# 第六章 软件质量保证

周 瑞

QQ群：235458708

信息与软件工程学院

# 本章学习目标

1. 掌握质量保证的概念和常用方法
2. 掌握软件测试的概念和常用方法
3. 理解质量保证活动在软件工程中的重要作用和意义

# 本章内容

- 软件质量
- 软件质量保证的常用方法
- 软件测试技术
  - 白盒测试
  - 黑盒测试
- 软件测试策略
  - 单元测试
  - 集成测试
  - 系统测试
  - 验收测试

# 软件质量为什么重要?

# 软件质量为什么重要？

## 爱国者导弹

- 美国爱国者导弹曾应用于海湾战争对抗伊拉克的飞毛腿导弹。但它发生过一次严重事故：1991年，爱国者导弹在追踪飞毛腿导弹时，其中一枚在沙特阿拉伯的多哈导致了28名美国士兵丧生。

- 事故原因：导弹的软件含有一个累加计时故障。一个很小的系统时钟错误累加起来，导致时间拖延了14小时，从而造成跟踪系统失去准确度。

# 软件质量为什么重要？

## 阿丽亚娜5型火箭

- 欧洲航天局的阿丽亚娜系列运载火箭在国际上享有盛誉。但是1996年阿丽亚娜5型火箭初次飞行时发生了灾难。发射后仅仅37秒，火箭偏离它的飞行路径，解体并爆炸了。火箭上载4颗通信卫星。50亿美元付之一炬。


- 事故原因：控制惯性导航系统的计算机向控制引擎喷嘴的计算机发送了一个无效数据。它没有发送飞行控制信息，而是送出了一个诊断位模式，表明在将一个64位浮点数转换成16位有符号整数时，产生了溢出。

# 软件质量为什么重要？

## ATM机

- 2012年在英国汉普郡利明顿附近的富裕小镇米尔福德，一台ATM机发生故障，在顾客取款时会吐出双倍数额的现金，有数百名顾客趁机取款。
- 2012年中国商丘，某先生去ATM机存钱，ATM机出现错误，吞掉其8400元。相隔不到几分钟，再存100元，却操作成功。

# 软件质量和软件质量保证

- 软件质量：明确表示是否符合功能和性能要求，明确地记载开发标准和所有专业开发软件的期望的隐性特点。
  - ① 与需求一致
  - ② 与隐含需求一致
  - ③ 与指定的开发标准一致
- 软件质量保证（SQA）：一个监控的软件工程以确保软件质量的过程
- 软件质量保证涵盖了整个软件开发过程

# 软件缺陷

- 至少满足下列一个条件，称发生了一个软件缺陷
  - 软件未实现产品说明书要求的功能
  - 软件实现了产品说明书未提到的功能
  - 软件出现了产品说明书指明不能出现的错误
  - 软件未实现产品说明书虽未明确提及但应该实现的目标
  - 软件难以理解、不易使用、运行缓慢、用户认为不好等

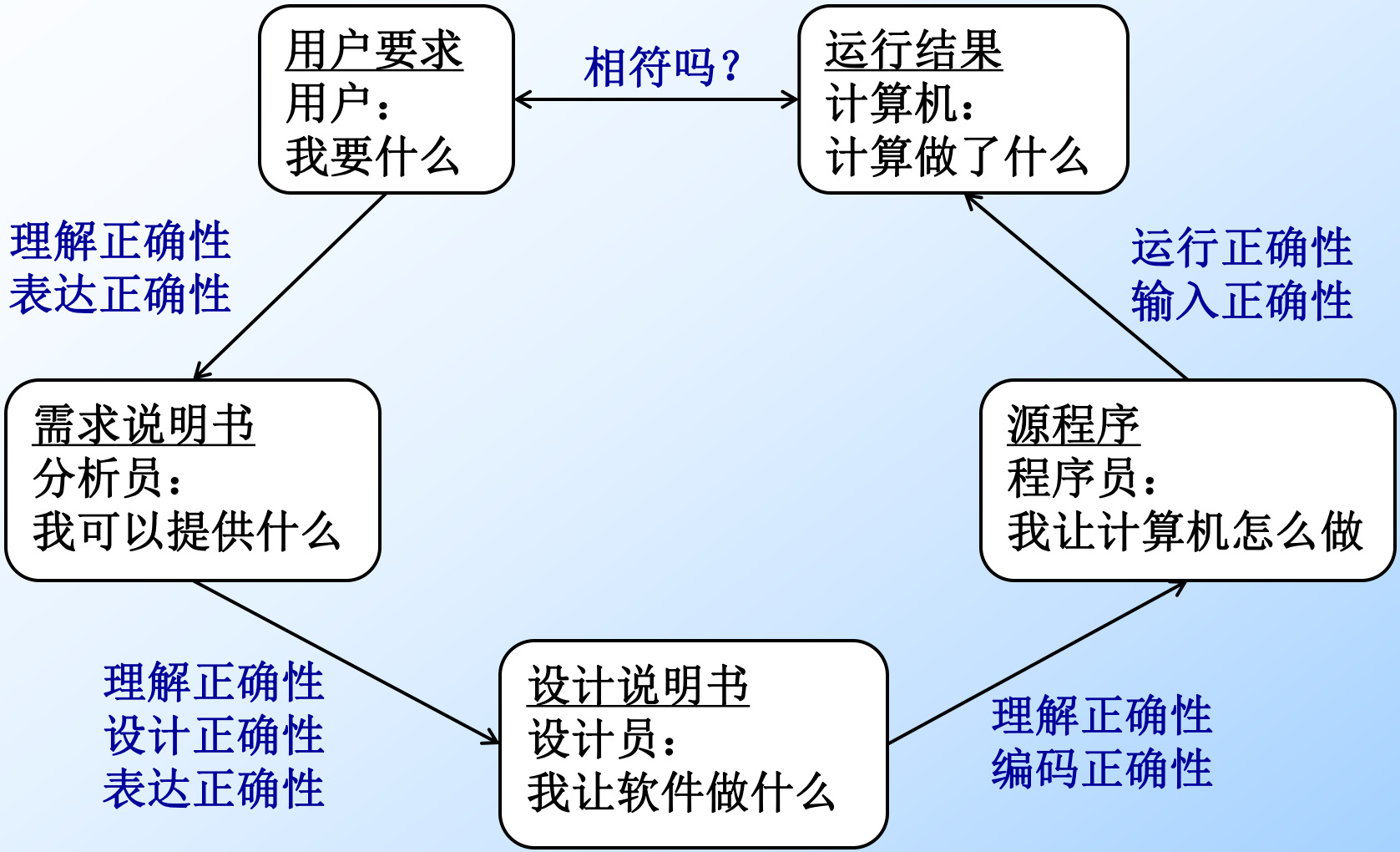
# 软件质量属性

- 正确性/Correctness：满足需求规格和用户目标的程度
- 可靠性/Reliability：一定时间内无故障运行的能力
- 有效性/Effectiveness：用户完成特定任务和达到特定目标时所具有的正确和完整程度
- 可用性/Availability：用户能否用软件完成他的任务，效率如何，主观感受怎样
- 可维护性/Maintainability：为修改Bug、增加功能、提高质量而诊断并修改软件的难易程度
- 可测试性/Testability：对软件进行测试的难易程度
- 灵活性/Flexibility：反映软件适应变化的能力，修改或改进一个已投入运行的软件所需的工作量

# 软件的质量属性

- 可移植性/Portability：软件不经修改或稍加修改就可以运行于不同软硬件环境的难易程度
- 可重用性/Reusability：重用软件或构件的难易程度
- 互操作性/Interoperability：本软件与其它系统交换数据和相互调用服务用以协同运作的难易程度
- 安全性/Security：向合法用户提供服务，阻止非授权使用服务
- 健壮性/Robustness：异常情况下软件能够正常运行的能力
- 易用性/Usability：用户使用软件的容易程度

# 软件质量保证的对象


用户要求
用户：
我要什么
相符吗？
运行结果
计算机：
计算做了什么
理解正确性
表达正确性
运行正确性
输入正确性
需求说明书
分析员：
我可以提供什么
源程序
程序员：
我让计算机怎么做
理解正确性
设计正确性
表达正确性
设计说明书
设计员：
我让软件做什么
理解正确性
编码正确性

# 软件质量保证的常用方法

- 评审/静态分析方法
- 软件测试
- 形式化建模与验证

# 形式化建模与验证

- 运用形式化语法和语义（集合论和逻辑符号体系）描述需求和设计
- 采用基于数学的方法建模，具有检验模型正确性的能力
- 能够形式化证明程序的正确性
- 能够创建具有极低故障率的软件
- 形式化规格说明语言：OCL、Z等
- 净室软件工程（Cleanroom software engineering）
  - 程序构造之前进行正确性验证
  - 将软件可靠性认证作为测试活动的一部分
- 缺点：不易掌握，没有广泛使用

# 评审/静态分析方法

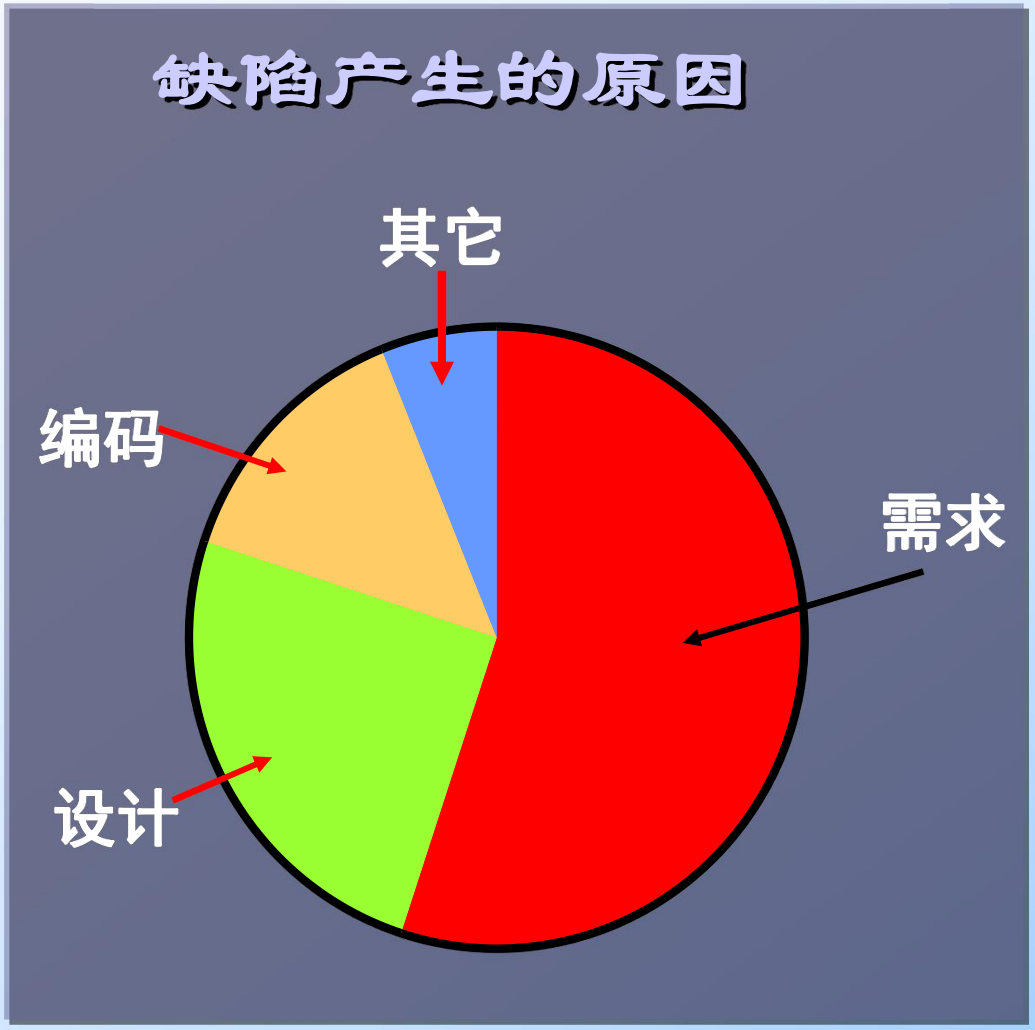
- 静态分析方法指不实际运行程序，通过检查和阅读等手段来发现错误并评估代码质量的软件质量保证技术。
- 作用
  - 通过对代码标准及质量的监控提高代码可靠性
  - 尽可能早地通过对源代码的检查发现缺陷
  - 组织代码审核定位易产生错误的模块
- 非常有效的质量保证手段
  - 是软件开发早期查错最有效的机制
  - 越来越多地被采用

# 评审

- 非正式评审/同事审查
  - 简单桌面检查、临时会议、结对编程评审
- 走查（walk through）
  - 开发组内部进行，采用讲解、讨论和模拟运行的方式查找错误
- 审查（inspection）
  - 开发组、测试组和相关人员联合进行，采用讲解、提问、以及Checklist方式进行错误查找，以会议的形式进行，制定目标、流程、规则和结果报告

# 评审

- 检查需求
- 检查设计
- 检查代码
- 发现软件缺陷的有效手段！



# 评审：检查需求

- 正确性：对系统功能性能等的描述正确反应用户真正需求
- 无二义性：每项需求对任何人都只有一种解释
- 一致性：任何一项需求不能和其它需求矛盾
- 完整性：包括用户需要的每一项功能、性能、外部接口、约束等
- 可行性：每项需求都是可以实施的
- 必要性：每项需求都是客户需要的
- 可验证性：每项需求都能写出测试用例或其它检验方法
- 可跟踪性：每项需求能与它的根源和设计、代码、测试用例之间建立链接
- 可修改性：格式和组织方式应保证后续的修改容易进行

# 需求检查练习

- 例1：产品必须在固定的时间间隔内提供状态信息，并且每次时间间隔不得小于60秒。
  - 什么样的状态信息？什么情况下显示？在什么地方显示？通过什么样的方式提供？
- 例2：如果可能的话，应当根据系统货物编号列表，在线确认输入的货物编号。
  - “如果可能的话”是什么意思？
- 例3：产品不应该提供将带来灾难性后果的查找和替换选择？
  - 真正的需求是什么？

# 评审：检查设计

- 在设计完成、编码开始前进行
- 检查软件设计说明
  - 功能的用意、总体位置
  - 输入、输出
  - 可能的错误/例外
  - 接口定义
  - 交互细节
  - 实施建议
  - ……

# 评审：检查代码

- 数据声明错误
- 数据引用错误
- 计算错误
- 比较错误
- 控制流程错误
- 子程序参数错误
- 输入/输出错误
- ……

# 软件测试(Software Testing)

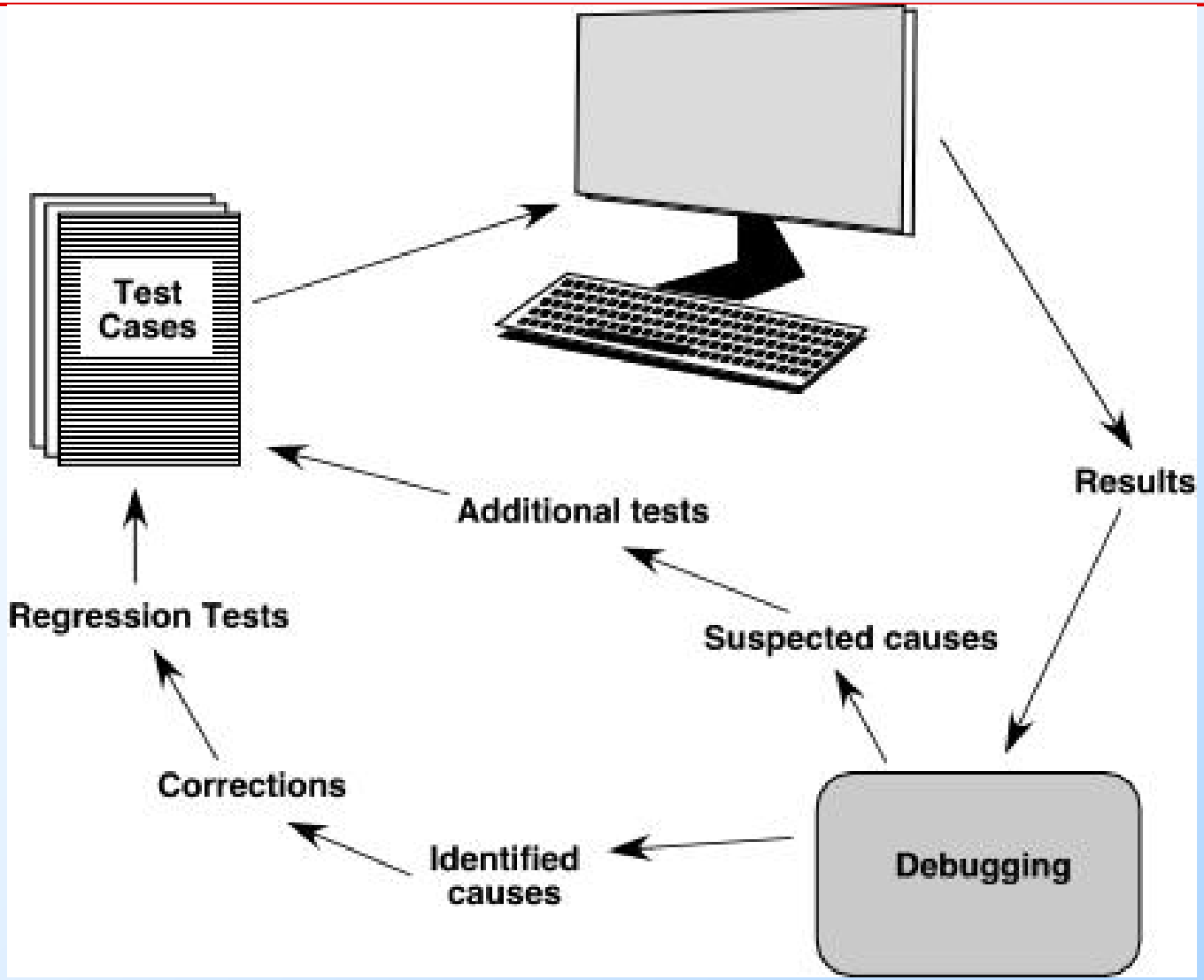
软件测试指在某种指定的条件下对系统或组件操作，观察或记录结果，对系统或组件的某些方面进行评估的过程。分析软件各项目以检测现有的结果和应有结果之间的差异（即软件缺陷），并评估软件各项目的特征的过程。

