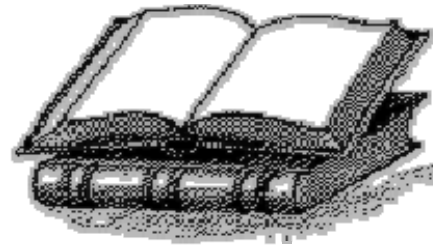




《软件工程》

Software Engineering





5.1 编码

- 高质量代码？
 - 代码规范
- 怎样达到高质量
 - 代码复审
 - 结对编程
- 伟大的代码？
 - 怎么做到？
- 如何影响别人



代码质量

- 我们写代码给谁看？ machine, or people?

- Both



代码还是给人读/维护

- 你的代码决定了你的RP
- 软件团队就是一个小社会
 - 不要搞得大家看不起你的 RP
- 越早发现问题约好
- 代码复审越早约好
- 推到极致
 - 当你写的时候，就有人复审 = Pair Programming



Broken Window (破窗理论)

- Broken Window Theory

- ☐ A few broken windows ->
- ☐ more broken windows ->
- ☐ vandalizing the house
- ☐ Litter on a sidewalk ->
- ☐ more litter ->
- ☐ becomes an dumping ground ->
- ☐ unsafe street

- Example - New York City in 1980's

- ☐ Start from small violations, strictly enforce the law
- ☐ Clean up subway and subway cars, every line, every car, everyday



破窗理论在软件工程中

- 一样存在!
- 如果小问题不修复， 那何必注意细节？
- 如果长时间都没有测试和每日构建， 为何要不断提到代码质量？
- 如果我签入代码前， 整个项目就编译不了， 那我签入一些编译不了的代码也无妨



What's solid code?

- Professionals need solid code
- ZERO defect
- easy to understand
 - Define your assumption explicitly
- Easy to maintain
 - Define your dependency explicitly



Header Comment

- Comment What (high level) and How
- Use consistent format
- Document owner, important changes
- Keep other detail information in check-in comments



In-body comment

- Comment How, not What
 - sparse enough that you only notice them when you need them,
 - but detailed enough that they make the surprises apparent.
- Avoid common knowledge
 - don't tell us about common language features, standard library functions,
 - or standard idioms like iteration.
 - Dev can get such information from the code itself
- Comment style in language Text book is different from Comment Style in product
 - Text book: explain basic ideas, as part of teaching



Code Review

名 称	形 式	目 的
自我复审 (self review)	自己 vs. 自己	用同伴复审的标准来要求自己。不一定最有效，因为开发者对自己总是过于自信。如果能持之以恒，则对个人有很大好处
同伴复审 (Peer Review)	复审者 vs. 开发者	简便易行
团队复审 (Team Review)	团队 vs. 开发者	有比较严格的规定和流程，用于关键的代码，以及复审后不再更新的代码。 覆盖率高——有很多双眼睛盯着程序。但是有可能效率不高（全体人员都要到会）



复审应该多严格？

- 完美主义的复审？ 大发脾气的复审？
 - 例子 Linus Torvals 对于注释的格式
- 如果复审者没有发现任何错误，这个代码复审还有价值么？
- 有
 - 让你把所有相关文档都准备好
 - 分享了知识
 - 别人学到了很多（多于你讲的话）
 - 当给别人描述代码逻辑上，很多人意识到了自己的错误。



代码复审检查表

■ 概要部分

- ☐ 1) 代码符合需求和规格说明么？ 2) 代码设计是否考虑周全？
- ☐ 3) 代码可读性如何？ 4) 代码容易维护么？ 5) 代码的每一行都执行并检查过了吗？

■ 设计规范部分

- ☐ 1) 设计是否遵从已知的设计模式或项目中常用的模式？
- ☐ 2) 有没有硬编码或字符串/ 数字等存在？
- ☐ 3) 代码有没有依赖于某一平台，是否会影响将来的移植（如Win32到Win64）？
- ☐ 4) 开发者新写的代码能否用已有的Library/SDK/Framework中的功能实现？在本项目中 是否存在类似的功能可以调用而不用全部重新实现？
- ☐ 5) 有没有无用的代码可以清除？



代码复审检查表

具体代码部分

- 1) 有没有对错误进行处理？对于调用的外部函数，是否检查了返回值或处理了异常？
- 2) 参数传递有无错误，字符串的长度是字节的长度还是字符（可能是单/双字节）的长度，是以0开始计数还是以1开始计数？
- 3) 边界条件是如何处理的？**switch**语句的**default**分支是如何处理的？循环有没有可能出现死循环？
- 4) 有没有使用断言（**Assert**）来保证我们认为不变的条件真的得到满足？
- 5) 对资源的利用，是在哪里申请，在哪里释放的？有无可能存在资源泄漏（内存、文件、各种**GUI**资源、数据库访问的连接，等等）？有没有优化的空间？
- 6) 数据结构中有没有用不到的元素？



代码复审检查表

■ 效能

- ☐ 1) 代码的效能（**Performance**）如何？最坏的情况是怎样的？
- ☐ 2) 代码中，特别是循环中是否有明显可优化的部分（**C++**中反复创建类，**C#**中 **string** 的操作是否能用**StringBuilder**来优化）？
- ☐ 3) 对于系统和网络的调用是否会超时？如何处理？

■ 可读性

- ☐ 代码可读性如何？有没有足够的注释？

■ 可测试性

- ☐ 代码是否需要更新或创建新的单元测试？针对特定领域的开发（如数据库、网页、多线程 等），可以整理专门的检查表。



5.2 测试

在开发软件的过程中，我们使用了保证软件质量的方法分析、设计和实现软件，但难免还会在工作中犯错误。这样，在软件产品中就会隐藏着许多错误和缺陷。特别是对于规模大、复杂性高的软件更是如此。在这些错误中，有些是致命性的错误如果不排除，就会导致生命与财产的重大损失。

DO 5 I = 1, 3

→

DO 5 I = 1. 3



一、 软件测试的基础

什么是软件测试

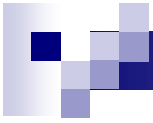
软件测试是为了发现错误而执行程序的过程。或者说，软件测试是根据软件开发各阶段的规格说明和程序内部结构而精心设计的一批测试用例（即输入数据及预期的输出结果），并利用这些测试用例去运行程序，以发现程序错误的过程。



软件测试人员

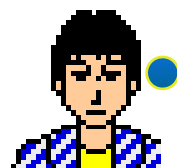
测试工具软件开发工程师
(Software Development Engineer in Test, 简称SDE/T)

软件测试工程师
(Software Test Engineer , 简称STE)



SDE/T

负责写测试工具代码，并利用测试工具对软件进行测试；或者开发测试工具为软件测试工程师服务。

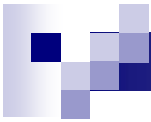


STE

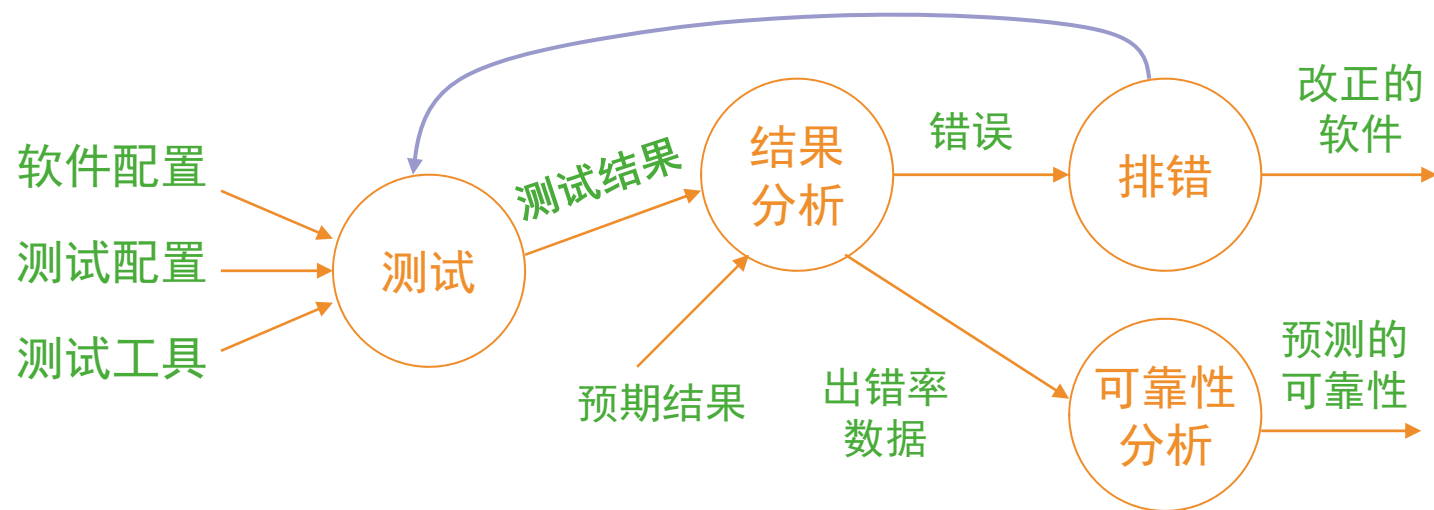
负责理解产品的功能要求，
然后对其进行测试，检查软件
有没有错误(**Bug**)，决定软件
是否具有稳定性，并写出
相应的测试规范和测试案例



