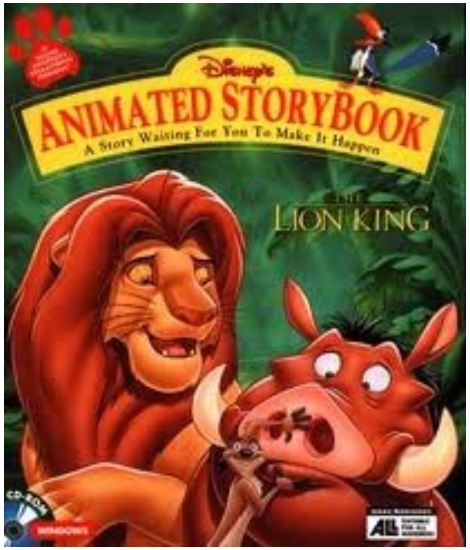
著名的软件错误案例研究

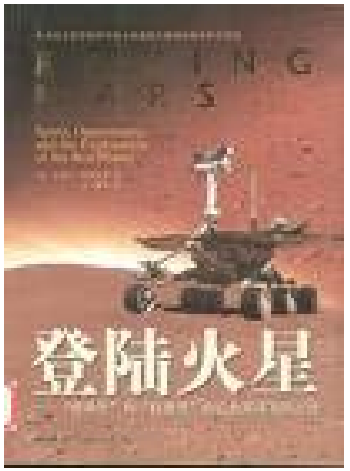
1、迪斯尼的狮子王

- 1994 年秋天，迪斯尼公司发布了第一个面向儿童的多媒体光盘游戏 LionKing Animated Storybook （狮子王动画故事书）。该游戏成为孩子们那个夏季的“必买游戏”。
- 12 月 26 日，圣诞节后的一天，迪斯尼公司的客户支持部电话开始响个不停。很快，电话支持部门就淹没在愤怒的家长和哭诉玩不成游戏的孩子们的电话狂潮之中。



原因：迪斯尼公司没有对市场上投入使用的各种 PC 机型进行正确的测试。软件在少数系统中工作正常——例如迪斯尼的程序员用于开发游戏的系统——但在大众使用的常见系统中却不行。

2、美国航天局火星基地登陆，1999



- 1999 年 12 月 3 日，美国航天局的火星基地登陆飞船在试图登陆火星表面时失踪。错误修正委员会观测到故障，并认定出现误动作的原因极可能是某一个数据被意外更改。

大家一致声讨，问题为什么没有在内部测试时解决。

- 从理论上看，登陆计划是：当飞船降落在火星表面时，它将打开降落伞减缓飞船的下落速度。降落伞打开后的几秒种内，飞船的三条腿将迅速撑开，并在预定地点着陆。当飞船离地面 1800 米时，它将丢弃降落伞，点燃登陆推进器，在余下的高度缓缓降落地面。

- 美国航天局为了省钱，简化了确定何时关闭推进器的装置。为了替代其他太空船上使用的贵重雷达，他们在飞船的脚上装了一个廉价的触点开关，在计算机中设置一个数据位来关掉燃料。即，飞船的脚不“着地”，引擎就会着火。
- 错误修正委员会在测试中发现，当飞船的脚迅速撑开并准备着陆时，机械震动在大多数情况下也会触发着地开关，设置错误的数据位。设想飞船开始着陆时，计算机极有可能关闭推进器，而火星登陆飞船下坠 1800 米之后冲向地面，撞成碎片。

原因： 登陆飞船经过了两个小组测试，其中一个小组测试飞船的脚落地过程，另一个小组测试此后的着陆过程。前一个小组不去注意着地数据位是否置位，后一个小组总是在开始测试之前重置计算机、清除数据位。双方独立工作很好，但从未在一起进行集成测试。

3、爱国者导弹防御系统，1991

美国爱国者导弹防御系统是里根总统提出的主动战略防御（即星球大战）程序的缩略版本。它首次应用在海湾战争中对抗伊拉克飞毛腿导弹的防御战争中。尽管关于此系统的赞誉不绝于耳，但是它确实在几次对抗导弹战役中失利，其中一枚在沙特阿拉伯的多哈击毙 28 名美国士兵。

原因：存在一个软件缺陷。一个很小的系统时钟错误积累起来就可能拖延 14 小时，造成跟踪系统失去准确度。在多哈袭击战中，系统被拖延 100 多小时。

4、千年虫，大约 1974

➢20 世纪 70 年代某位程序员——假设他叫 Dave —— 负责本公司的工资系统。他使用的计算机存储空间很小，迫使他尽量节省每一个字节。 Dave 自豪地将自己的程序压缩得比其他人都小。

➢他使用的其中一个办法是把 4 位数日期，例如 1973 缩减为 2 位数，例如 73 。因为工资系统极度依赖数据处理， Dave 得以节省可观的存储空间。他认为只有在到达 2000 年时程序计算 00 或 01 这样的年份时才会出现问题。他知道这样会出问题，但是在 25 年之内程序肯定会更改或升级，而且眼前的任务比现在计划遥不可及的未来更加重要。

➢这一天毕竟是要到来的。 1995 年， Dave 的程序仍然在使用，而 Dave 退休了，谁也不会想到进入程序检查 2000 年兼容问题，更不用说去修改了。

估计世界各地更换或升级类似 Dave 程序以解决原有 2000 年错误的费用以及超过数亿美元了。

8.1 软件测试的定义和目标

软件测试：检测和评价软件以确定其质量的过程和方法，即评价软件或程序的属性和能力，以确定它是否满足所需结果的过程与方法。

只是检查和审阅

运行和使用软件

软件测试可分为静态分析和动态测试：

（1）进行静态分析时，不必运行软件，只是通过对源代码进行分析，检测程序的控制流和数据流，以及发现执行不到的“死代码”、无限循环、未初始化的变量、未使用的数据、重复定义的数据等；也可能包括对多种复杂性度量值的计算。静态分析虽然不能取代动态测试，但它是动态测试开始前有用的质量检测手段。

（2）动态测试技术借助于输入样例（即测试用例）来执行软件，一般又分为功能测试（即黑盒测试）以及结构测试（即白盒测试）。

From:《计算机科学技术百科全书》（第二版），张效祥主编，清华大学出版社，2005

软件测试目标：

（1）预防错误

（2）发现错误

一般只有符合下列 5 个规则才叫软件错误：
1. 软件未达到产品说明书标注的功能 .
2. 产品出现了产品说明书指明不会出现的错误 .
3. 软件功能超出产品说明书的范围 .
4. 软件未达到产品说明书虽未指出但应达到的目标 .
5. 软件测试员认为软件难以理解、不易使用、运行速度缓慢，或者最终用户认为不好。

From: 《软件测试》，（美） Patton, R. 著. 北京：机械工业出版社， 2002.2.

8.2 软件测试与软件调试的区别

软件调试：发现所编写软件中的错误，确定错误的位置并加以排除，使之能由计算机或相关软件正确理解与执行的方法与过程。

在进行调试工作以前，首先要发现存在着某种错误的迹象。随后的调试过程通常分为两步：

（1）确定问题的性质并且找到该错误在软件中所处的位置；

（2）修正这一错误。

– **From:《计算机科学技术百科全书》（第二版），张效祥主编，清华大学出版社，2005 年 11 月。**

软件测试和软件调试的主要区别：

（1）测试从一个侧面证明程序员的“失败”，而调试是为了证明程序员的正确。

（2）测试以已知条件开始，使用预先定义的程序，且有预知的结果，不可预见的仅是程序是否通过测试。调试一般是以不可知的内部条件开始，除统计性调试外，结果是不可预见的。