测试用例是测试输入、执行条件、以及预期结果的集合，是为特定的目的开发的，例如执行特定的程序路径或验证与指定的需求相符合。

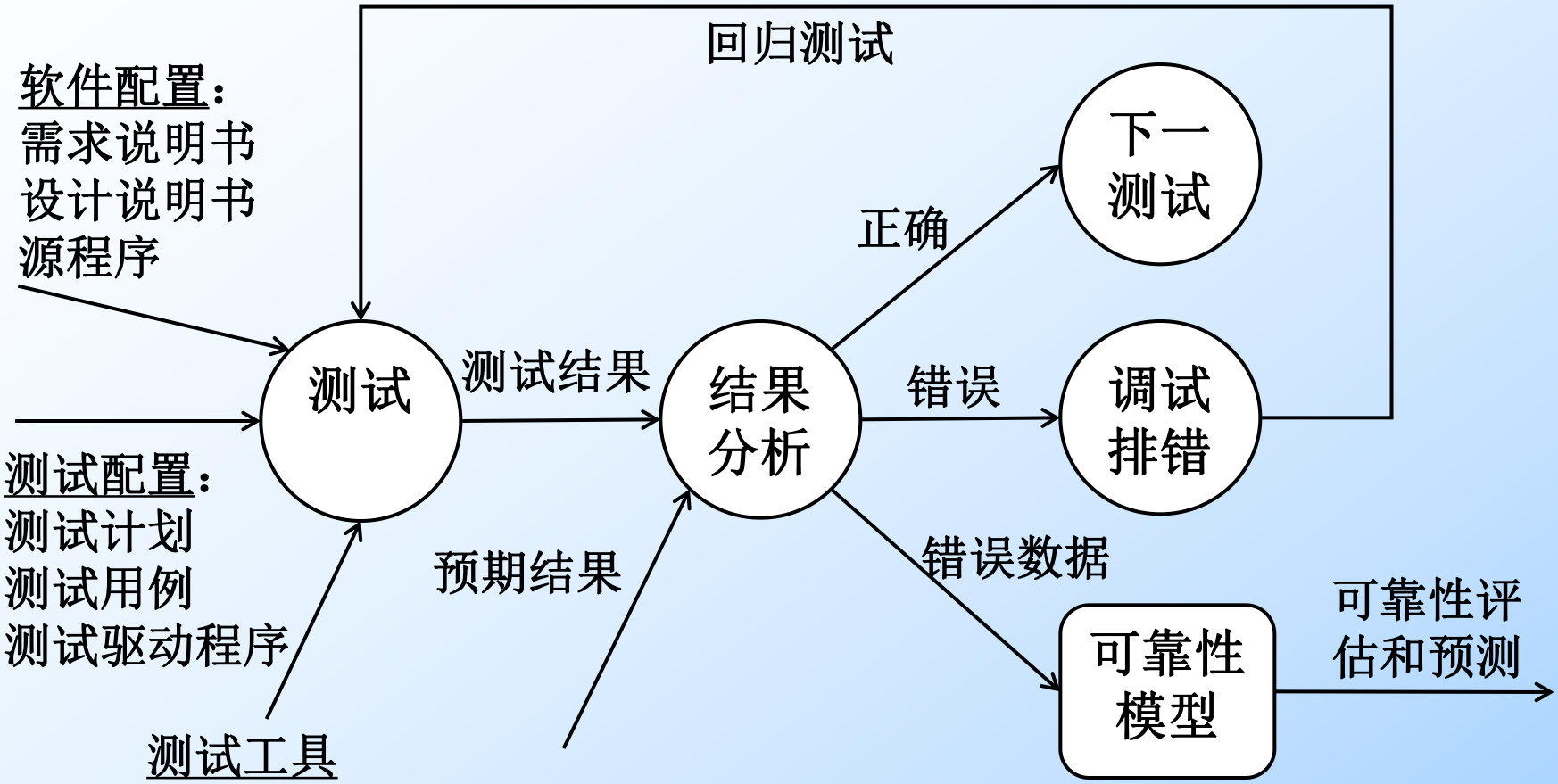
# 软件测试和调试

- 测试与调试的共同点
  - 两者都包含有处理软件缺陷和查看代码的过程
- 测试与调试的区别
  - 测试的目标是发现软件缺陷的存在
  - 调试的目标是定位与修复缺陷

# 软件调试过程


Test Cases
Results
Additional tests
Regression Tests
Suspected causes
Corrections
Identified causes
Debugging

# 软件测试过程


回归测试
软件配置：
需求说明书
设计说明书
源程序
测试配置：
测试计划
测试用例
测试驱动程序
测试工具
测试
测试结果
结果分析
正确
下一测试
错误
调试排错
预期结果
错误数据
可靠性模型
可靠性评估和预测

# 软件测试

- 软件质量保证的主要方法
- 测试占总成本的40%以上
- 对于安全关键（safety critical）软件，是其它总费用的3-5倍
- 测试的目标：
  - 测试是为了发现程序中的错误
  - 好的测试方案：极可能发现新的错误
  - 成功的测试方案：发现了新的错误
- 测试需要由专门的测试小组进行测试

# 软件测试

- 穷举测试是不可能的！！ 为什么？
- 路径条数

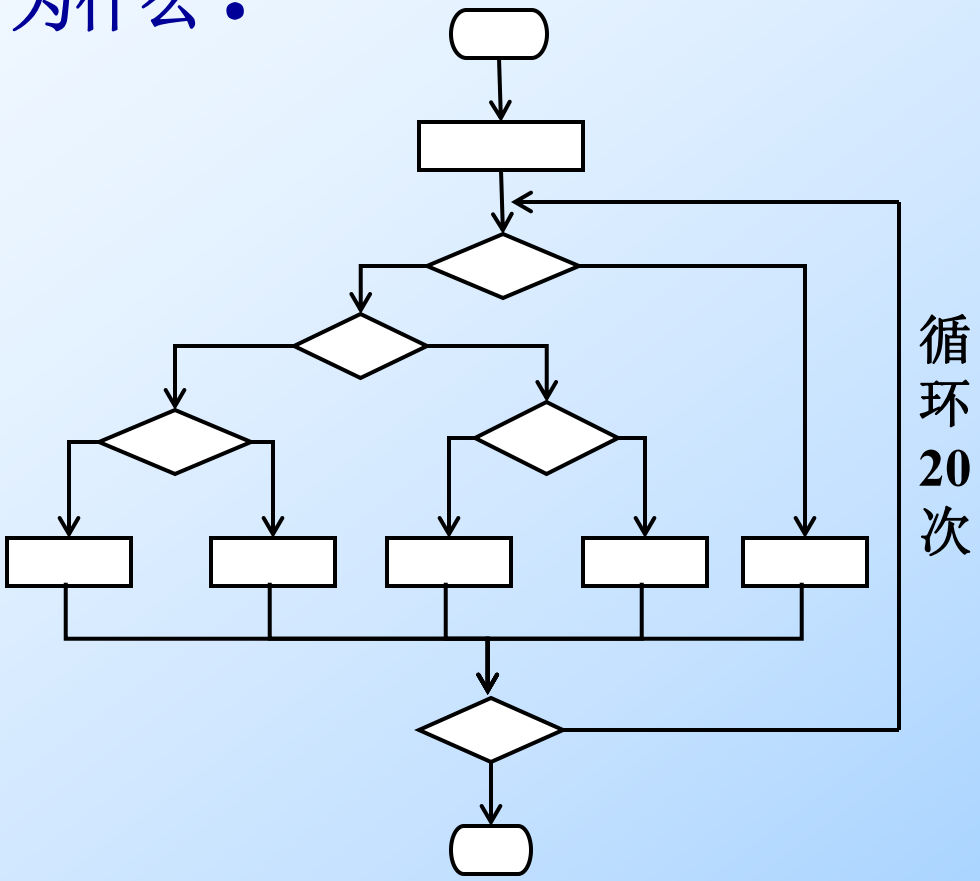
  $5^{20} \approx 10^{14}$
- 测试时间

  假设测试1条路径需要1毫秒

  $5^{20} \times 10^{-3}$秒

  $5^{20} \times 10^{-3} / (3600 \times 24 \times 365)$

  =3024年
- 软件测试只能是部分测试！



# 软件测试

如果测试没有发现错误，

能说软件没有错误吗？

# 软件测试

测试只能证明程序中有错误，

而不能证明程序是正确的！

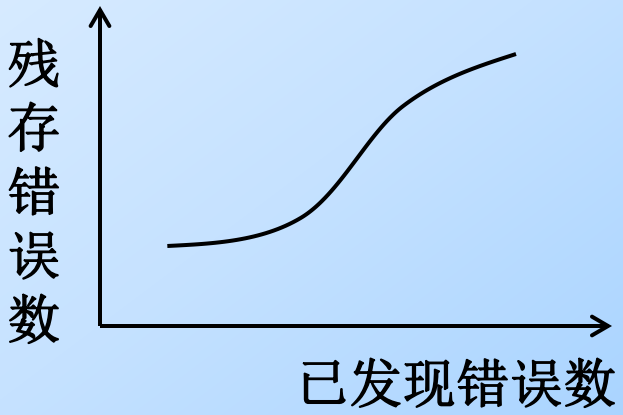
# 软件测试的目标

- 确认系统满足其预期的使用和用户的需要
- 确认解决了所需解决的问题（如实现商业规则）
- 便于及早发现软件和系统的异常
- 及早提供软件和系统的性能的评估
- 为管理提供真实信息，以决定在当前状态下发布产品在商业上的风险
- 鉴别出程序在功能等方面的异常集聚之处

# 软件测试准则

- 尽早制定测试计划，不断进行软件测试
- 严格执行测试计划，排除测试随意性
- 从小规模测试开始，逐步进行较大规模测试
- 由专门的测试小组进行测试
- 设计测试用例时，既要测试合理的输入条件，也要测试不合理的输入条件
- 测试用例由测试输入数据和预期结果组成，对测试结果做全面检查
- 注意错误群集性现象

为什么？



# 软件测试的评估准则

- 覆盖率
- 故障插入
- 变异分值

# 软件测试的评估准则-覆盖率

- 给定一个测试需求集合TR和一个测试集合T，覆盖率可以定义为T满足的测试需求占TR总数的比例
- 100%覆盖率在实际中是不现实的
- 商用自动化测试工具（loadrunner等）

# 软件测试的评估准则-故障插入

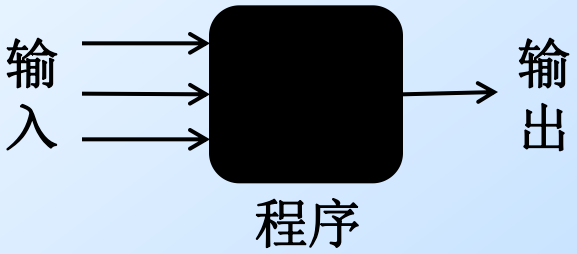
- 故障插入（Fault Seeding）是一种统计方法，用于评价遗留在一个程序中的故障的数量和种类。
- 在测试前有意地插入一些故障到程序中，在测试执行中，有一部分插入的故障会因测试而显露出来，但可能一些故障在测试中没有暴露出来，仍存在于程序中。
- 原本错误总数 =（播入的错误总数/发现的播入错误数）×发现的原本错误数
- 残留错误数 = 原本错误总数 - 发现的原本错误数

# 软件测试的评估准则-变异分值

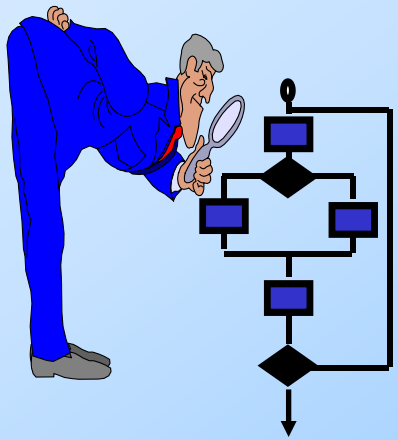
- 该指标和变异测试密切相关。
- 变异测试是一种特殊的测试方法，在这种测试方法中，程序进行两个或更多个变异，然后用同样的测试用例执行测试，可以评估这些测试用例探测程序变异间的差异的能力。

# 软件测试技术

- 黑盒测试
  - 忽略程序的内部机制，测试程序的每项功能是否符合要求
  - 也称功能性测试

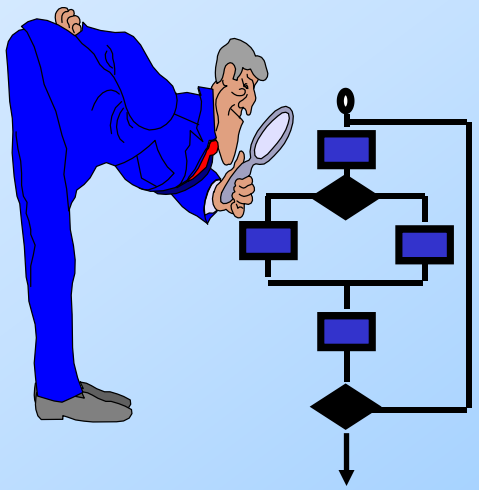



- 白盒测试
  - 考虑程序的内部机制，测试程序的每种内部操作是否符合要求
  - 也称结构性测试



# 白盒测试技术

- 白盒测试指考虑系统或组件的内部机制的测试形式（如分支测试、路径测试、语句测试等），也称结构性测试、逻辑驱动测试。
- 考虑程序的内部机制，测试程序的每种内部操作是否符合要求
  - 对程序的所有独立执行路径进行测试
  - 对所有的逻辑判定的“真”和“假”的情况进行测试
  - 在循环的边界测试
  - 测试内部数据结构的有效性



# 白盒测试方法

- 逻辑覆盖
- 控制流图覆盖
- 独立路径测试

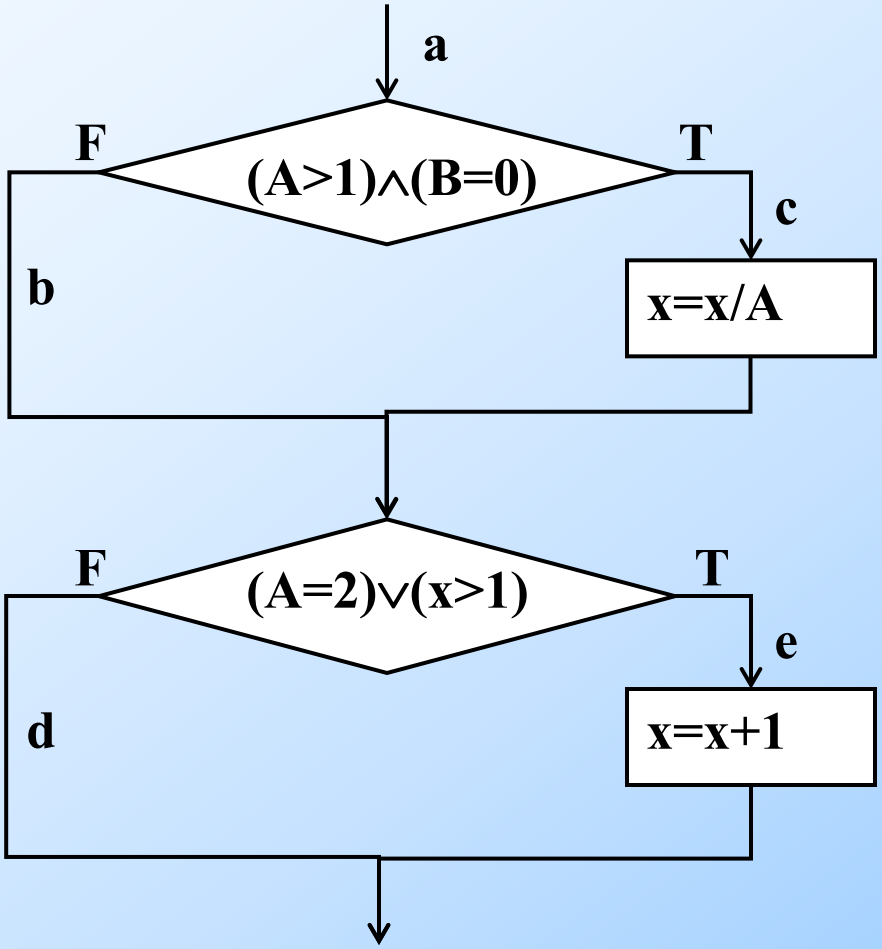
# 白盒测试技术-逻辑覆盖

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
- 判定覆盖/分支覆盖
- 条件覆盖
- 判定/条件覆盖
- 条件组合覆盖
- 路径覆盖