软件测试人员的任务很清楚，就是站在使用者的角度上，通过不断地使用和攻击刚开发出来的软件产品尽量多地找出产品存在的问题，也就是我们所称的 Bug 。



软件测试信息流





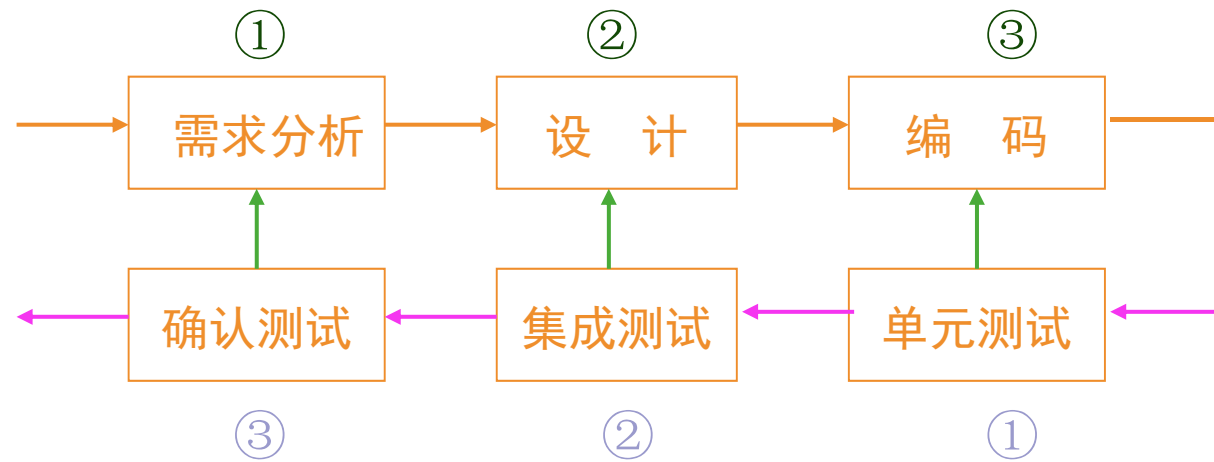
软件测试的对象

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。因此，需求分析、概要设计、详细设计以及程序编码等所得到的文档资料，包括需求规格说明、概要设计说明、详细设计规格说明以及源程序，都应成为软件测试的对象。



测试与软件开发阶段的关系

软件开发过程是一个自顶向下、逐步细化的过程，而测试则是依相反的顺序安排的，**自底向上、逐步集成**的过程。低一级为上一级测试准备条件。





有关测试的误解

测试的时候尽量用**Debug**版本，便于发现 **bug**

- 如果你的目的是尽快让问题显现，尽快找到问题，那我建议用**Debug**版本，“尽快发现问题”在软件开发周期的早期特别重要。
- 如果你的目的是尽可能测试用户所看到的软件，则用**Release**版本，这在软件开发的后期很有价值，特别是在运行效能 (**performance**) 和压力 (**stress**) 测试的时候。

Different kind of test





二、 软件测试设计的方法

软件的测试设计与软件产品的设计一样，是一项需要花费许多人力和时间的工作，我们希望以最少量的时间和人力，最大可能地发现最多的错误。

测试技术

1、 白盒测试
(White Box Testing)

2、 黑盒测试
(Black Box Testing)



白盒测试(White Box Testing)

也叫**玻璃盒测试**(Glass Box Testing)。

对软件的过程性细节做细致的检查。这一方法是把测试对象看作一个打开的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，来设计或选择**测试用例**，对程序所有**逻辑路径**进行测试。



白盒测试 的内容

对程序模块的所有独立
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
都能至少测试一次。

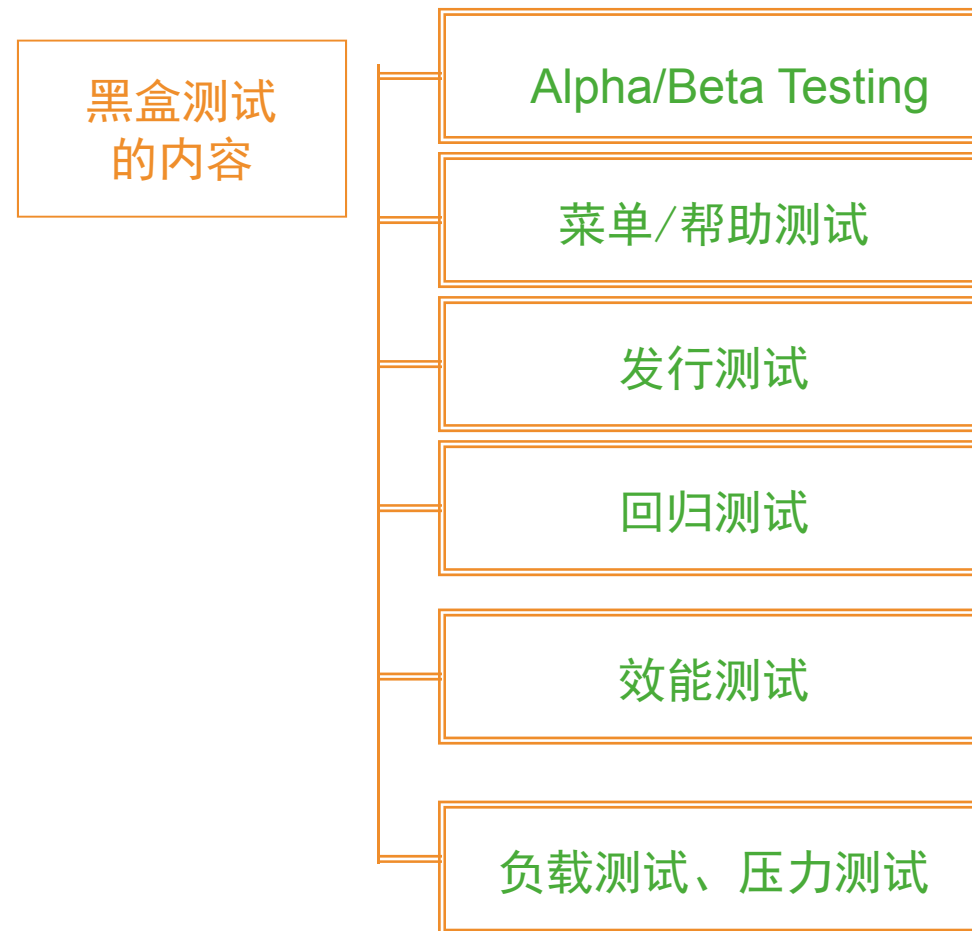
在循环的边界和运行边
界限内执行循环体

测试内部数据结构的有效
性。



黑盒测试(Black Box Testing)


已知产品的功能设计规格，可以进行测试证明每个实现了的功能是否符合要求。





Example -如何测试效能

- 效能测试：在100个用户的情况下，产品搜索必须在3秒钟内返回结果。
- 负载测试：在2 000 用户的情况下，产品搜索必须在5秒钟内返回结果。
- 压力测试：在高峰压力（4 000 用户）持续48小时的情况下，产品搜索的返回时间必须保持稳定。系统不至于崩溃。



Example - 旅客列车

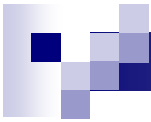
- 效能测试：
 - 在80%上座率的情况下，
 - 期望：列车按时到达，并且乘客享受到优质服务（每小时清洁，保障水，食物，卫生）。乘务员不要太累。
- 负载测试：
 - 在100%上座率的情况下，
 - 期望：列车大部分按时到达，乘客享受到基本服务。乘务员的疲劳在可恢复范围内。
- 压力测试：
 - 在高峰压力是200%上座率，全国铁路系统增加20%列车，持续15天的情况下（春运）
 - 期望：列车能到站，无出轨；乘客能活着下车，系统不至于崩溃。乘务员也能活着下车。



三、 白盒测试用例的设计

一、逻辑覆盖法

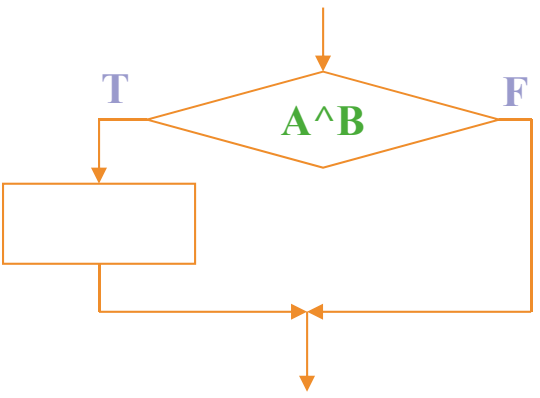
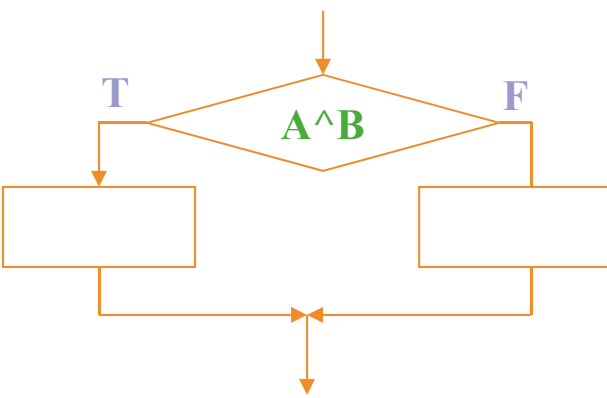
逻辑覆盖测试方法通常采用流程图来设计测试用例，它考察的重点是图中的判定框，因为这些判定通常是与选择结构有关或是与循环结构有关，是决定程序结构的关键成分。



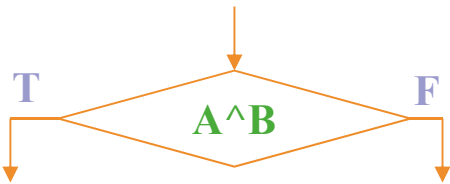
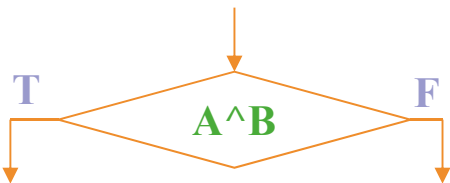
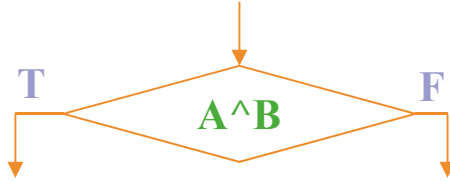
逻辑覆盖测试的5种标准

发现错误的 能力	标 准	含 义
1(弱)	语句覆盖	每条语句至少执行一次
2	判定覆盖	每一判定的每个分支至少执行一次
3	条件覆盖	每一判定中的每个条件，分别按“真”、“假”至少各执行一次
4	判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
5 (强)	条件组合覆盖	求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次