（3）测试是有计划的，并要进行测试设计；而调试是不受时间约束的。

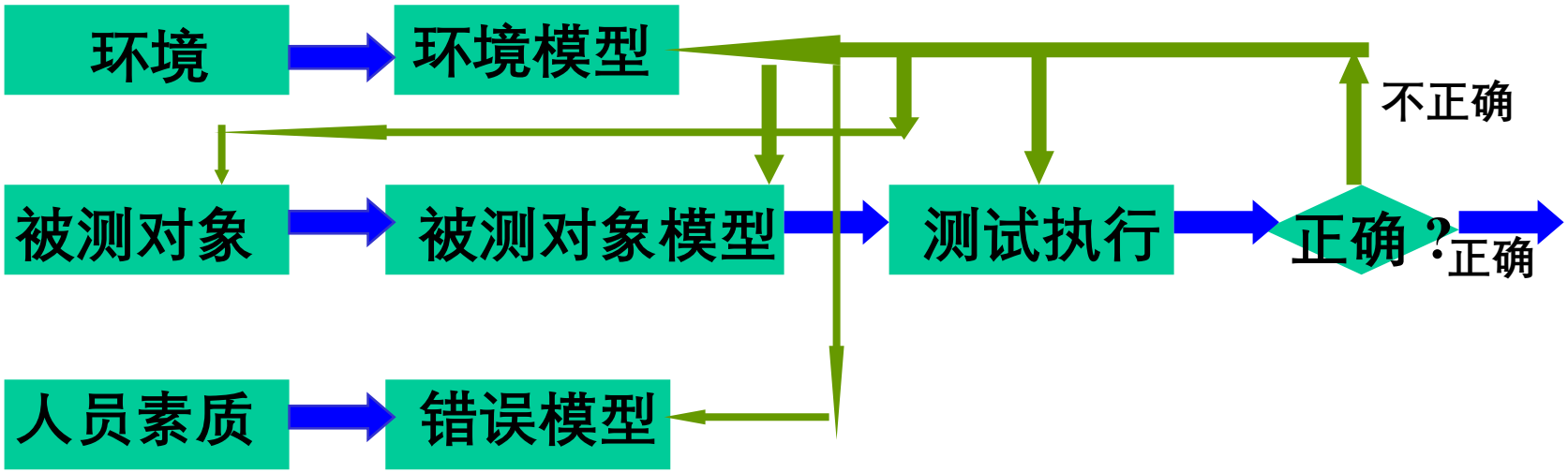
（4）测试是一个发现错误、改正错误、重新测试的过程；而调试是一个推理过程。

（5）测试的执行是有规律的，而调试的执行往往要求程序员进行必要推理以至知觉的“飞跃”。

（6）测试经常是由独立的测试组在不了解软件设计的条件下完成的；而调试必须由了解详细设计的程序员完成。

（7）大多数测试的执行和设计可由工具支持，而调试时，程序员能利用的工具主要是调试器。

8.3 软件测试过程模型



- 软件测试过程所涉及的要素，以及
- 这些要素之间的关系

•环境：包括支持其运行的硬件、固件和软件；

•被测对象模型：为了测试，形成被测对象的简化版本。不同的测试技术，对同一被测对象（程序），可产生不同的对象模型：

  •简化注重程序的控制结构－－－形成“白盒”测试

  •简化注重程序的处理过程－－－形成“黑盒”测试

•错误模型：为了统一认识，定义“什么是错误”。

几个关键性的概念：

- 错误（error）是指“与所期望的设计之间的偏差，该偏差可能产生不期望的系统行为或失效”。
- 失效（failure）是指“与所规约的系统执行之间的偏差”。失效是系统故障或错误的后果。
- 故障（fault）是指“导致错误或失效的不正常的条件”。故障可以是偶然性的或是系统性的。

三者关系：

- 程序员编写程序，在这个过程中，他无意或有意地犯一个错误（error）。
- 故障（fault）是一个或多个错误的表现。
- 当执行程序中那段有故障的代码时，就会引起失效（failure），导致程序出现不正确的状态，影响程序的输出结果。

8.4 软件测试的原则

（1）所有的测试都应当追朔到用户需求。软件测试的目的在于发现错误，而从用户角度看，最严重的错误就是那些致使程序无法满足需求的错误。
（2）在测试工作开始前，要进行测试计划的设计。测试计划可以在需求分析一完成时开始，详细的测试用例定义可以在设计模型被确定后立即开始。
（3）测试应从小规模开始，逐步转向大规模。最初的测试通常放在单个程序模块上，测试焦点逐步转移到在集成的模块簇内寻找错误，最后在整个系统中寻找错误。
（4）穷举测试是不可能的。一个大小适度的程序，其路径排列的数量是惊人的。
（5）为了尽可能发现错误，应由独立的第三方来测试。
（6）在一般情况下，在分析、设计、实现阶段的复审和测试工作能够发现和避免 80 %的 bug，而系统测试又能找出其余一些 bug，最后剩下的 bug 可能只能在用户的大范围、长时间的使用后才会暴露。因此测试只能保证尽可能多地发现错误，无法保证能够发现所有的错误。

8.5 软件测试技术

黑盒测试：

黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能，通过测试来检测每个功能是否都能正常使用，在测试时，把程序看作一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数锯而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。

黑盒测试方法主要有等价类划分、边界值分析、因果图、错误推测等，主要用于软件确认测试。

“黑盒”法是穷举输入测试，只有把所有可能的输入都作为测试情况使用，才能以这种方法查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但是可能的输入进行测试。

黑盒测试试图发现以下错误类型：

（１）功能不对或遗漏；

（２）界面错误；

（３）数据结构或外部数据库访问的错误；

（４）性能错误；

（５）初始化和终止错误．

问题：黑盒测试依据什么？

白盒测试：

白盒测试也称结构测试或逻辑驱动测试，它是知道产品内部工作过程，可通过测试来检测产品内部动作是否按照规格说明书的规定正常进行，按照程序内部的结构测试程序，检验程序中的每条通路是否都有能按预定要求正确工作，而不顾它的功能．

白盒测试的主要方法有逻辑覆盖、基本路径测试等，主要用于软件验证。

问题：白盒测试依据什么？

8.5.1 白盒测试技术

依据程序的逻辑结构 - 白盒测试技术

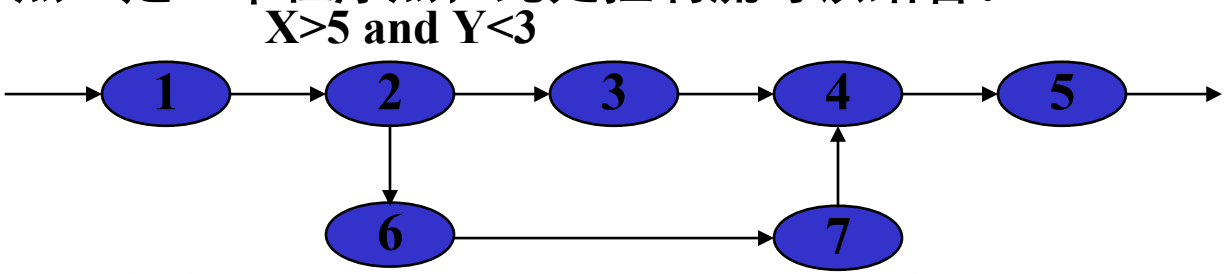
（1）关于建立被测对象模型

控制流程图：一种表示程序控制结构的图形工具，其基本元素是节点、判定、过程块。

其中：过程块是既不能由判定、也不能由节点分开的一组程序语句；

判定：是一个程序点，此处控制流可以分叉；

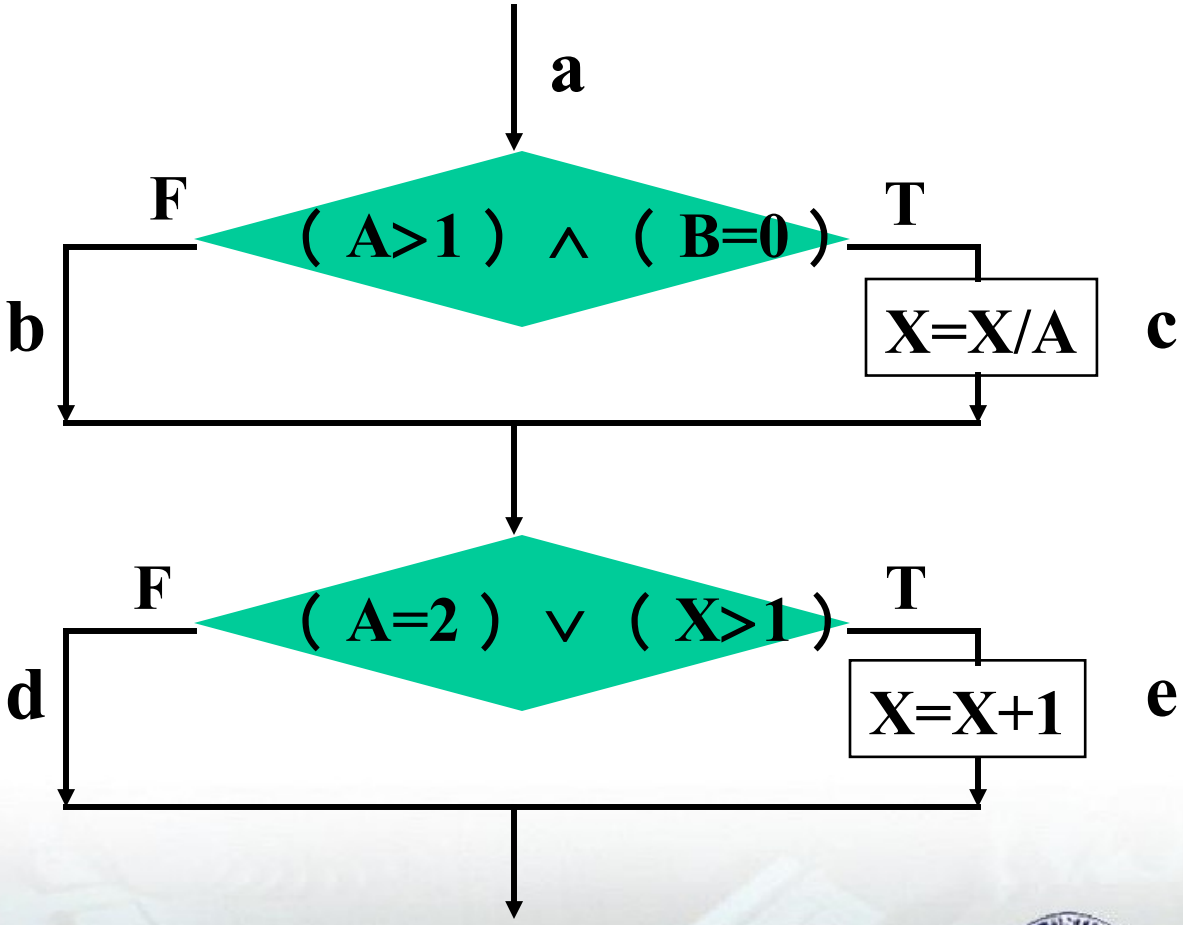
节点：是一个程序点，此处控制流可以结合。



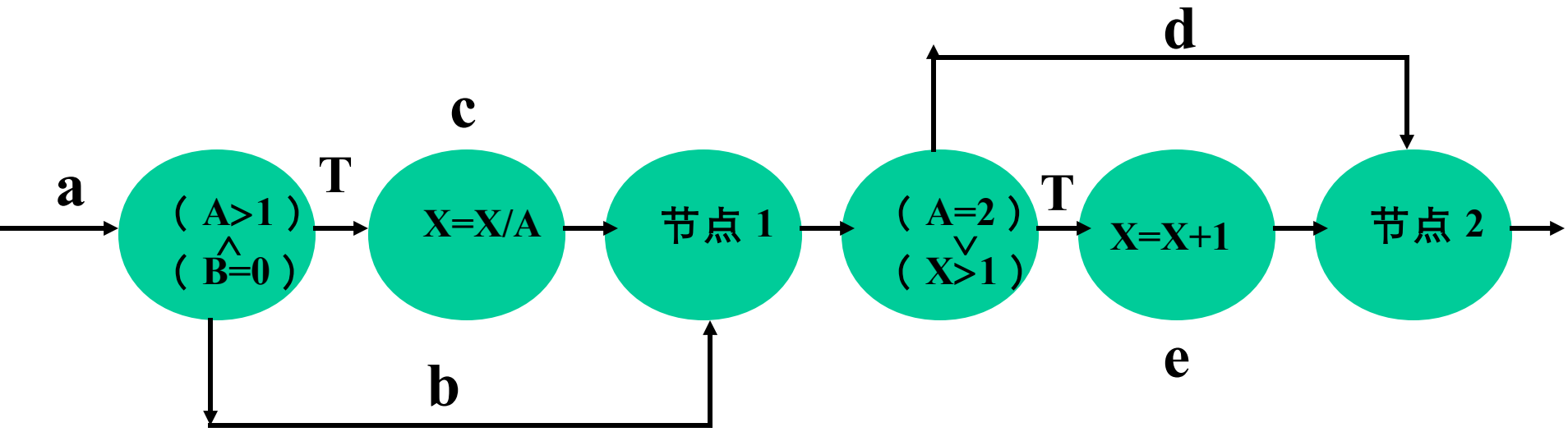
其中：1、3、5、6、7 为过程块

2 为判定，4 为节点

例如：以下为一个程序流程图，其中该例子中有两个判断，每个判断都包含复合条件的逻辑表达式。



其控制流程图为：



（2）各种测试方法

该控制流程图有 4 条不同的路径。4 条路径可表示为：

L1 （a→c→e）简写 ace 、L2 （a→b→d）简写 abd

L3 （a→b→e）简写 abe 、L4 （a→c→d）简写 acd

- 路径测试（PX）：执行所有可能的穿过程序的控制流程路径。

一般来说，这一测试严格地限制为所有可能的入口 / 出口路径。如果遵循这一规定，则我们说达到了 100% 路径覆盖率。在路径测试中，该策略是最强的，但一般是不可实现的。

针对该例子，要想实现路径覆盖，可选择以下一组测试用例（规定测试用例的设计格式为：【输入的（A，B，X），输出的（A，B，X）】）。

测试用例	覆盖路径
【（2，0，4），（2，0，3）】	L1
【（1，1，1），（1，1，1）】	L2
【（1，1，2），（1，1，3）】	L3
【（3，0，3），（3，0，1）】	L4

- 语句测试（P1）：至少执行程序中所有语句一次。

如果遵循这一规定，则我们说达到了 100% 语句覆盖率（用 C1 表达）。

在该例子中，只要设计一种能通过路径 ace 的测试用例，就覆盖了所有的语句。所以可选择测试用例如下：

【（2，0，4），（2，0，3）】 覆盖 L1

语句覆盖是最弱的逻辑覆盖准则。

问题：就该程序而言，如果两个判断的逻辑运算写错，例如，第一个判断中的逻辑运算符“∧”错写成了“∨”，或者第二个判断中的逻辑运算符“∨”错写成了“∧”，利用上面
的测试用例，仍可覆盖其中 2 个语句，而发现不了判断中逻辑运算符出现的错误。

- 分支测试（P2）：至少执行程序中每一分支一次。如果遵循这一规定，则我们说达到了 100% 分支覆盖率（用 C2 表示）。

分支覆盖是一种比语句覆盖稍强的逻辑覆盖。但若程序中分支的判定是由几个条件联合构成时，它未必能发现每个条件的错误。

例如对于以上例子，如果选择路径 L1 和 L2，就可得到实现分支覆盖的测试用例：

【（2，0，4），（2，0，3）】 覆盖 L1

【（1，1，1），（1，1，1）】 覆盖 L2

如果选择路径 L3 和 L4，还可得另一组可用的测试用例：

【（2，1，1），（2，1，2）】 覆盖 L3

【（3，0，3），（3，1，1）】 覆盖 L4

问题：分支覆盖还不能保证一定能查出在判断的条件中存在的错误。例如，在该例子中，若第二个分支 X>1 错写成 X<1，利用上述两组测试用例进行测试，无法查出这一错误。因此，需要更强的逻辑覆盖准则去检验判定的内部条件。

- 条件组合测试

条件组合测试是一种具有更强逻辑覆盖的测试。

条件组合测试，就是设计足够的测试用例，使每个判定中的所有可能的条件取值组合至少执行一次。如果遵循这一规定，则我们说就实现了条件组合覆盖。只要满足了条件组合覆盖，就一定满足分支覆盖。

在条件组合覆盖技术发展过程中，最初，在设计测试用例时，人们只考虑使分支中各个条件的所有可能结果至少出现一次。但发现该测试技术未必能覆盖全部分支。例如，在上图的例子中，程序段中有四个条件： A>1， B=0， A=2， X>1

条件 A ＞ 1 取真值标记为 T1，取假值标记为 F1

条件 B=0 取真值标记为 T2，取假值标记为 F2

条件 A=2 取真值标记为 T3，取假值标记为 F3

条件 X ＞ 1 取真值标记为 T4，取假值标记为 F4

在设计测试用例时，要考虑如何选择测试用例实现 T1、F1、T2、F2、T3、F3、T4、F4 的全部覆盖：

例如，可设计如下测试用例实现条件覆盖：

测试用例	通过路径	条件取值	覆盖分支
【（1，0，3），（1，0，4）】	L3	F1 T2 F3 T4	b,e
【（2，1，1），（2，1，2）】	L3	T1 F2 T3 F4	b,e

从上面的测试用例，可以看到该组测试用例虽然实现了判定中各条件的覆盖，但没有实现分支覆盖，因为该组测试用例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。为此，人们又进一步提出了条件组合覆盖技术。

例如，在该例子中，前一个判定有 4 种条件组合：

（1）(A > 1)，(B=0)，标记为 T1 、T2 ；

（2）(A > 1)，(B≠0)，标记为 T1 、F2, ；

（3）(A≤1)，(B=0)，标记为 F1 、T2 ；

（4）(A≤1)，(B≠0)，标记为 F1 、F2 ；

后一个判定又有 4 种条件组合：

（5）(A=2)，(X > 1)，标记为 T3 、T4 ；

（6）(A=2)，(X≤1)，标记为 T3 、F4 ；

（7）(A≠2)，(X > 1)，标记为 F3 、T4 ；

（8）(A≠2)，(X≤1)，标记为 F3 、F4 。

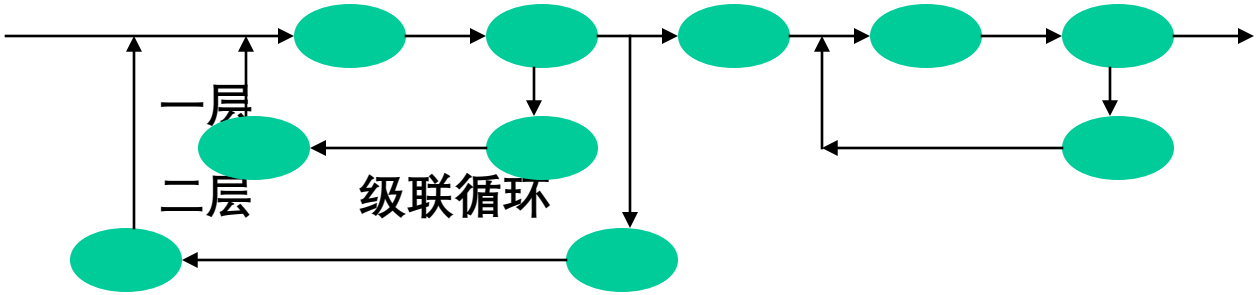
可以采用以下四组测试数据，从而实现条件组合覆盖。

测 试 用 例	覆盖条件	覆盖组合号	通过路径	
【（2，0，4），（2，0，3）】	T1 T2 T3 T4	(1)(5)		L1
【（2，1，1），（2，1，2）】	T1 F2 T3 F4	(2)(6)		L3
【（1，0，3），（1，0，4）】	F1 T2 F3 T4	(3)(7)		L3
【（1，1，1），（1，1，1）】	F1 F2 F3 F4	(4)(8)		L2

这组测试用例实现了分支覆盖，也实现了条件的所有可能取值的组合的覆盖。

问题：语句覆盖、分支覆盖、条件组合覆盖、路径覆盖之间的关系?

（3）循环情况的路径选取



嵌套循环

还要考虑循环变量的具体情况

关键路径的选取

主要功能路径

没有功能的路径

最短路径

...

8.5.2 黑盒测试（功能测试）- 依据软件行为的描述的测试

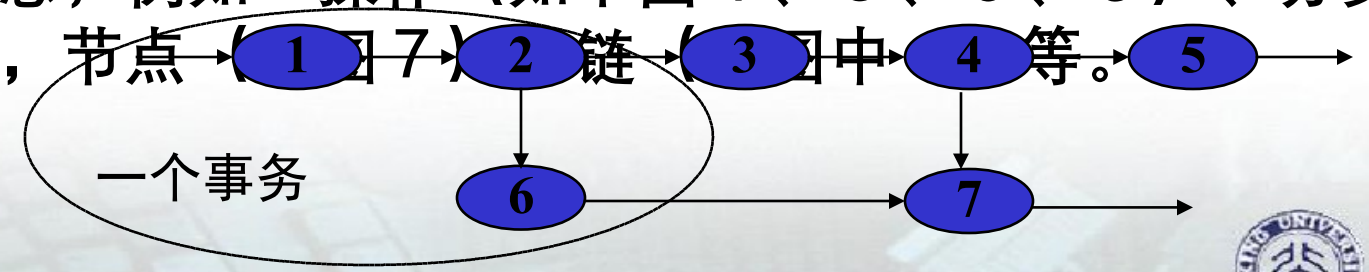
8.5.2.1 事务流测试技术

（1）基本概念：

① 事务：以用户的角度所见的一个工作单元。

一个事务由一系列操作组成。其中某些操作可含有系统执行成分，或含有设备执行成分，它们共同协作，完成用户的一项工作。

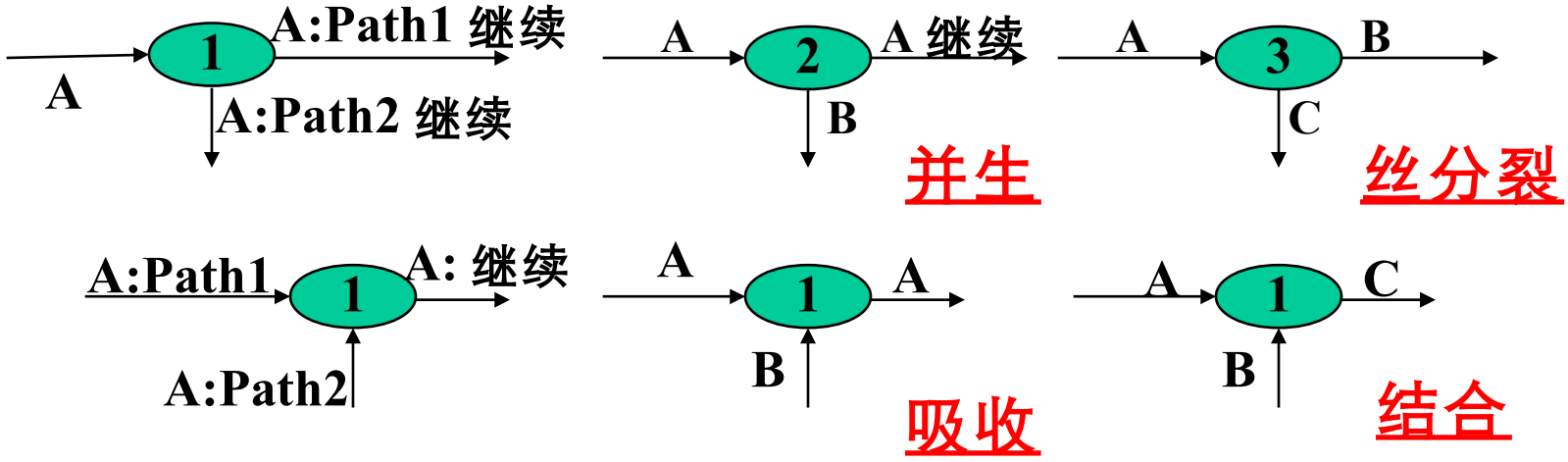
② 事务处理流程（图）：系统行为的一种表示方法，为功能测试建立了软件动作模式。其中使用了白盒测试中的一些概念，例如：操作（如下图1、3、6、5）、分支（下图2），节点（如图7）、链（如图中等。



（2）与程序控制流程图的比较：

① 事务流图是一种数据流图，即从操作应用的历史，观察数据对象。

② 事务流图中的分支：“抽象”了一个复杂的过程。



③ 事务流图存在“中断”，把一个过程等价地变换为具有繁多出口的链支。

测试设备：路径分析器，测试用例数据库，测试执行调度器

（3）测试步骤

第一步：获取事务流程图，即建立被测对象模型；

第二步：浏览与复审

主要对事务进行分类，为设计用例奠定基础；

第三步：用例设计

设计足够测试用例，实现基本事务覆盖。

涉及：覆盖策略，事务选取等；

第四步：测试设备开发：

路径分析器，测试用例数据库，

测试执行调度器，…

第五步：测试执行；

第六步：测试结果比较。

8.5.2.2 等价类划分技术

(1) 基本概念

① 等价类：输入域的一个子集，在该子集中，各个输入数据对于揭示程序中的错误都是等效的。即：以等价类中的某代表值进行的测试，等价于对该类中其他取值的测试。

② 有效等价类：指那些对于软件的规格说明书而言，是合理的、有意义的输入数据所构成的集合。

- 用于实现功能和性能的测试。

③ 无效等价类：指那些对于软件的规格说明书而言，是不合理的、无意义的输入数据所构成的集合。

- 用于测试那些所实现的功能和性能不符合规格说明书的要求。

（2）等价类划分原则（指南）

① 如果输入条件规定了输入数据的取值范围或值的个数，则可以确定一个有效等价类和二个无效等价类。

例如：输入条件：“... 项数可以是 1 到 999...”

② 如果输入条件规定了输入数据的一组值，而且软件要对每个输入值进行处理，则可以为每一个输入值确定一个有效等价类，为所有不允许的输入值确定一个无效等价类。

例如：对大一、大二、大三、大四的学生进行管理，可确定 4 个有效等价类，为大一、大二、大三、大四的学生，一个无效等价类为不符合上述身份的人。

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 如果输入条件规定了输入值必须符合的条件，则可以确定一个有效等价类（符合条件）和一个无效等价类（不符合条件）。

例如：“标识符是一字母打头的…串。” 则

字母打头的 -- 为一个有效等价类，而

其余的 -- 为一个无效等价类

注意：如果在已确定的等价类中各元素在软件中的处理方式不同，则应根据需要对等价类进一步进行划分。

（3） 测试用例设计

在确定了等价类之后，建立等价类表：

输入条件	有效等价类	无效等价类
……	……	……
……	……	……

再根据等价类来设计测试用例，过程如下：

（1）为每一个等价类规定一个唯一的编号；

（2）设计一个新的测试用例，使其尽可能多的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）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。

问题：为什么设计无效等价类的测试用例时仅包括一个未被覆盖的无效等价类？

（3）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。

问题：为什么设计无效等价类的测试用例时仅包括一个未被覆盖的无效等价类？

因为某些程序中对某一输入错误的检查往往会屏蔽对其它输入错误的检查。因此设计无效等价类的测试用例时应该仅包括一个未被覆盖的无效等价类。

8.5.2.3 边界值分析

边界值分析是一种最常用的黑盒测试技术。因为测试工作经验表明，大量的错误经常发生在输入或输出范围的边界上。因此设计一些测试用例，使程序运行在边界情况附近，这样揭露错误的可能性比较大。

使用边界值分析设计测试用例可遵循以下原则：

（1）如果某个输入条件规定了输入值的范围，则应选择正好等于边界值的数据，以及刚刚超过边界值的数据作为测试数据。

（2）如果某个输入条件规定了值的个数，则可用最大个数、最小个数、比最大个数多 1、比最小个数少 1 的数作为测试数据。

（3）根据规格说明的每个输出条件，使用前面的原则（1）。

（5）如果程序的规格说明中，输入域或输出域是一个有序集合，在实践中，则经常选取集合的第一个元素、最后一个元素以及典型元素作为测试用例。

（6）如果程序中使用了内部数据结构，则应选择这个内部数据结构的边界上的值作为测试用例。

（7）分析规则说明，找出其他可能的边界条件。

8.5.2.4 因果图

因果图：是设计测试用例的一种工具，它着重检查各种输入条件的组合。例如，两个输入值的乘积超过了存储器的限制，程序将发生错误。等价划分和边界值分析够不能发现这类错误，因为它们未考虑输入情况的各种组合。

因果图的基本原理是通过画因果图，把用自然语言描述的功能说明转换为判定表，最后为判定表的每一列设计一个测试用例。

8.6 软件测试步骤

8.6.1 单元测试

主要检查软件设计的最小单位－模块。该测试以详细设计文档为指导，测试模块内的重要控制路径。对应程序编码，主要采用白盒测试技术。

1、测试重点：

单元测试期间，可以考虑的模块的以下几个特征：

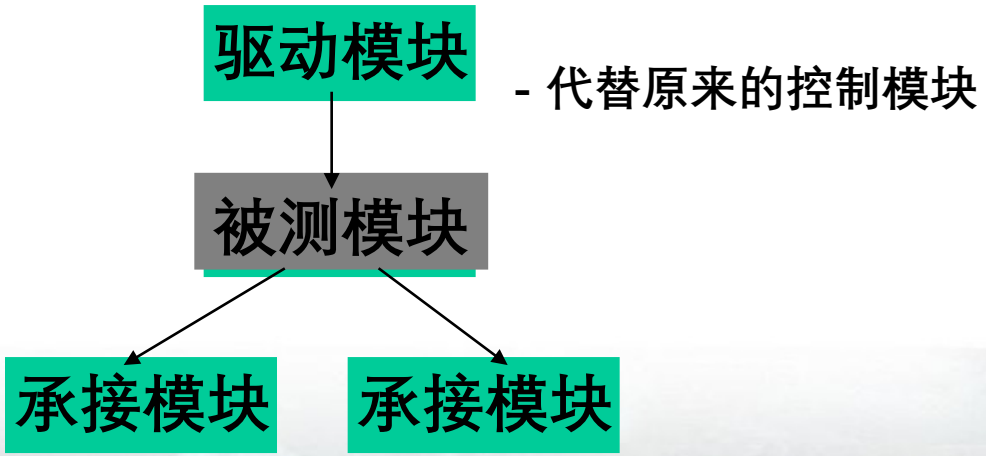
（1）模块接口；

（2）局部数据结构；

（3）“重要的”执行路径；

（4）错误执行路径

（5）边界条件



2、代码审查

人工测试源程序可以由程序的编写者本身非正式地进行，也可以由审查小组正式进行。后者称为代码审查，对于典型的程序来说，可以查出 30%-70% 的逻辑设计错误和编码错误。

（1）审查小组的构成（最好由下述 4 人组成）：

- 组长，应该是一个很有能力的程序员，且没有直接参与这项工程。
- 程序的设计者。
- 程序的编写者。
- 程序的测试者。

（2）审查的步骤

- 审查之前，小组成员应先研究设计说明书，力求理解这个设计
- 在审查会上，由程序的编写者解释他是如何用代码实现这个设计的，通常是逐个语句地讲述程序的逻辑，小组成员倾听，并力图发现错误
- 审查会上，对照上面五类错误，分析审查这个程序，发现的错误由组长记录下来
- 审查会上，还可以由一人扮演“测试者”（他需要在会前做好测试方案），其他人扮演“计算机”，这样由扮演计算机的成员模拟计算机执行被测程序。

8.6.2 集成测试

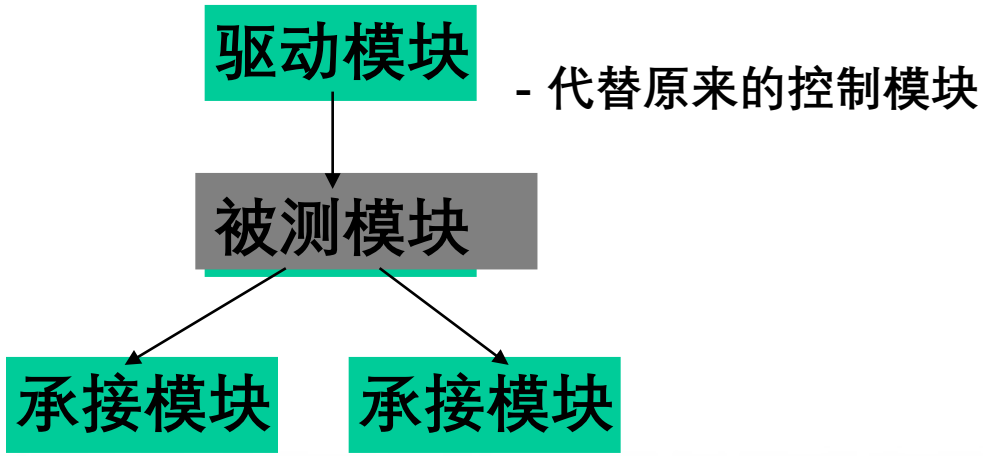
对应软件的设计，主要检测各个单元集成过程中相互之间的接口错误以及形成新的组合后功能中的错误，多采用黑盒测试技术并辅以一些白盒测试技术。

集成测试是一种软件集成化技术

- **方式：自顶向下或自底向上**
- **设计测试设备**

  驱动模块

  承接模型



8.6.3 有效性测试（确认测试，验收测试）

有效性测试的目标是发现软件实现的功能与需求规格说明书不一致的错误。

（1）确认测试的范围

确认测试对应系统需求，以确保软件符合所有功能、行为、性能的需求测试，确认测试只使用黑盒测试技术。

确认测试必须有用户积极参与，或者以用户为主进行.

用户应参与设计测试方案，使用用户界面输入测试数据并分析评价测试的输出结果.

（2）软件配置检查

确认测试的一个重要内容是复查软件配置，保证软件配置的所有成分都齐全，质量符合要求，文档与程序完全一致，且已经编号目录．在确认测试过程中，还应该严格遵循用户指南及其他操作程序，以便检查这些使用手册的完整性和正确性.

（3） Alpha 和 Beta 测试

如果软件专为某个客户开发，可以进行一系列确认测试，以便用户确认所有需求都得到满足.

- 如果一个软件是为许多客户开发的，让每个客户进行确认测试是不现实的．这时，软件开发商使用 Alpha 和 Beta 测试，来发现那些看起来只有最终用户才能发现的错误
  - α 测试：产品发布前开发单位的内部综合测试。
  - β 测试：产品正式投入市场前，由有关用户试用测试。

8.6.4 系统测试

集中检验系统所有元素（包括硬件、信息等）之间协作是否合适，整个系统的性能、功能是否达到。

系统测试实际上是一系列不同的测试，以下是用于系统测试的几种典型软件系统测试：

- 恢复测试：是一种系统测试，它指采取各种人工干预方式强制性地使软件出错，使其不能正常工作，进而检验系统的恢复能力。
  - 如果恢复是自动的，则重新初始化、检测点设置、数据恢复以及重新启动等都是对其正确性的评价；
  - 如果恢复需要人员的干预，则要估算出修复的平均时间，以及确定它是否能在可接受的限制范围内。

•**安全性测试**：**就是试图去验证建立在系统内的预防机制，以防止来自非正常的侵入**：

- 充当任何角色，测试者可以通过外部书写的方式得到口令；
- 可以用用户设计的软件去破坏已构造好的任何防御的袭击系统；
- 可以破坏系统，使系统不能为他人服务；
- 可以跳过侵入的恢复过程，而故意使系统出错；
- 也可以跳过找出系统的入口钥匙，而放过看到的不安全数据等；

•**强度测试**：**是在要求一个非正常数量、频率或容量资源方式下运行一个系统。**

•**性能测试**：**就是测试软件在被组装进系统的环境下运行时的性能：**

- 性能测试应覆盖测试过程的每一步，即使在单元层，单个模块的性能也可以通过白盒测试来评价；
- 性能测试有时与强度测试联系在一起，常常需要硬件和软件的测试设备。

- **可用性测试：**从使用的合理性、方便性等角度对软件系统进行检验，以发现人为因素或使用上的问题。

- **部署测试（配置测试）**：软件必须在多种平台及操作系统环境中运行。有时将部署测试称为配置测试，是在软件将要在其中运行的每一种环境中测试软件。另外，部署测试检查客户将要使用的所有安装程序及专业安装软件，并检查用于向最终用户介绍软件的所有文档。

8.7 程序正确性证明（这部分不要求掌握）

问题的提出：

以上介绍的测试技术，可以发现错误，但不能表明程序没有错误！

正确性证明方法的研究：

开发了一些程序正确性证明方法。其主要思路是：

① 针对一种特定的语言

② 以人工智能和谓词演算为基础

基本评价：已开发的程序正确性证明技术，对大型软件而言，是不适用的。

霍尔系统 - 基于公理的证明技术

①Hoare 公理系统

对一个特定的顺序语言而言，其主要构造可能有：

赋值语句，复合语句，选择语句，循环语句

（其中没有考虑输入 / 输出语句）

因此，可以通过对以上“构造‘的性质的研究、分析，

为推导程序的正确性（程序的一个性质），得出“一个

公认的事实”，即公理

程序的性质，可以采用**前置条件**和**后置条件**予以表达，因此一个程序可表示为：

{ P } Q { R }

其中：{ P } 为程序 Q 的**前置条件**；

{ R } 为程序 Q 的**后置条件**；

北京大学

- 关于赋值的公理

考虑如下赋值语句：

x ： =f　　其中， x 是一个简单变量的标识符

f 是一个没有副作用的表达式，但可能含有 x

如果在赋值之后断定 P （ x ）为真，那么在赋值之前 P （ f ）一定为真。

例如： x ： = x↑2+5 。

假定： x 在赋值语句执行前，有值为 2 。

在赋值之后， P （ x ） >9, 显然在赋值之前 P （ x↑2+5 ） >9

这一“事实”可形式化地表达为：

D_0 赋值公理

$P_0\{ x:=f \} P$

其中， P_0 是通过以 f 替代 P 中所有出现的 x 而得到的。

实际上， D_0 是一个公理模式，描述了一个无限的公理集合，其中的公理共享着这一样式。

- 关于复合语句的推理规则

考虑以下复合语句：

$$(\ s_1;\ s_2;\ \cdots;\ s_n\ )$$

可以表达为：$(\ s_1;\ \ (\ s_2;\ \ (\cdots;\ (\ \ s_{n-1};\ s_n\ )\ \cdots)\ )\ )$

进一步可记为：$(\ q_1\ ,\ \ q_2\ )$

❶ 如果程序的第一部分的证明结果与程序第二部分的前提是相等的； $P\ \{q_1\}\ R_1 = R_1\{\ q_2\}R$

❷ 并在该前提下，程序产生了预期结果，即 $P\{\ q_1;q_2\}R$

那么，只要程序满足“ P $\{q_1\}$ R_1 和 $R_1\{q_2\}R$”，则整个程序

将产生其预期结果 $P\{q_1;q_2\}R$ 。

这一“事实”可形式化地表达为：

D1 复合规则：

if $P\{q_1\}R_1$ and $R_1\{q_2\}R$ then $P\{q_1;q_2\}R$

- 结果规则—作为该系统的推导规则

if $P\{Q\}R$ and $R \supset S$ then $p\{Q\}S$

if $P\{Q\}R$ and $S \supset P$ then $S\{Q\}R$

- 关于选择语句的推理规则

考虑以下语句：

$$\text{if e then } s_1 \text{ else } s_2;$$

如果在 P 的条件下，执行这一语句，当 e 为真时，具有 Q ；

并在 e 为假时，也具有 Q 。那么

可以把这一“事实”形式化地表述为：

D2　选择规则：

$$\text{if} \quad P \wedge e \{S_1\} Q \quad \text{and} \quad P \wedge \neg e \{S_2\} Q \quad \text{then}$$

$$P \{\text{if e then } s_1 \text{ else } s_2\} Q$$

- 关于重复语句的推理规则

考虑以下语句： While B do S;

假定 :P 是一个断言，只要它在 S 初始时为真，则在 S 完成时也是真的。

从而，在 S 语句的任一次迭代之后， P 一直为真。

另外，在迭代结束时， B 为假。

以上“事实”，可以形式地表述为：

D3 迭代规则：

if $P \wedge B \{S\} P$ then $P\{\text{While B do S}\} \neg B \wedge P$

- 综上，有：

1、空语句　{P }skip{p}

2、赋值　{P_E}x:=E{P}

其中，P_E 表示以 E 代替 P 中每一自由出现的 x 之后所产生的

3、选择　$\frac{\{P \wedge B\}S_1\{Q\},\ \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\ \text{if B then } S_1 \text{ else } S_2\{Q\}}$

4、循环　$\frac{\{P \wedge B\}S\{P\}}{\{P\}\ \text{while B do S}\ \{P \wedge \neg B\}}$

5、复合　$\frac{\{P_1\}S_1\{P_2\},\ \{P_2\}S_2\{P_3\},\ \ldots,\ \{P_n\}S_1\{P_{n+1}\}}{\{P_1\}\ \text{begin } S_1;\ S_2;\ \ldots;\ S_n \text{ end}\ \{P_{n+1}\}}$

6、推理　$\frac{\{P_1\}S\{Q_1\},\ P \to P_1,\ Q_1 \to Q}{\{P_1\}\ \ S\ \ \{Q\}}$

② 程序性质的证明，就是通过使用以上 1-6 ，给出一个

$$\{P\}\ \ S\ \ \{Q\}$$

例如：求 x 除以 y 得其商 q 和余数 r 。该程序可以写为：

((r:=x;q:=0);while y≤r do

r:=r-y;q:=1+q))

为了证明，假定：

1）所有变量为非负整数；

2）所有变量满足以下公理：

$x+y=y+x$　　$x \boxtimes y=y \boxtimes x$

$(x+y)+z=x+(y+z)$　　$(x \boxtimes y) \boxtimes z=x \boxtimes (y \boxtimes z)$

$x \boxtimes (y+z)=x \boxtimes y+x \boxtimes z$

$y \leq x$　$(x-y)+y=x$　　$x+0=x$

$x \boxtimes 0=0$　　$x \boxtimes 1=x$

根据以上公理，可以证明：

$x=x+y \cdot 0$

证明： $x=x$

$= x+0$

$=x+y \cdot 0$

$y \le r \quad r+y \cdot q=(r-y)+y \cdot (1+q)$

证明：

5 $(r-y)+y \cdot (1+q)=(r-y)+(y \cdot 1)+(y \cdot q)$

9 $=(r-y)+(y+y \cdot q)$

3 $=(r-y+y)+y \cdot q$

6 $=r+y \cdot q$ 只要 $y \le r$

根据以上给出的程序可知，其主要性质是 $x=r+y \ast q$ ，

因此，一个证明就是要得到：

$true\{((r:=x;q:=0);\text{while } y \le r \text{ do } (r;=r-y;q:=1+q))\}$

$\neg y \le r \wedge x=r+y \ast q$

1 、 $true \supset x=x+y \ast 0$ 引理 1

2 、 $x=x+y \ast 0 \{r:=x\} x=r+y \ast 0$ 赋值

3 、 $x=r+y \ast 0 \{q:=0\} x=r+y \ast q$ 赋值

4 、 $true\{ r:=x\}x=r+y \ast 0$ 推理

5 、 $true \{ r:=x;q:=0\} x=x+y \ast q$ 复合

6 、 $x=r+y \times q \wedge y \le r \supset x=(r-y)+y \times (1+q)$ 引理 2

7 、 $x=(r-y)+y \times (1+q)\{r:=r-y\}\ x=r+y \times (1+q)$ 赋值

8 、 $x=r+y \times (1+q)\ \{q:=1+q\}\ x=r+\ y \times q$ 赋值

9 、 $x=(r-y)+y \times (1+q)\ \{r:=r-y;q:=1+q\}x=r+\ y \times q$ 复合

10 、 $x=\ r+y \times q \wedge y \le r\ \ \{r:=r-y;q:=1+q\}x=r+\ y \times q$ 推理

11 、 $x=r+\ y \times q$ {while $y \le r$ do ($r:=r-y;q:=1+q$)}

$x=r+\ y \times q$ 循环

12 、 true{((r:=x;q:=0);while $y \le r$ do (r;=r-y;q:=1 + q))}

$\neg y \le r \wedge x=r+y \times q$ 复合

附：程序复杂性度量与错误之间的关系

代码行度量

- 以源代码的行数 -- 度量程序的复杂性。
- Thayer 指出，程序出错率为 100 行源程序中可能存在 0.04 -7 个错误

出错率与源程序行数之间不存在简单的线性关系

Lipow 指出，对于小程序，每行代码（执行部分）的出错率为 1.3%-1.8%

对于大程序，每行代码（执行部分）的出错率为 2.7%-3.2%

对于少于 100 个语句的程序，出错率与源程序行数之间是线性关系的；

随着程序的增大，出错率以非线性方式增长。

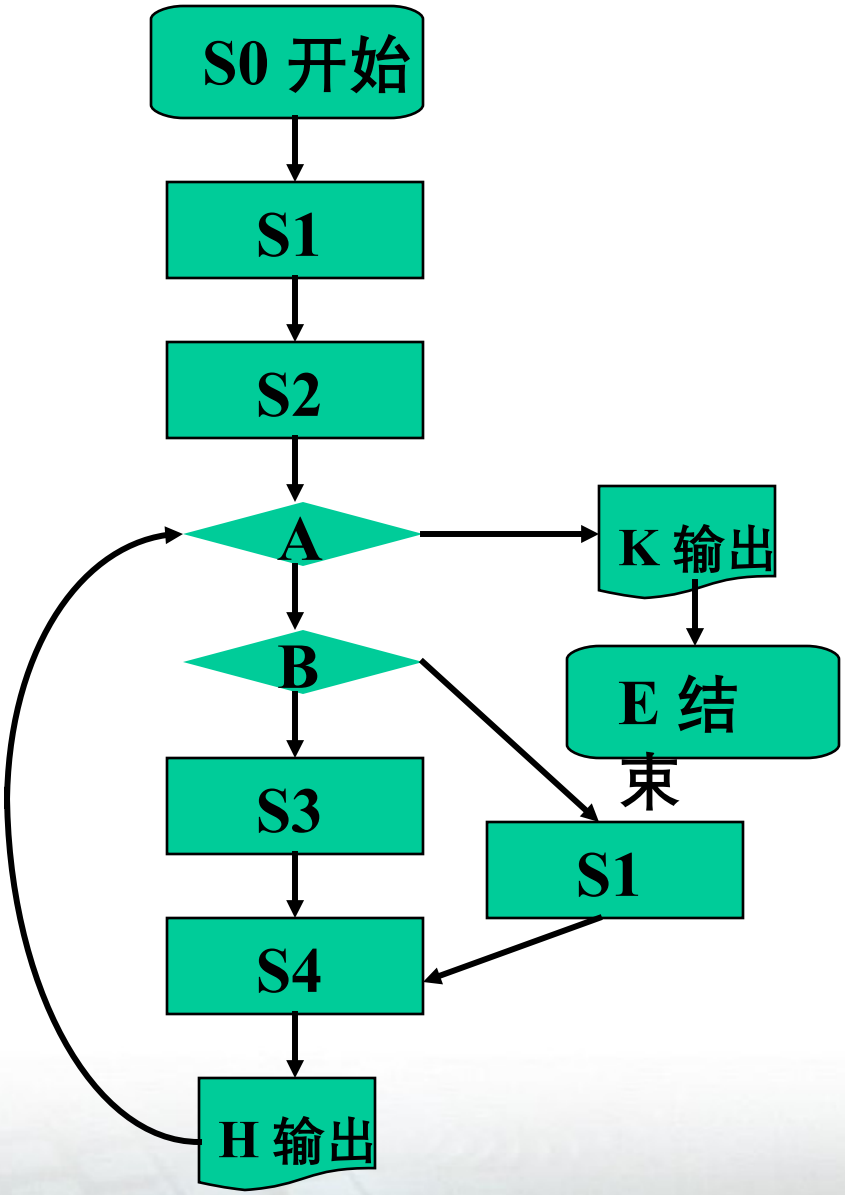
环路复杂性度量 -McCabe 度量法

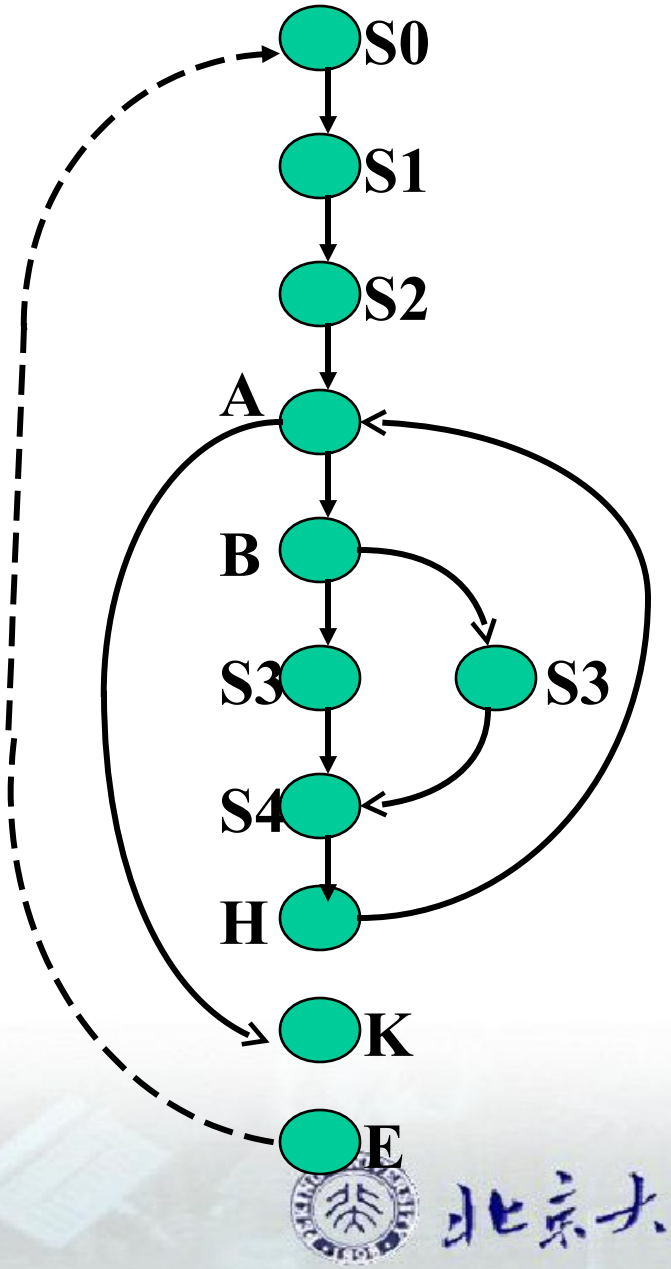
- 基于程序控制流的复杂性度量方法

程序图：把程序流程图中的每个处理符号退化为一个结点，把连接每个处理符号的链转变为连接结点的有向弧，所得到的有向图称为程序图。

程序图的语义：描述程序内部的控制流程，不表现对数据的具体操作以及分支和循环的具体条件。

即该方法把简单的条件语句与复杂的循环语句的复杂性视为一样；把嵌套的 IF 语句与 CASE 语句的复杂性视为一样。





根据图论，在一个强连通的有向图中，环的个数为：

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

其中： m 为图中的弧数， n 为图中的结点数， p 为图中强连通分量数

程序总是连通的，但一般不是强连通的，为此，需加一条从入口到出口的虚线。

据此，上例中的环数为：13-11+1=3

即 McCabe 度量法的度量值为 3 。

注：环路复杂度是可加的。

为软件测试的难度提供了一种定量度量方法。

McCabe 发现，在 276 个程序中，有 23% 的子程序的复杂度超过 10 ，而在这些子程序中发现的错误占总错误的 53% ，因此，他建议，如果程序的复杂度超过 10 ，那么就应该对这一程序进行分解。

McCabe 度量法的主要缺点：

不能区分不同种类控制流的复杂性

简单的条件语句与复杂的循环语句的复杂性等同看待

嵌套的 IF 语句与 CASE 语句的复杂性等同看待；

模块间的接口当成一个简单的分支一样处理；

一个具有 1000 行的顺序程序与一行语句的复杂性相同。

使用：

- 错误估算
- 排错费用估算
- 方案选择等

程序长度、程序量度量与错误数 -Halstead 方法

- 程序长度

$$H=n_1*\log_2 n_1+ n_2*\log_2 n_2$$

其中：H 是程序长度的预测值，而不是程序中的语句条数；

n_1 表示程序中不同运算符（包括关键字）的个数；

n_2 表示程序中不同对象的个数。

注：成对运算符，如“ BEGIN　END” 等都当作单一运算符；

运算对象包括变量名和常数。

- 程序量 V

$$V=N*\log_2 （ n_1+ n_2 ）$$

其中：N 为实际的 Halstead 长度，即 N=N1+N2

N1 为程序中实际出现的运算符总个数

N2 为程序中实际出现的运算对象总个数

n_1 表示程序中不同运算符（包括关键字）的个数；

n_2 表示程序中不同对象的个数。

例如：用 Fortran 语言编写的交换排序程序为：

```
SUBROUTINE SORT(X,N)
DIMENSION X(N)
IF (N.LT.2) RETURN
DO 20 I=2,N
   DO 10 J=1,I
   IF(X(I).GE.X(J)) GO TO 10
   SAVE=X(I)
   X(I)=X(J)
  10 CONTINE
 20  CONTINE
RETURN
END.
```

运算符	计数	运算对象	计数
可执行语句结束	7	x	6
数组下标	6	I	5
=	5	J	4
IF ()	2	N	2
DO	2	2	2
,	2	SAVE	2
程序结束	1	1	1
.LT.	1	n_2=7	N2=22
.GE.	1		
GO TO 10	1		
n_1=10	N1=28		

$H=10*\log_2 10+\log_2 7=52.87$

$N=28+22=50$

$V=(28+22)*\log_2(10+7)=204$

北京大学

- 程序量比率（语言的抽象级别）

$$L = （2/n_1） * （n_2/N_2）$$

其中， $N_2 = n_2 * \log_2 n_2$, 程序量它表明了一个程序的最紧密形式的程序量与实际程序量之比，其倒数 D=1/L ，表明了实现算法的困难程度。有时，用 L 表达语言的抽象级别。

例如：经验得出的一些语言的抽象级别：

语言	L 的平均值
English Prose	2.16
PL/1	1.53
ALGOL68	2.12
FORTRAN	1.14
Assembler	0.88

- 程序员工作量

$E=V/L$ 或 $E=H^* \log_2 (n_1+n_2) * [(n_1 * n_2) / (2* n_2$

- 程序的潜在错误

Halstead 认为，程序中的错误应与程序的容量成正比，因此程序潜在错误的预测公式为：

$$B= (N1+N2) * \log_2 (n_1+n_2) /3000$$

例如：一个程序对 75 个数据库项共访问 1300 次，对 150 个运算符共使用了 1200 次，那么该程序的预测错误数为：

$$B= (1300+1200) * \log_2 (75+150) /3000$$

$$= 6.5$$

即该程序可能含有 6-7 个错误。