# 白盒测试技术-语句覆盖

- 每条语句至少执行一次




- 路径ace
- 测试用例：

  输入：（A=2，B=0，x=4）

  输出：（A=2，B=0，x=3）

- 问题：只覆盖一条路径

# 白盒测试技术-判定覆盖

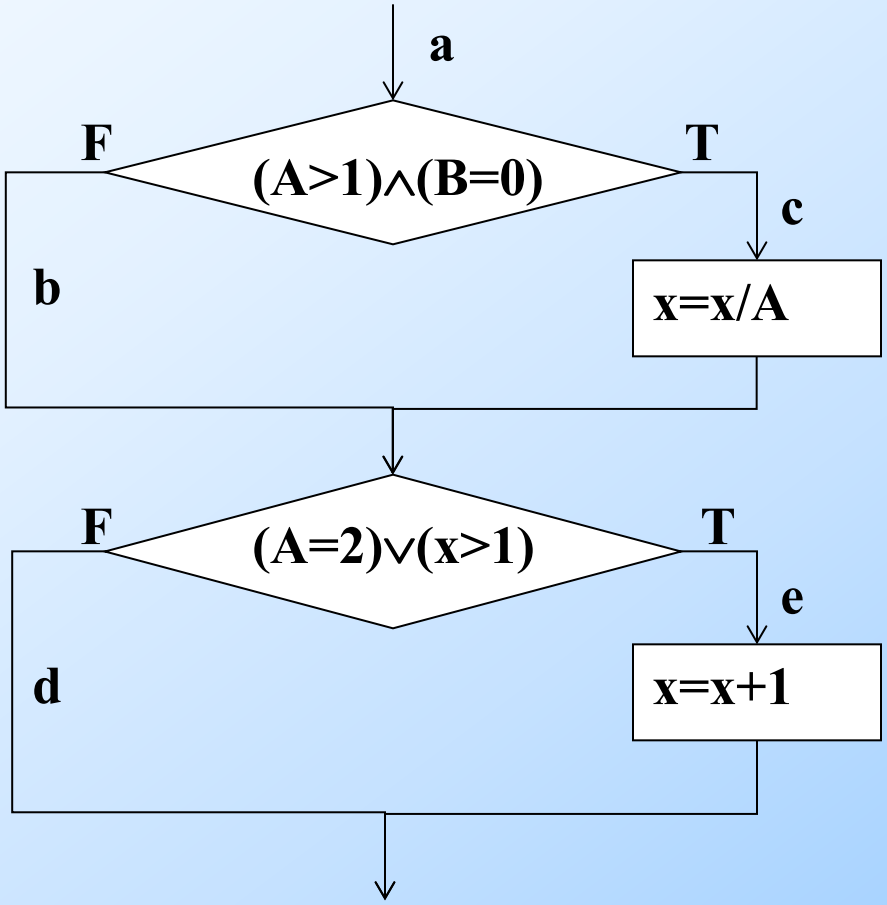
- 每个判定的取真分支和取假分支至少执行一次
- 路径ace和abd
- 测试用例1

  输入（A=2，B=0，x=4）

  输出（A=2，B=0，x=3）
- 测试用例2

  输入（A=1，B=1，x=1）

  输出（A=1，B=1，x=1）



# 白盒测试技术-判定覆盖

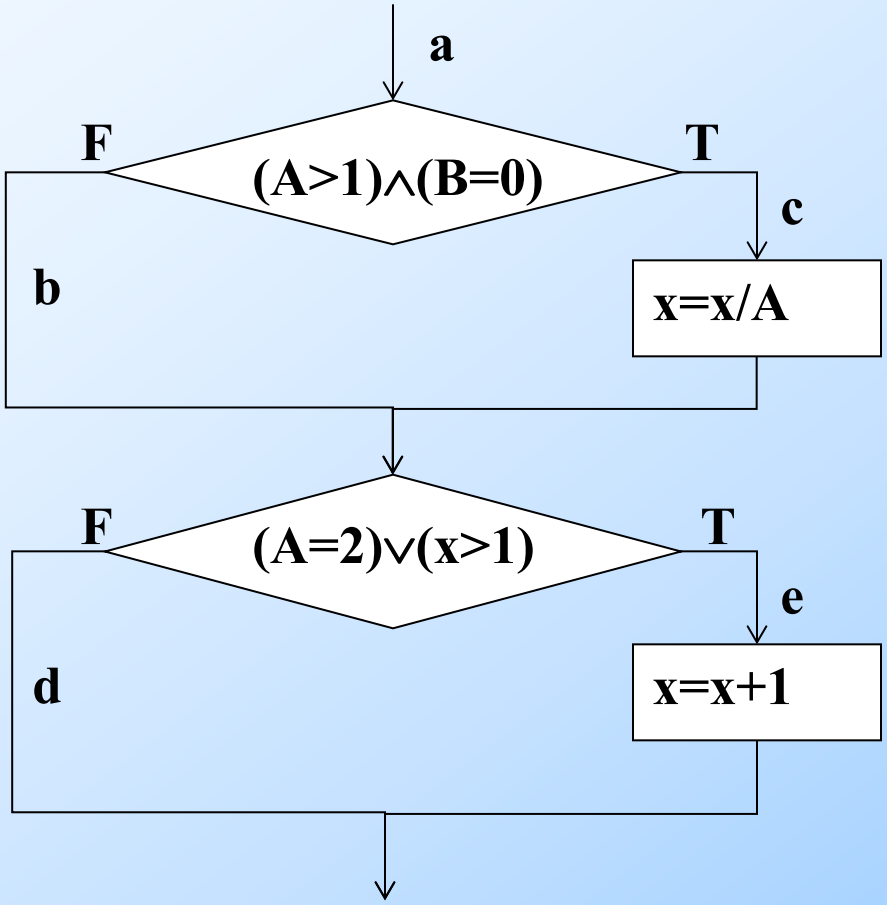
- 每个判定的取真分支和取假分支至少执行一次
- 路径abe和acd
- 测试用例1

  输入：（A=2，B=1，x=1）

  输出：（A=2，B=1，x=2）
- 测试用例2

  输入：（A=3，B=0，x=3）

  输出：（A=3，B=0，x=1）
- 问题：(A=2)∨(x<1)？



# 白盒测试技术-条件覆盖

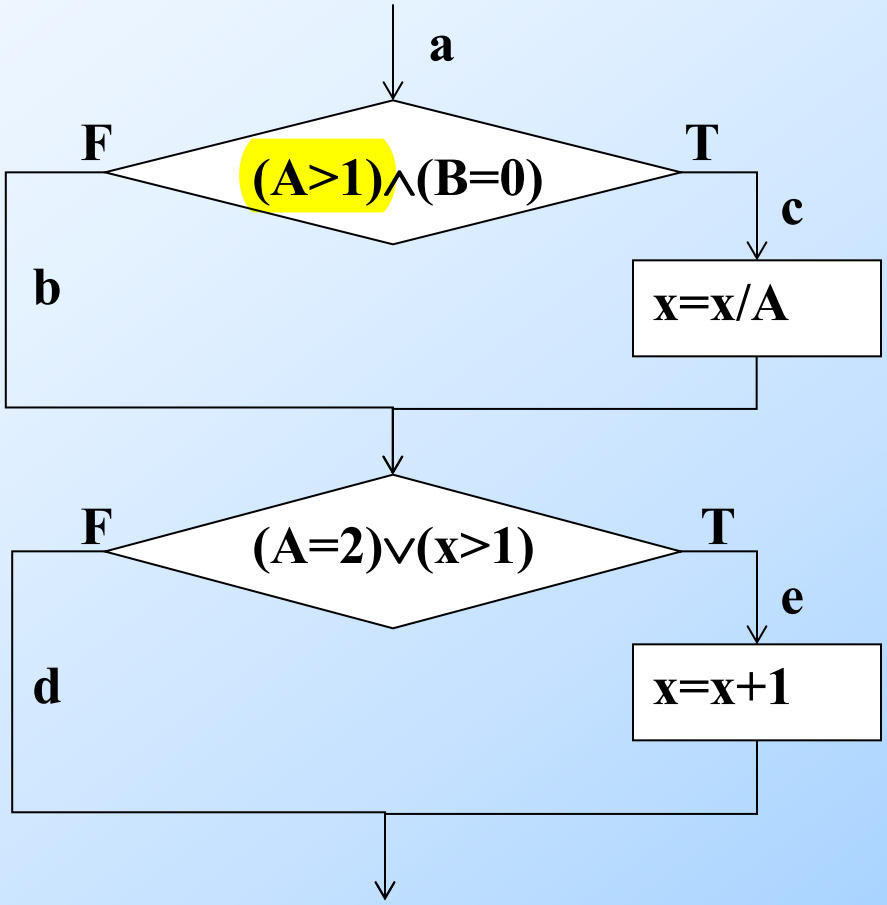
- 每个判定的每个条件都能取到可能的结果
- 测试用例1：FTFT

  输入：（A=1，B=0，x=3）

  输出：（A=1，B=0，x=4）
- 测试用例2：TFTF

  输入：（A=2，B=1，x=1）

  输出：（A=2，B=1，x=2）
- 问题：只执行了路径be



# 白盒测试技术-判定条件覆盖

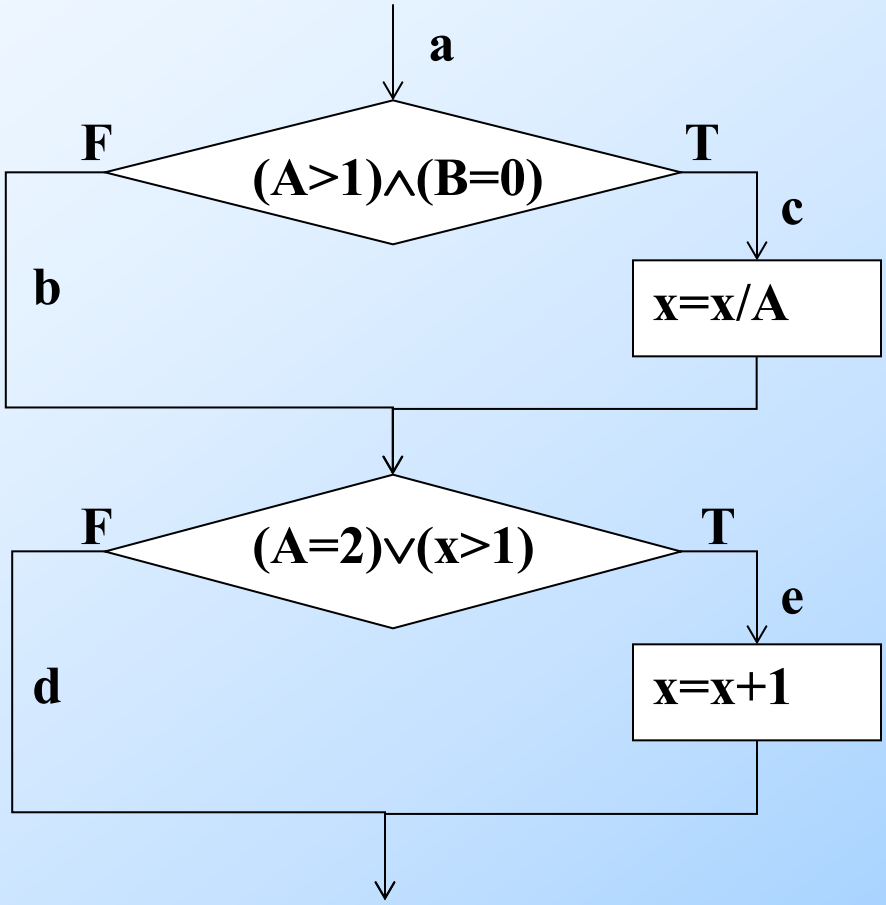
- 不仅每个判定的取真分支和取假分支至少执行一次，并且每个判定的每个条件都能取到可能的结果
- 测试用例1：TTTT，ace

  输入：(A=2, B=0, x=4)

  输出：(A=2, B=0, x=3)
- 测试用例2：FFFF，abd

  输入：(A=1, B=1, x=1)