示例

覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		$A^B = .T.$
判定覆盖		$A^B = .T.$ $A^B = .F.$



覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		$A=.T. \quad A=.F.$ $B=.T. \quad B=.F.$
判定/条件覆盖		$A \wedge B=.T. \quad A \wedge B=.F.$ $A=.T. \quad A=.F. \quad B=.T. \quad B=.F.$
条件组合覆盖		$A=.T. \quad \wedge \quad B=.T.$ $A=.T. \quad \wedge \quad B=.F.$ $A=.F. \quad \wedge \quad B=.T.$ $A=.F. \quad \wedge \quad B=.F.$



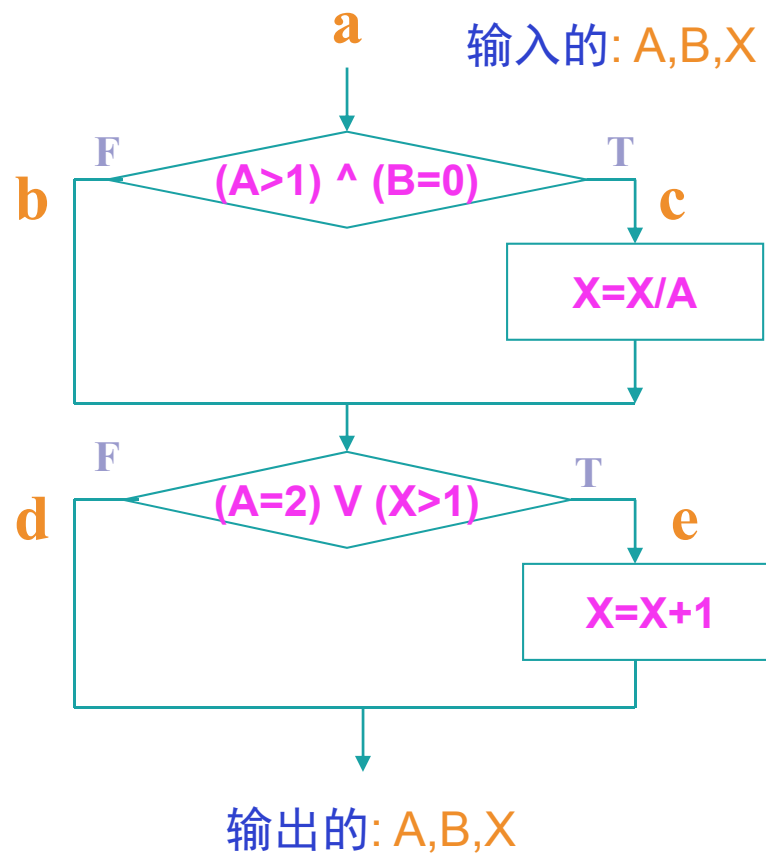
语句覆盖

ace (L1)

abd (L2)

abe (L3)

acd (L4)



满足语句覆盖的
测试用例如下

2, 0, 4

2, 0, 3

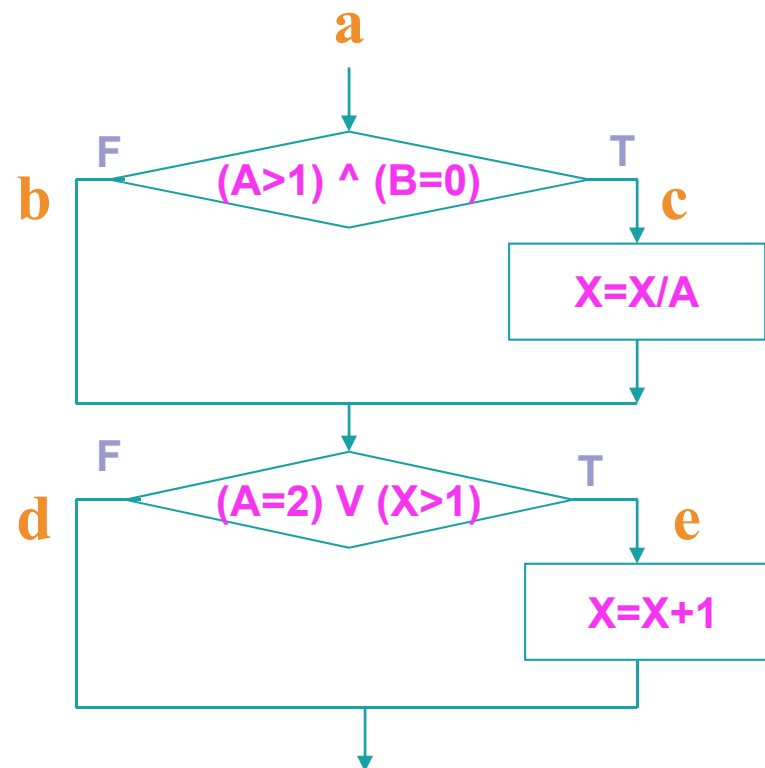
a -- c -- e

L1



判定覆盖(分支覆盖)

