字，如图中所标的a、b、c等。

示例（续）

	a			
		c	b	
			d	
				e
f				

图4-2-3

- 为了方便控制流图在机器上表示，可以将它表示成矩阵的形式，即控制流图矩阵。图4-2-3表示了图4-2-2的控制流图矩阵，这个矩阵有5行5列，是由该控制图中5个节点决定的。矩阵中6个元素a、b、c、d、e和f的位置决定了它们所连接节点的号码。
- 例如，弧d在矩阵中处于第3行第4列，那是因为它在控制流图中连接了节点3至节点4。这里必须注意方向，图中节点4到节点3没有弧，所以矩阵中第4行第3列也就没有元素。

3.2.3 覆盖测试

1 测试覆盖率

2 逻辑覆盖法

3 面向对象的覆盖

4 测试覆盖准则

1、 测试覆盖率

- 测试覆盖率：用于确定测试所执行到的覆盖项的百分比。其中的覆盖项是指作为测试基础的一个入口或属性，比如语句、分支、条件等。
- 测试覆盖率可以表示出测试的充分性，在测试分析报告中可以作为量化指标的依据，测试覆盖率越高效果越好。但覆盖率不是目标，只是一种手段。

1、测试覆盖率

- 测试覆盖率通常分为需求覆盖率和逻辑覆盖率(结构覆盖率)。

➢ 需求覆盖率：用于表示软件已经测试验证的需求数量与软件需要实现的需求之间的比例关系。其含义是通过设计一定的测试用例，要求每个需求点都被测试到。公式如下：

需求覆盖率 ＝被验证到的需求数量 /总的需求数量。

说明：需求不仅仅是指功能需求，还要包括性能需求。衡量需求覆盖率的最直观的方式是有多少功能点、多少性能点要求？它们是分母；写了多少测试用例，覆盖了多少模块，多少功能点，性能测试用例考虑了待测程序多少性能点等等，这些作为分子。

1、测试覆盖率

- 结构（逻辑）覆盖率：是指针对代码中某个对象所做的测试的占代码中该对象总数的比例。具体包括：**语句覆盖率、分支覆盖率、循环覆盖率、路径覆盖率**等等。
- 如，语句覆盖率，是监视每行代码是否在用例中被执行到，或者说测试用例里面大概执行了百分之多少的语句/代码行数。

2、 逻辑覆盖测试法

- 逻辑覆盖测试是依据被测程序的逻辑结构设计测试用例，驱动被测程序运行完成测试。其中的一个重要问题是测试进行到什么地步就达到要求？可以结束测试了？这就是说需要给出结构测试的覆盖准则。

2、 逻辑覆盖测试法

- 根据覆盖目标的不同，逻辑覆盖又可分为语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、组合覆盖和路径覆盖。

- 语句覆盖：选择足够多的测试用例，使得程序中的每个可执行语句至少执行一次。
- 判定覆盖：通过执行足够的测试用例，使得程序中的每个判定至少都获得一次“真”值和“假”值， 也就是使程序中的每个取“真”分支和取“假”分支至少均经历一次，也称为“分支覆盖”。
- 条件覆盖：设计足够多的测试用例，使得程序中每个判定包含的每个条件(表达式）的可能取值（真/假）都至少满足一次。

逻辑覆盖法（续）

- 判定/条件覆盖：设计足够多的测试用例，使得程序中每个判定包含的每个条件的所有情况（真/假）至少出现一次，并且每个判定本身的判定结果（真/假）也至少出现一次。

  ——满足**判定/条件**覆盖的测试用例集一定同时满足判定覆盖和条件覆盖。

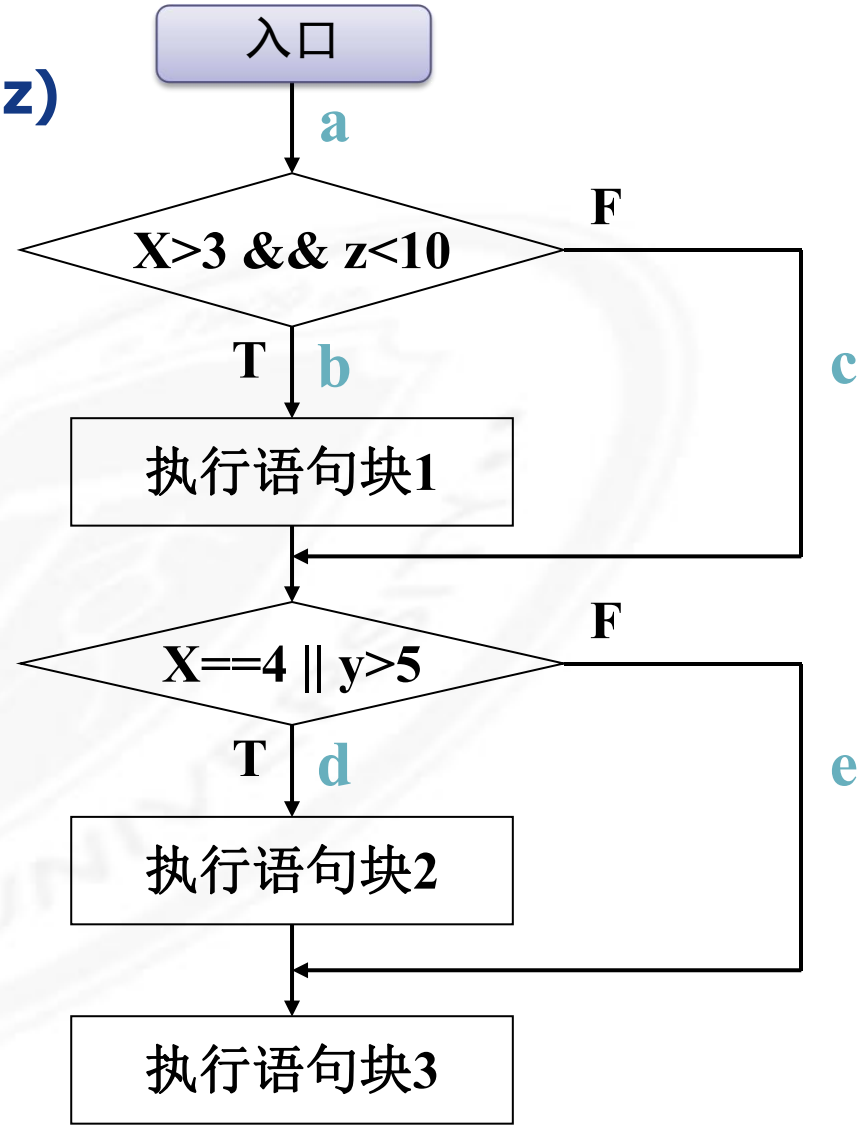
- 组合覆盖：通过执行足够的测试用例，使得程序中每个判定的所有可能的条件取值组合都至少出现一次。