  输出：(A=1, B=1, x=1)
- 问题：没有测试所有条件的取值



# 白盒测试技术-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

  ①A>1, B=0； T； ②A>1, B≠0； F；

  ③A≤1, B=0； F； ④A≤1, B≠0； F；

  ⑤A=2, x>1； T； ⑥A=2, x≤1； T；

  ⑦A≠2, x>1； T； ⑧A≠2, x≤1； F；

- 测试用例

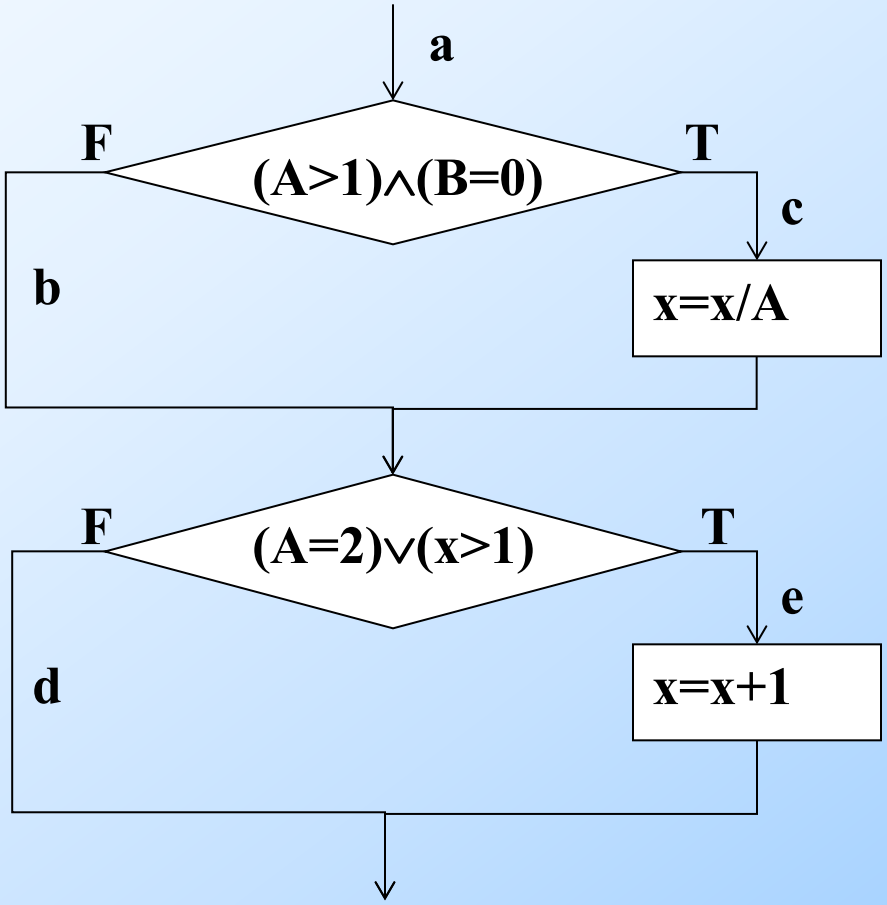
  (A=2, B=0, x=4)， ①⑤， ace

  (A=2, B=1, x=1)， ②⑥， abe

  (A=1, B=0, x=3)， ③⑦， abe

  (A=1, B=1, x=1)， ④⑧， abd

- 问题：未覆盖acd



# 白盒测试技术-路径覆盖

- 程序中每条可能路径都至少执行一次
- 路径

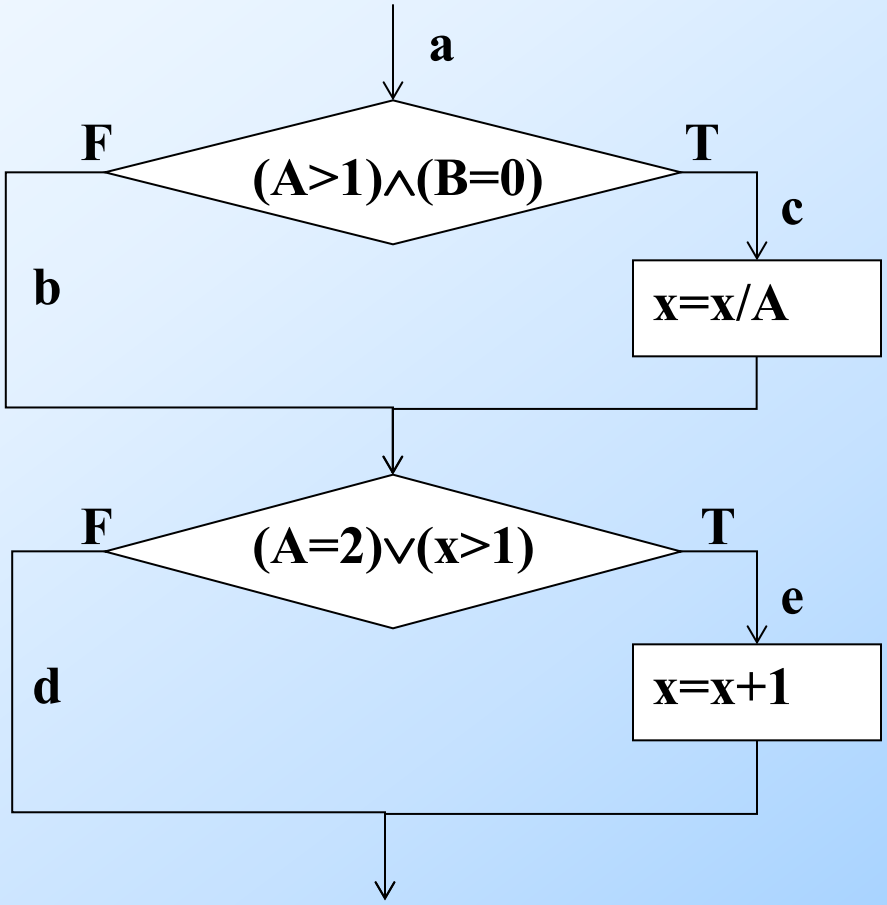
  ace、abd、abe、acd
- 测试用例

  (A=2, B=0, x=4)， ace；

  (A=1, B=1, x=1)， abd；

  (A=1, B=1, x=2)， abe；

  (A=3, B=0, x=3)， acd



# 白盒测试技术-控制流图覆盖

- 控制流图覆盖测试是将代码转变为控制流图（CFG），基于其进行测试的技术
- 控制流图覆盖属于白盒测试
  ① 节点覆盖
  ② 边覆盖
  ③ 路径覆盖

# 控制流图

- 控制流图（CFG、流图、程序图）

① 退化的程序流程图，仅描述程序内部的控制流程

② 将程序流程图中的每个处理符号都退化成一个结点

③ 分支的汇聚处应该有一个汇聚结点

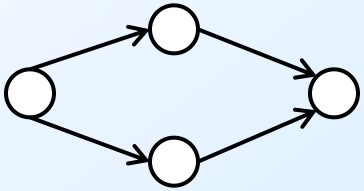
④ 如果判断中的条件表达式是复合条件表达式，则改为嵌套的简单条件表达式

⑤ 将程序流程图中连接处理符号的流线变成连接结点的有向弧

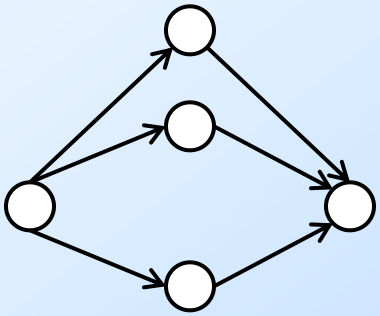
# 控制流图

- 控制流图的结构

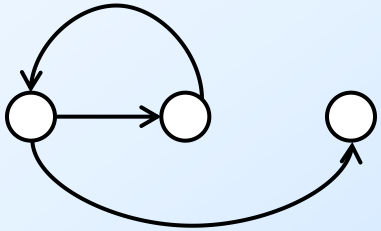
顺序结构



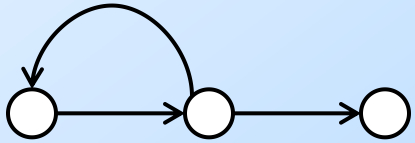
IF选择结构



CASE多分支结构

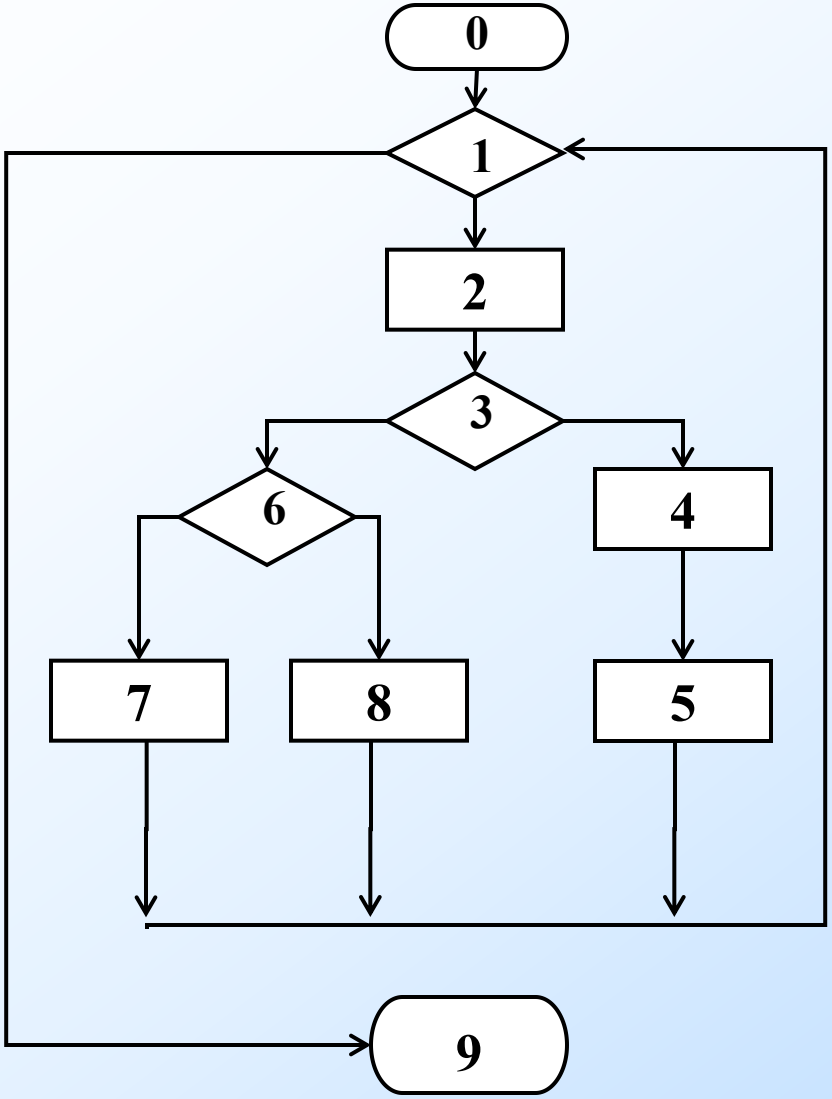


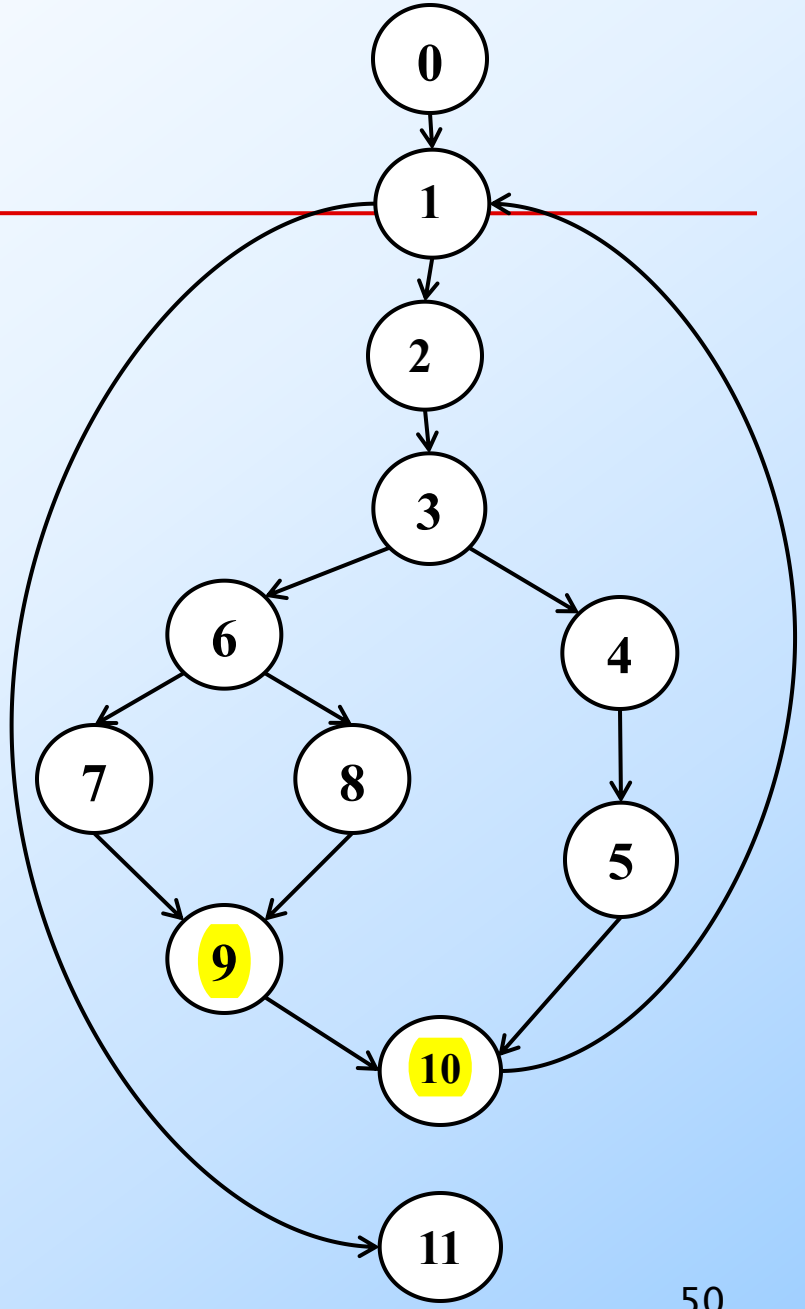
WHILE型循环结构



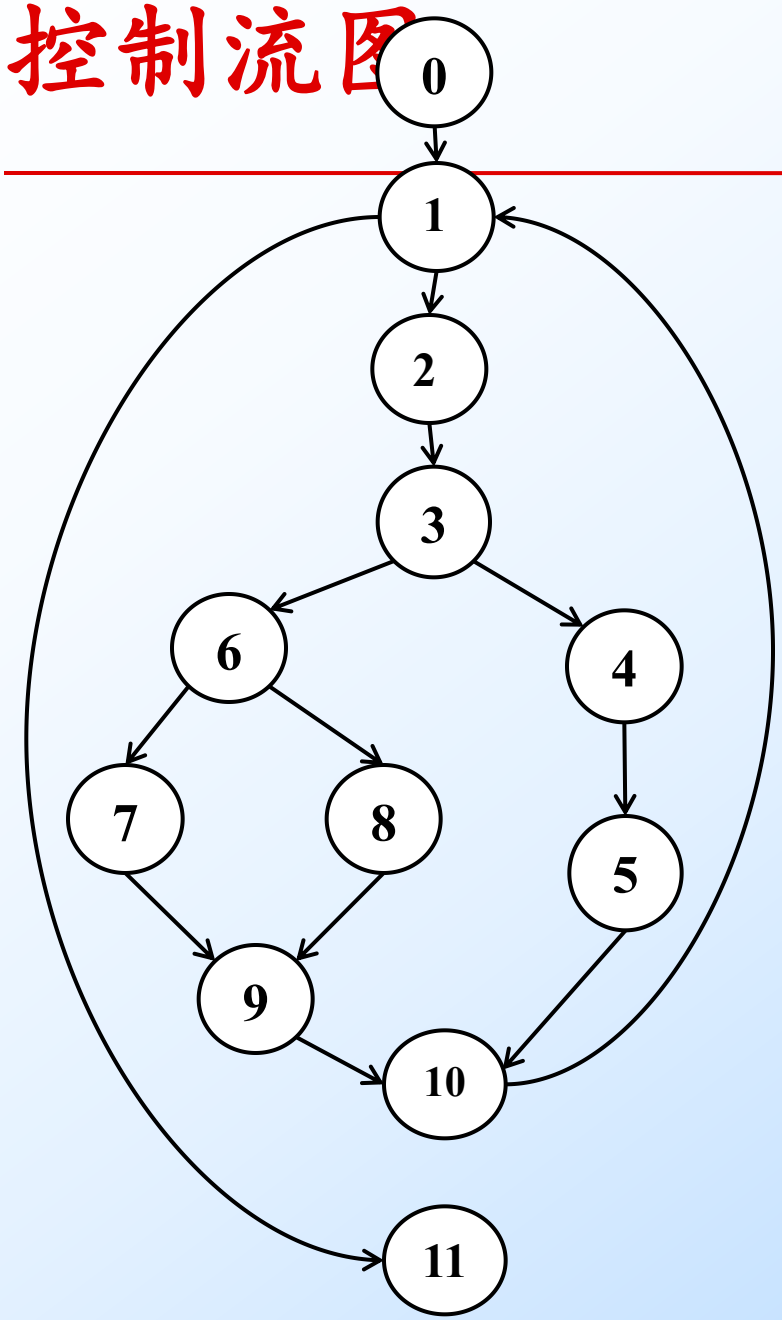
UNTIL型循环结构

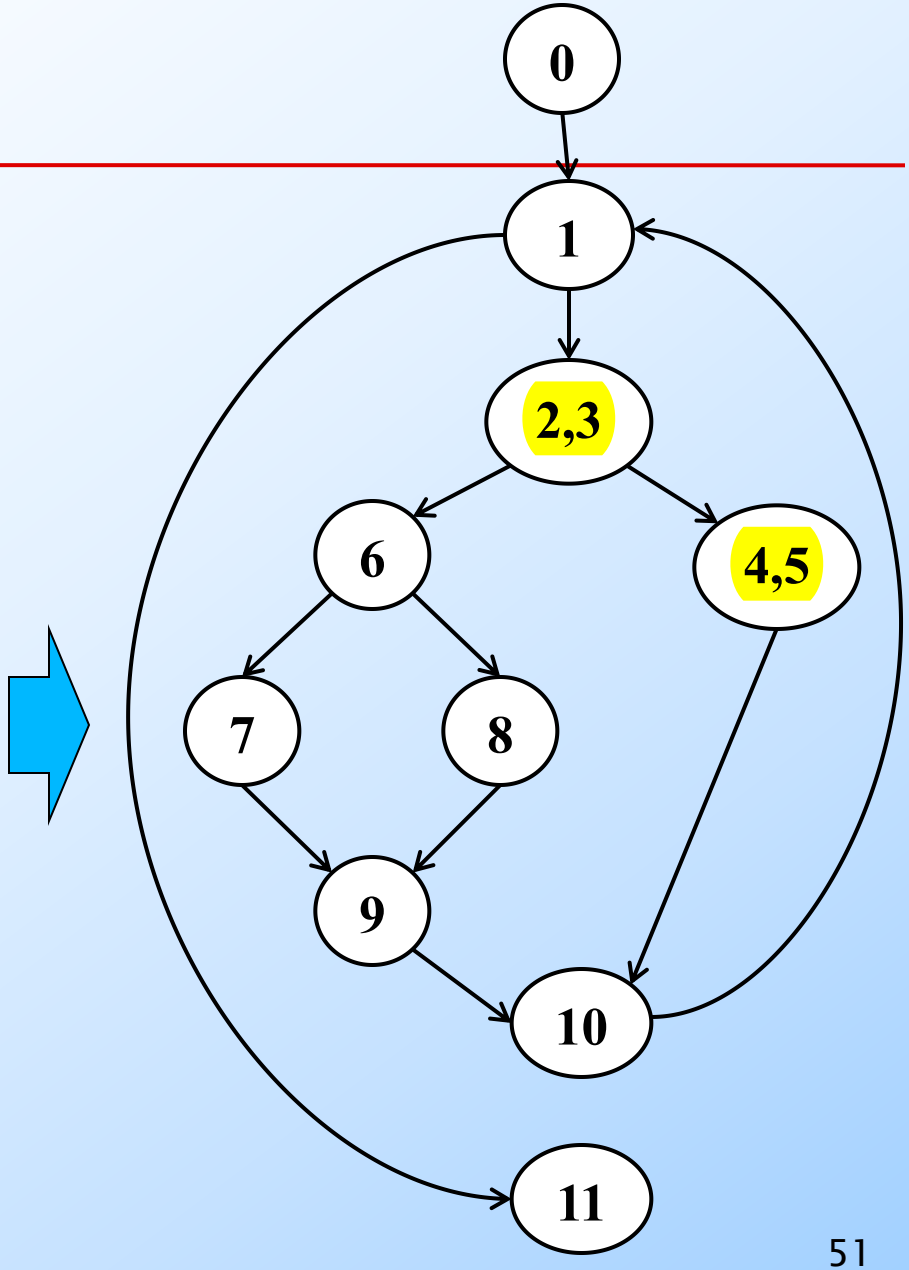
# 控制流图



# 控制流图



# 白盒测试技术-控制流图节点覆盖

- 对于控制流图中每个语法上可达的节点，测试用例所执行的测试路径的集合中至少存在一条测试路径访问该节点。
- 节点覆盖和语句覆盖是等价的。

# 白盒测试技术-控制流图边覆盖

- 对于控制流图中每一个可到达的长度小于等于1的路径，测试用例所执行的测试路径的集合中至少存在一条测试路径游历该路径
- 边覆盖包含节点覆盖，且边覆盖也可以实现分支覆盖