所谓的判定覆盖就是设计若干个测试用例，运行被测程序，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。



取“真”分支
测试用例如下

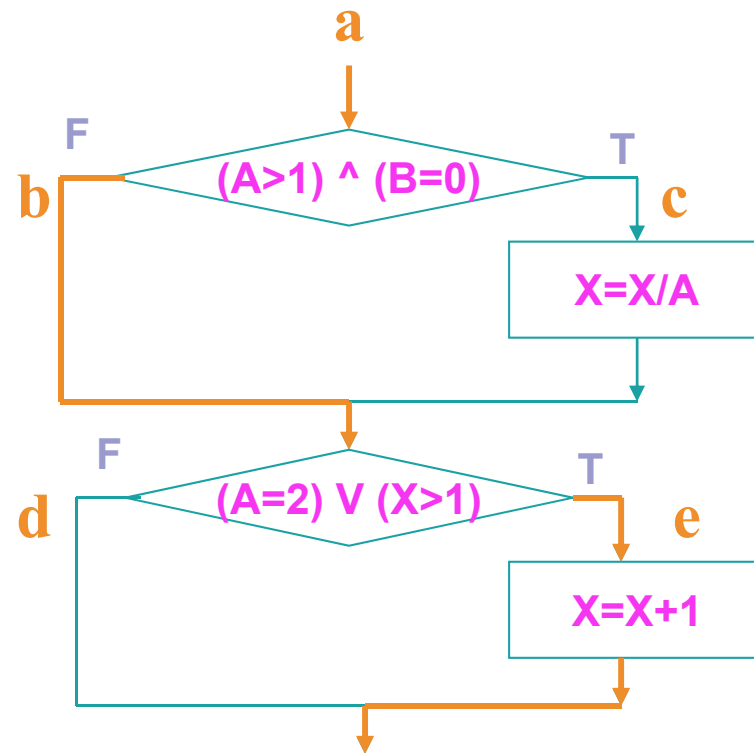
2, 0, 4
2, 0, 3

1, 1, 1
1, 1, 1

取“假”分支
测试用例如下

a -- c -- e L1

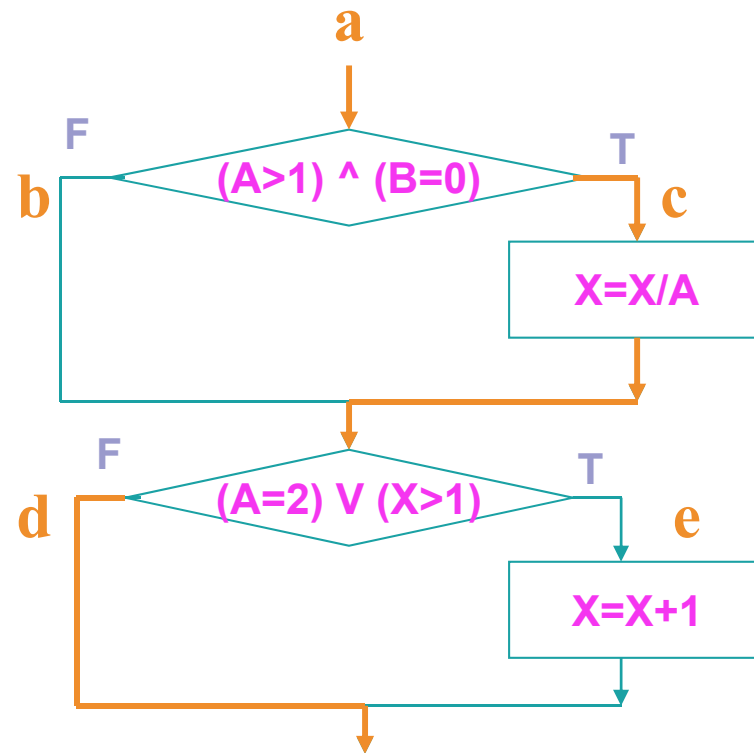
a -- b -- d L2



取“真假”分支
测试用例如下

2	1	1
2	1	2

a -- b -- e L3



取“真假”分支
测试用例如下

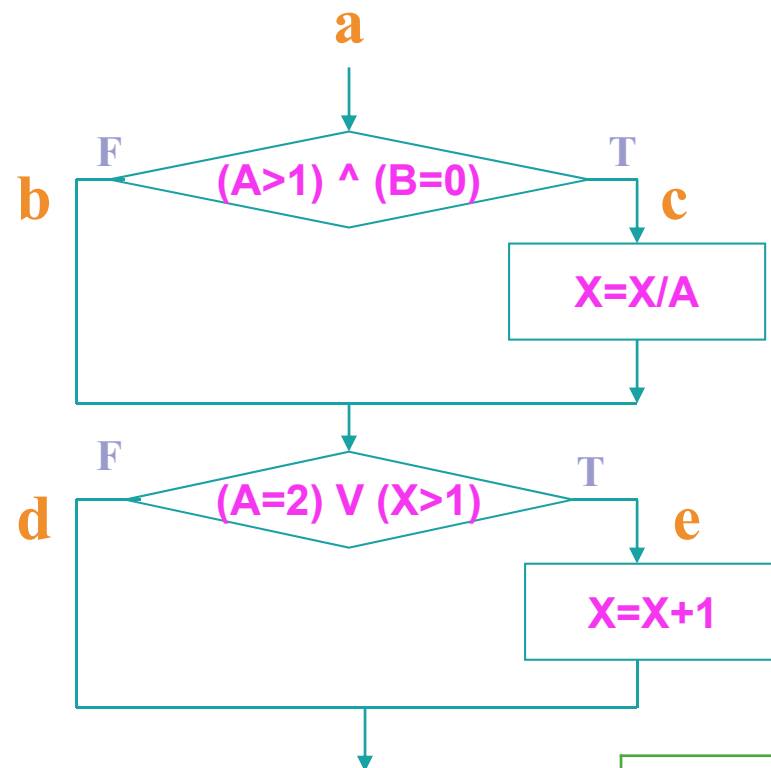
3	0	3
3	0	1

a -- c -- d L4



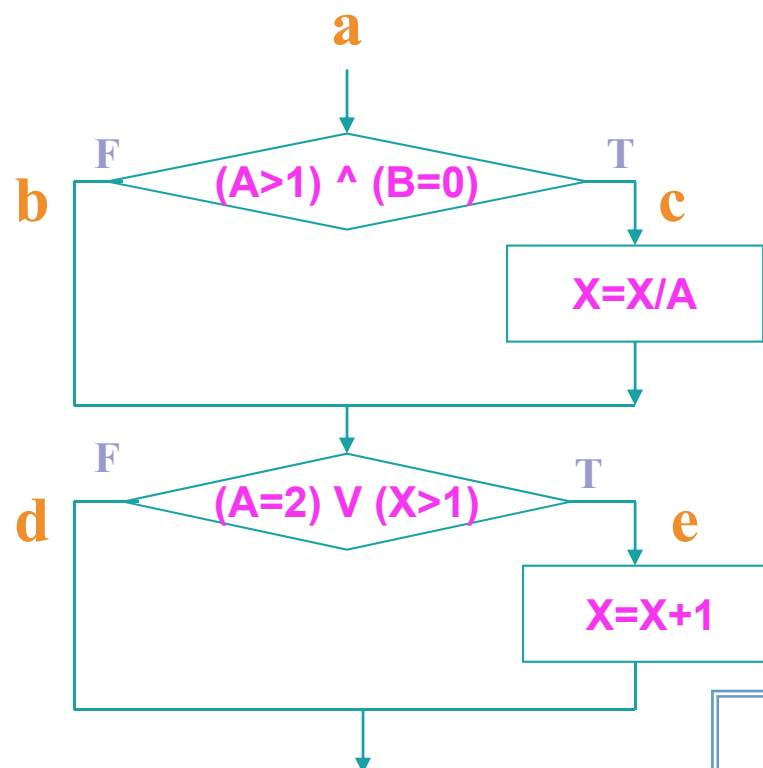
条件覆盖

所谓的条件覆盖就是设计若干个测试用例，运行被测程序，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。



设条件的取值标记

判断	条件	取真值	取假值
判断 (一)	A>1	T1	$\overline{T1}$
	B=0	T2	$\overline{T2}$
判断 (二)	A=2	T3	$\overline{T3}$
	X>1	T4	$\overline{T4}$

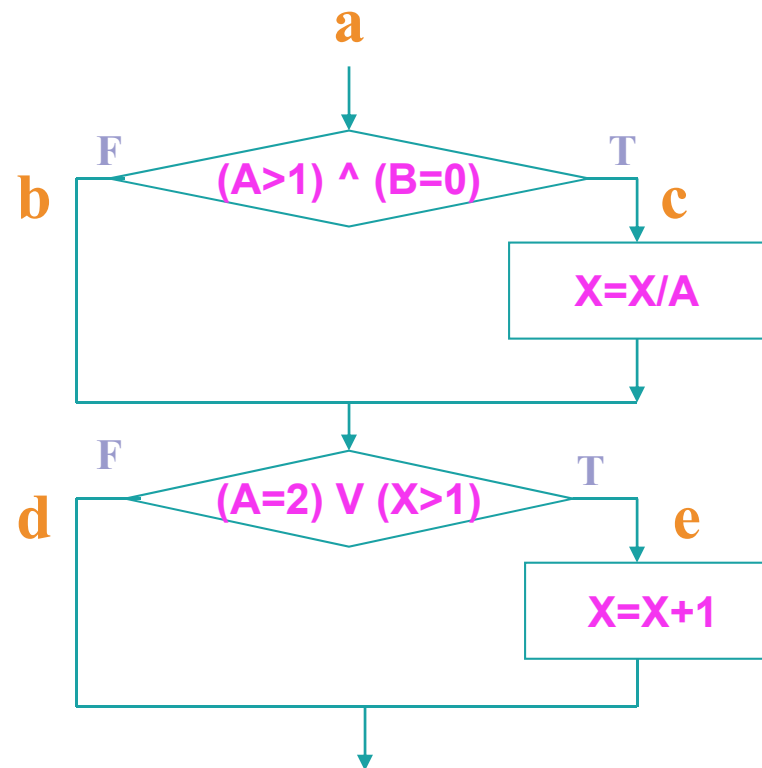


设条件的取值标记

判断	条件	取真值	取假值
判断 (一)	$A>1$	T1	$\overline{T1}$
	$B=0$	T2	$\overline{T2}$
判断 (二)	$A=2$	T3	$\overline{T3}$
	$X>1$	T4	$\overline{T4}$

条件覆盖可选取的
(第一组测试用例) 如下表

测试用例	通过路径	条件取值	覆盖分支
$(2,0,4), (2,0,3)$	ace(L1)	T1 T2 T3 T4	c,e
$(1,0,1), (1,0,1)$	abd(L2)	$\overline{T1}$ T2 $\overline{T3}$ $\overline{T4}$	b,d



设条件的取值标记

判断	条件	取真值	取假值
判断 (一)	$A>1$	T1	$\overline{T1}$
	$B=0$	T2	$\overline{T2}$
判断 (二)	$A=2$	T3	$\overline{T3}$
	$X>1$	T4	$\overline{T4}$

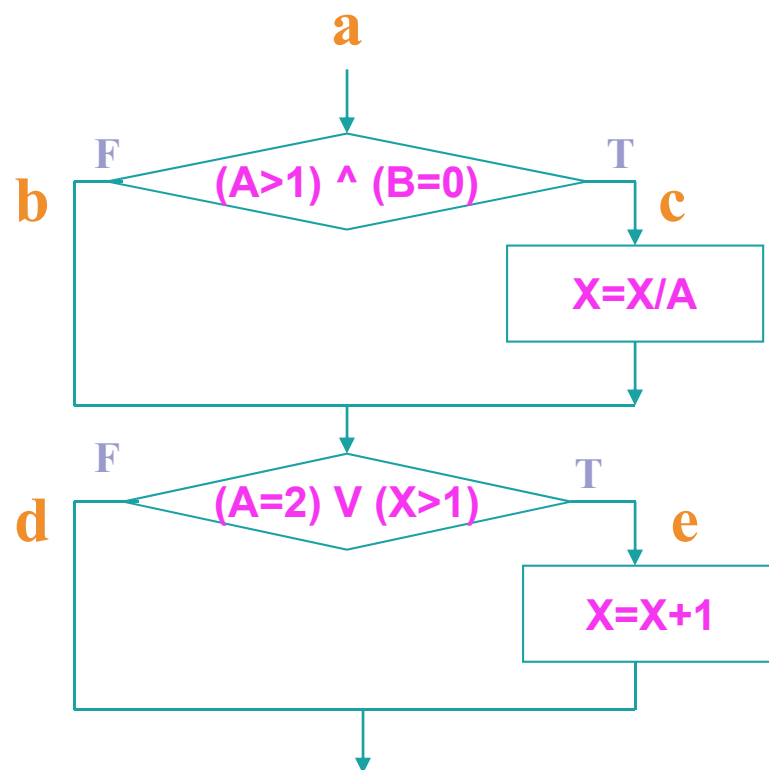
条件覆盖可选取的 (第二组测试用例) 如下表

测试用例	通过路径	条件取值	覆盖分支
(1,0,3),(1,0,4)	abe(L3)	$\overline{T1}$ T2 $\overline{T3}$ T4	b,e
(2,1,1),(2,1,2)	abe(L3)	T1 $\overline{T2}$ T3 $\overline{T4}$	b,e



判定/条件覆盖

所谓的判定/条件覆盖就是设计足够的测试用例，使得 判断中每个条件的所有可能取值至少执行一次， 同时每个判断本身的所有可能判断结果至少执行一次。



设条件的取值标记

判断	条件	取真值	取假值
判断 (一)	A>1	T1	$\overline{T1}$
	B=0	T2	$\overline{T2}$
判断 (二)	A=2	T3	$\overline{T3}$
	X>1	T4	$\overline{T4}$

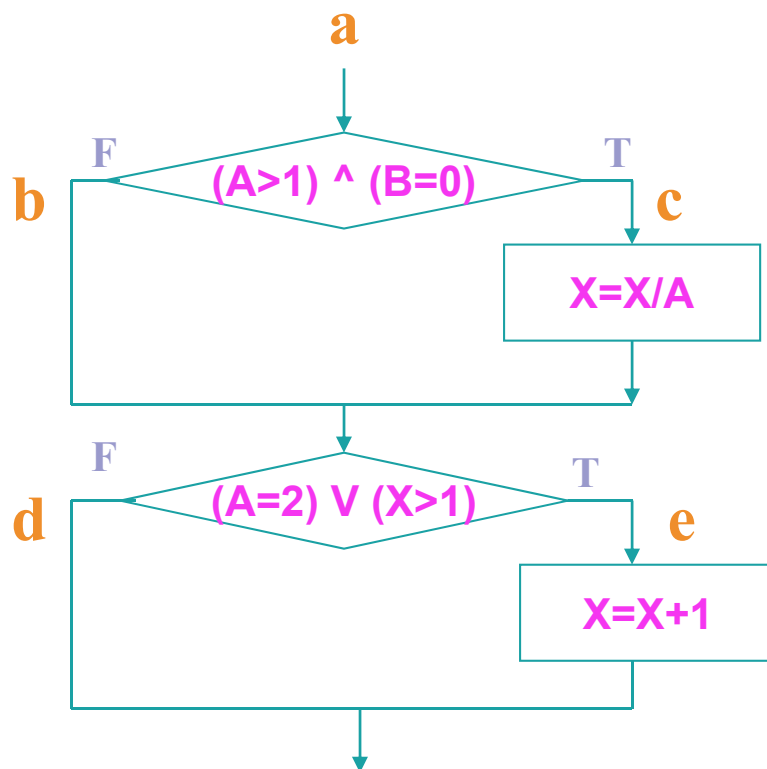
判定/条件覆盖可选取的 测试用例 如下表

测试用例	通过路径	条件取值	覆盖分支
(2,0,4),(2,0,3)	ace(L1)	T1 T2 T3 T4	c,e
(1,1,1),(1,1,1)	abd(L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$	b,d