  ——满足组合覆盖的测试用例一定满足判定覆盖、条件覆盖和判定/条件覆盖。

- 路径覆盖：设计足够多的测试用例，要求覆盖程序中所有可能的路径。

应用举例

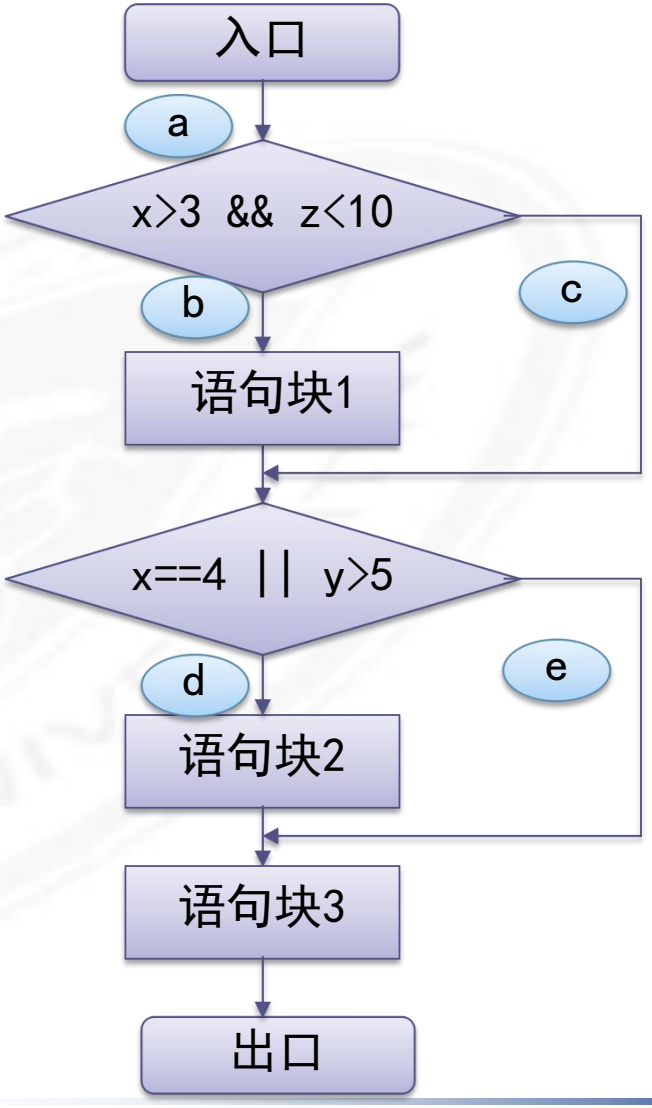
```
void  DoWork(int x,int y,int z)
{
  int  k=0,j=0;
  if(( x>3 ) && ( z<10 ))
  {
     k=x*y-1;     //语句块1
     j=sqrt(k);
  }
  if(( x= =4 ) || ( y>5 ))
  {
     j=x*y+10;   //语句块2
  }
  j=j%3;              //语句块3
}
```



语句覆盖

- 测试用例输入为：
  - { x=4、y=5、z=5}

程序执行的路径是：abd



语句覆盖

- 要实现DoWork函数的语句覆盖，只需设计一个测试用例就可以覆盖程序中的所有可执行语句。
- 测试用例输入为：{ x=4、y=5、z=5 }
- 程序执行的路径是：abd
- 分析：

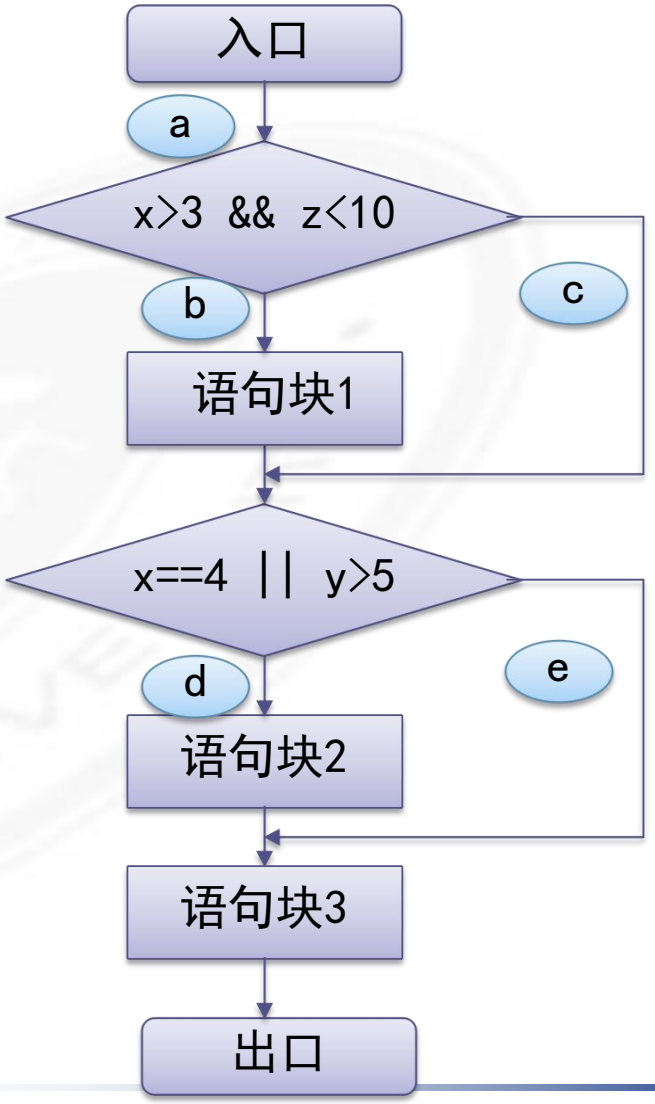
语句覆盖可以保证程序中的每个语句都得到执行，但发现不了判定中逻辑运算的错误，即它并不是一种充分的检验方法。例如在第一个判定((x>3)&&(z<10))中把“&&”错误的写成了“||”，这时仍使用该测试用例，则程序仍会按照流程图上的路径abd执行。可以说语句覆盖是最弱的逻辑覆盖准则。

判定（分支）覆盖

- 测试用例的输入为：
  - { x=4、y=5、z=5}
  - { x=2、y=5、z=5}

程序执行的路径是：abd

程序执行的路径是：ace



判定覆盖

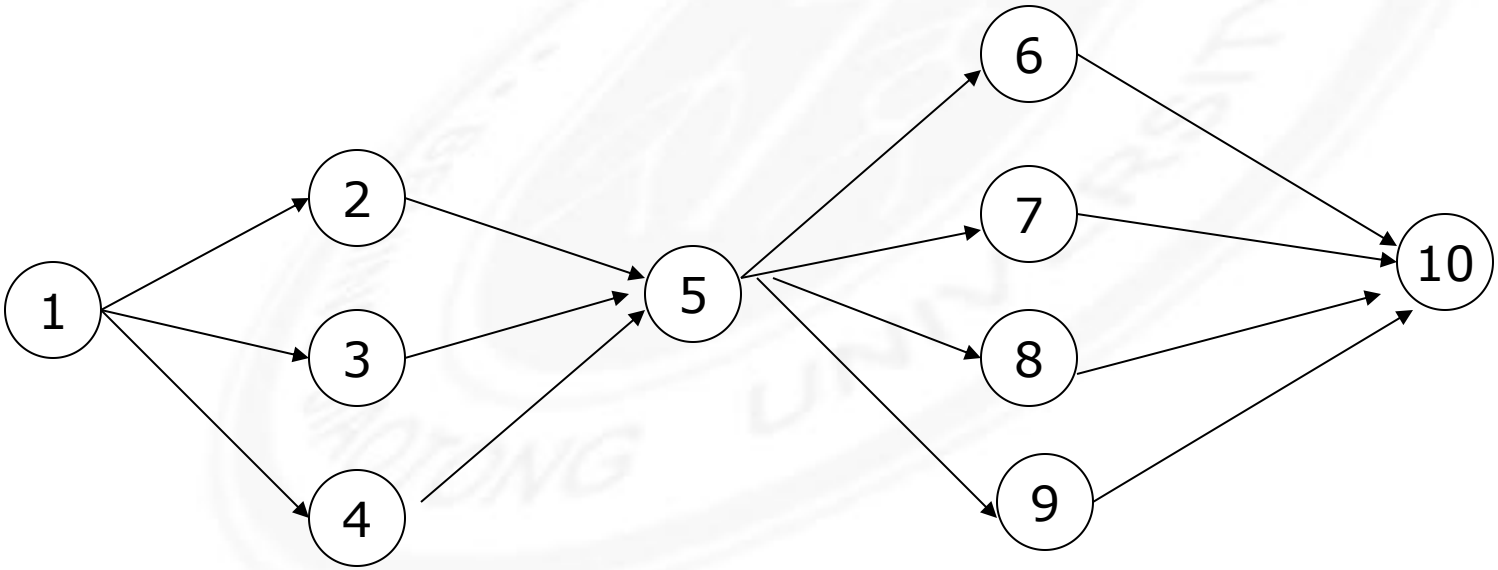
- 要实现DoWork函数的判定覆盖，需要设计两个测试用例。
- 测试用例的输入为：{x=4、y=5、z=5}；{x=2、y=5、z=5}
- 程序执行的路径分别是：abd；ace
- 分析：

  述两个测试用例不仅满足了判定覆盖，同时还做到语句覆盖。从这点看似乎判定覆盖比语句覆盖更强一些，但仍然无法确定判定的内部条件的错误。

  - 例如把第二个判定中的条件y>5错误写为y<5，使用上述测试用例，照样能按原路径执行而不影响结果。因此，需要有更强的逻辑覆盖准则去检验判定内的条件。

判定覆盖（续）

- 说明：以上仅考虑了两出口的判断，我们还应把判定覆盖准则扩充到多出口判断（如Case语句）的情况。因此，判定覆盖更为广泛的含义应该是使得每一个判定获得每一种可能的结果至少一次。



条件覆盖

- 在实际程序代码中，一个判定中通常都包含若干条件。条件覆盖的目的是设计若干测试用例，在执行被测程序，要使每个判定中每个条件的可能值至少满足一次。
- 对DoWork函数的各个判定的各种条件取值加以标记。

➢ 对于第一个判定（ (x>3)&&(z<10) ）：

条件x>3　　取真值记为T1，取假值记为-T1

条件z<10　取真值记为T2，取假值记为-T2

➢ 对于第二个判定（ (x==4)||(y>5) ）：

条件x==4　取真值记为T3，取假值记为-T3

条件y>5　　取真值记为T4，取假值记为-T4

条件覆盖（续）

- 根据条件覆盖的基本思想，要使上述4个条件可能产生的8种情况至少满足一次，设计测试用例如下：

测试用例	执行路径	覆盖条件	覆盖分支
x=4、y=6、z=5	abd	T1、T2、T3、T4	bd
x=2、y=5、 z=15	ace	-T1、-T2、-T3、-T4	ce

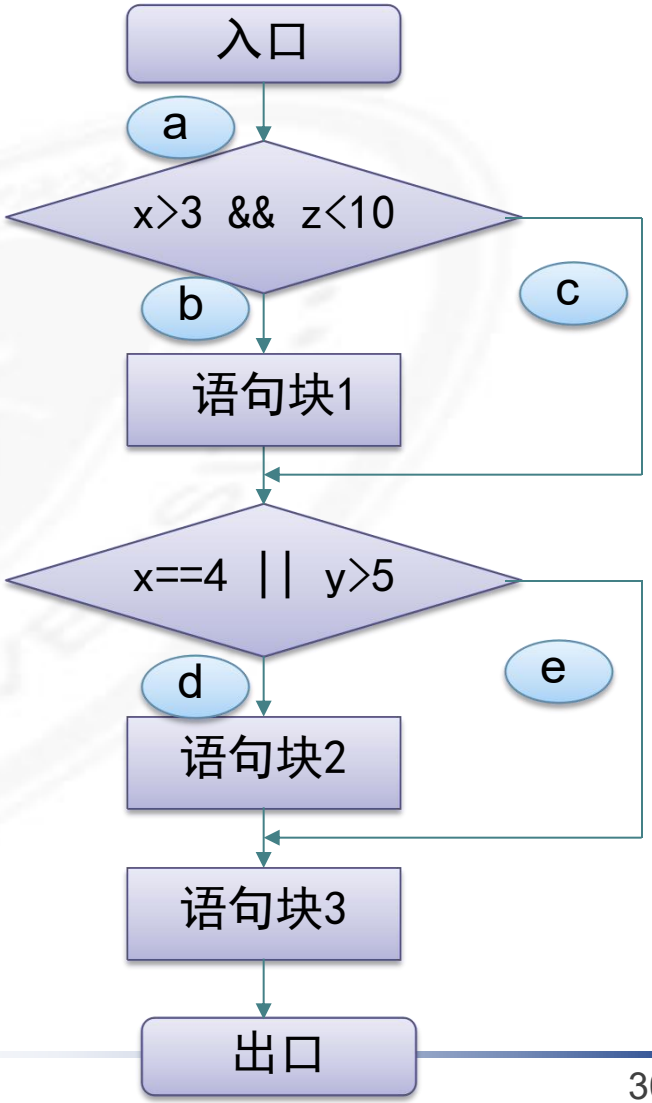
- 分析：上面这组测试用例不但覆盖了4个条件的全部8种情况，而且将两个判定的4个分支b、c、d、e也同时覆盖了，即同时达到了条件覆盖和判定覆盖。

条件覆盖

- 测试用例的输入为：
  - { x=4、y=6、z=5}
  - { x=2、y=5、z=15}

程序执行的路径是：abd

程序执行的路径是：ace



条件覆盖（续）

- 说明：虽然前面的一组测试用例同时达到了条件覆盖和判定覆盖，但是，并不是说满足条件覆盖就一定能满足判定覆盖。如果设计了下表中的这组测试用例，则虽然满足了条件覆盖，但只是覆盖了程序中第一个判定的取假分支c和第二个判定的取真分支d，不满足判定覆盖的要求。

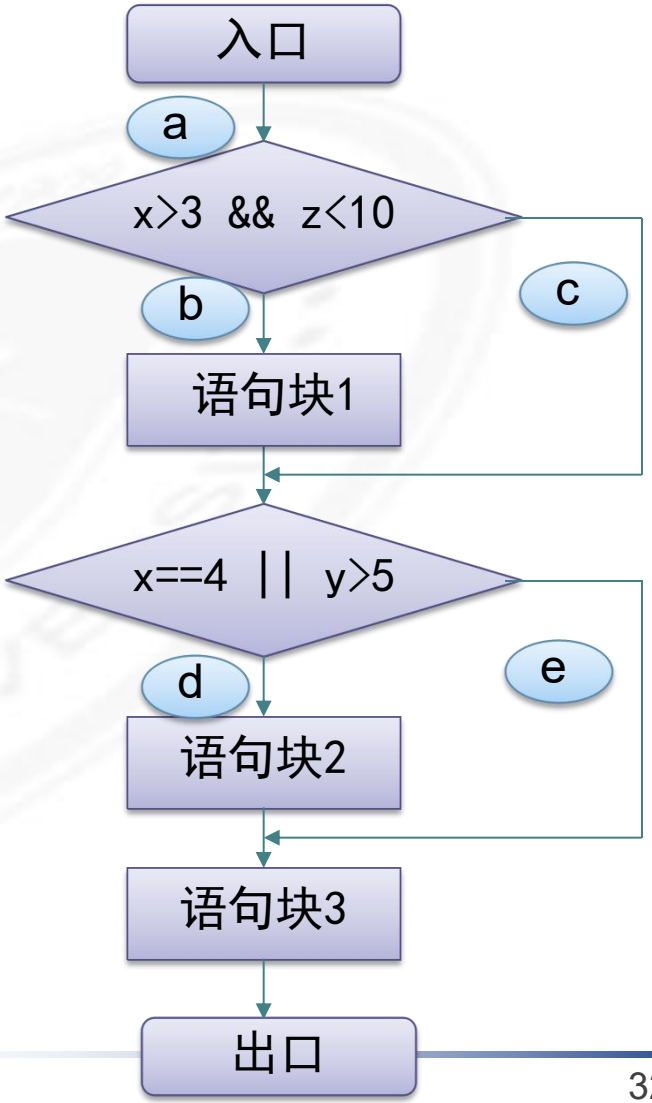
测试用例	执行路径	覆盖条件	覆盖分支
x=2、y=6、z=5	acd	-T1、T2、-T3、T4	cd
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	cd

条件覆盖

- 测试用例的输入为：
  - { x=2、y=6、z=5}
  - { x=4、y=5、z=15}

程序执行的路径是：acd

程序执行的路径是：acd



判定/条件覆盖

- 判定/条件覆盖实际上是将判定覆盖和条件覆盖结合起来的一种方法，即：设计足够的测试用例，使得判定中每个条件的所有可能取值至少满足一次，同时每个判定的可能结果也至少出现一次。
- 根据判定/条件覆盖的基本思想，只需设计以下两个测试用例便可以覆盖4个条件的8种取值以及4个判定分支。

测试用例	执行路径	覆盖条件	覆盖分支
x=4、y=6、z=5	abd	T1、T2、T3、T4	bd
x=2、y=5、z=15	ace	-T1、-T2、-T3、-T4	ce

判定/条件覆盖（续）

- 分析：从表面上看，判定/条件覆盖测试了各个判定中的所有条件的取值，但实际上，编译器在检查含有多个条件的逻辑表达式时，某些情况下的某些条件将会被其它条件所掩盖。因此，判定/条件覆盖也不一定能够完全检查出逻辑表达式中的错误。

- 例如：对于第一个判定(x>3)&&(z<10)来说，必须x>3和z<10这两个条件同时满足才能确定该判定为真。如果x>3为假，则编译器将不再检查z<10这个条件，那么即使这个条件有错也无法被发现。对于第二个判定(x==4)||(y>5)来说，若条件x==4满足，就认为该判定为真，这时将不会再检查y>5，那么同样也无法发现这个条件中的错误。

组合覆盖

- 组合覆盖的目的是要使设计的测试用例能覆盖每一个判定的所有可能的条件取值组合。
- 对DoWork函数中的各个判定的条件取值组合加以标记：

1、x>3, z<10　　记做T1 T2，第一个判定的取真分支

2、x>3, z>=10　　记做T1 -T2，第一个判定的取假分支

3、x<=3, z<10　　记做-T1 T2，第一个判定的取假分支

4、x<=3, z>=10　　记做-T1 -T2，第一个判定的取假分支

5、x==4, y>5　　记做T3 T4，第二个判定的取真分支

6、x==4, y<=5　　记做T3 -T4，第二个判定的取真分支

7、x!=4, y>5　　记做-T3 T4，第二个判定的取真分支

8、x!=4, y<=5　　记做-T3 -T4，第二个判定的取假分支

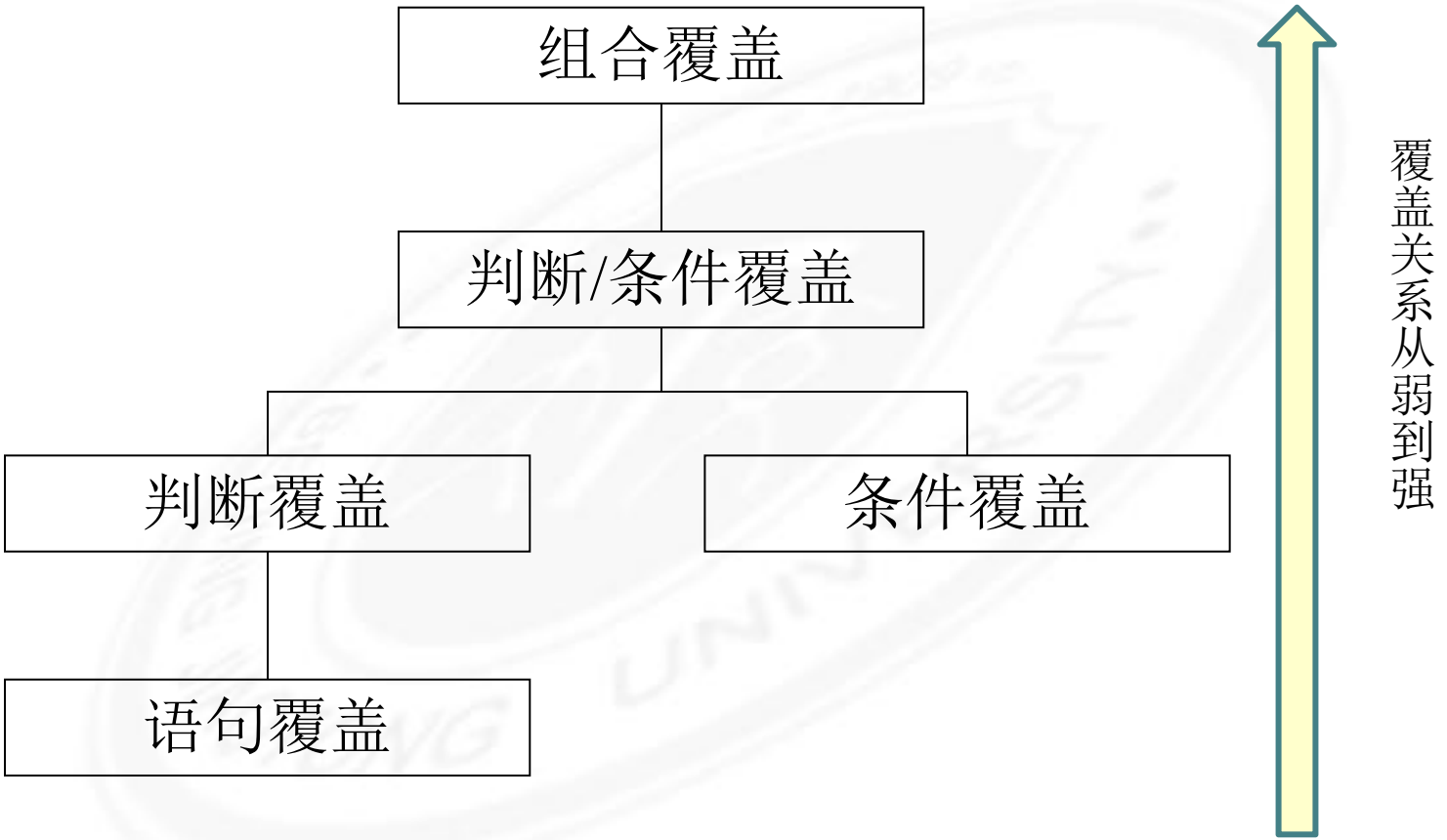
组合覆盖（续）

- 根据组合覆盖的基本思想，设计测试用例如下：

测试用例	执行路径	覆盖条件	覆盖组合号
x=4、y=6、z=5	abd	T1、T2、T3、T4	1和5
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	2和6
x=2、y=6、z=5	acd	-T1、T2、-T3、T4	3和7
x=2、y=5、z=15	ace	-T1、-T2、-T3、-T4	4和8

- 分析：上面这组测试用例覆盖了所有8种条件取值的组合，覆盖了所有判定的真假分支，但是却丢失了一条路径abe。

逻辑覆盖法



路径覆盖

- 前面提到的5种逻辑覆盖都未涉及到路径的覆盖。事实上，只有当程序中的每一条路径都受到了检验，才能使程序受到全面检验。路径覆盖的目的就是要使设计的测试用例能覆盖被测程序中所有可能的路径。
- 根据路径覆盖的基本思想，在前述满足组合覆盖的测试用例中修改其中一个测试用例，则可以实现路径覆盖：

测试用例	执行路径	覆盖条件
x=4、y=6、z=5	abd	T1、T2、T3、T4
x=4、y=5、z=15	acd	T1、-T2、T3、-T4
x=2、y=5、z=15	ace	-T1、-T2、-T3、-T4
x=5、y=5、z=5	abe	T1、T2、-T3、-T4

路径覆盖（续）

- 分析：虽然前面一组测试用例满足了路径覆盖，但并没有覆盖程序中所有的条件组合（丢失了组合3和7），即满足路径覆盖的测试用例并不一定满足组合覆盖。
- 说明：

① 对于比较简单的小程序，实现路径覆盖是可能做到的。但如果程序中出现较多判断和较多循环，可能的路径数目将会急剧增长，要在测试中覆盖所有的路径是无法实现的。为了解决这个难题，只有把覆盖路径数量压缩到一定的限度内，如程序中的循环体只执行一次。

② 在实际测试中，即使对于路径数很有限的程序已经做到路径覆盖，仍然不能保证被测试程序的正确性，还需要采用其它测试方法进行补充。

3 面向对象的覆盖*

继承上下文覆盖

- 由于传统的结构化度量没有考虑面向对象的一些特性（如多态、继承和封装等），所以在面向对象领域，传统的结构化覆盖必须被加强，以满足面向对象特性。
- 继承上下文覆盖考虑在每个类的上下文内获得的覆盖率级别。它是扩展到面向对象领域里的一种覆盖率度量方法，用于度量在系统中的多态调用被测试的程度。
- 继承上下文定义将基类上下文内例行程序的执行作为独立于继承类上下文内例行程序的执行。同样，它们在考虑继承类上下文内例行程序的执行，也独立于基类上下文内例行程序的执行。为了获得100%继承上下文覆盖，代码必须在每个适当的上下文内被完全执行。

面向对象的覆盖（续）

- 基于状态的上下文覆盖

① 在绝大多数面向对象的系统中存在这样的一些类：这些类的对象可以存在于众多不同状态中的任何一种，并且由于类的行为依赖于状态，每个类的行为在每个可能的状态中其性质是不同的。

② 基于状态的上下文覆盖对应于被测类对象的潜在状态。

③ 这样基于状态的上下文覆盖把一个状态上下文内的一个例行程序的执行认为是独立于另一个状态内相同例行程序的执行。为了达到100%的基于状态的上下文覆盖，例行程序必须在每个适当的上下文（状态）内被执行。

4 测试覆盖准则

- 逻辑覆盖的出发点是合理的、完善的。所谓“覆盖”，就是想要做到全面而无遗漏，但逻辑覆盖并不能真正做到无遗漏。
- 例如：我们不小心将前面提到的程序段中的

  if (x>3 && Z<10) { ..... }

  错写成 if (x>=3 && Z<10) { ..... }

  按照我们前面设计的测试用例（x的值取2或4）来看，逻辑覆盖对这样的小问题都无能为力。

  ✓ 分析：出现这一情况的原因在于错误区域仅仅在x=3这个点上，即仅当x的值取3时，测试才能发现错误。面对这类情况，我们应该从中吸取的教训是测试工作要有重点，要多针对容易发生问题的地方设计测试用例。

测试覆盖准则（续）

- 测试覆盖准则主要讨论ESTCA（Error Sensitive Test Cases Analysis，错误敏感测试用例分析）和LCSAJ（Linear Code Sequence And Jump，线性代码序列与跳转）。
- Foster提出了ESTCA覆盖准则：在容易发生问题的地方设计测试用例，即重视程序中谓词（条件判断）的取值。
- ESTCA覆盖准则是一套错误敏感用例分析规则。这一规则虽然并不完备，但在普通程序中却是有效的。原因在于这是一种经验型的覆盖准则，规则本身针对了程序编写人员容易发生的错误，或是围绕着发生错误的频繁区域，从而提高了发现错误的命中率。

测试覆盖准则（续）

- ESTCA具体规则如下：

➢ [规则1] 对于A rel B型（rel可以是<、= 或 >）的分支谓词，应适当的选择A与B的值，使得测试执行到该分支语句时，A<B、A=B、A>B的情况分别出现一次。

——这是为了检测逻辑符号写错的情况，如将“A<B”错写为“A>B”。

测试覆盖准则（续）

- [规则2] 对于A rel C型 (rel可以是>或<，A是变量，C是常量)的分支谓词。(1)当rel为<时，应适当的选择A的值，使A=C-M （M是距C最小的机器允许正数，若A和C都为正整数时，M=1）；(2)当rel为>时，应适当的选择A的值，使A=C+M。

  ——这是为了检测“差1”之类的错误，如“A>1”错写成“A>0”。

- [规则3] 对外部输入变量赋值，使其在每一个测试用例中均有不同的值与符号，并与同一组测试用例中其它变量的值与符号不同。

  ——这是为了检测程序语句中的错误，如应该引用某一变量而错成引用另一个常量。

测试覆盖准则（续）

- Woodward等人提出了层次LCSAJ覆盖准则。

- LCSAJ是指一组顺序执行的代码，以控制跳转为其结束点。
- LCSAJ的起点是根据程序本身决定的。它的起点可以是程序第一行或转移语句的入口点，或是控制流可跳达的点。
- 如果有几个LCSAJ首尾相接，且第一个LCSAJ起点为程序起点，最后一个LCSAJ终点为程序终点，这样的LCSAJ串就组成了程序的一条路径称为LCSAJ路径。一条LCSAJ程序路径可能是由2个、3个或多个LCSAJ组成的。

测试覆盖准则（续）

- 层次LCSAJ覆盖准则：
- 基于LCSAJ与路径的关系，提出了层次LCSAJ覆盖准则。它是一个分层的覆盖准则，可以概括地描述为：

① 第一层 — 语句覆盖。

② 第二层 — 分支覆盖。

③ 第三层 — LCSAJ覆盖，即程序中的每一个LCSAJ都至少在测试中经历过一次。

④ 第四层 — 两两LCSAJ覆盖，即程序中的每两个相连的LCSAJ组合起来在测试中都要经历一次。

⑤ 第n+2层 — 每n个首尾相连的LCSAJ组合在测试中都要经历一次。

- 越是高层的覆盖准则越难满足。在实施测试时，若要实现上述的层次LCSAJ覆盖，需要产生被测程序的所有LCSAJ。

举例：

- 例：找出前面DoWork函数的所有LCSAJ（代码块）和LCSAJ路径。

➢ LCSAJ（5个）：

（1）int k=0, j=0;
　　if ((x>3)&&(z<10))
（2）k=x*y-1;　　j=sqrt(k);
　　if ((x==4)||(y>5))
（3）if ((x==4)||(y>5))
（4）j=x*y+10;　　j=j%3
（5）j=j%3

➢ LCSAJ路径（4条）：

（1）-（2）-（4）、（1）-（2）-（5）
（1）-（3）-（4）、（1）-（3）-（5）

```
void  DoWork (int x,int y,int z)
{
  int  k=0,j=0;
  if ( (x>3)&&(z<10) )
  {  k=x*y-1;
     j=sqrt(k);
  }                          //语句块1
  if ( (x==4)||(y>5) )
  { j=x*y+10; }              //语句块2
  j=j%3;                     //语句块3
}
```

3.2.4 路径测试

1 路径表达式

2 基本路径测试方法

3 循环测试方法

4 补充测试用例设计

1、路径表达式

- 为了满足路径覆盖，必须首先确定具体的路径以及路径的个数。我们通常采用控制流图的边（弧）序列和节点序列表示某一条具体路径，更为概括的表示方法为：

（1）弧a和弧b相乘，表示为ab，它表明路径是先经历弧a，接着再经历弧b，弧a和弧b是先后相接的。

（2）弧a和弧b相加，表示为a+b，它表明两条弧是“或”的关系，是并行的路段。

- 路径数的计算：

在路径表达式中，将所有弧均以数值1来代替，再进行表达式的相乘和相加运算，最后得到的数值即为该程序的路径数。

2、 基本路径测试方法

- 路径测试就是从一个程序的入口开始，执行所经历的各个语句的完整过程。从广义的角度讲，任何有关路径分析的测试都可以被称为路径测试。
- 完成路径测试的理想情况是做到路径覆盖，但对于复杂性大的程序要做到所有路径覆盖（测试所有可执行路径）是不可能的。
- 在不能做到所有路径覆盖的前提下，如果某一程序的每一条独立路径都被测试过，那么可以认为程序中的每个语句都已经检验过了，即达到了语句覆盖。这种测试方法就是通常所说的基本路径测试方法。

基本路径测试方法（续）

- 基本路径测试方法是在控制流图的基础上，通过分析控制结构的环形复杂度，导出执行路径的基本集，再从该基本集设计测试用例。
- 基本路径测试方法包括以下4个步骤：

（1）画出程序的控制流图。

（2）计算程序的环形复杂度，导出程序基本路径集中的独立路径条数，这是确定程序中每个可执行语句至少执行一次所必须的测试用例数目的上界。

（3）导出基本路径集，确定程序的独立路径。

（4）根据（3）中的独立路径，设计测试用例的输入数据和预期输出。

基本路径测试方法（续）

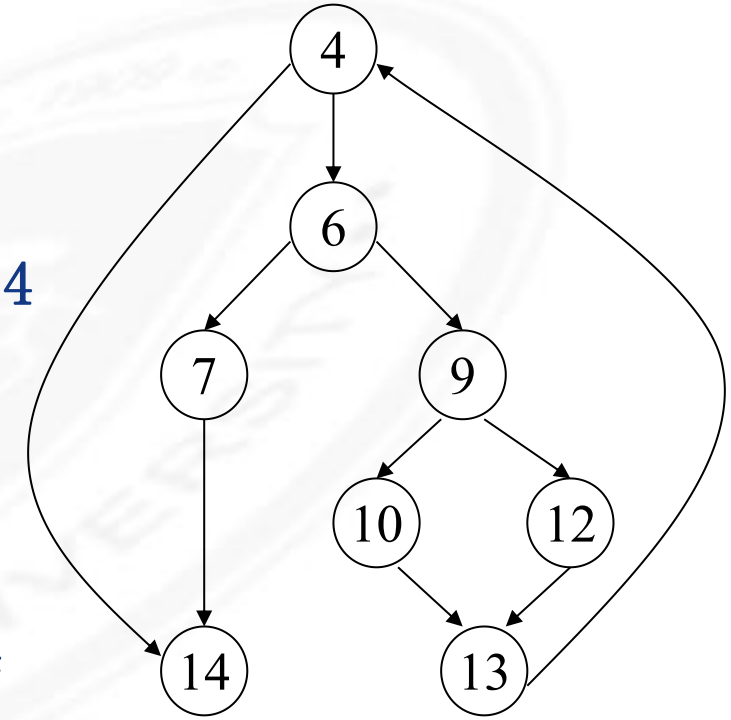
例：

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

基本路径测试方法（续）

- 画出控制流图：

  如右图所示



- 计算环形复杂度：

  10（条边）- 8（个节点）+ 2 = 4

- 导出独立路径（用语句编号表示）

  路径1：4→14

  路径2：4→6→7→14

  路径3：4→6→9→10→13→4→14

  路径4：4→6→9→12→13→4→14

基本路径测试方法（续）

- 设计测试用例：

	输入数据	预期输出
测试用例1	irecordnum = 0 itype = 0	x = 0 y = 0
测试用例2	irecordnum = 1 itype = 0	x = 2 y = 0
测试用例3	irecordnum = 1 itype = 1	x = 10 y = 0
测试用例4	irecordnum = 1 itype = 2	x = 0 y = 20

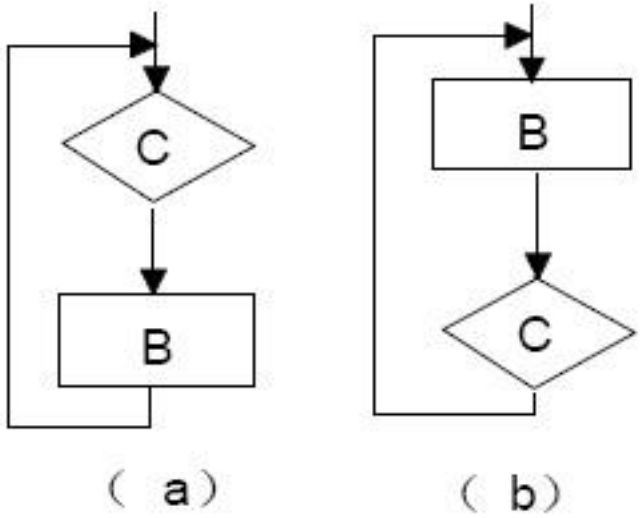
3、 循环测试方法

- 从本质上说，循环测试的目的就是检查循环结构的有效性。
- 通常，循环可以划分为简单循环、嵌套循环、串接循环和非结构循环4类。

（1）测试简单循环

设其循环的最大次数为n ，可采用以下测试集：

- 跳过整个循环；
- 只循环一次；
- 只循环两次；
- 循环 m 次，其中m<n；
- 分别循环 n-1、n 和 n+1 次。



（a） （b）

循环测试方法（续）

（2）测试嵌套循环

如果将简单循环的测试方法用于嵌套循环，可能的测试次数会随嵌套层数成几何级数增加。 可采用以下办法减少测试次数：

① 测试从最内层循环开始，所有外层循环次数设置为最小值；

② 对最内层循环按照简单循环的测试方法进行；

③ 由内向外进行下一个循环的测试，本层循环的所有外层循环仍取最小值，而由本层循环嵌套的循环取某些“典型”值；

④ 重复上一步的过程，直到测试完所有循环。

循环测试方法（续）

（3）测试串接循环

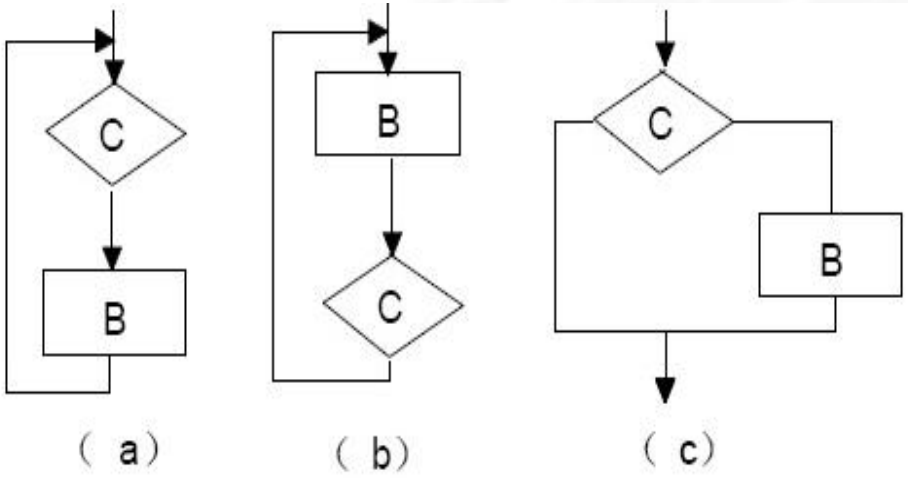
若串接的各个循环相互独立，则可分别采用简单循环的测试方法；否则采用嵌套循环的测试方法。

（4）非结构循环

对于非结构循环这种情况，无法进行测试，需要按结构化程序设计的思想将程序结构化后，再进行测试。

Z路径覆盖下的循环测试方法

- Z路径覆盖是路径覆盖的一种变体，它是将程序中的循环结构简化为选择结构的一种路径覆盖。
- 循环简化的目的是限制循环的次数，不考虑循环的形式和循环体实际执行的次数，简化后的循环测试只考虑执行循环体一次和零次（不执行）两种情况，即考虑执行时进入循环体一次和跳过循环体这两种情况。



（a）　（b）　（c）

在循环简化的思路下，循环与判定分支的效果是一样的，即：循环要么执行、要么跳过。

4、 补充测试用例设计

- 在实践中，除了前面给出的各种方法外，通常还可以采用以下三种方法来补充设计测试用例：

（1）通过非路经分析得到测试用例

——这种方法得到的测试用例是在应用系统本身的实践中提供的，基本上是测试人员凭工作经验的得到，甚至是猜测得到的。

（2）寻找尚未测试过的路径并生成相应的测试用例

——这种方法需要穷举被测程序的所有路径，并与前面已测试路径进行对比。

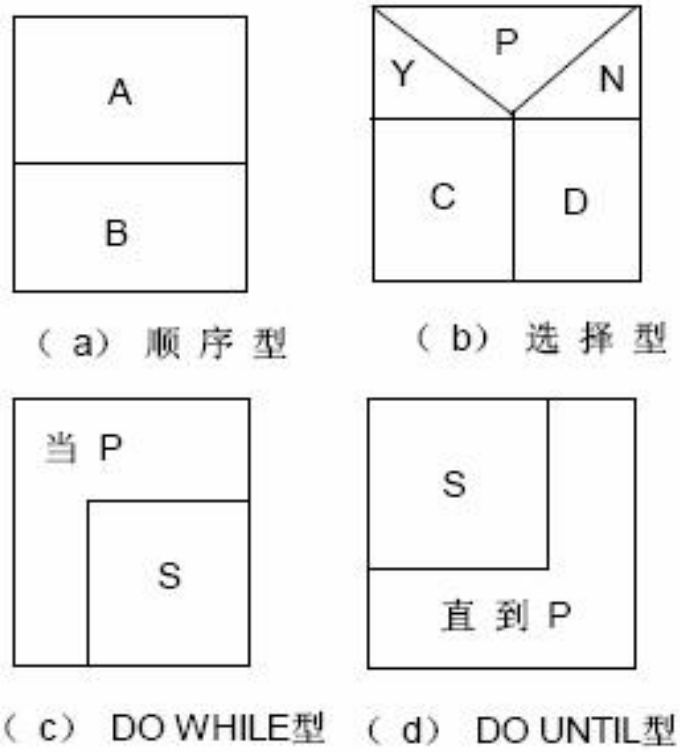
（3）通过指定特定路径并生成相应的测试用例

3.2.5 最少测试用例数计算

- 为实现测试的逻辑覆盖，必须设计足够多的测试用例，并使用这些测试用例执行被测程序，实施测试。我们关心的是：对于某个具体的程序来说，至少需要设计多少个测试用例。这里提供一种估算最少测试用例数的方法。
- 我们知道，结构化程序是由 3 种基本控制结构组成：顺序型（构成串行操作）、选择型（构成分支操作）和重复型（构成循环操作）。
- 为了把问题化简，避免出现测试用例极多的组合爆炸，把构成循环操作的重复型结构用选择结构代替。这样，任一循环便改造成进入循环体或不进入循环体的分支操作了。

最少测试用例数计算（续）

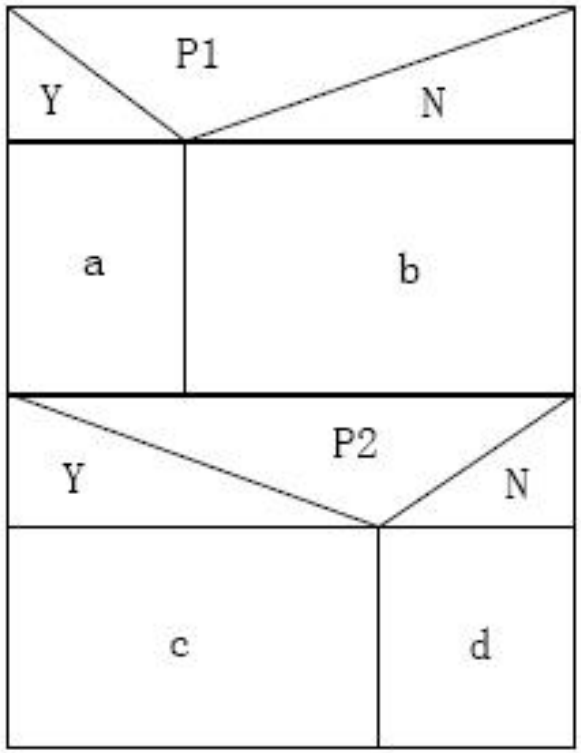
- 盒图或N-S图是结构化编程中的一种可视化建模。 NS图几乎是流程图的同构，任何的N-S图都可以转换为流程图，而大部分的流程图也可以转换为N-S图。
- **用N-S图表示程序的3种基本控制结构：**



- 图中A、B、C、D、S均表示要执行的操作，P是可取真假值的谓词，Y表真值，N表假值。
- 图中的 (c) 和 (d) 两种重复型结构代表了两种循环。在做了简化循环的假设以后，对于一般的程序控制流，我们只考虑选择型结构。事实上它已经能体现顺序型和重复型结构了。

最少测试用例数计算（续）

- 例如，下图表达了两个顺序执行的分支结构。当两个分支谓词P1和P2取不同值时，将分别执行a或b及c或d操作。



显然，要测试这个小程序，需要至少提供4个测试用例才能作到逻辑覆盖，使得ac、ad、bc及bd操作均得到检验。其实，这里的4是图中的第1个分支谓词引出的两个操作，及第2个分支谓词引出的两个操作组合起来而得到的，即2×2=4。并且，这里的2是由于两个并列的操作，即1+1=2 而得到的。

最少测试用例数计算（续）

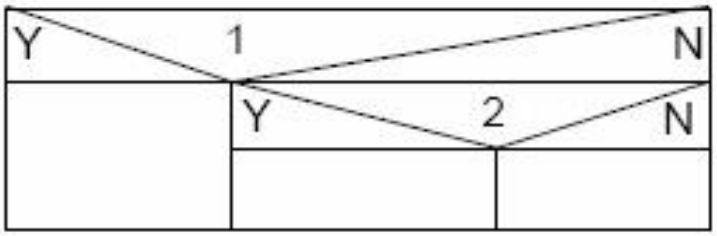
- 对于一般的、更为复杂的问题，估算最少测试用例个数的原则也是同样的：

① 如果在N-S图中存在有并列的层次A1、A2，A1和A2的最少测试用例个数分别为a1、a2，则由 A1、A2 两层所组合的 N-S图对应的最少测试用例数为a1×a2。

② 如果在N-S图中不存在有并列的层次，则对应的最少测试用例数由并列的操作数决定，即N-S图中除谓词之外的操作框的个数。

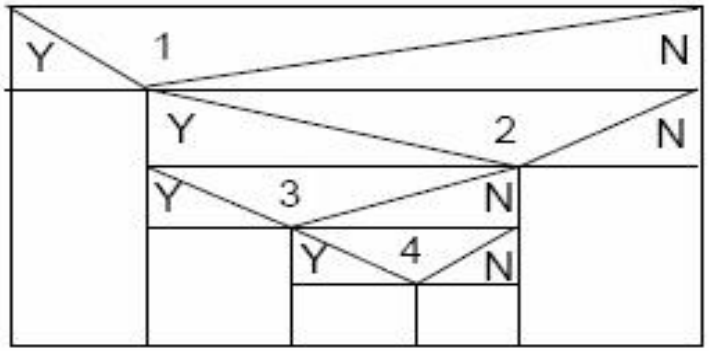
最少测试用例数计算（续）

- 例：如下图所示的两个N-S图，至少需要多少个测试用例完成逻辑覆盖？



- 对于第一个N-S图：

由于图中并不存在并列的层次，最少测试用例数由并列的操作数决定，即为1+1+1=3。

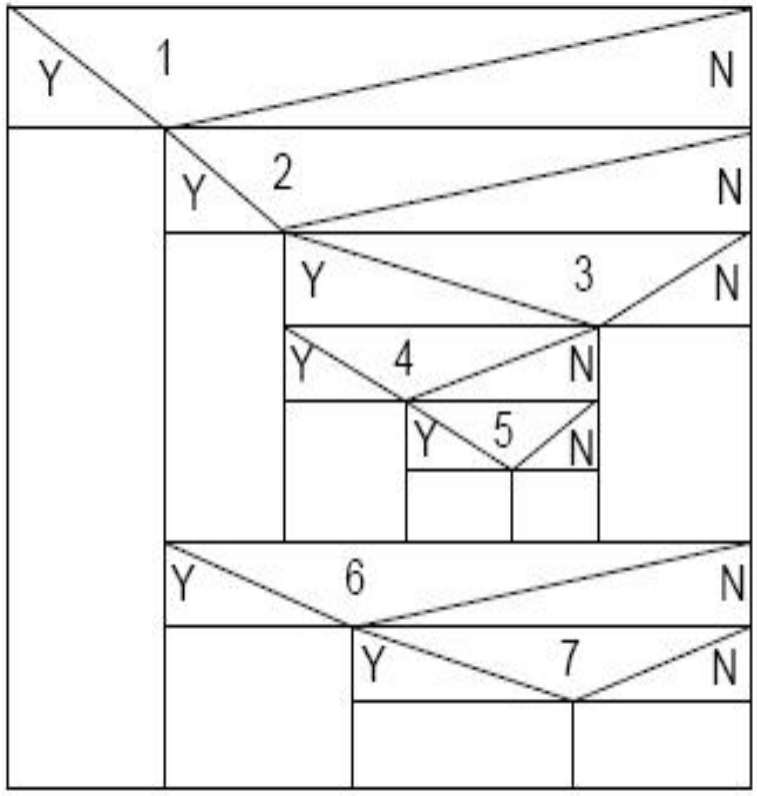


- 对于第二个N-S图：

由于图中没有包含并列的层次，最少测试用例数仍由并列的操作数决定，即为1+1+1+1+1=5。

最少测试用例数计算（续）

- 例：如下图所示的N-S图，至少需要多少个测试用例完成逻辑覆盖？



- 分析该N-S图：

图中的2345和67是并列的两层。其中，2345层对应的最少测试用例数为1+1+1+1+1=5，67层对应的测试用例数为1+1+1=3，2345和67这两层组合后对应的测试用例数为5×3=15。最后，由于两层组合后的部分是不满足谓词1时所要做的操作，还要加上满足谓词1要做的操作，因此整个程序所需测试用例数为15+1=16。

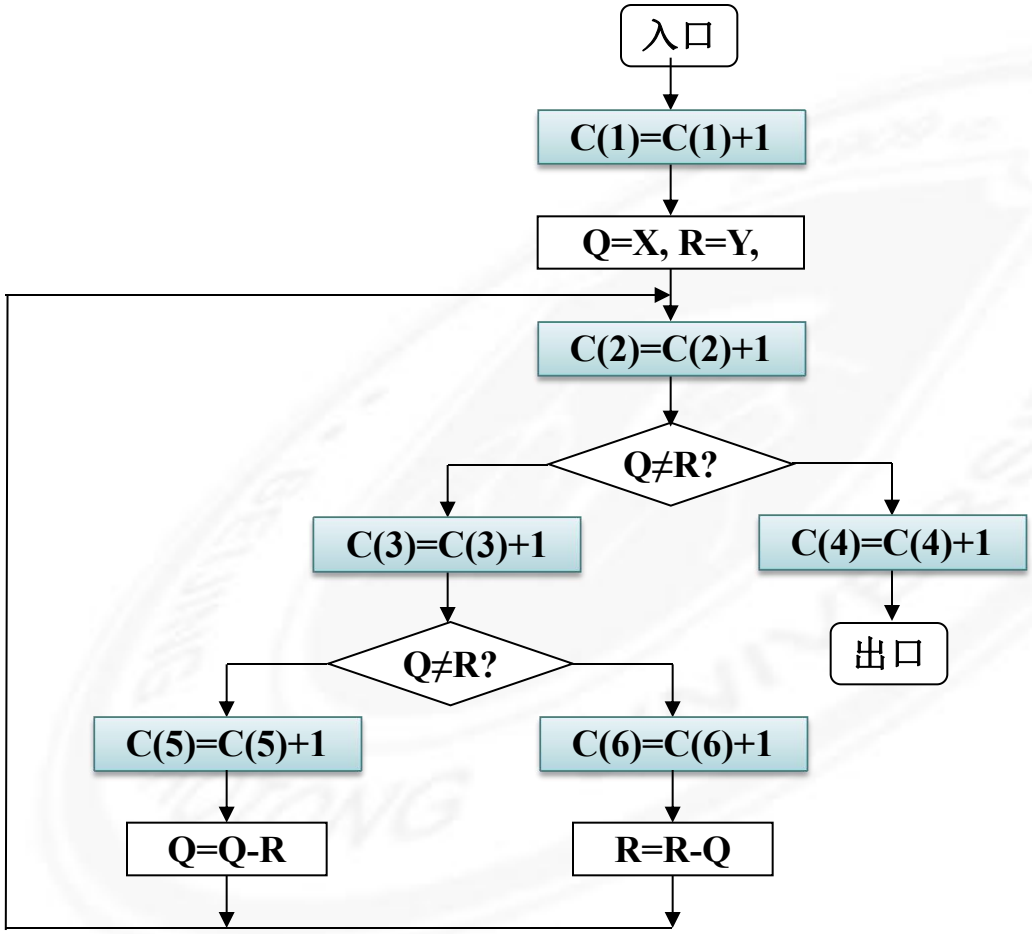
3.2.6其他白盒测试方法简介

- **程序插装测试方法**
- 程序插装（Program Instrumentation）是一种基本的测试手段，在软件测试中有着广泛的应用。
- **方法概述：程序插装的基本原理是在不破坏被测试程序原有逻辑完整性的前提下，在程序的相应位置上插入一些探针。这些探针本质上就是进行信息采集的代码段，可以是赋值语句或采集覆盖信息的函数调用。通过探针的执行并输出程序的运行特征数据。基于对这些特征数据的分析，揭示程序的内部行为和特征。**
  - 断言语句：在所测试源程序中，在指定位置按一定格式，用注释语句写出的断言叫做断言语句。在程序执行时，对照断言语句检查事先指定的断言是否成立。可以帮助复杂系统的检验、调试和维护。

程序插装测试

- 程序插装
  - 向被测程序中插入操作来实现测试目的
- 关注的问题
  - 探测哪些信息？
  - 程序的什么位置设置探测点？
  - 需要多少探测点？
- 程序插装类型
  - 用于测试覆盖率和测试用例有效性度量的程序插装
  - 用于断言检测的程序插装
    - **程序执行到插入点时必须满足的条件，否则就会产生错误**
    - **在进行除法运算之前，加一条分母不为0的断言语句，可以有效地防止程序出错。**

程序插装测试



3.2.6 其他白盒测试方法简介

域测试	域测试（Domain Testing）是一种基于程序结构的测试方法。
符号测试	符号测试的基本思想是允许程序的输入不仅仅是具体的数值数据，而且包括符号值。
Z路径覆盖	舍掉一些次要因素，对循环机制进行简化路径覆盖为Z路径覆盖。
程序变异	程序变异方法与前面提到的结构测试和功能测试都不一样，它是一种错误驱动测试。

3.2.7 白盒测试方法选择的策略

在白盒测试中，使用各种测试方法的综合策略参考如下：

1. 在测试中，应尽量先用人工或工具进行静态结构分析。
2. 测试中可采取先静态后动态的组合方式：先进行静态结构分析、代码检查并进行静态质量度量，再进行覆盖率测试。
3. 利用静态分析的结果作为引导，通过代码检查和动态测试的方式对静态分析结果进行进一步的确认，使测试工作更为有效。
4. 覆盖率测试是白盒测试的重点，一般可使用基本路径测试法达到语句覆盖标准；对于软件的重点模块，应使用多种覆盖率标准衡量代码的覆盖率。
5. 在不同的测试阶段，测试的侧重点不同：在单元测试阶段，以代码检查、逻辑覆盖为主；在集成测试阶段，需要增加静态结构分析、静态质量度量；在系统测试阶段，应根据黑盒测试的结果，采取相应的白盒测试。

本节要点

- 白盒测试方法的基本概念
- 白盒测试的覆盖理论
- 白盒测试的路径表达
- 白盒测试的基本路径测试法

练习题

1、使用条件覆盖测试方法，为以下程序段设计测试用例。

```
      void Do (int X,int A,int B)
      {
1       if ( (A>1)&&(B=0) )
2           X = X/A;
3       if ( (A=2)||(X>1) )
4           X = X+1;
5   }
```

2、在三角形问题中，要求输入三个边长：a，b，c。当三边不可能构成三角形时提示错误，可构成三角形时计算三角形的周长。若是等腰三角形打印“等腰三角形”，若是等边三角形，则打印“等边三角形”。画出相应的程序流程图，并采用基本路径测试方法为该程序设计测试用例。

本章完