# 白盒测试技术-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。

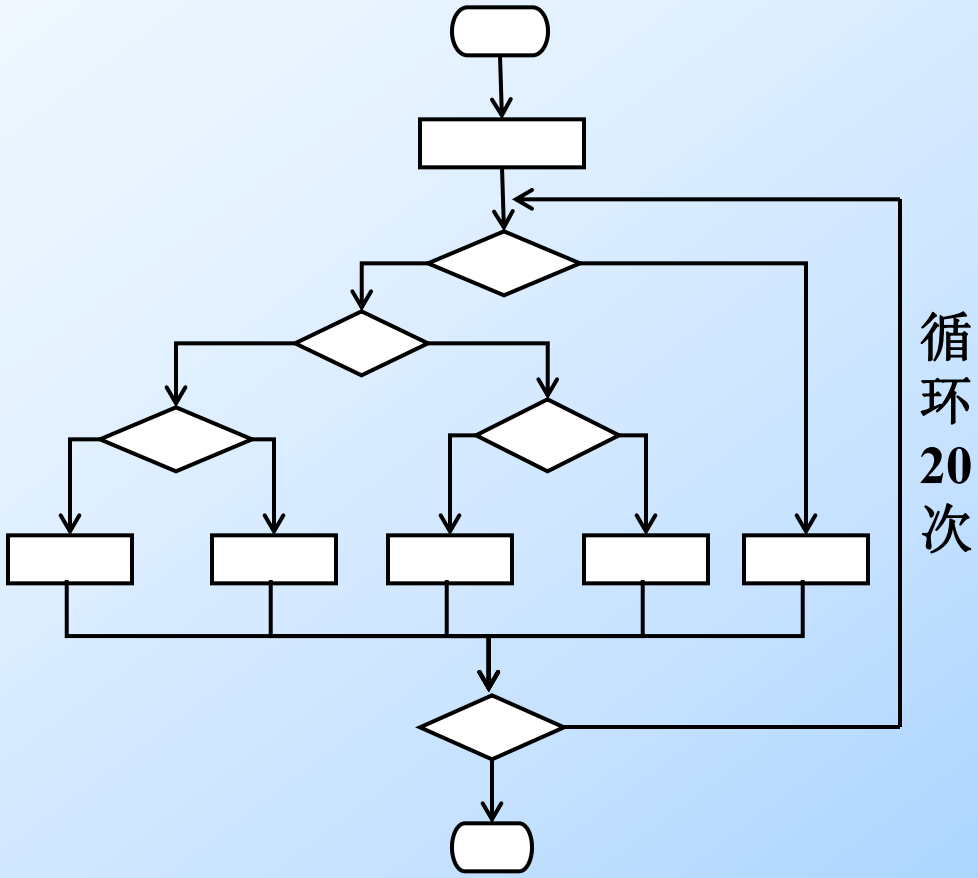
# 白盒测试技术-基本路径测试

- 基本路径/独立路径测试

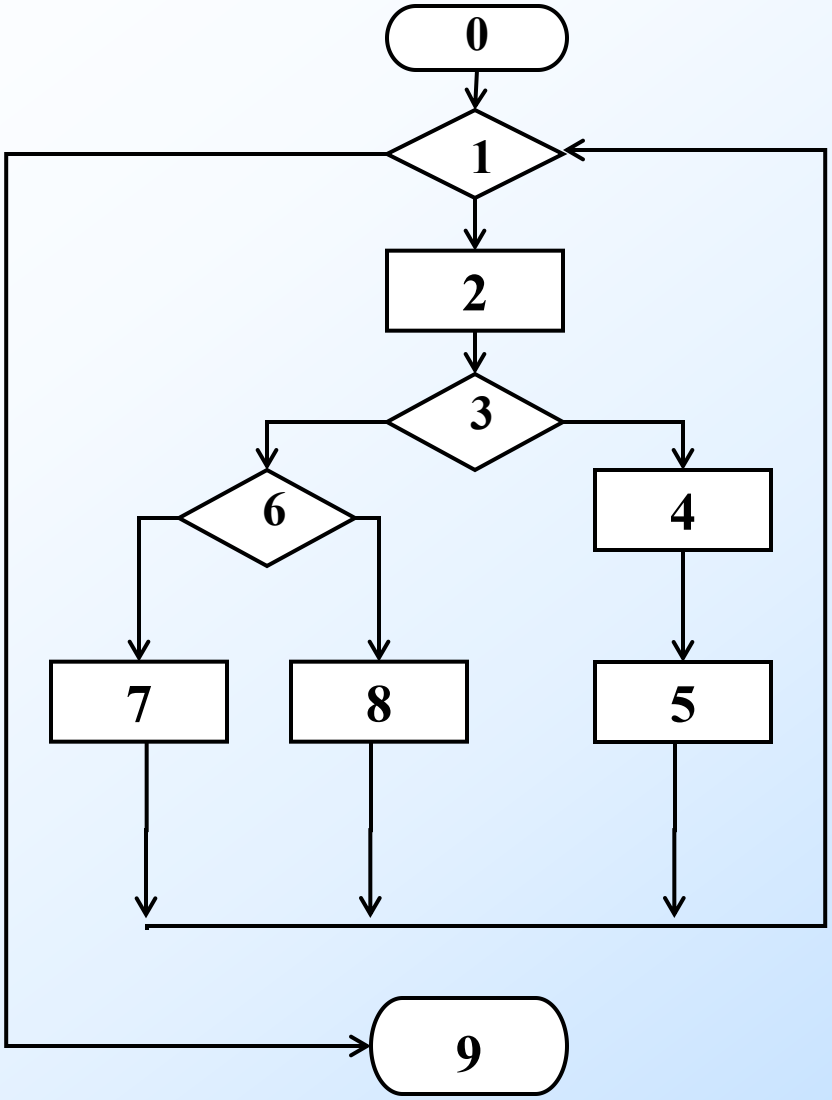
  ①根据流程图画出程序的控制流图（流图、程序图）

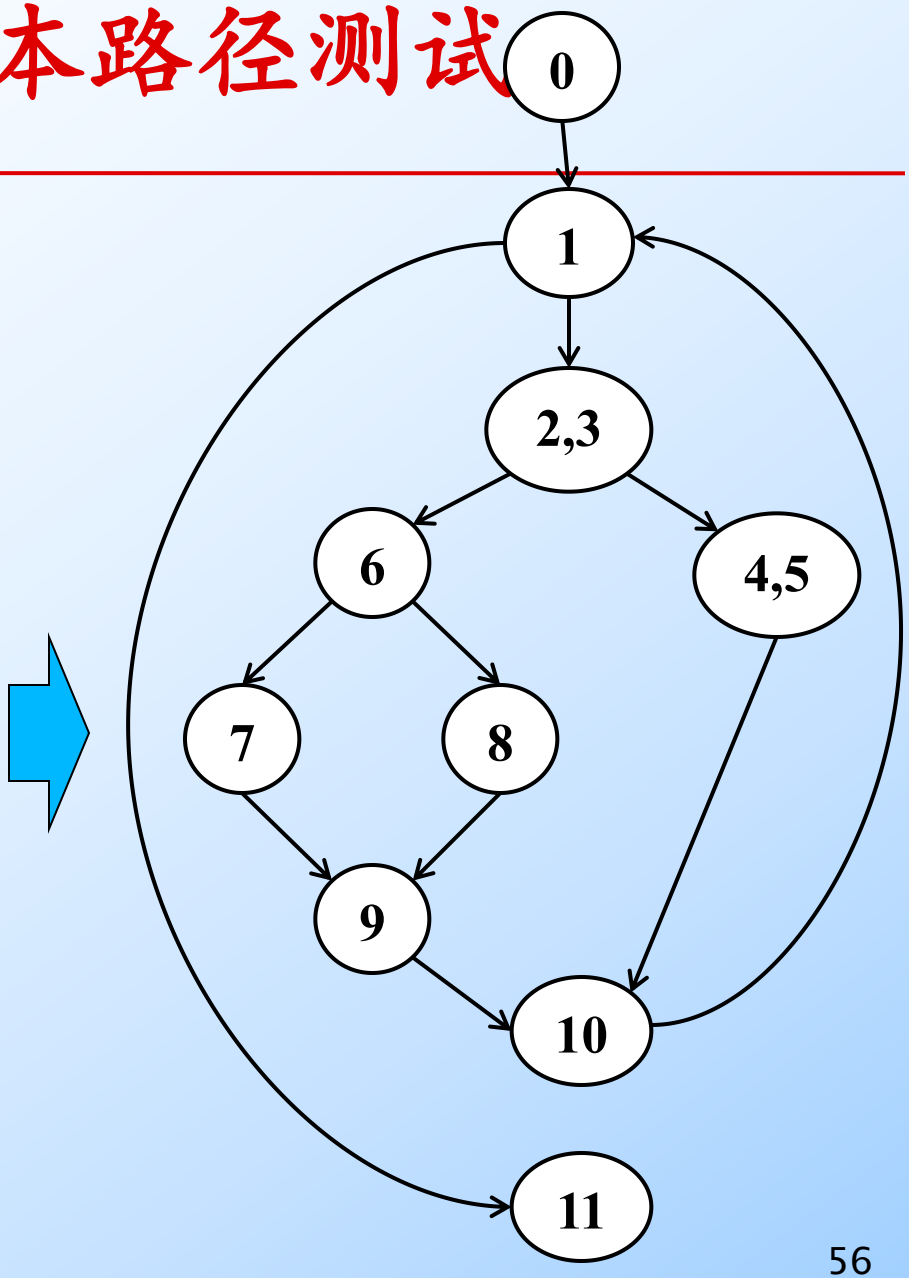
  ②确定基本路径（独立路径）集

  ③设计测试用例，执行基本路径集中的每条路径



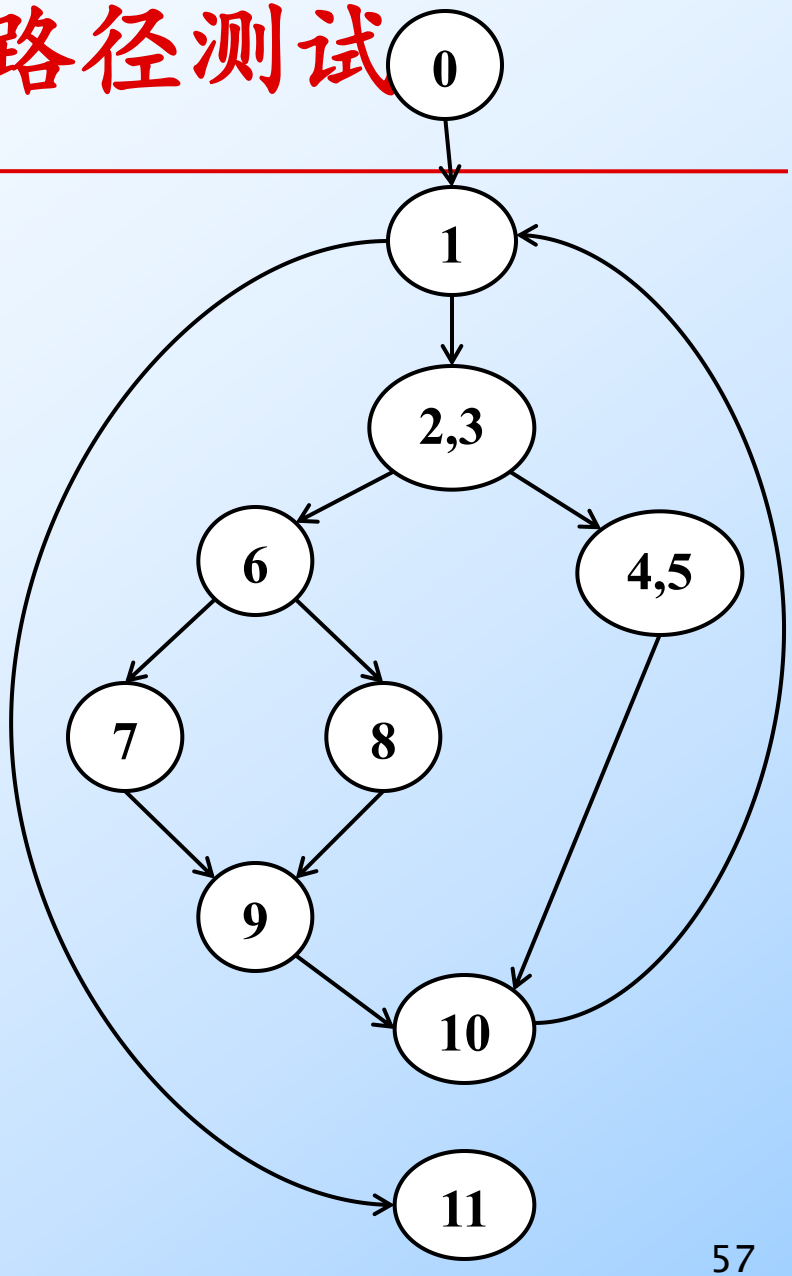
# 白盒测试技术-基本路径测试



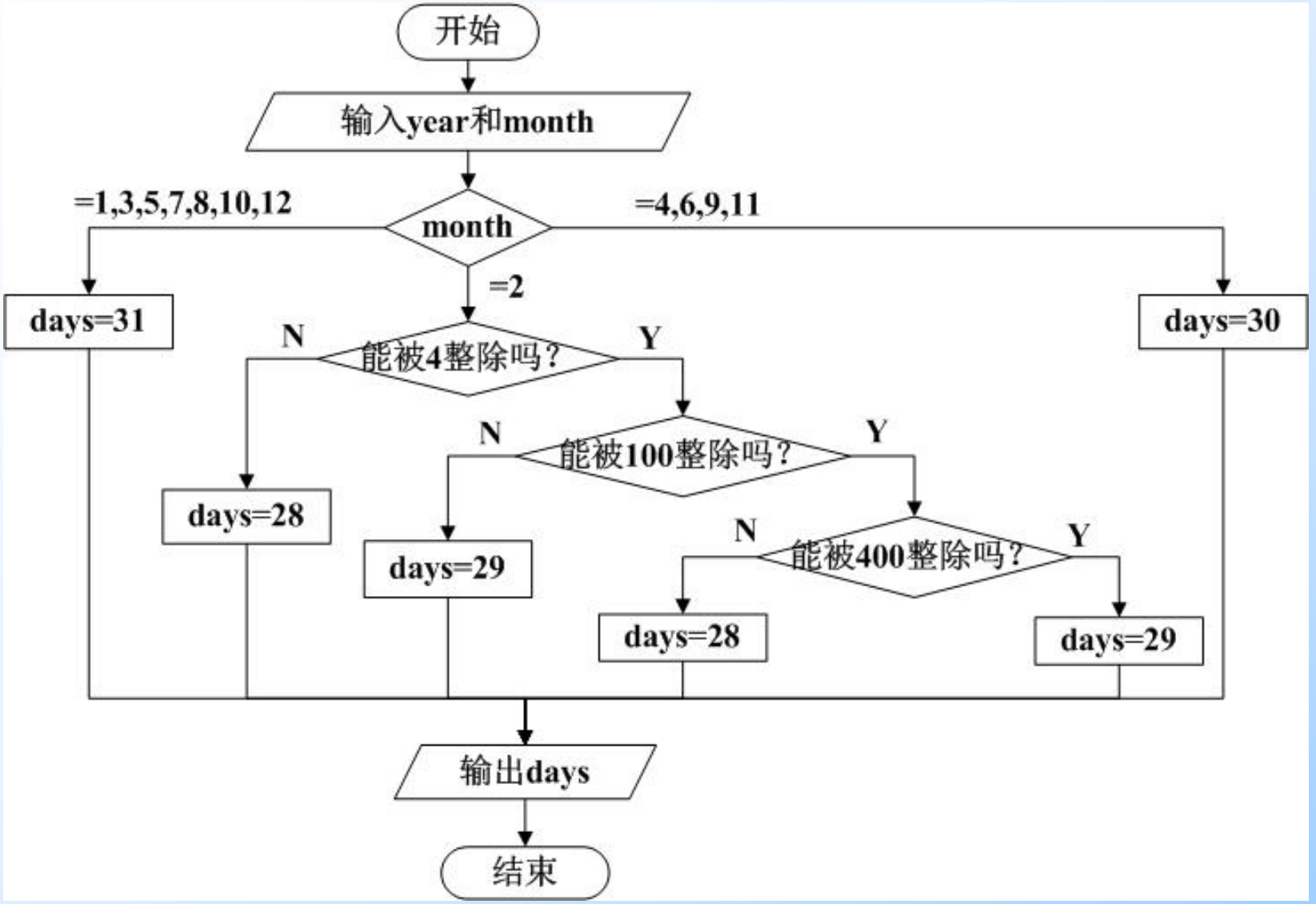



# 白盒测试技术-基本路径测试

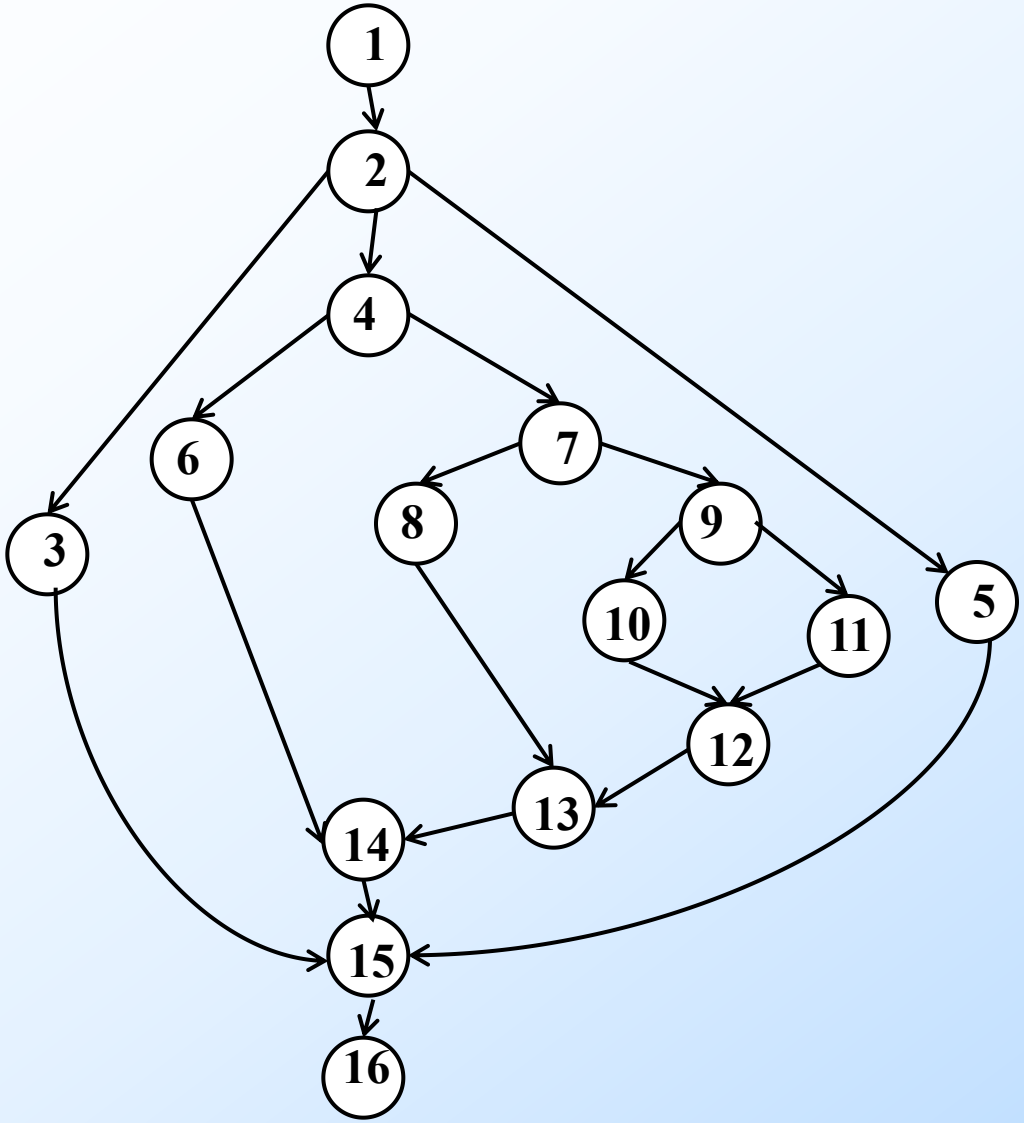
- 独立路径条数：4
  - 0-1-11
  - 0-1-2-3-4-5-10-1-11
  - 0-1-2-3-6-8-9-10-1-11
  - 0-1-2-3-6-7-9-10-1-11



# 案例：基本路径测试


开始
输入year和month
month
=1,3,5,7,8,10,12
=4,6,9,11
=2
days=31
days=30
N
能被4整除吗?
Y
N
能被100整除吗?
Y
days=28
days=29
N
能被400整除吗?
Y
days=28
days=29
输出days
结束