条件组合覆盖

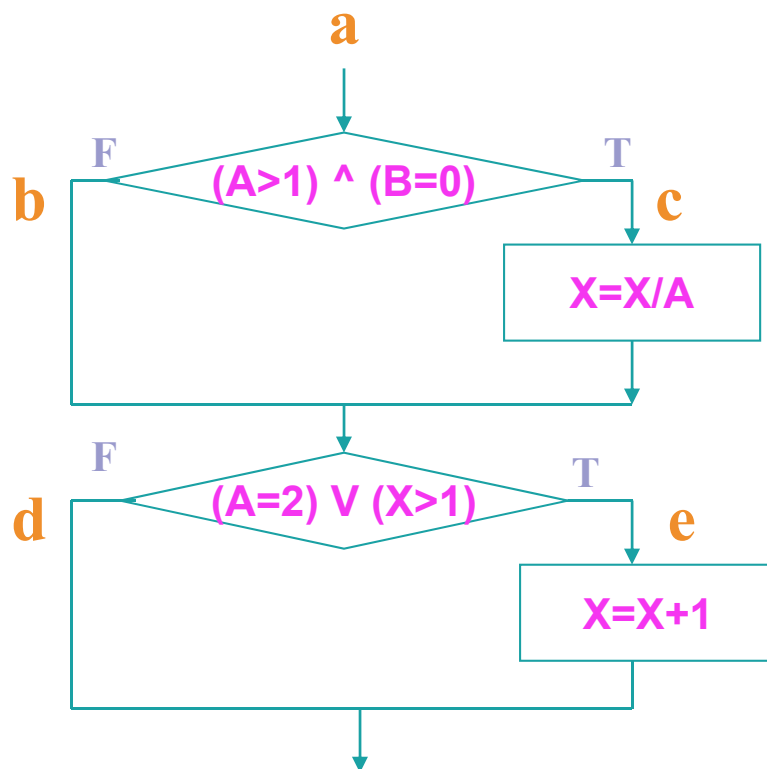
所谓的 条件组合覆盖就是设计足够的测试用例，运行被测程序，使得 每个判断的所有可能的条件取值组合至少执行一次。



设条件的取值标记

判断	条件	取真值	取假值
判断 (一)	A>1	T1	$\overline{T1}$
	B=0	T2	$\overline{T2}$
判断 (二)	A=2	T3	$\overline{T3}$
	X>1	T4	$\overline{T4}$

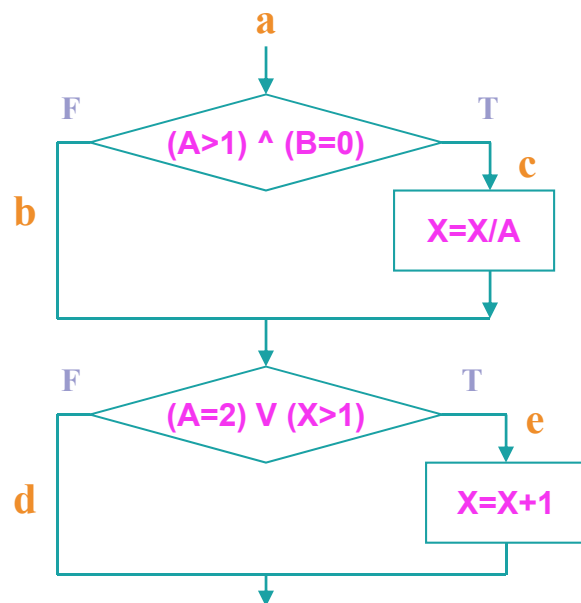
条件	标记	第一个判断取真假分支
A>1 B=0	T1 T2	① 取真分支
A>1 B≠0	T1 $\overline{T2}$	② 取假分支
A≠1 B=0	$\overline{T1}$ T2	③ 取假分支
A≠1 B≠0	$\overline{T1}$ $\overline{T2}$	④ 取假分支



设条件的取值标记

判断	条件	取真值	取假值
判断 (一)	$A>1$	T1	$\overline{T1}$
	$B=0$	T2	$\overline{T2}$
判断 (二)	$A=2$	T3	$\overline{T3}$
	$X>1$	T4	$\overline{T4}$

条件	标记	第二个判断取真假分支
$A=2$ $X>1$	T3 T4	⑤ 取真分支
$A=2$ $X \ngtr 1$	T3 $\overline{T4}$	⑥ 取真分支
$A \neq 2$ $X>1$	$\overline{T3}$ T4	⑦ 取真分支
$A \neq 2$ $X \ngtr 1$	$\overline{T3}$ $\overline{T4}$	⑧ 取假分支



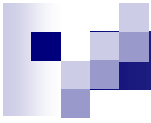
条件	标记	第一个判断取真假分支
$A > 1 \ B = 0$	T1 T2	① 取真分支
$A > 1 \ B \neq 0$	T1 $\overline{T2}$	② 取假分支
$A \ngtr 1 \ B = 0$	$\overline{T1}$ T2	③ 取假分支
$A \ngtr 1 \ B \neq 0$	$\overline{T1} \ \overline{T2}$	④ 取假分支

测试用例	通过路径	条件取值	覆盖组合号
$(2, 0, 4), (2, 0, 3)$	ace L1	T1 T2 T3 T4	①, ⑤
$(2, 1, 1), (2, 1, 2)$	abe L3	T1 $\overline{T2}$ T3 $\overline{T4}$	②, ⑥
$(1, 0, 3), (1, 0, 4)$	abe L3	$\overline{T1}$ T2 $\overline{T3}$ T4	③, ⑦
$(1, 1, 1), (1, 1, 1)$	abd L2	$\overline{T1} \ \overline{T2} \ \overline{T3} \ \overline{T4}$	④, ⑧



二、路径测试法

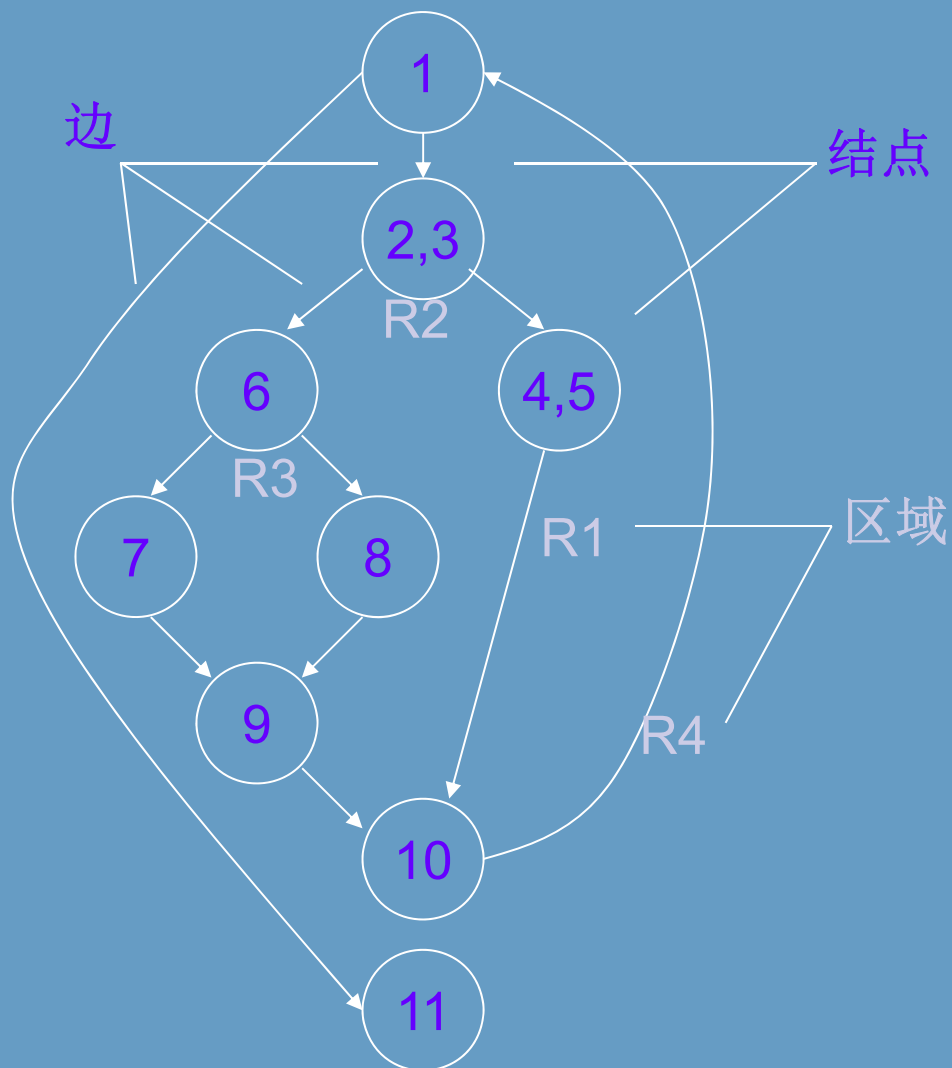
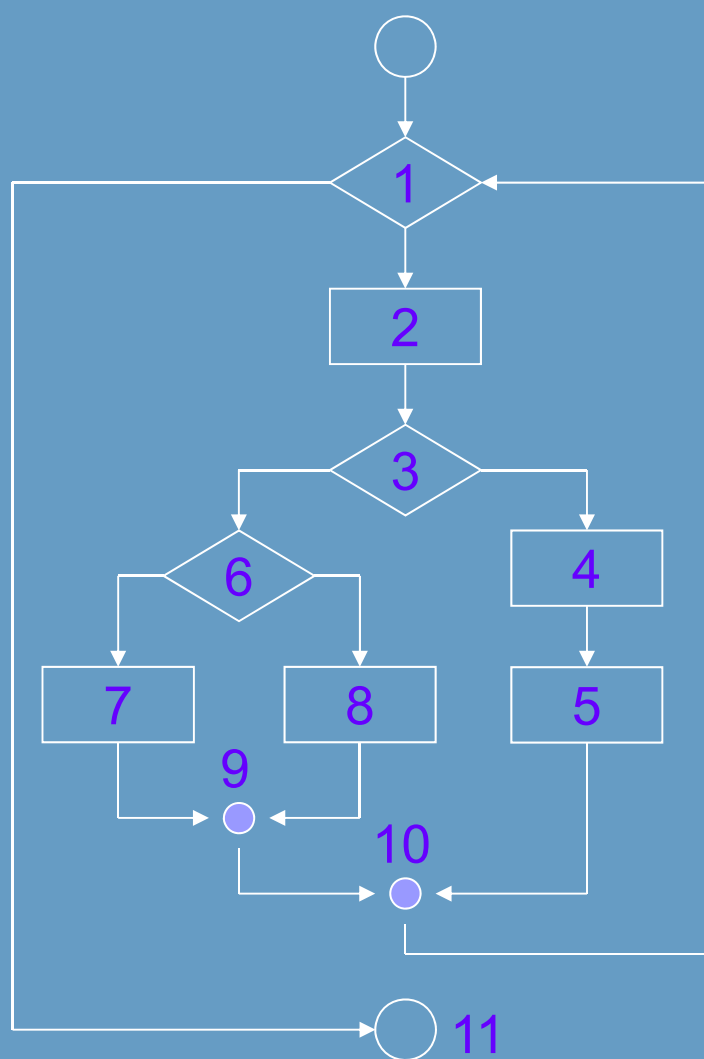
路径测试就是设计足够的测试用例，覆盖程序中每一条可能的程序执行路径至少测试一次，如果程序中含有循环(在程序图中表现为环)则每个循环至少执行一次。



程序的控制流图

- 使用目的：突出表示程序的控制流
- 仅仅描述程序的控制流程，完全不表现对数据的具体操作和分支或循环的具体条件

程序流程图→程序控制流图





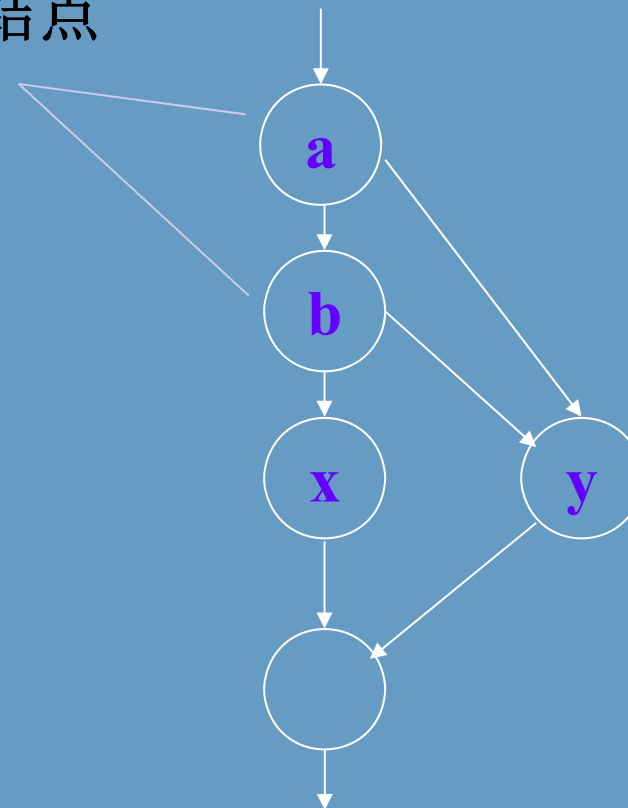
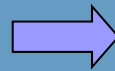
程序的控制流图

1. 一个结点对应一条或多条语句
2. 流程图中，一个顺序执行的处理框序列和一个菱形判断框，可以映射成控制流图中的一个结点
3. 一条边必须终止于一个结点，哪怕是空语句的结点
4. 在计算流图中的区域数时，应包括图外部未被围起来的敞开区域
5. 若判断框中的条件是复合条件时，则需要将复合条件的判断分解为一系列只有单个条件的嵌套判断

复合条件（and）下的流图

...
If a and b
then procedure x
else procedure y
endif
...

判定结点

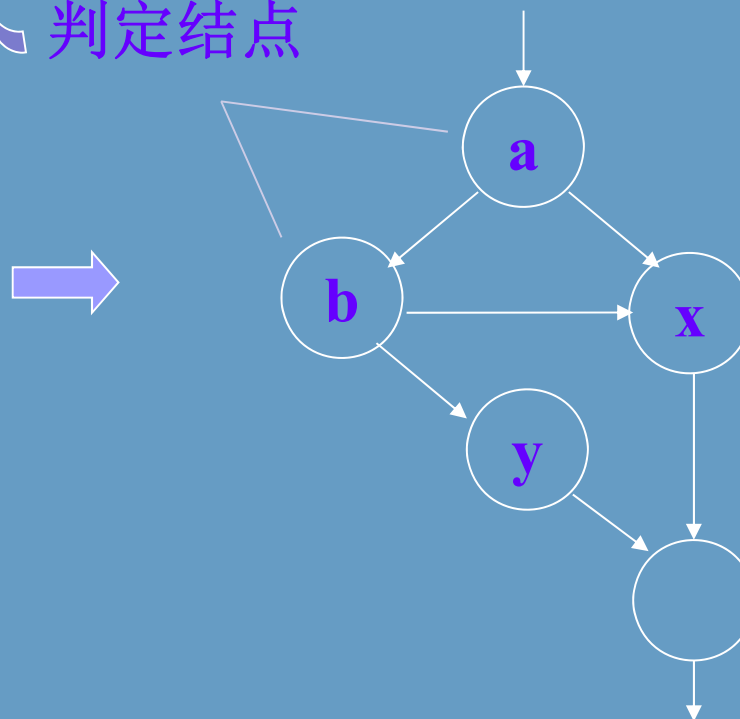



复合条件（or）下的流图

从每个判定结点引出两条或多条边

判定结点

...
If a or b
 then procedure x
 else procedure y
endif
...





程序环路复杂性 $V(G)$

- 度量程序的逻辑复杂性，通过流图导出
- 计算方法
 1. $V(G) =$ 流图中的区域数
 2. $V(G) = P + 1$ ，其中 P 判断结点数
 3. $V(G) = E - N + 2$ ，其中 E 边数， N 结点数

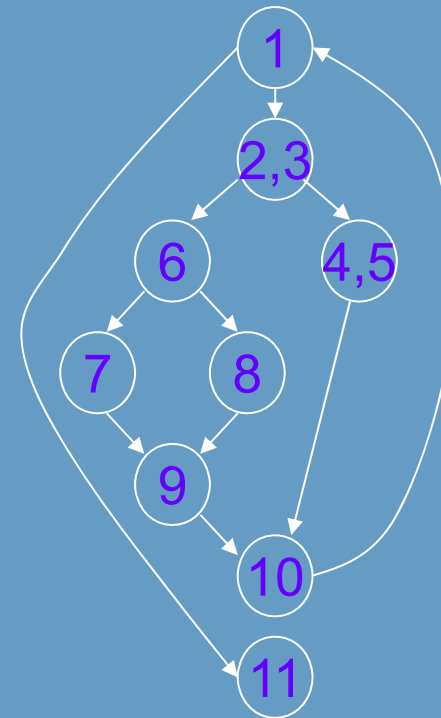


独立路径

- 指包括一组以前没有处理的语句或条件的一条路径。
从流图来看，一条独立路径至少包含有一条在其他独立路径中从未有过的边的路径
- 独立路径条数 = $V(G)$
 - 是确保程序中每个可执行语句至少执行一次所需的测试用例数目的上界
 - 基本路径集不唯一，但条数是唯一的

上图中的一组独立路径

- 共4条独立路径
 - 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
- 这4条独立路径组成了流图中的一个基本路径集





导出测试用例

主要步骤：

- ① 以详细设计或源代码为基础，导出程序的流图
- ② 计算得到的流图 G 的环路复杂性 $V(G)$
- ③ 确定线性无关的路径的基本集合
- ④ 生成测试用例，确保基本路径集中每条路径的执行



作业

- 写一个函数判断三角形的种类。参数a, b, c分别为三角形的三边，返回的参数值：
 - * 为0，表示非三角形；
 - * 为1，表示普通三角形；
 - * 为2，表示等腰三角形；
 - * 为3，表示等边三角形。

设计语句覆盖测试用例、判定覆盖测试用例、条件覆盖测试用例、条件-判定覆盖测试用例、条件组合覆盖测试用例，给出：

用例编号	输入	期望输出	覆盖分支	测试结果截图
------	----	------	------	--------

- * 提交的文档中需包括函数、函数流程图、以及测试用例。



四、黑盒测试用例的设计

黑盒测试法是根据被测程序功能来进行测试，所以通常也称为功能测试。用黑盒测试法设计测试用例，有4种常用技术：

等价分类法

边界值分析

错误猜测法

因果图法



(一) 等价分类法

所谓等价分类，就是把输入数据的可能值划分为若干**等价类**（等价类是指某个输入域的子集合。在该集合中，各个输入数据对于揭露程序中的错误都是等价的）。因此，可以把全部输入数据合理地划分为若干等价类，在每一个等价类中取一个数据作为测试的输入条件，这样就可以少量的代表性测试数据，来取得较好的测试结果。



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。利用它可以检验程序是否实现预先规定的功能和性能。



无效等价类

是指对于程序的规格说明来说，是不合理的，是无意义的输入数据构成的集合。程序员主要利用这一类测试用例来检查程序中功能和性能的实现是否不符合规格说明要求。



确定等价类的原则:

1、如果输入条件规定了取值范围，或者是值的个数，则可以确立一个有效等价类和两个无效等价类。

例如：... .. 序号值可以从 1到999

一个有效等价类： $1 \leq \text{序号值} \leq 999$

两个无效等价类： 序号值 < 1

序号值 > 999



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 C 语言中对变量标识符规定为“以字母打头的... 串”。所有以字母打头的构成为有效等价类；而不在集合内(不以字母打头)归于无效等价。



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是一组值，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再针对这组确立一个无效等价类，它应是所有不允许输入值的集合。

例如：在教师分房方案中规定对教授、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，以及 1个无效等价类它应是所有不符合以上身份的人员的输入值的集合。



5、如果规定了输入数据必须遵守的规则，则可以确定一个有效等价类(符合规则)，和若干个无效等价类(从不同角度违反规则)。

例如：在C语言中规定了“一个语句必须以分号 ‘;’ 作为结束”，这时，可以确定一个有效等价类，以 “;” 结束，而若干个无效等价类应以 “: , \” 等。



6、如果确知，已划分的等价类中各元素在程序中的处理方式不同，则应将此等价类进一步划分成更小的等价类。



采用这一技术要注意以下两点：

1、划分等价类不仅要考虑代表“有效”输入值的有效等价类，还需考虑代表“无效”输入值的无效等价类。

2、每一无效等价类至少要用一个测试用例，不然就可能漏掉某一类错误，但允许若干有效等价类合用同一个测试用例，以便进一步减少测试的次数。



确立测试用例

输入条件	有效等价类	无效等价类
.....



确立测试 用例原则

为每一个等价类规定一个唯一的编号。

设计一个新的测试用例，使其尽可能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。



请利用等价分类法为以下提供的内容设计测试用例

在某一个PASCAL 语言版本中规定

- 1、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；
- 2、标识符必须先说明，后使用；
- 3、在同一个说明语句中，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)， 数字(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)



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)

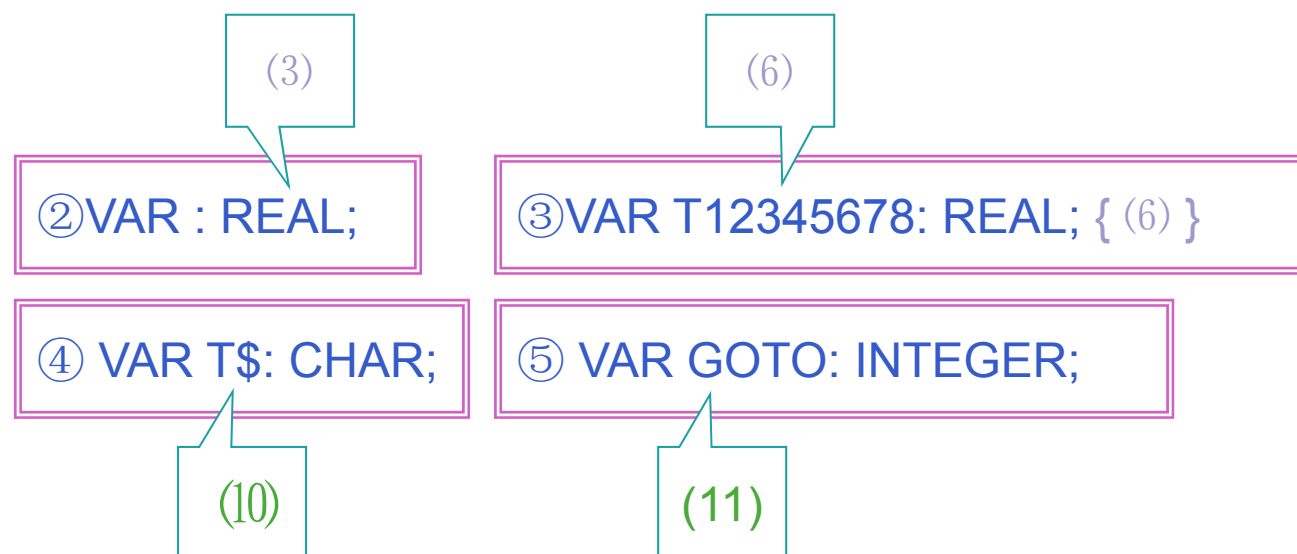
(1)

(2) (4) (8) (9) (12) (14)

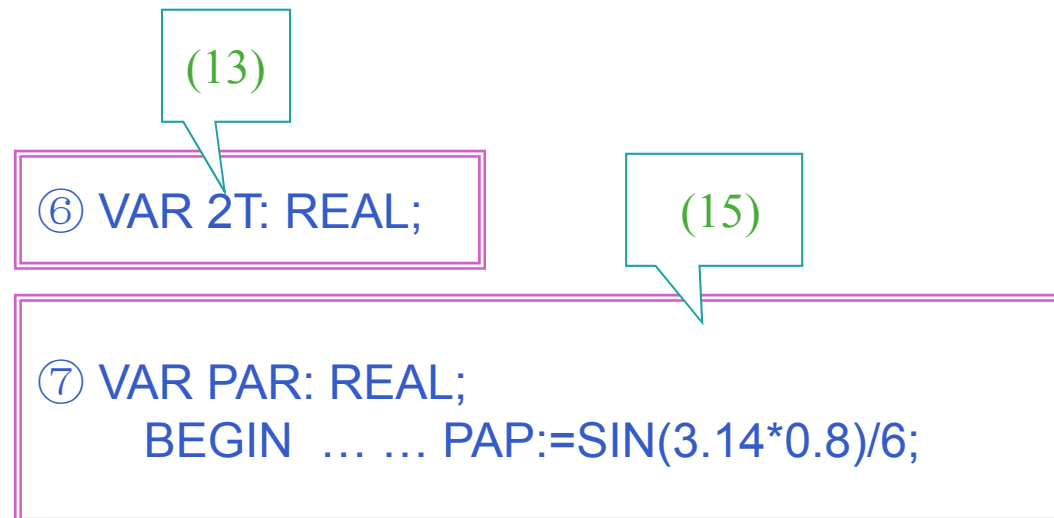
① VAR x, T1234567: REAL;
BEGIN x:=3.414; T1234567:=2.732;



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)



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)





二、请利用等价分类法为以下提供的内容设计测试用例

某工厂公开招工，规定报名者年龄应在**16~35** 周岁之间（到1995年6月30日为止），即出生年月不早于1960年7月，不晚于1979年6月。 报名程序具有自动检验输入数据的功能。如出生年月不在上述范围内， 将拒绝接受，并显示“年龄不合格”等出错信息。

请试用等价分类法， 设计出生年月的等价分类表 。



1、划分出生年月等价分类表

假定已知出生年月是由 6 位数字字符表示，前4 位代表年，后2 位代表月，则可以划分为 3 个有效等价类和 7 个无效等价类。

输入数据	有效等价类	无效等价类
出生年月	① 6位有效数字字符	② 有非数字字符 ③ 少于6个数字字符 ④ 多于6个数字字符
对应数值	⑤ 196007-197906	⑥ < 196007 ⑦ > 197906
月份对应数值	⑧ 在1-12之间	⑨ 等于 “0” ⑩ >12



2、设计有效等价类需要的测试用例

输入数据	有效等价类	无效等价类
出生年月	① 6位有效数字字符	② 有非数字字符 ③ 少于6个数字字符 ④ 多于6个数字字符
对应数值	⑤ 196007-197906	⑥ < 196007 ⑦ > 197906
月份对应数值	⑧ 在1-12之间	⑨ 等于 “0” ⑩ >12

测试数据	期望结果	测试范围
197011	输入有效	①、⑤、⑧



输入数据	有效等价类	无效等价类
出生年月	① 6位有效数字字符	② 有非数字字符 ③ 少于6个数字字符 ④多于6个数字字符
对应数值	⑤ 196007-197906	⑥ < 196007 ⑦ > 197906
月份对应数值	⑧ 在1-12之间	⑨ 等于 “0” ⑩ >12

3、为每一个无效等价类至少设计一个测试用例

测试数据	期望结果	测试范围
MAY,70	输入无效	② 有非数字字符
19705	输入无效	③ 少于6个数字字符
1968011	输入无效	④ 多于6个数字字符
196008	年龄不合格	⑥ <196007
195512	年龄不合格	⑦ >197906
196200	输入无效	⑨ 等于 “0”
197222	输入无效	⑩ >12



(二) 边界值分析法

采用边界值分析法来选择测试用例，可使
得被测程序能在边界值及其附近运行，从而更
有效地暴露程序中潜藏的错误。

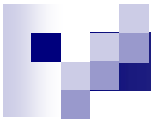


为了接受年龄合格的报名者则程序中
可能设有语句为：

```
If (196007 < value(birthdate) <= 197906)  
  Then read(birthday)  
  Else write “invalid age! ”
```



输入等价类	测试用例说明	测试数据	期望结果
出生年月	1 个数字字符 5 个数字字符 7 个数字字符 有1个非数字字符 全是非数字字符	5 19705 1968011 19705 AUGUS T	输入无效
对应数值	35 周岁 16 周岁	196007 197906	合格年龄
	>35 周岁 <16周岁	196006 197907	不合格年龄
月份 对应数值	月份值为 1 月 月份值为 12 月	196701 197412	输入有效
	月份值 < 1 月份值 >12	196700 197413	输入无效



等价分类法与边界值分析法的比较

- 1、等价分类法的测试数据是在各个等价类允许的**值域内**任意选取的，而边界值分析法的测试数据必须在**边界值附近**选取。
- 2、在公开招工的例子中，采用等价分类法设计了 **8**个测试用例而边界值分析法则设计了**13** 个， 所以，一般来说，用边界值分析法设计的测试用例要比等价分类法的代表性更广，发现错误的能力也更强。但是对边界的分析与确定比较复杂，它要求测试人员具有更多的经验和长找性。



(三) 错误猜测法

所谓猜测，就是猜测被测程序在哪些地方容易出错，然后针对可能的薄弱环节来设计测试用例。显然它比前两种方法更多地依靠测试人员的直觉与经验。所以一般都先用前两方法设计测试用例然后再用猜测法去补充一些例子作为辅助的手段。



(四) 因果图法

因果图是借助图形来设计测试用例的一种系统方法。它适用于被测程序具有多种输入条件，程序的输出又依赖于输入条件的各种组合的情况。

因果图是一种简化了的逻辑图，它能直观地表明程序输入条件（原因）和输出动作（结果）之间的相互关系。



测试方法的选用

测试策略

- 1、在任何情况下都应该使用边界值分析的方法。
- 2、必要时用等价类划分法补充测试方案。
- 3、必要时再用错误猜测法补充测试方案。
- 4、对照程序逻辑，检查已经设计出的测试方案。可以根据对程序可靠性的要求采用不同的逻辑覆盖标准，如果现有测试方案的逻辑程度没有达到要求的覆盖标准则应再补充一些测试方案。

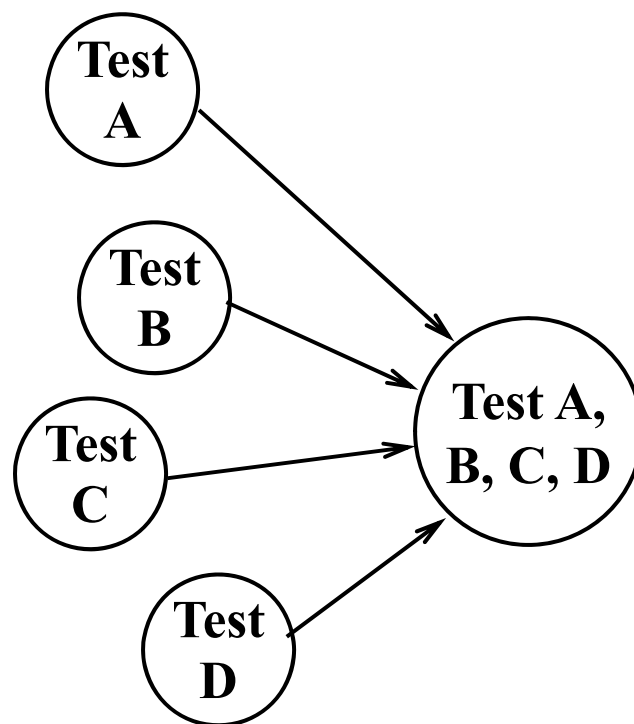


测试方法总结

策略种类	黑盒测试	白盒测试
测试对象	程序的功能	程序的结构
测试要求	逐一验证 程序的功能	程序的每一组成部分 至少被测试一次
采用技术	等价分类法 边界分析法 错误猜测法 因果图法	逻辑覆盖法 路径测试法

五、集成测试 (Integration Testing)

1、非渐增式测试 (Big-bang testing)





2、渐增式测试 (Incremental testing)

两种方式的比较：

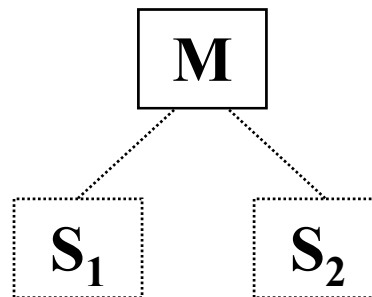
- ♠ Incremental testing 可以较早发现模块间的接口错误；Big-bang testing 最后才组装，因此错误发现得晚。
- ♠ Big-bang testing 中发现错误后难以诊断定位；Incremental testing 中，出现的错误往往跟最新加入的模块有关。
- ♠ Incremental testing 在不断集成的过程中使模块不断在新的条件下受到新的检测，测试更彻底。
- ♠ Incremental testing 较 Big-bang testing 费时。
Big-bang testing 可以同时并行测试所有模块，能充分利用人力。

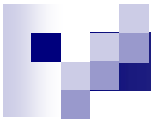
3、Incremental testing 的几种策略

(1) Top-down testing

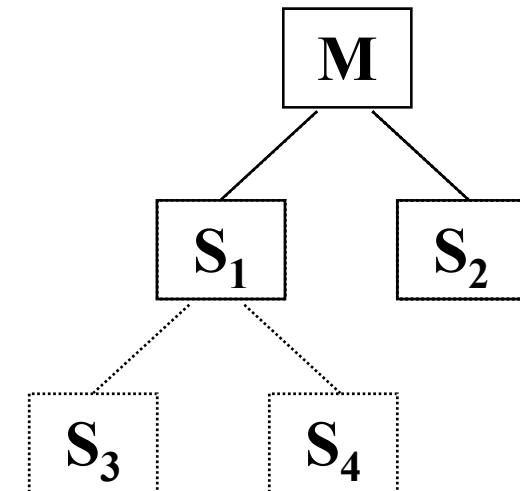
第1步：测试顶端模块，用存根程序(stub)代替直接附属的下层模块

Stub: to simulate the activity of the component which is not yet tested.





第2步：根据深度优先或宽度优先的策略，每次用一个实际模块代换一个stub。



第3步：在结合进一个模块的同时进行测试。

第4步：回归测试(regression testing)——全部或部分地重复以前做过的测试。

优点：在早期即对主要控制及关键的抉择进行检验。

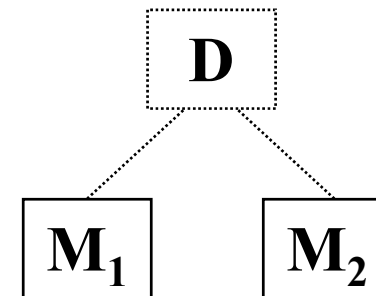
问题：Stub只是对低层模块的模拟，测试时没有重要的数据自下往上流，许多重要的测试须推迟进行，而且在早期不能充分展开人力。

(2) Bottom - up testing

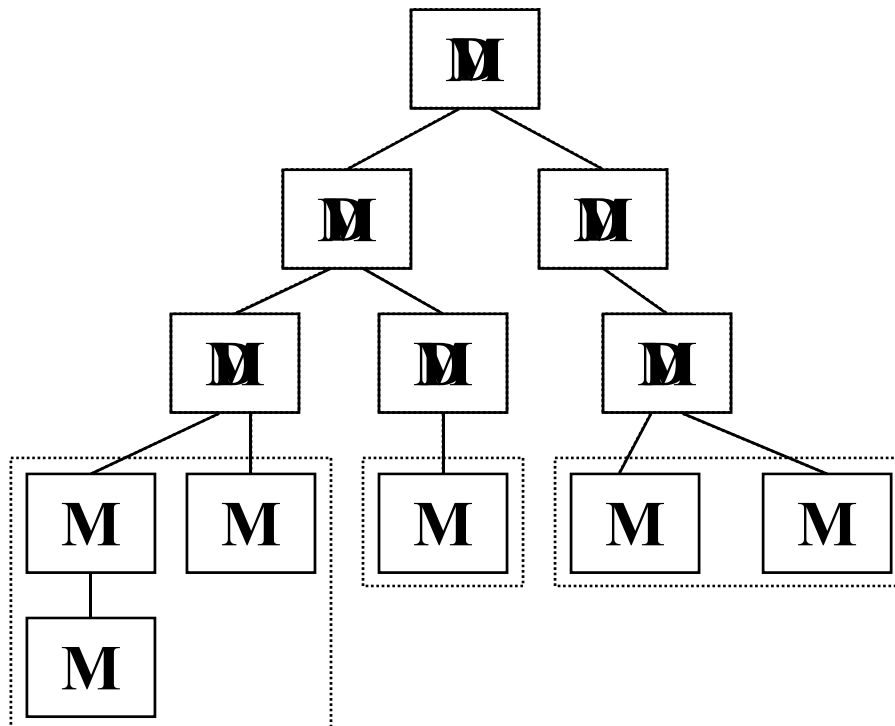
第1步：把低层模块组合成族，每族实现一个子功能。

第2步：用驱动程序(Driver)协调测试数据的I\O，测试子功能族。

Driver: to call a particular component and passes a test case to it.



第3步：去掉Driver，自下而上把子功能族合成更大的子功能族。



注意：两种策略的优、缺点刚好互补，但单用其中任何一种都不实际，通常根据软件的特点将二者混用。

4 验收测试(Acceptance testing)

任务：验收软件的有效性（功能和性能达标）；

手段：黑盒测试；用户参与；主要用实际数据进行测试；

内容：按合同规定审查软件配置；设计测试计划，使通过测试保证软件能满足所有功能、性能要求；
文档与程序一致，具有维护阶段所必须的细节；
严格按用户手册操作，以检查手册的完整性和正确性。



5 静态测试

手工测试（Manual Testing）

- 通过人工检查、代码审查和走查程序。
- 能有效地发现30%~70%的逻辑设计和编码错误，从而↓测试的总工作量。



自动测试（Automation Testing）

- 编写测试工具，让它们自动运行来查找Bug
- 优点：
 - 能够快速、广泛地查找Bug
- 缺点：
 - 只能检查一些最主要的问题，如崩溃、死机，无法发现一些通过人眼很容易发现的日常错误
 - 编写测试工具的工作量大



自动测试案例一

- 在测试Exchange Server时，常常需要几万个甚至几十万个用户同时把E-mail发送到Server，以保证Server不会出现死机或崩溃的现象。可是，需要几万人同时发送E-mail，在现实生活中很难人为实现。但是，利用测试工具能够非常容易地做到。测试工具可以自动产生几万个帐号，并且让它们在同一时间从不同机器上（一个机器可以有多个不同的帐号）同时发送E-mail信息。



自动测试案例二

- 在测试网站的Server时，要求50 000个用户同时浏览一个Web页面，以保证网站的Server不会死机。一般来说，找到50 000 个用户同时打开一个网页不现实，就算能够找到50 000个测试者，成本也非常高。但通过测试工具则很容易做到，并且工具还可以自动判断浏览结果是否正确。