# 案例：基本路径测试




基本路径集：

path1：1-2-3-15-16

path2：1-2-4-6-14-15-16

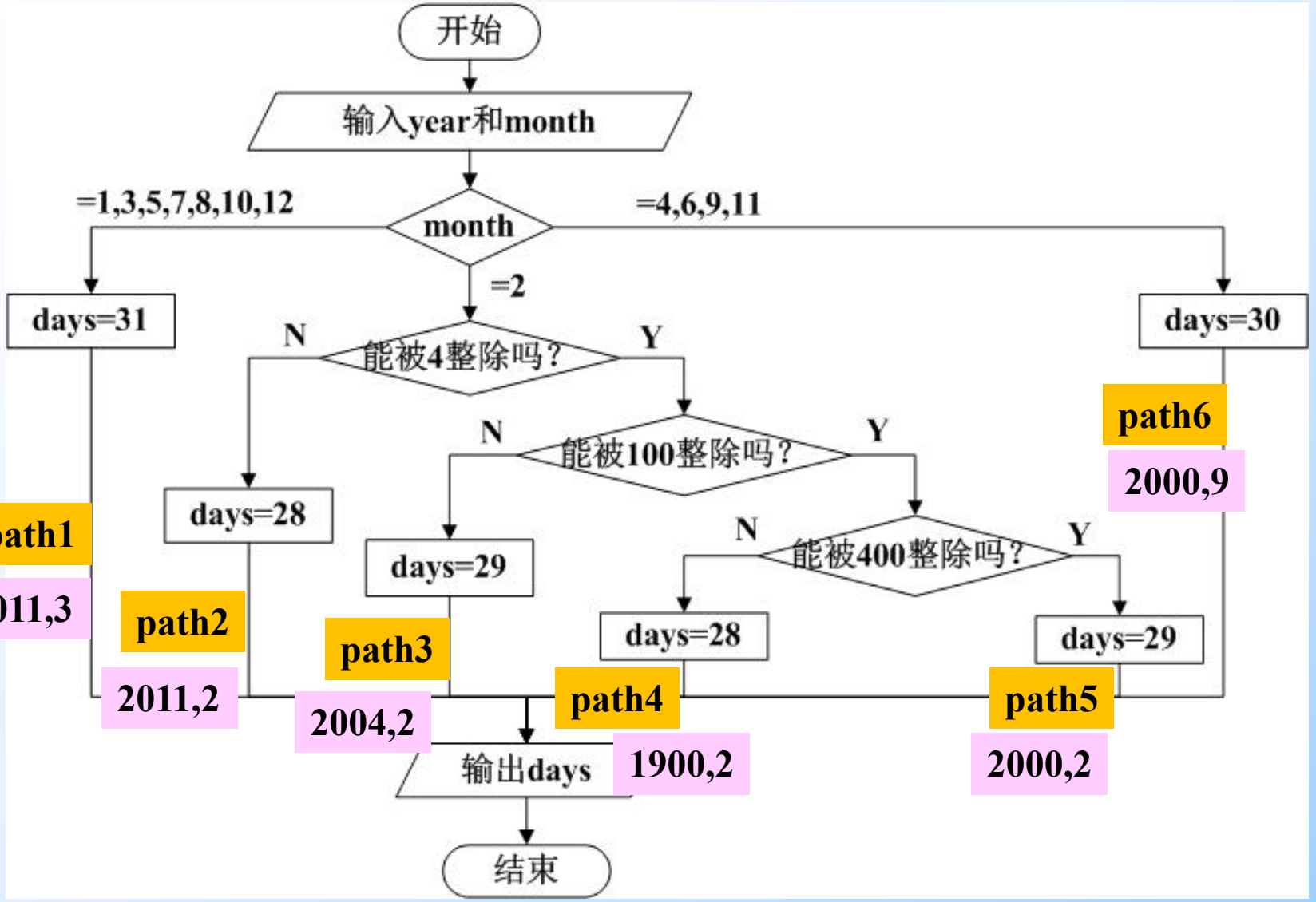
path3：1-2-4-7-8-13-14-15-16

path4：1-2-4-7-9-10-12-13-14-15-16

path5：1-2-4-7-9-11-12-13-14-15-16

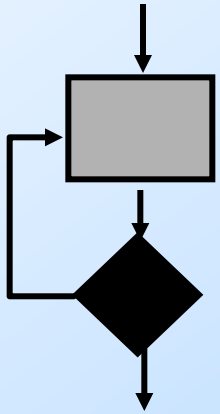
path6：1-2-5-15-16

# 案例：基本路径测试


开始
输入year和month
month
=1,3,5,7,8,10,12
=2
=4,6,9,11
days=31
N
能被4整除吗?
Y
days=30
N
能被100整除吗?
Y
days=28
days=29
N
能被400整除吗?
Y
days=28
days=29
输出days
结束
path1
2011,3
path2
2011,2
path3
2004,2
path4
1900,2
path5
2000,2
path6
2000,9

# 白盒测试技术-循环测试

简单循环的测试集：

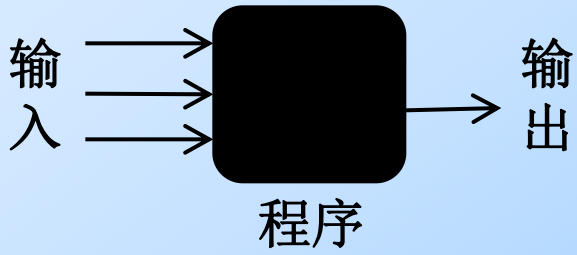


1. 跳过整个循环
2. 一次通过循环
3. 两次通过循环
4. m次通过循环，其中m < n
5. (n-1), n, 和 (n+1) 次通过循环

其中n是允许通过的最大次数

# 黑盒测试技术

- 黑盒测试是把测试对象看做一个黑盒子，测试人员完全不考虑程序内部的逻辑结构和内部特性，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。
- 黑盒测试又叫做功能测试或数据驱动测试。
- 在输入和输出条件中确定测试用例，检查程序能否产生正确输出
  - 是否有不正确或遗漏的功能
  - 输入能否正确接收
  - 能否输出正确结果
  - 是否有数据结构或数据文件访问错误
  - 性能是否满足要求
  - 是否有初始化或终止性错误
  - ……



# 黑盒测试技术

- 穷举测试不可能
  - 假设一个程序有输入量X和Y，以及输出量Z，X和Y只取整数。
  - 穷举测试组合数：
    - $2^{32} \times 2^{32} = 2^{64}$
  - 需要时间：
    - 假设执行一次测试需要1微秒
    - $2^{64} \times 10^{-6}$秒
    - $2^{64} \times 10^{-6}/(3600 \times 24 \times 365)$ =58万年

# 黑盒测试方法

- 等价类划分
- 边界值分析
- 状态测试

# 黑盒测试技术-等价类划分

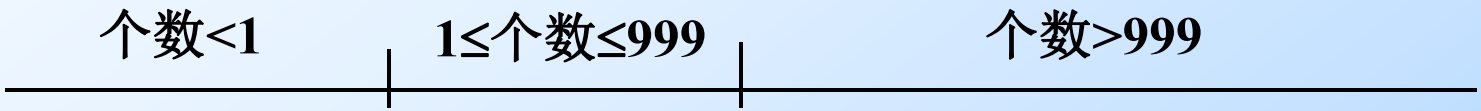
- 等价类划分是一种典型的黑盒测试方法，使用这一方法时，完全不考虑程序的内部结构，只依据程序的规格说明来设计测试用例。
- 等价类划分方法把所有可能的输入数据，即程序的输入域划分成若干部分，然后从每一部分中选取少数有代表性的数据做为测试用例。
- 把输入域划分成若干等价类，据此导出测试用例
- 等价类：输入域的子集，该子集合中，各个输入数据对于揭露程序中的错误是等效的
- 有效等价类和无效等价类

# 黑盒测试技术-等价类划分

- 等价类划分的原则

  ①如果输入数据规定了取值范围，则可以确定一个有效等价类和两个无效等价类

  - 例如：输入数据的个数可以从1到999



# 黑盒测试技术-等价类划分

② 如果规格说明中规定的是一个条件数据，则可确定一个有效等价类和一个无效等价类

- 例如：汽车驾驶员需年满18岁

③ 如果已划分的等价类中元素的处理方式不同，则可将等价类进一步划分

- 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
- 可以确定四个有效等价类：教授、副教授、讲师和助教，教授又可分为教授和博导
- 一个无效等价类：所有不符合以上身份的人员

# 黑盒测试技术-等价类划分

等价类划分方法的测试步骤：

① 根据输入条件确立等价类，建立等价类表，列出所有划分出的等价类；

② 为每一个等价类规定一个唯一编号；

③ 设计一个测试用例，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；

④ 设计一个测试用例，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。

| 输入条件 | 有效等价类 | 无效等价类 |
| --- | --- | --- |
| …… | …… | …… |
| …… | …… | …… |

# 案例：等价类划分测试

- PASCAL语言版本中规定：
  - 标识符由字母开头，后跟字母或数字的任意组合。
  - 有效字符数为8个，最大字符数为80个。
  - 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。

# 案例：等价类划分测试

- 用等价类划分的方法，建立输入等价类表，并为每一个等价类规定一个唯一编号

| 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） |

# 案例：等价类划分测试

- 设计一个测试用例，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

① VAR *x*, T1234567：REAL;
VAR temp: INTEGER
BEGIN *x* := 3.414;
temp := 10;
T1234567 := 2.732;
……

覆盖的等价类：(1)，(2)，(4)，(8)，(9)，(12)，(14)

| (1) | (2) | (4) | (8) | (9) | (12) | (14) |
|---|---|---|---|---|---|---|
| 标识符个数：1个 | 标识符个数：多个 | 标识符字符数：1-8 | 标识符组成：字母 | 标识符组成：数字 | 第一个字符：字母 | 先说明后使用 |

# 案例：等价类划分测试

- 设计一个测试用例，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

①VAR　：REAL;　　(3) 标识符个数为0

②VAR *x*, ：REAL;　　(5) 第2个标识符字符数为0

③VAR T12345678：REAL;　　(6) 标识符字符数多于8个

④VAR T12345.....：REAL;　　(7) 标识符字符数多于80个

⑤VAR T$：CHAR;　　(10) 非字母数字字符

⑥VAR GOTO：INTEGER;　　(11) 保留字

⑦VAR 2T：REAL;　　(13) 第一个字符非字母

⑧VAR PAR：REAL;　　(15) 未说明已使用

```
BEGIN  ......
    PAP := SIN (3.14 * 0.8) / 6;
```

# 黑盒测试技术-边界值分析

- 大量错误发生在输入边界
- 稍大于、稍小于或者等于边界值
- 例如，在做三角形计算时，要输入三角形的三个边长：A、B和C。这三个数值应当满足以下条件才能构成三角形：

$$A>0、B>0、C>0、$$

$$A+B>C、A+C>B、B+C>A$$

- 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“≥”，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
- 边界值分析方法是对等价类划分方法的补充，在等价类的边界上选择测试用例

# 黑盒测试技术-边界值分析

- 边界值选择的原则

①若输入条件指定以a和b为边界的范围，则测试用例应该包括：*a*、*b*、略大于*a*、略小于*a*、略大于*b*、略小于*b*

②若输入条件指定为一组值，则测试用例应该包括其中的：最大值、最小值、略大于最大值、略小于最大值、略大于最小值、略小于最小值

③若内部数据结构有预定义的边界值，则在边界处设计测试用例

④在等价类的边界选择测试用例

# 状态测试

- 在黑盒测试阶段，程序内部的逻辑结构无从得知，可通过对状态的测试间接地加以验证
- 建立状态图
  - 标识出软件可能进入的每一种独立状态。
  - 找出从一种状态转入另一种状态所需的输入和条件。
  - 找出进入或退出某种状态时的设置条件及输出结果。

# 状态测试

- 根据状态转换图设计测试用例
  - 每种状态至少访问一次
  - 测试看起来最常见和最普遍的状态转换
  - 测试状态之间最不常用的分支
  - 测试所有错误状态及其返回值
  - 测试状态的随机转换

# 软件测试策略

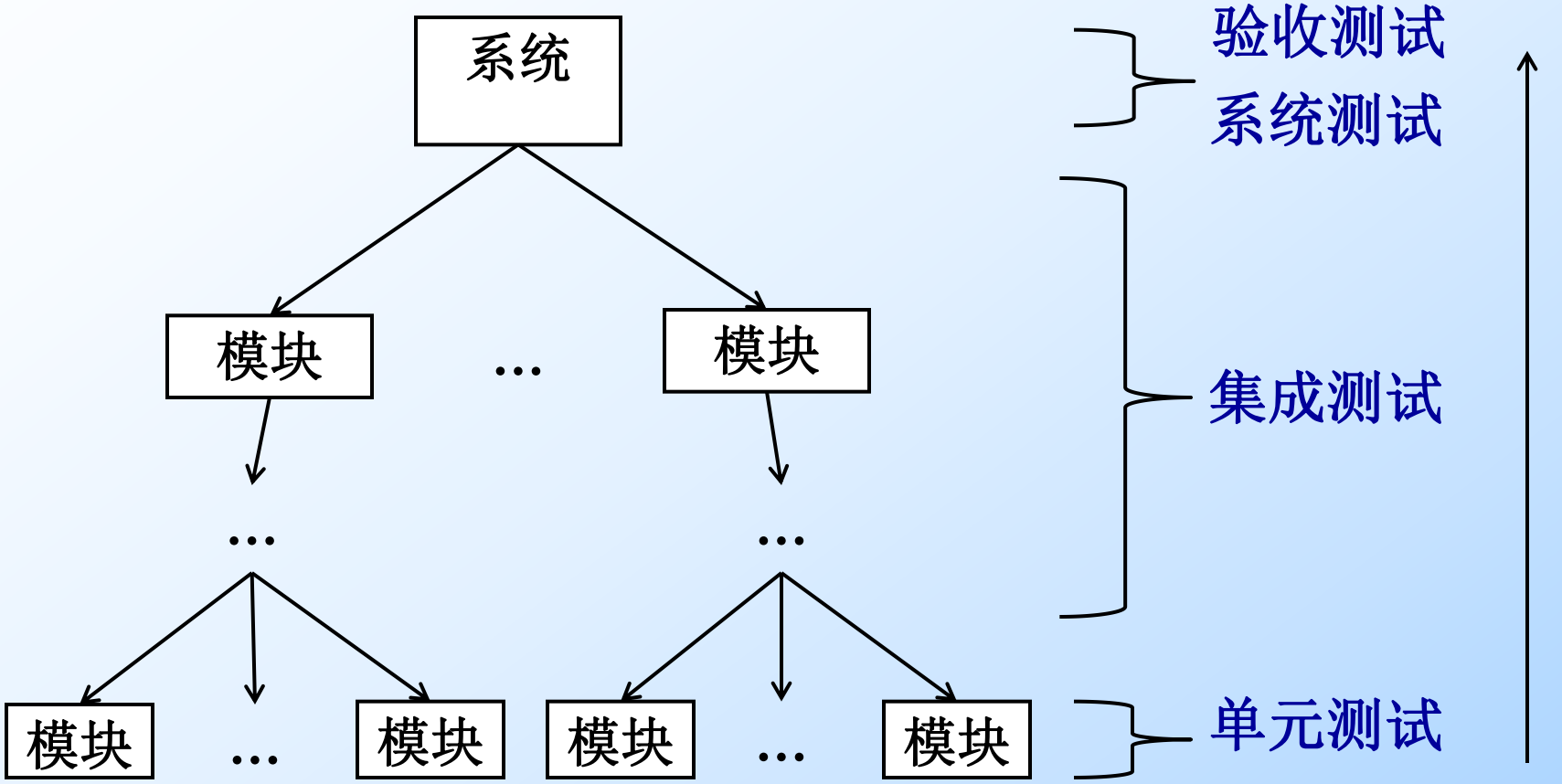
- 如何进行测试？
- 是否应该制定正式的测试计划？
- 是将程序作为一个整体来测试？还是测试其中的一部分？
- 当向系统中加入新的构件时，对于已经做过的测试，是否需要重新测试？
- 什么时候客户参与测试工作？...

- 软件测试策略：路线图

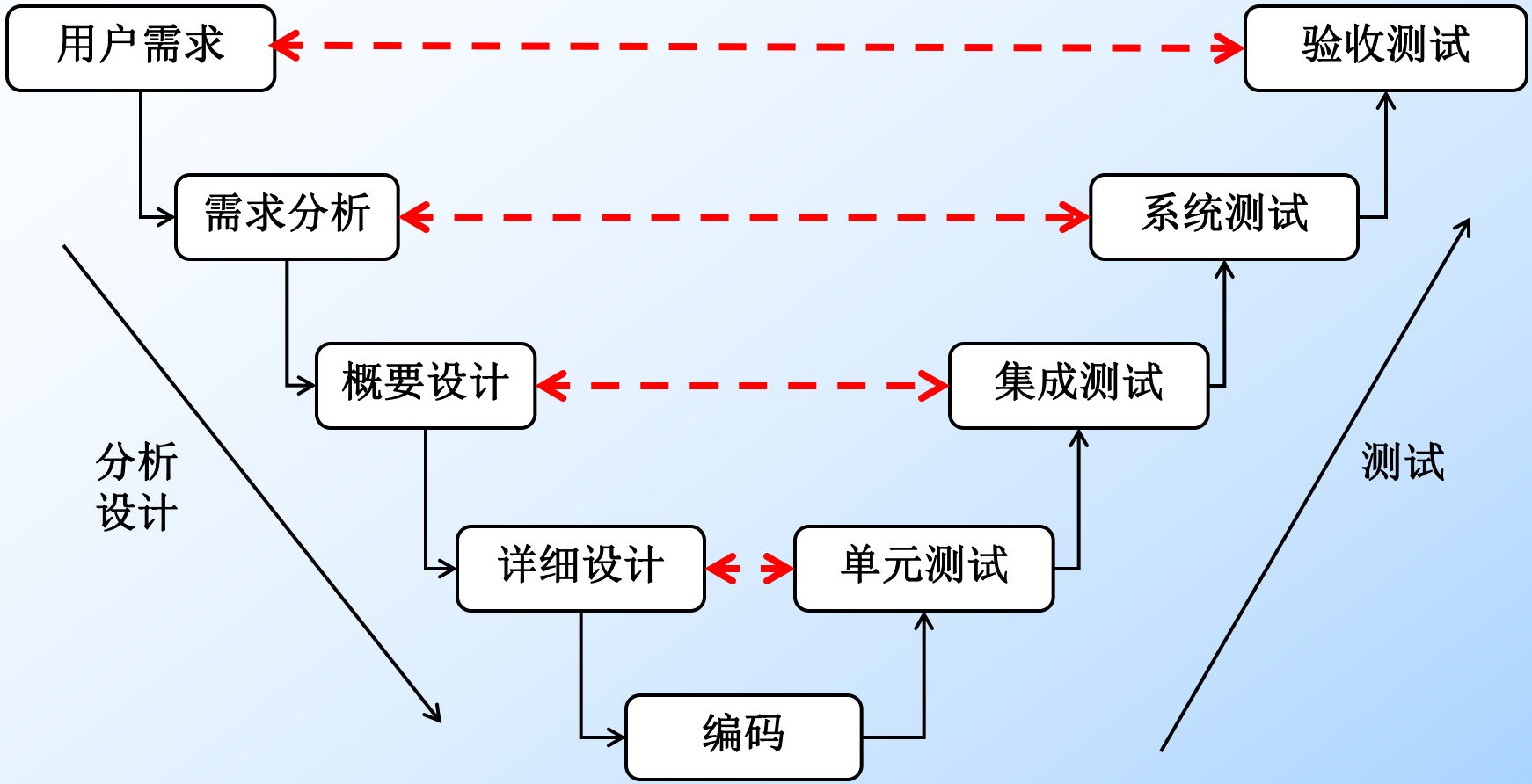
# 软件测试策略

- 软件测试策略为软件开发人员、质量保证组织、和客户提供了一个路线图，规定了测试的主要步骤
- 测试策略必须和测试计划、测试用例设计、测试执行、还有测试结果数据的收集与分析结合在一起
- 测试策略应当具备足够的灵活性，这样在必要的时候它能够有足够的可塑性来应付所有的大软件系统
- 测试策略必须保证足够的严格，这样才能保证对项目的整个进程进行合理的计划和跟踪管理

# 软件测试策略



# 软件测试和分析设计的关系:V模型



# 软件测试和分析设计的关系:V模型

V模型明确标明了测试过程中存在的不同级别，并且清楚描述了这些测试阶段和开发过程期间各阶段的对应关系：

1、单元测试的主要目的是验证软件模块是否按详细设计的规格说明正确运行

2、集成测试主要目的是检查多个模块间是否按概要设计说明的方式协同工作

3、系统测试的主要目的是验证整个系统是否满足需求规格说明

4、验收测试从用户的角度检查系统是否满足合同中定义的需求，以及以确认产品是否能符合业务上的需要

# 软件测试的基本步骤

- 单元测试、集成测试和系统测试的基本步骤：

1. 计划与准备阶段
   - 制定计划
   - 编写与评审测试用例
   - 编写测试脚本和准备测试环境
2. 执行阶段
   - 搭建环境、构造测试数据
   - 执行测试并记录问题
   - 和开发人员一起确认问题
   - 撰写测试报告
3. 返工与回归测试阶段

# 回归测试（Regression testing）

- 在软件测试的各个阶段，需要修正发现的软件缺陷或增加新功能，对软件进行修改可能会引入新的软件缺陷或其他问题。为解决这些问题，需要进行回归测试。
- 回归测试是指有选择地重新测试系统或其组件，以验证对软件的修改没有导致不希望出现的影响，以及系统或组件仍然符合其指定的需求。
- 回归测试可以在所有的测试级别执行，并应用于功能和非功能测试中。
- 回归测试应该尽量采用自动化测试。

# 回归测试的范围

1. 缺陷再测试：重新运行所有发现故障的测试，而新的软件版本已经修正了这些故障。
2. 功能改变的测试：测试所有修改或修正过的程序部分。
3. 新功能测试：测试所有新集成的程序。
4. 完全回归测试：测试整个系统。

# 单元测试（Unit testing）

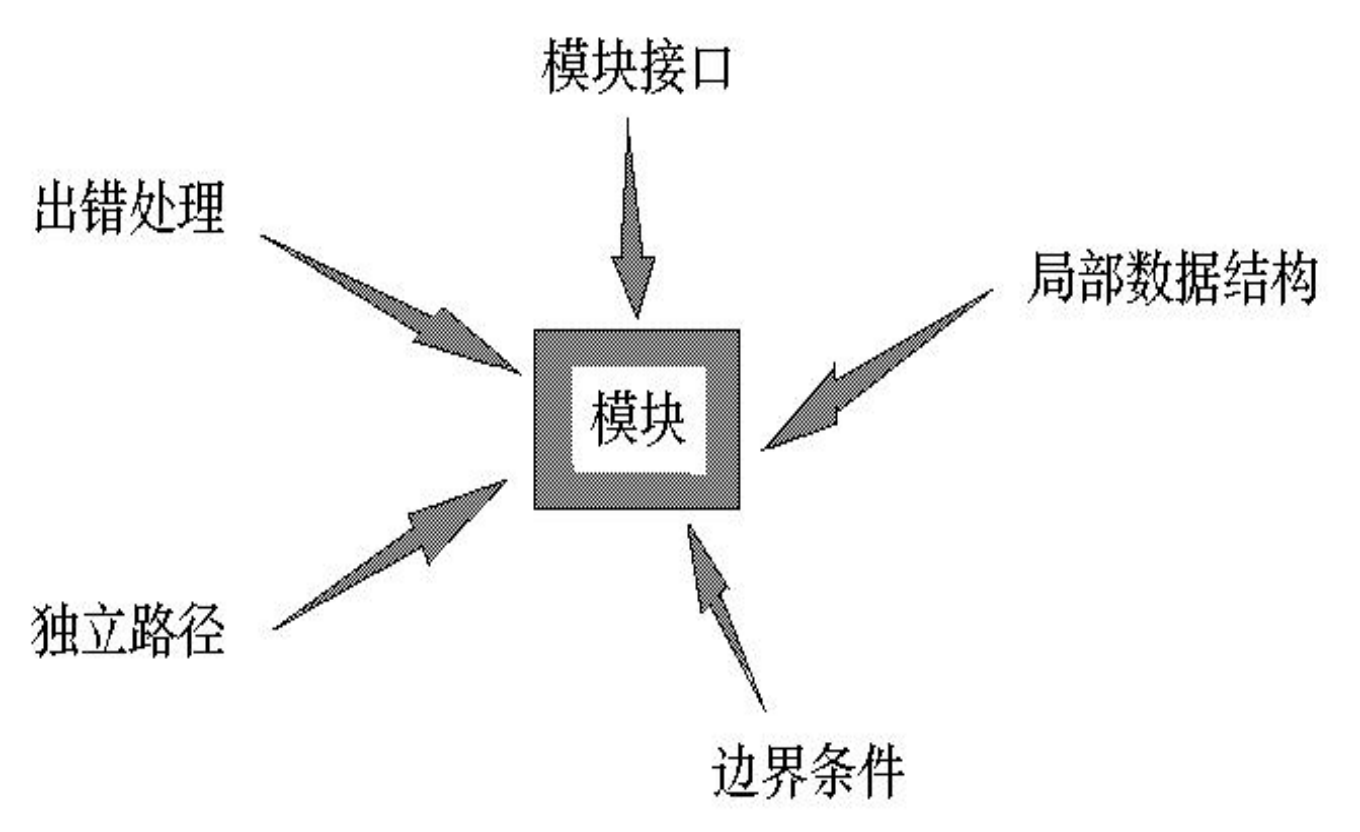
- 单元测试又称模块测试，是针对软件设计的最小单位 — 程序模块，进行正确性检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。
- 编码之后进行，由编码者进行
- 各模块可以独立并行进行
- 单元测试的依据是详细设计
- 测试方法
  - 代码审查
  - 动态测试
- 以白盒测试为主，辅以黑盒测试

软件工程师

# 单元测试

- 进入条件：
  - 被测代码编译链接通过
  - 被测代码静态检查工具检查通过
  - 已完成至少一轮代码审查或走查
  - 单元测试用例的审查通过
  - 单元测试代码写完并通过检测
- 退出条件：
  - 所用测试用例执行通过
  - 单元测试覆盖率达到预定要求
  - 单元测试未被执行的代码进行正式审查

# 单元测试的内容



# 单元测试的内容：模块接口测试

- 对通过本模块的数据流进行测试，包括：
  - 调用本模块的输入参数是否正确
  - 本模块调用子模块时，输入给子模块的参数是否正确
  - 全局量的定义在各模块中是否一致
- 对输入输出操作进行测试，包括：
  - 文件属性是否正确
  - Open与Close语句是否正确
  - 在进行读写操作之前是否打开了文件
  - 文件处理结束后是否关闭了文件
  - I/O错误是否检查并做了处理

# 单元测试的内容：局部数据结构测试

- 不正确或不一致的数据类型说明
- 使用尚未赋值或尚未初始化的变量
- 错误的初始值或错误的缺省值
- 变量名拼写错误或书写错误
- 不一致的数据类型
- 全局数据对模块的影响

# 单元测试的内容：路径测试

- 对模块中重要的执行路径进行测试；
- 查找由于错误的运算导致的错误，如：
  - 运算符不正确，如：“=”和“==”
  - 运算的优先次序不正确
  - 运算对象在类型上不相容
  - 算法错误
- 查找由于不正常的控制流导致的错误，如：
  - 差“1”错，即不正确的多循环一次或少循环一次
  - 错误的或不可能的循环终止条件
  - 不适当地修改了循环变量

# 单元测试的内容：错误处理测试

- 设计良好的模块要求能预见出错的条件，并设置适当的出错处理：
  - 出错的描述是否难以理解？
  - 出错的描述是否能够对错误定位？
  - 显示的错误与实际的错误是否相符？
  - 对错误条件的处理正确与否？
  - 在对错误进行处理之前，错误条件是否已经引起系统的干预？
  - ……

# 单元测试的内容：边界测试

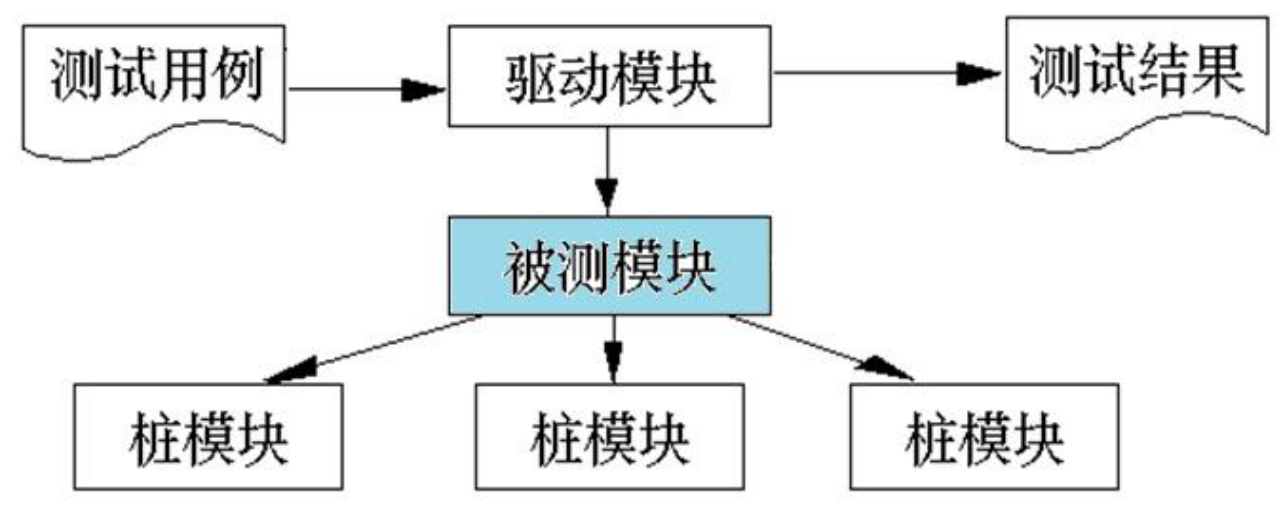
- 边界是容易出错的地方
- 测试数据流或控制流中刚好等于、大于或小于边界值
  - 循环边界
  - 最大值、最小值
- 如果对模块运行时间有要求的话，还要专门进行关键路径测试，以确定最坏情况下和平均意义下影响模块运行时间的因素

# 单元测试用例的设计

- 在单元测试时，测试者需要依据详细设计说明书和源程序清单，了解该模块的I/O条件和模块的逻辑结构
- 主要采用白盒测试的测试用例，辅之以黑盒测试的测试用例
- 对任何合理的输入和不合理的输入，都能鉴别和响应

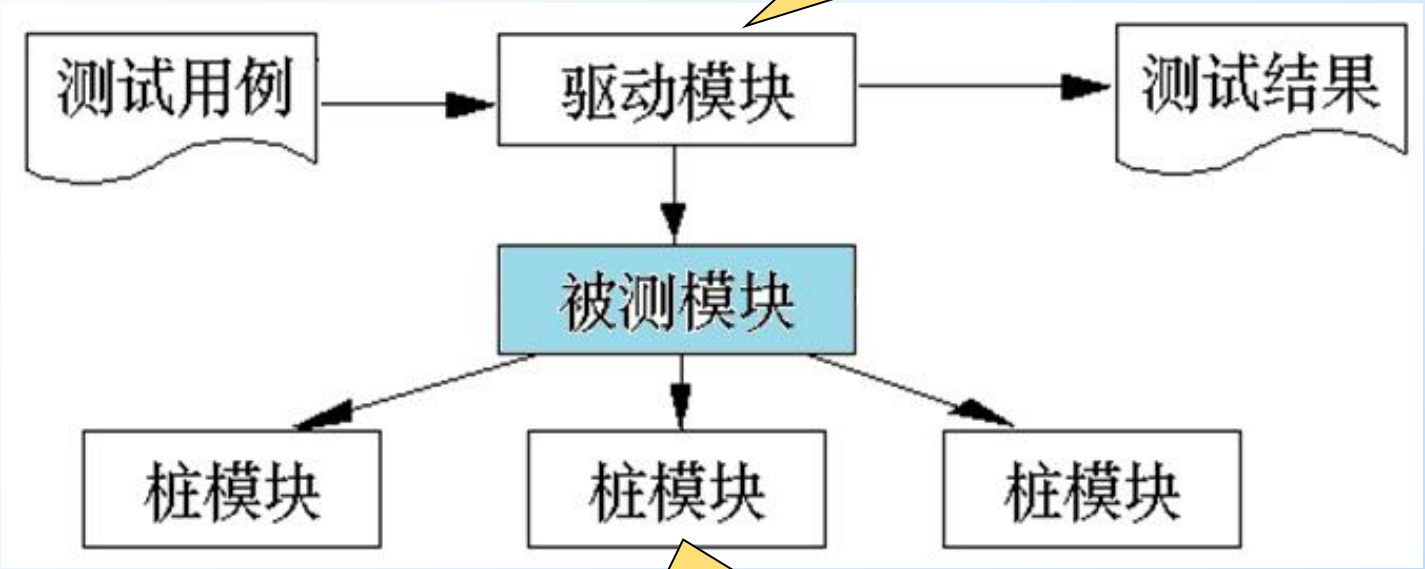
# 单元测试的环境

- 模块并不是一个独立的程序，在考虑测试模块时，同时要考虑它和外界的联系，用一些辅助模块去模拟与被测模块相联系的其它模块。
- 驱动模块（driver）
- 桩模块（stub）—— 存根模块



# 单元测试的环境

代替被测模块的主程序
接收测试数据
传递测试数据给被测模块
输出测试结果




模块独立性高，测试环境就会简单！

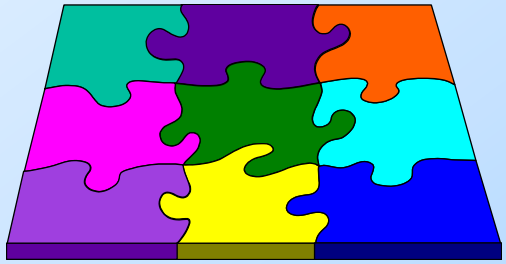
代替被测模块的子模块
不需要子模块的所有功能

# 集成测试

- 集成测试就是将软件集成起来后进行测试。又称为子系统测试、组装测试、部件测试等。集成测试主要可以检查诸如两个模块单独运行正常，但集成起来运行可能出现问题的情况
  - 穿越模块接口是否有数据丢失
  - 一个模块的功能是否对另一个模块产生不利影响
  - 各子功能组合起来，是否达到预期的父功能
  - 全局数据结构是否有问题
  - 单个模块的误差累积起来是否会放大达到不可接受的程度
- 集成测试是一种范围很广的测试，当向下细化时，就成为单元测试。

# 集成测试的方式

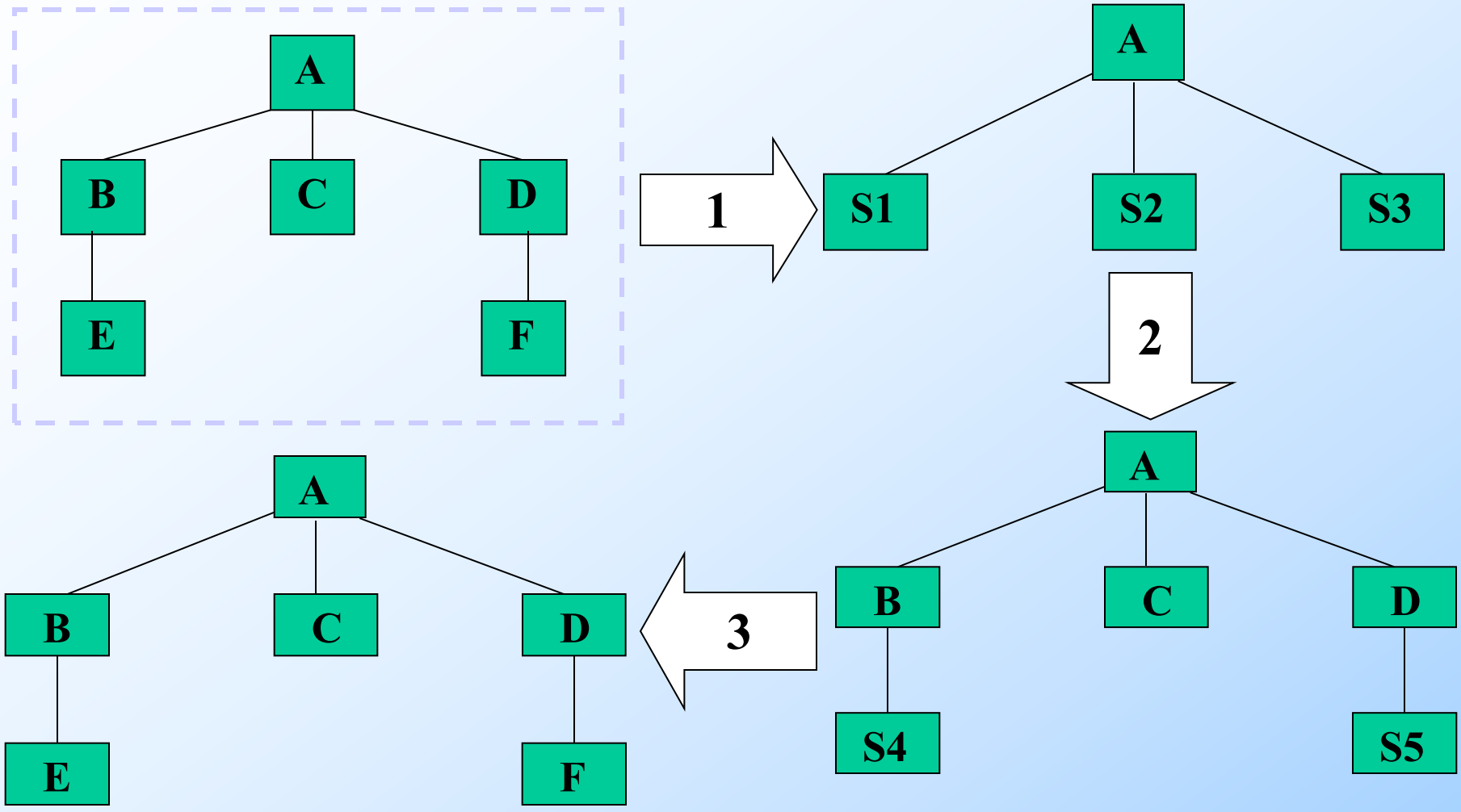
- 渐增式集成
  - 分别对每个模块进行测试
  - 然后将这些模块逐步组装成较大的模块
  - 边组装边测试，最终组装成整个系统
- 渐增式集成方式
  - 自顶向下集成
  - 自底向上集成
  - 混合集成



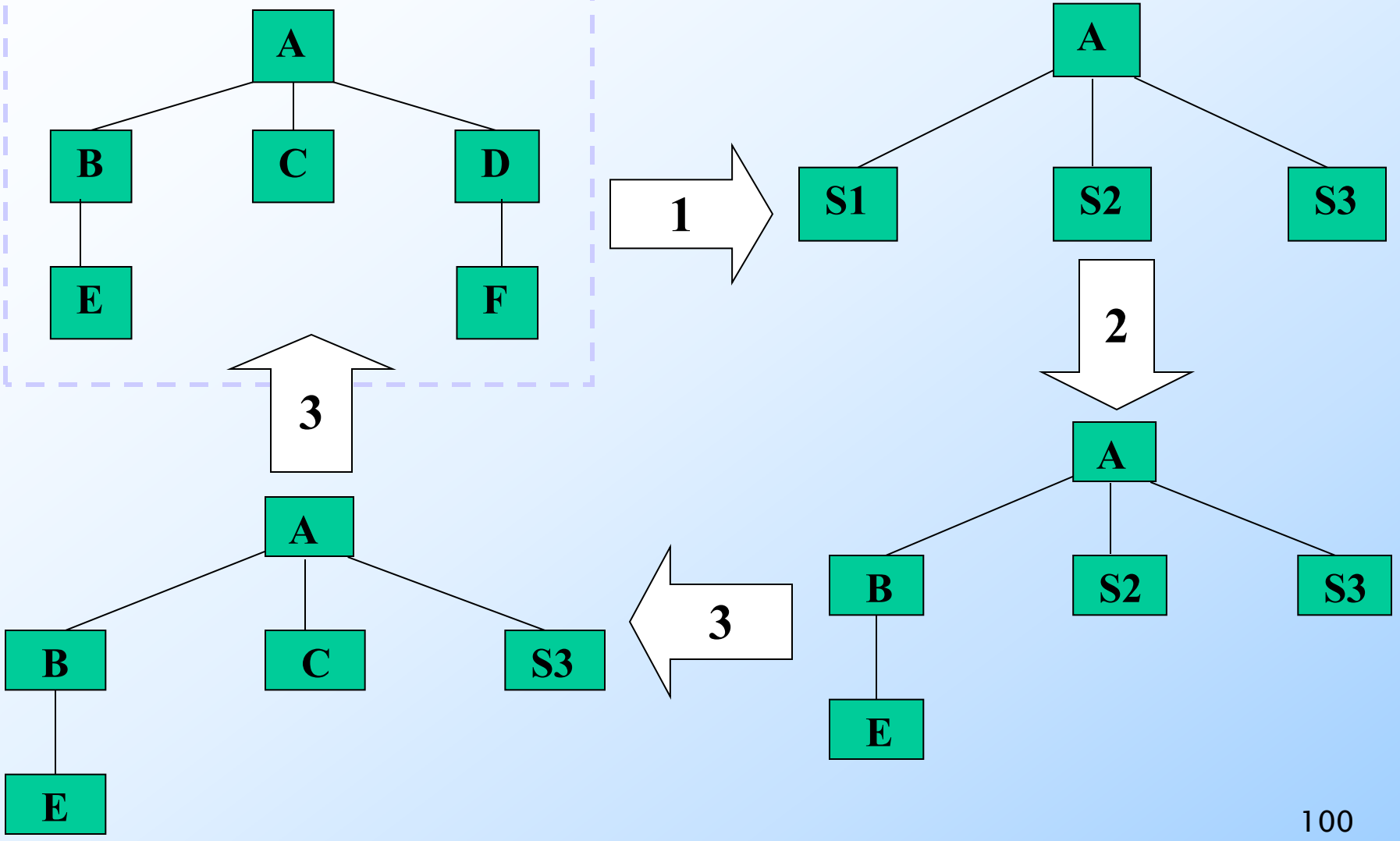
# 集成测试：自顶向下集成

- 按软件结构，沿控制层次自顶向下进行集成
- 两种方式
  - 深度优先方式
    - 从属于主模块的子模块按深度优先的方式（纵向）集成到结构中去
  - 广度优先方式
    - 从属于主模块的子模块按广度优先的方式（横向）集成到结构中去

# 自顶向下集成：广度优先方式



# 自顶向下集成：深度优先方式



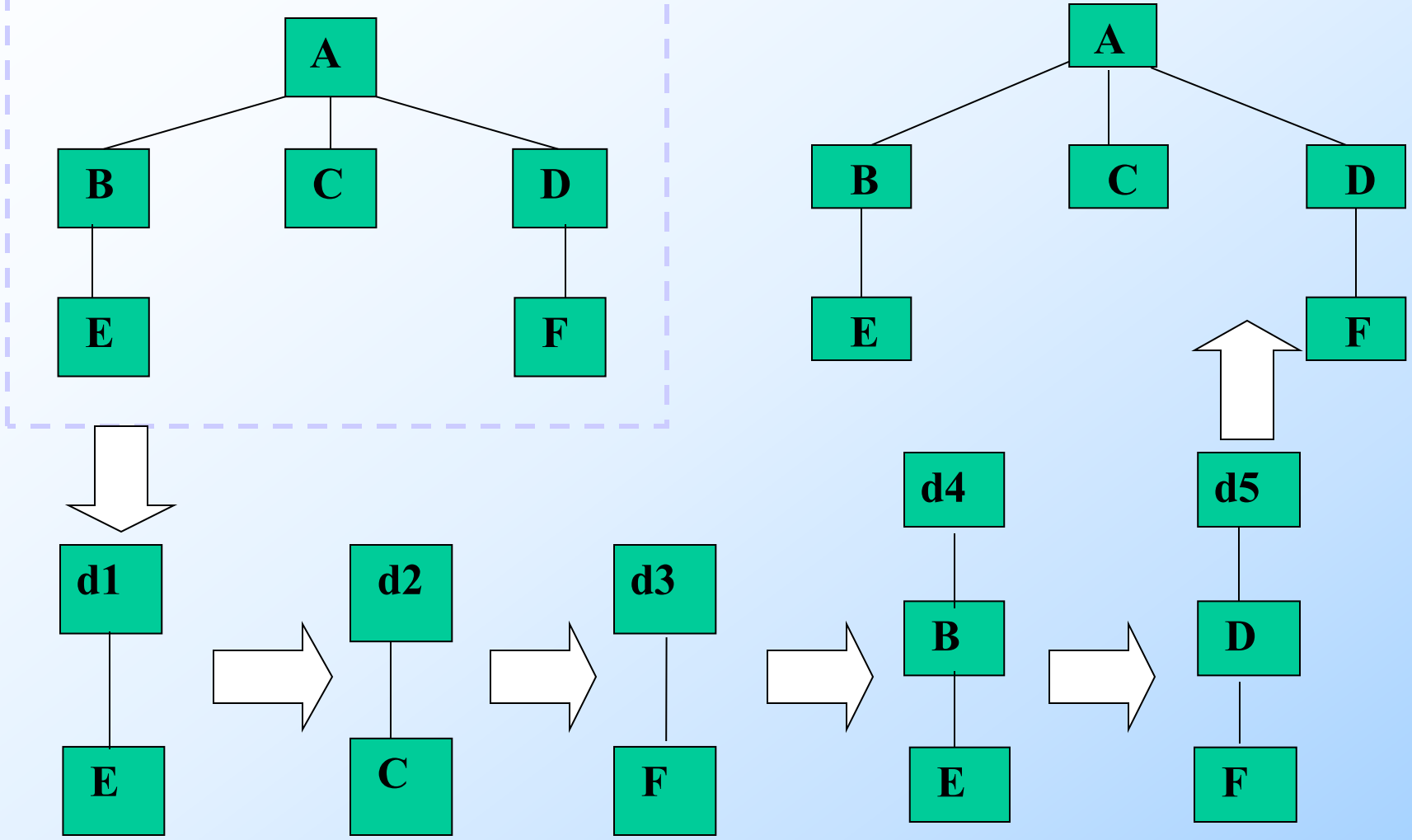
# 集成测试：自顶向下集成

- 优点：
  - 自顶向下的集成方式在测试过程中较早地验证了主要的控制和判断点
  - 选用按深度方向集成的方式，可以首先实现和验证一个完整的软件功能
- 缺点：
  - 桩模块的开发量较大

# 集成测试：自底向上集成

- 从软件结构最底层的模块开始，按照接口依赖关系逐层向上集成
- 优点：不需要使用桩模块进行辅助测试
  - 由于是从最底层开始集成，对于一个给定层次的模块，它的子模块及子模块的所有下属模块已经集成并测试完成
  - 在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到
- 缺点：每个模块都必须编写驱动模块

# 集成测试：自底向上集成



# 集成测试：混合集成方式

- 在实际工作中，常常是综合使用自底向上和自顶向下的混合集成方法
- 例如，按进度优先测试已经完成的模块
  - 如果已完成的模块所调用的模块没有完成，就采用自顶向下的方法，打桩进行测试
  - 如果已经完成模块的上层模块没有完成，可以采用自底向上集成方式

# 集成测试用例的设计

- 首先为通过性测试设计用例，验证需求和设计是否得到满足、软件功能是否得到实现。可以考虑等价类分法、场景分析法、状态图法等。
- 其次为失效性测试设计用例，以可能的缺陷为依据设计测试用例。可以考虑边界值法、错误猜测法、因果图法和状态图法等。
- 强调覆盖率的要求，集成测试的覆盖率有接口覆盖率、接口路径覆盖率等。
- 注意接口有显性和隐性之分。函数调用（API）接口属于显性接口，而消息、网络协议等都属于隐性接口。

# 系统测试

- 系统测试是从用户使用的角度来进行的测试，主要工作是将完成了集成测试的系统放在真实的运行环境下进行测试，用于功能确认和验证。
- 系统测试基本上使用黑盒测试方法
- 系统测试的依据主要是软件需求规格说明

# 为什么进行系统测试？

- 系统测试在软件开发过程中属于必不可少的一环，是软件质量保证的最重要环节。
- 从测试的内容上看，系统测试针对的是外部输入层的测试空间，如果不进行系统测试，那么外部输入层向接口层转换的代码就没有得到测试。此外，许多功能是系统所有组件相互协调中得到的，只能在系统测试级别进行观察和测试。
- 从测试的角度上看，在单元测试和集成测试阶段，测试针对的是各级技术规格说明，即从软件开发者的技术观点的角度考虑的。而系统测试是从客户的观点来考虑系统是否完全正确地满足了需求。

# 系统测试的内容

- 功能性测试
- 性能测试
- 压力测试
- 恢复测试
- 安全测试
- 其它系统测试还包括配置测试、兼容性测试、本地化测试、文档测试、易用性测试等

# 系统测试的内容

- 功能测试
  - 在规定的一段时间内运行软件系统的所有功能，以验证这个软件系统有无错误
- 性能测试
  - 检查系统是否满足在需求说明书中规定的性能,特别是对于实时嵌入式系统
  - 性能测试常常需要与压力测试结合起来进行，并常常要求同时进行硬件和软件检测
  - 通常对软件性能的检测表现在以下几个方面：响应时间、吞吐量、辅助存储区，例如缓冲区、工作区的大小等、处理精度，等等。

# 系统测试的内容

- 强度测试/压力测试
  - 检查系统在运行环境不正常乃至发生故障的情况下，系统可以运行到何种程度的测试。例如：
    - 把输入数据速率提高一个数量级，确定输入功能将如何响应
    - 设计需要占用最大存储量或其它资源的测试用例进行测试
    - 设计出在虚拟存储管理机制中引起“颠簸”的测试用例进行测试
    - 设计出会对磁盘常驻内存的数据过度访问的测试用例进行测试
  - 压力测试的一个变种就是敏感性测试。在程序有效数据界限内一个小范围内的一组数据可能引起极端的或不平稳的错误处理出现，或者导致极度的性能下降的情况发生。此测试用以发现可能引起这种不稳定性或不正常处理的某些数据组合

# 系统测试的内容

- 恢复测试
  - 测试在解决硬件故障（硬件或网络出错等）后，系统能否正常地继续进行工作，并不对系统造成任何损害
  - 可采用人工干预手段，模拟硬件故障，并由此检查：
    - 系统能否发现硬件失效与故障
    - 能否切换或启动备用硬件
    - 在故障发生时能否保护正在运行的作业和系统状态
    - 在系统恢复后能否从最后记录下来的无错误状态开始继续执行作业
  - 掉电测试：测试软件系统在发生电源中断时能否保护当时的状态且不毁坏数据，然后在电源恢复时从保留的断点处重新进行操作

# 系统测试的内容

- 安全测试
  - 检验系统的安全性和保密性措施是否发挥作用、有无漏洞
  - 力图破坏系统的保护机构以进入系统的主要方法有以下几种
    - 正面攻击或从侧面、背面攻击系统中易受损坏的那些部分；
    - 以系统输入为突破口，利用输入的容错性进行正面攻击；
    - 申请和占用过多的资源压垮系统，以破坏安全措施，从而进入系统
    - 故意使系统出错，利用系统恢复的过程，窃取用户口令及其它有用的信息
    - 通过浏览残留在计算机各种资源中的垃圾（无用信息），以获取如口令，安全码，译码关键字等信息
    - 浏览全局数据，期望从中找到进入系统的关键字
    - 浏览那些逻辑上不存在，但物理上还存在的各种记录和资料等

# 验收测试

- 验收测试是以用户为主的测试。软件开发人员和质量保证人员也应参加。由用户参加设计测试用例，使用生产中的实际数据进行测试。
- 通常使用黑盒测试技术
- 除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认
- 交付文档：
  - 验收测试分析报告
  - 最终的用户手册和操作手册
  - 项目开发总结报告

# 验收测试的主要形式

- 根据合同进行的验收测试
  - 重复执行相关的测试用例
- 用户验收测试
  - 客户和最终用户不同
- 现场测试
  - 客户代表执行
  - 分为α测试和β测试

# α 测试

- 在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。
- α 测试是由一个用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的测试。
- α 测试的目的是评价软件产品的FLURPS（即：Functionality功能，Localizability局域化，Usability可使用性，Reliability可靠性，Performance性能，Supportability支持）。尤其注重产品的界面和特色。
- α 测试可以从软件产品编码结束之时开始，或在模块（子系统）测试完成之后开始，也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。

# β 测试

- β 测试是由软件的多个用户在实际使用环境下进行的测试。这些用户返回有关错误信息给开发者。
- 测试时，开发者通常不在测试现场。因而，β 测试是在开发者无法控制的环境下进行的软件现场应用。
- 在 β 测试中，由用户记下遇到的所有问题，包括真实的以及主观认定的，定期向开发者报告。
- β 测试主要衡量产品的FLURPS。着重于产品的支持性，包括文档、客户培训和支持产品生产能力。
- 只有当 α 测试达到一定的可靠程度时，才能开始 β 测试。它处在整个测试的最后阶段。同时，产品的所有手册文本也应该在此阶段完全定稿。

# 本章复习要点

- 软件质量
- 软件质量保证的常用方法
  - 评审、软件测试
- 软件测试技术：
  - 白盒测试：（各种）覆盖、基本路径测试、循环测试
  - 黑盒测试：等价类划分、边界值分析、状态测试
- 软件测试策略：
  - 单元测试、集成测试、系统测试、验收测试
  - 回归测试、α测试、β测试

# 作业

1. 阅读下面的程序，并按要求进行解答。

```
public String foo(int x, int y)
{
    boolean z;
    if(y>10)
        return "C";
    if(x<y)
        z=true;
    else
        z=false;
    if(z&&x+y==10)
        return "A";
    else
        return "B";
}
```

（1）设计测试用例，达到分支覆盖标准。

（2）设计测试用例，达到条件组合覆盖标准。

（3）绘出代码对应的CFG图。

（4）设计测试用例，达到节点覆盖标准。

（5）设计测试用例，进行基本路径测试。

# 作业

2. 用等价类划分方法设计足够的测试用例进行如下测试。要求：在对应表格中完善等价类表和对应的测试用例。

某镇的行政代码有3部分组成：

地区：空白或3位数字；

前缀：非‘0’或‘1’开头的3位数字；

后缀：任意4位数字。

程序应接受符合条件的号码，拒绝不符合条件的号码。

# 等价类表

| 输入条件 | 有效等价类 | 无效等价类 |
| --- | --- | --- |
| 地区码 | | |
| 前　缀 | | |
| 后　缀 | | |

# 测试用例

| 方案 | 内容 | | | 输　入 | 预期输出 |
|---|---|---|---|---|---|
| | 地区码 | 前缀 | 后缀 | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |