

软件工程

第十二章 软件测试

乔立民

qlm@hit.edu.cn

2011年5月25日

主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.3 软件测试战术

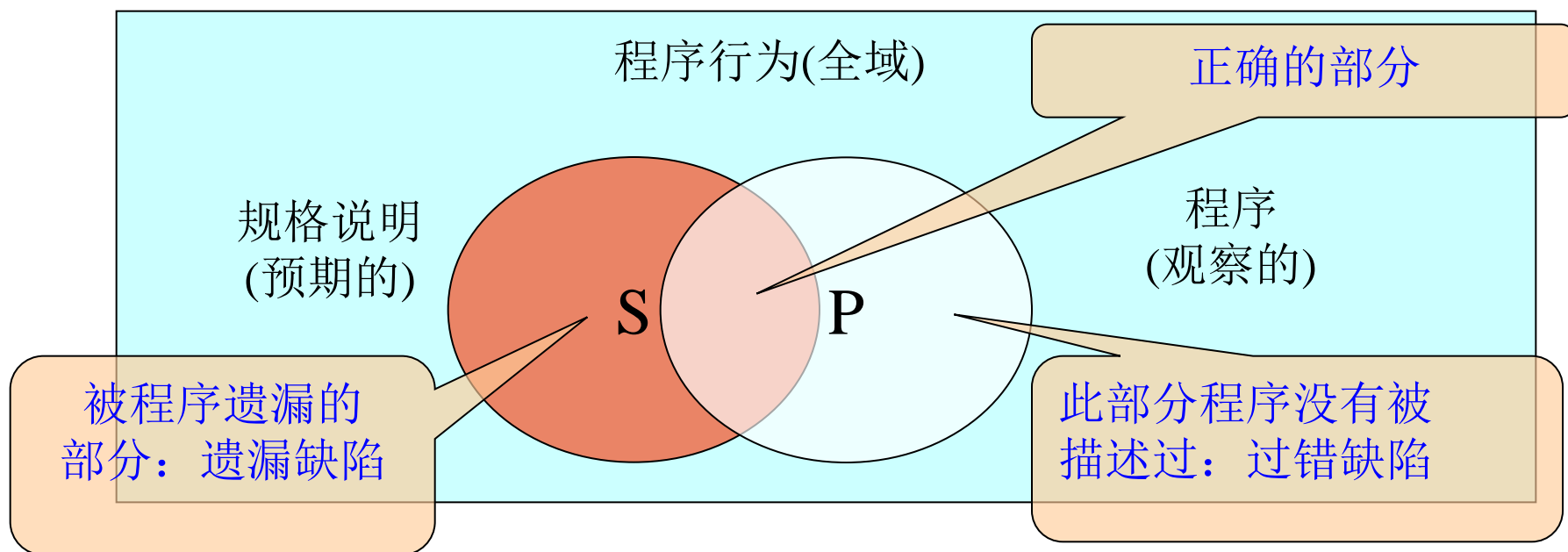
1 软件测试的概念

- **IEEE:** 测试是使用人工和自动手段来运行或检测某个系统的过程，其目的在于检验系统是否满足规定的需求或弄清预期结果与实际结果之间的差别。

该定义明确提出了软件测试以“检验是否满足需求”为目标。

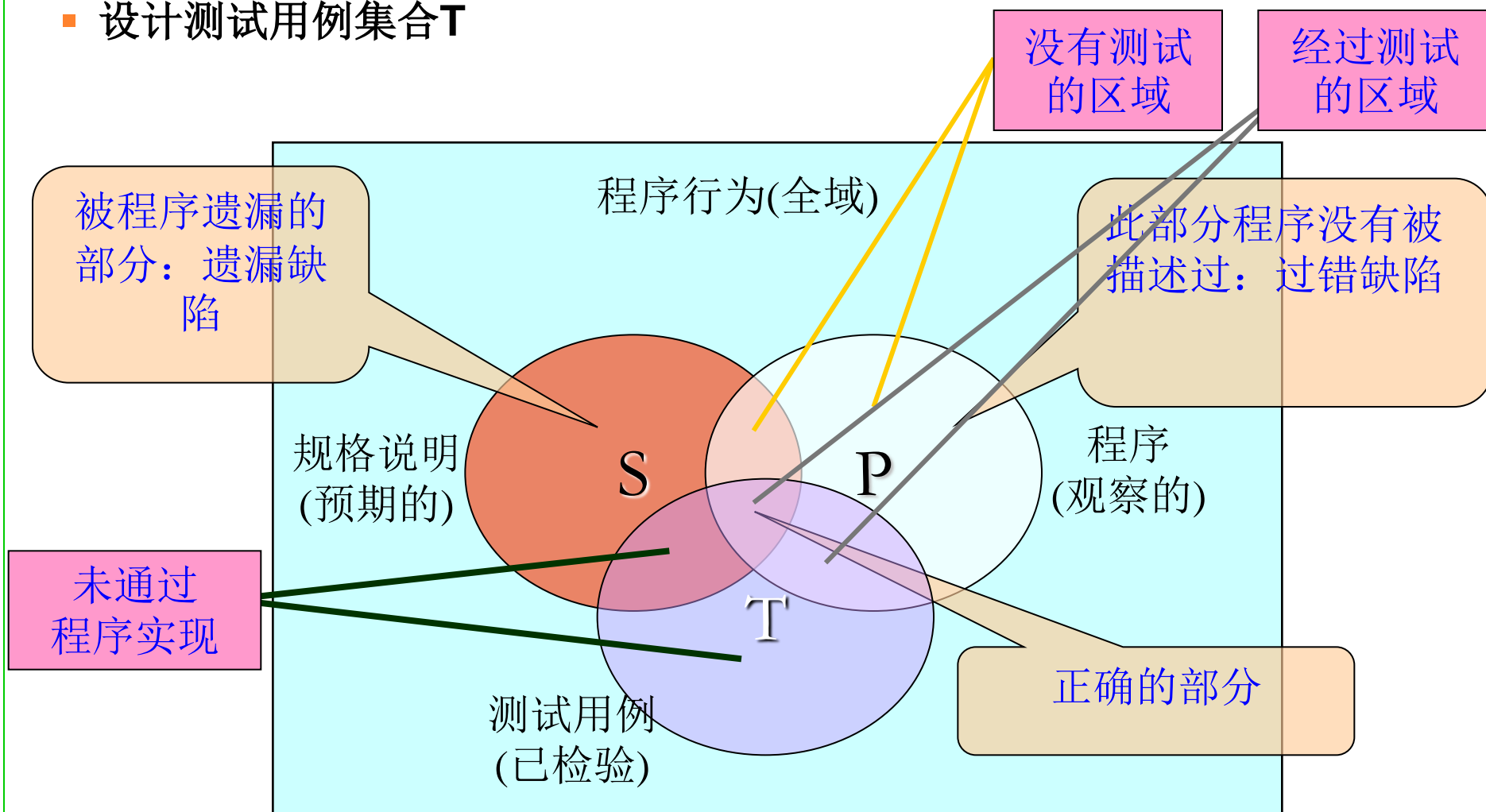
用Venn Diagram来理解测试

- 考虑一个程序行为全域，给定一段程序及其规格说明
 - 集合S是所描述的行为；
 - 集合P是用程序实现的行为；



用Venn Diagram来理解测试

- 设计测试用例集合T



2 软件测试的目标

- 找到错误
- **Glen Myers**关于软件测试目的提出以下观点：
 - 测试是为了发现错误而执行程序的过程
 - 测试是为了证明“程序有错”，而无法证明“程序正确”
 - 一个好的测试用例在于能够发现至今未发现的错误
 - 一个成功的测试是发现了至今未发现的错误的测试

3 软件测试组织



开发者

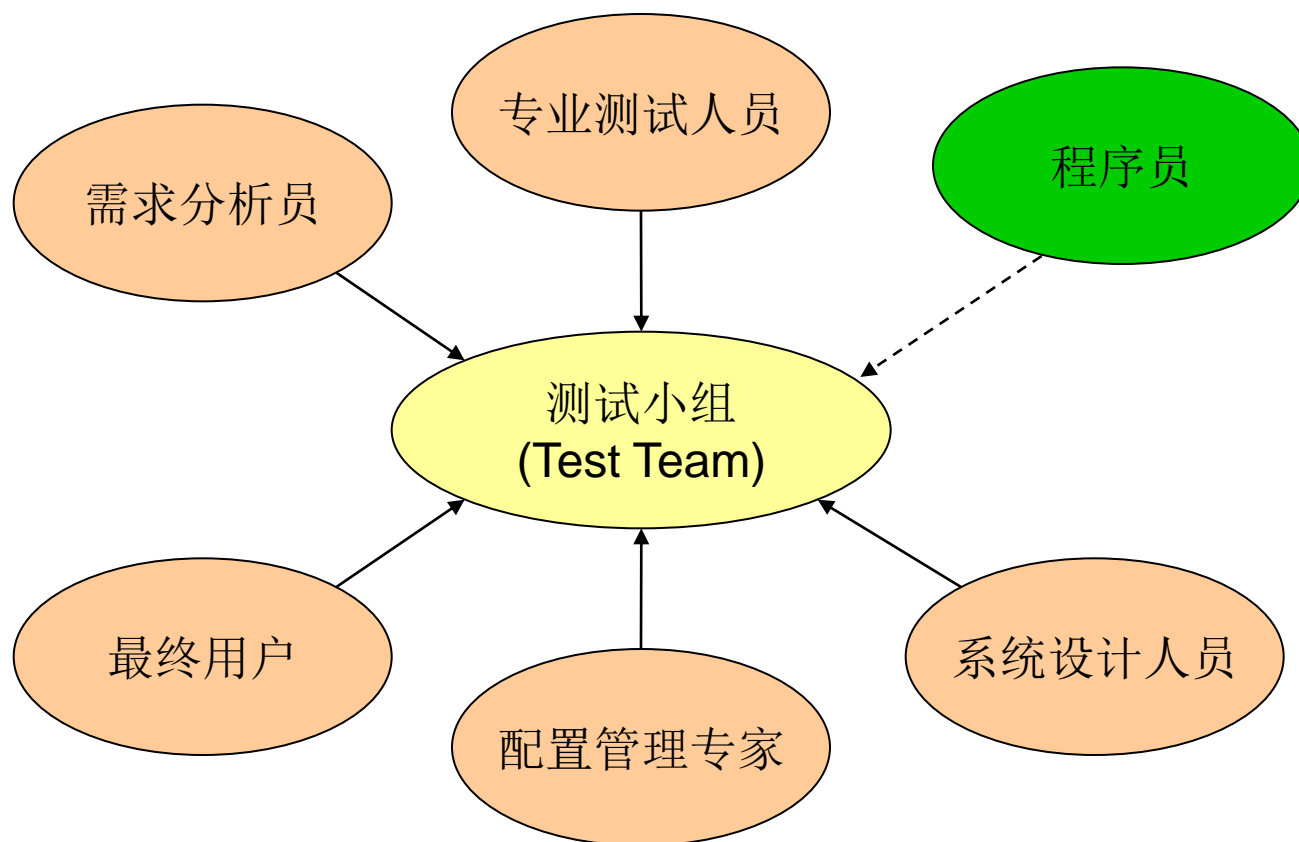
系统的构建者，理解系统
测试是“温和”的，试图证明正确性
发布驱动



独立测试组

必须学会系统
测试是破坏性的，试图证明错误存在
质量驱动

3 软件测试组织



软件测试人员的素质要求

■ 沟通能力

- 理想的测试人员必须能与测试涉及到的所有人进行沟通，具有与技术人员(开发者)和非技术人员(客户、管理人员)的交流能力。

■ 移情能力

- 和系统开发有关的所有人员(用户、开发者、管理者)都处于一种既关心又担心的状态中。测试人员必须和每一类人打交道，因此需要对每一类人都具有足够的理解和同情，从而将测试人员与相关人员之间的冲突和对抗减少到最低程度。

■ 技术能力

- 一个测试人员必须既明白被测软件系统的概念又要会使用工程中的那些工具，最好有几年以上的编程经验，从而有助于对软件开发过程的较深入理解。

软件测试人员的素质要求

■ 自信心

- 开发人员指责测试人员出了错是常有的事，测试人员必须对自己的观点有足够的自信心。

■ 外交能力

- 当你告诉某人他出了错时，就必须使用一些外交方法，机智老练和外交手法有助于维护与开发人员之间的协作关系。

■ 幽默感

- 在遇到狡辩的情况下，一个幽默的批评将是很有帮助的。

■ 很强的记忆力

- 理想的测试人员应该有能力将以前曾经遇到过的类似的错误从记忆深处挖掘出来，这一能力在测试过程中的价值是无法衡量的。

软件测试人员的素质要求

■ 耐心

- 一些质量保证工作需要难以置信的耐心，有时需要花费惊人的时间去分离、识别一个错误。

■ 怀疑精神

- 开发人员会尽他们最大的努力将所有的错误解释过去，测试人员必须听每个人的说明，但他必须保持怀疑直到他自己看过以后。

■ 自我督促

- 干测试工作很容易变得懒散，只有那些具有自我督促能力的人才能够使自己每天正常地工作。

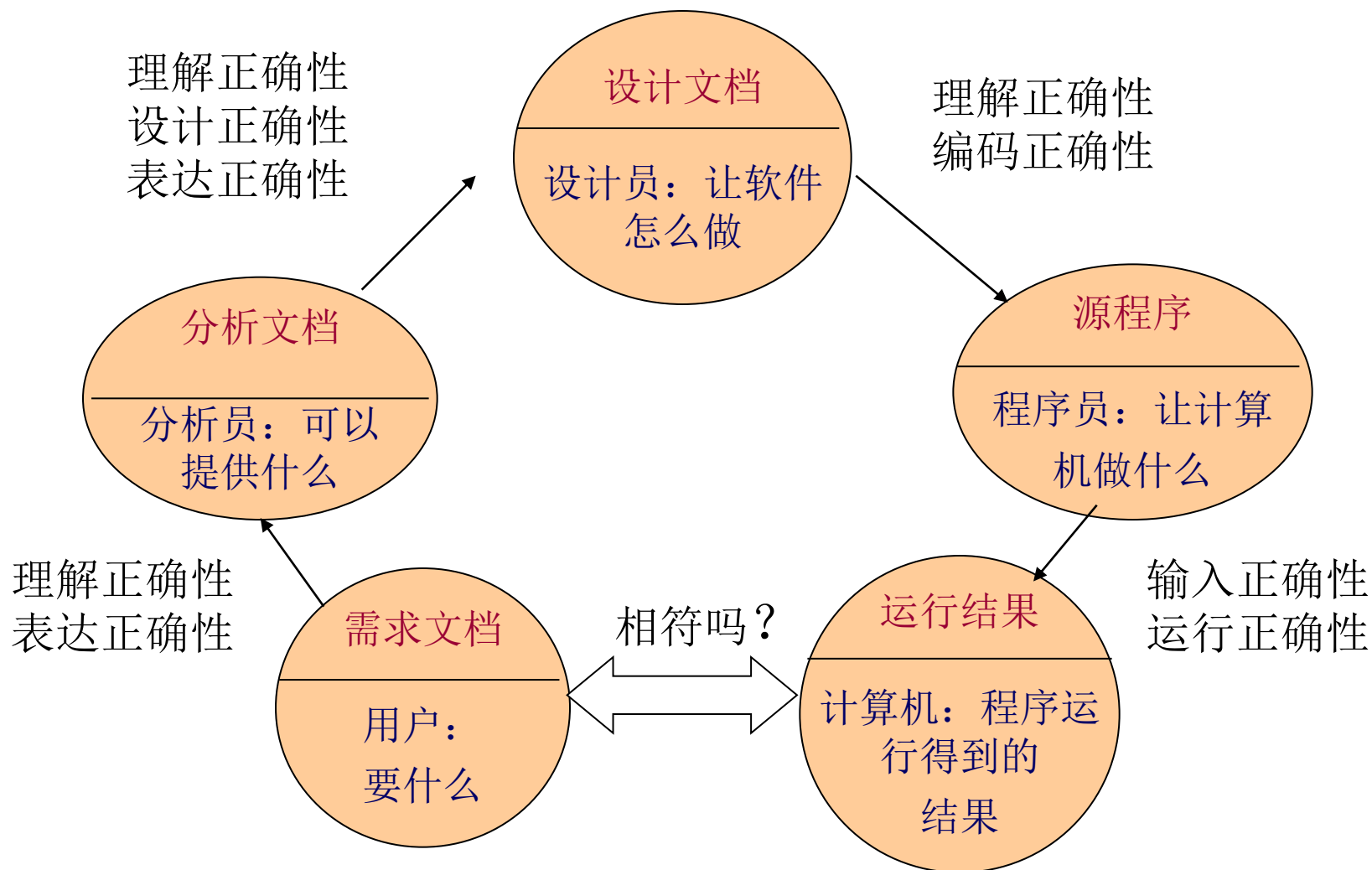
■ 洞察力

- 一个好的测试人员具有“测试是为了破坏”的观点、捕获用户观点的能力、强烈的质量追求、对细节的关注能力。

4 软件测试的对象

- 软件测试并不等于程序测试，应贯穿于软件定义与开发的各个阶段。
- 测试对象包括：
 - 需求规格说明
 - 设计规格说明
 - 源程序

软件测试对象之间关系



主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.2.1 软件测试策略

12.2.2 软件测试步骤

12.2.3 软件调试

12.3 软件测试战术

软件测试策略

- **测试策略：描述将要进行的测试步骤**
 - 测试计划
 - 测试用例设计
 - 测试执行
 - 测试结果的收集与评估

软件测试策略

■ 测试计划(Test Plan)

- 测试计划是测试工作的指导性文档，规定测试活动的范围、方法、资源和进度；明确正在测试的项目、要测试的特性、要执行的测试任务、每个任务的负责人，以及与计划相关的风险。
- 主要内容：测试目标、测试方法、测试范围、测试资源、测试环境和工具、测试进度表

软件测试策略

■ 测试用例(Test Case)

- 测试用例是数据输入和期望结果组成的对，其中“输入”是对被测软件接收外界数据的描述，“期望结果”是对于相应输入软件应该出现的输出结果的描述，测试用例还应明确指出使用具体测试案例产生的测试程序的任何限制。
- 测试用例可以被组织成一个测试系列，即为实现某个特定的测试目的而设计的一组测试用例。例如，一部分测试用例用来测试系统的兼容性，另一部分是用来测试系统在特定的环境中，系统的典型应用是否能够很好地运作。

软件测试策略

■ 测试结果评估

- 缺陷报告是编写在需要调查研究的测试过程期间发生的任何事件，简而言之，就是记录软件缺陷。
- 主要内容：缺陷编号、题目、状态、提出、解决、所属项目、测试环境、缺陷报告步骤、期待结果、附件
- 在报告缺陷时，一般要讲明缺陷的严重性和优先级。
 - 严重性表示软件的恶劣程度，反映其对产品和用户的影响。
 - 优先级表示修复缺陷的重要程度和应该何时修复。

主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.2.1 软件测试策略

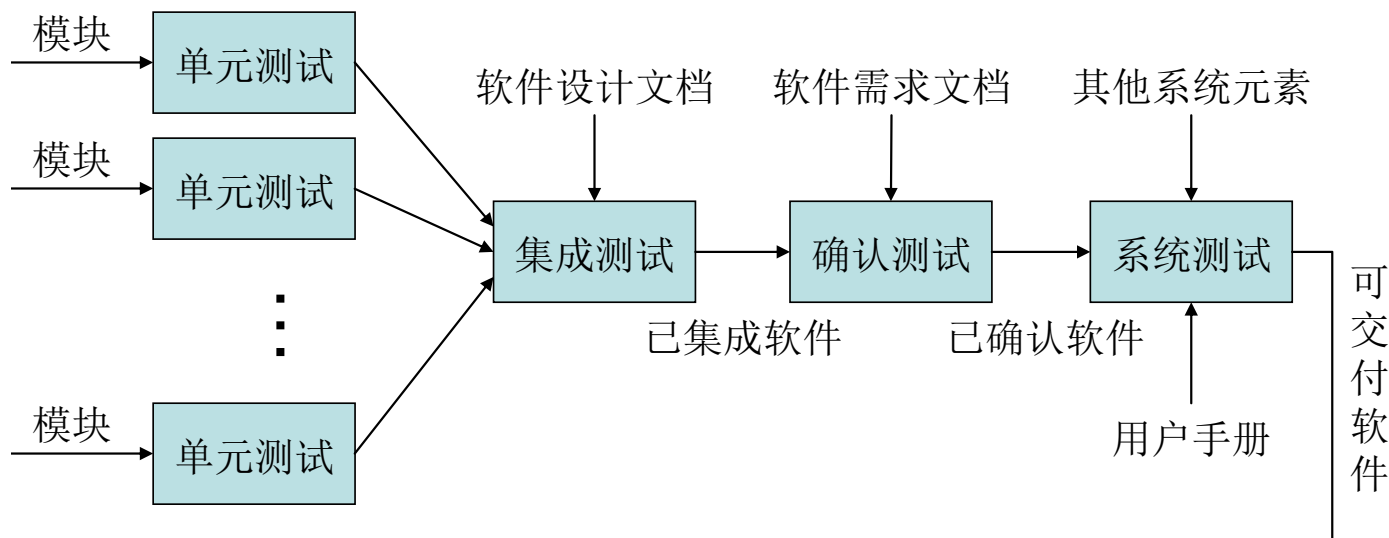
12.2.2 软件测试步骤

12.2.3 软件调试

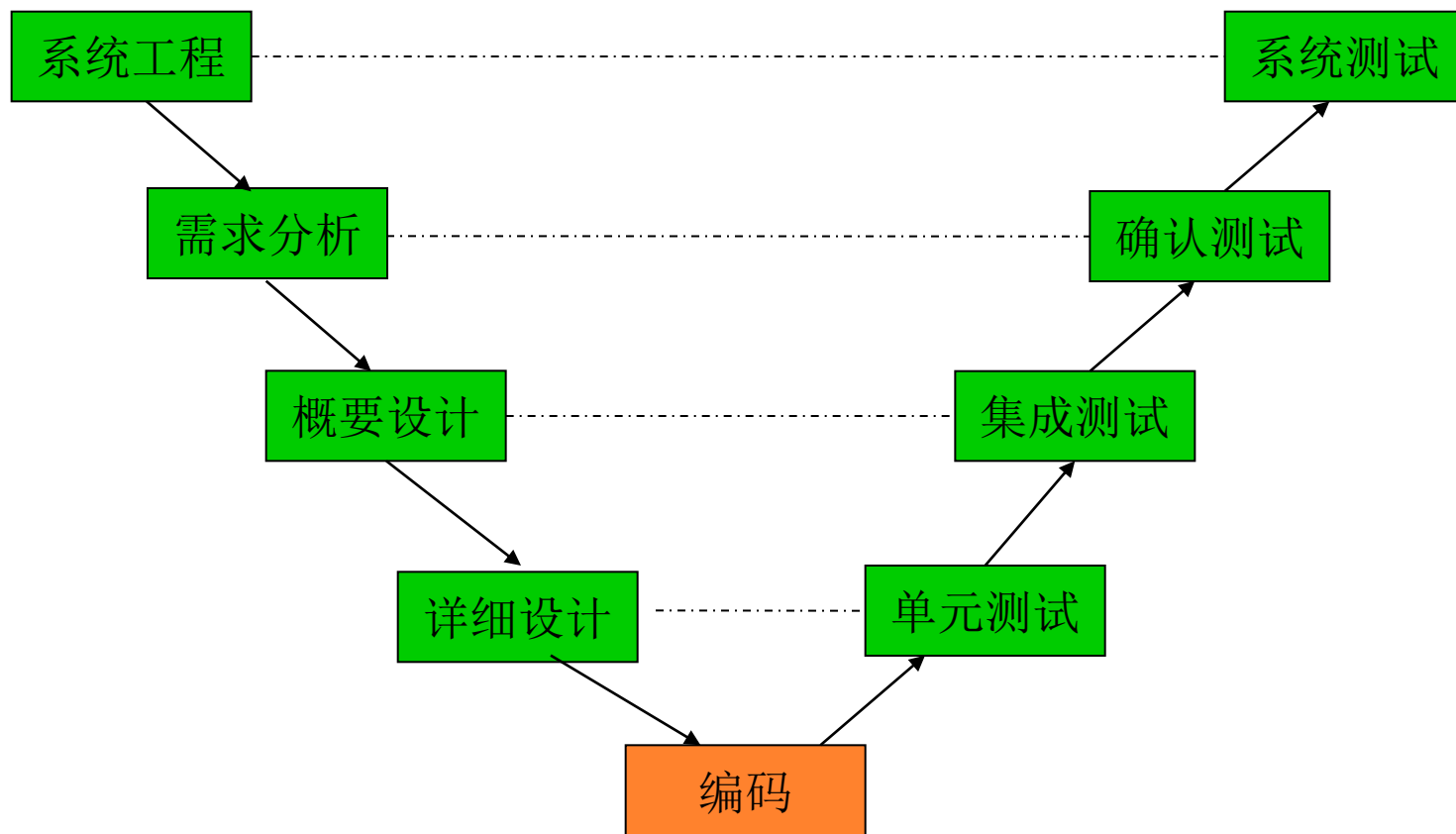
12.3 软件测试战术

软件测试步骤

- 单元测试 Unit Testing
- 集成测试 Integration Testing
- 确认测试 Validation Testing
- 系统测试 System Testing



软件测试的V模型



1 单元测试(Unit Testing)

■ 目的

- 验证开发人员所书写的代码是否可以按照其所设想的方式执行而产生符合预期值的结果，确保产生符合需求的可靠程序单元。

■ 范围

- 单元测试是对软件基本组成单元（构件或模块）进行的测试
- 单元测试侧重于构件中的内部处理逻辑和数据结构（依据详细设计）
- 结构化程序单元是模块，面向对象程序单元是类

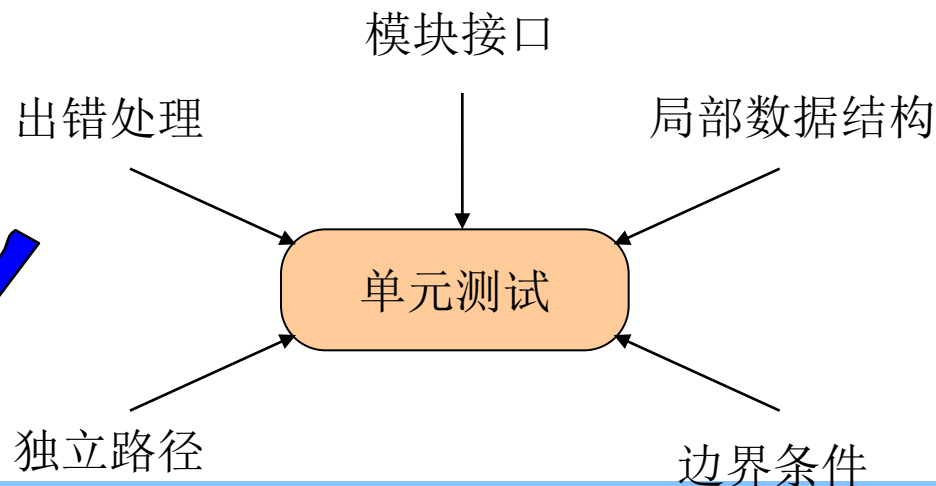
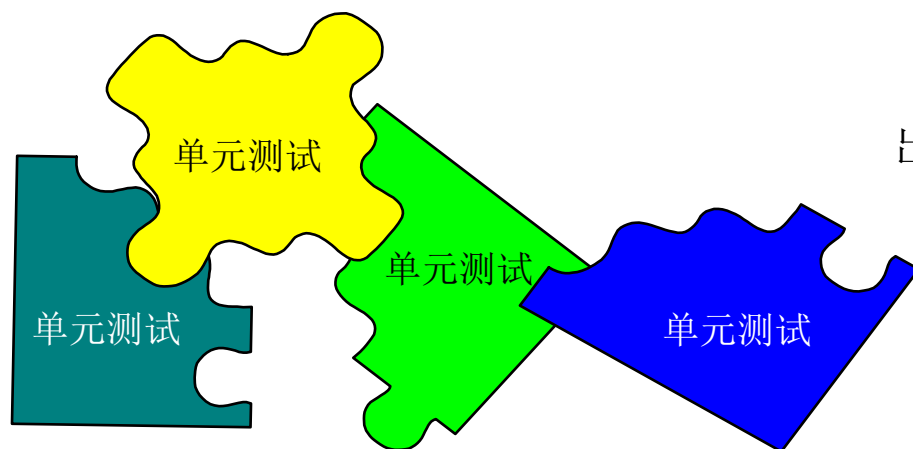
■ 规程

- 单元测试通常被认为是编码阶段的附属工作，单元测试可以再编码开始之前或源代码生成之后完成

单元测试

■ 内容

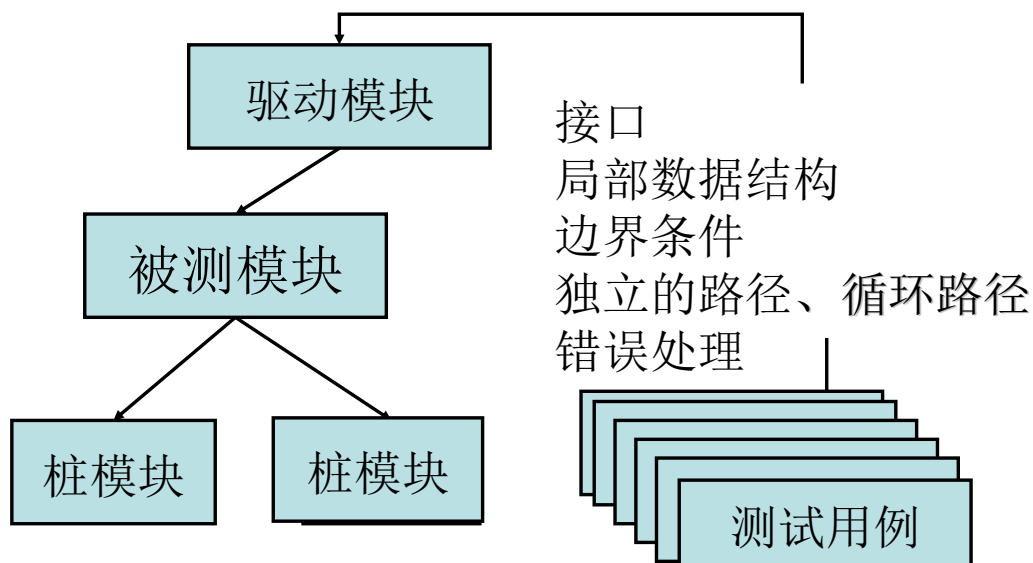
- 接口
- 局部数据结构
- 独立路径
- 边界条件
- 错误处理路径



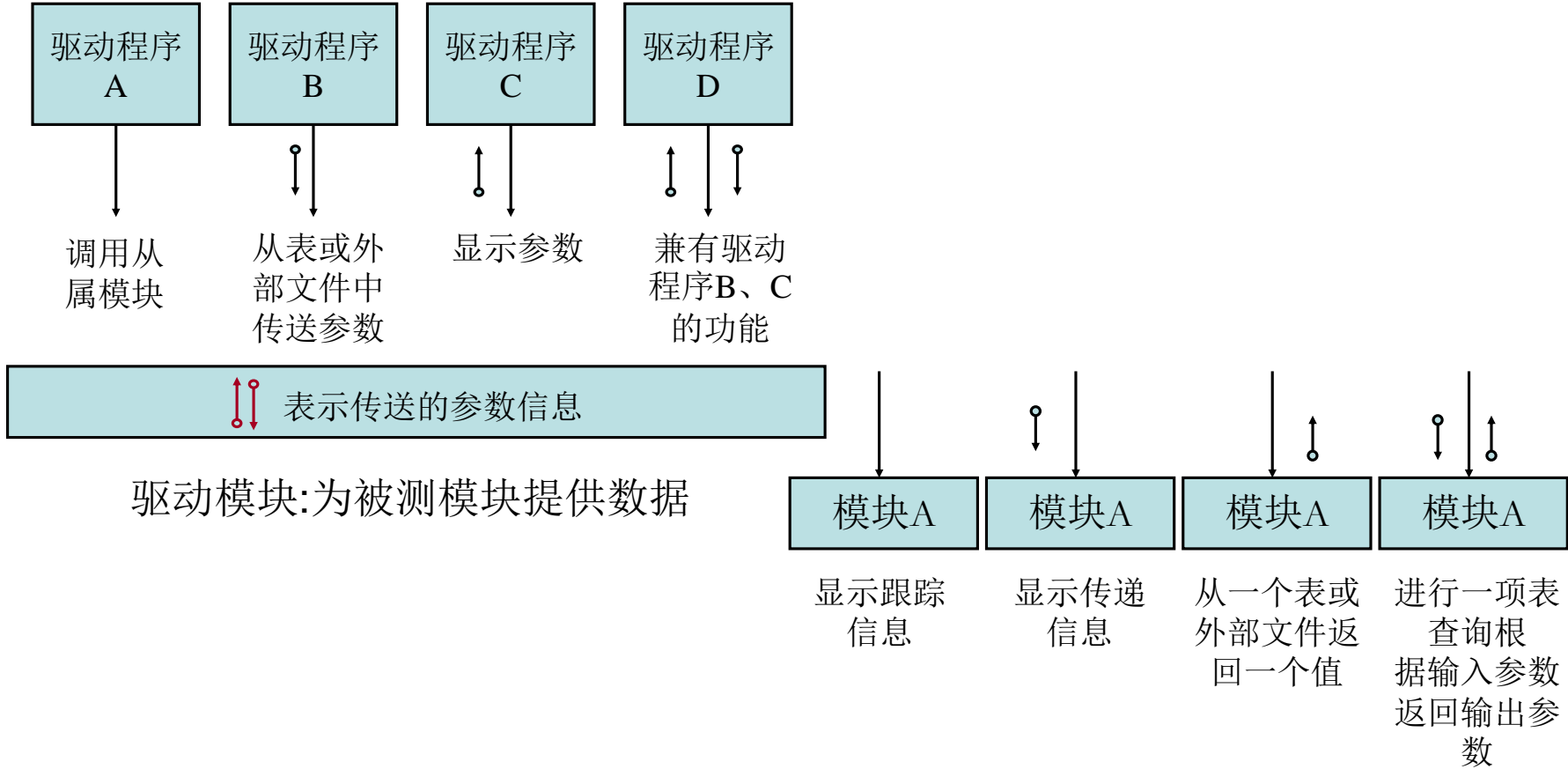
单元测试

■ 单元测试环境

- **驱动模块(driver)**: 模拟被测模块的上一级模块, 接收测试数据, 把这些数据传送给所测模块, 最后再输出实际测试结果;
- **桩模块(stub)**: 模拟被测单元需调用的其他函数接口, 模拟实现子函数的某些功能。

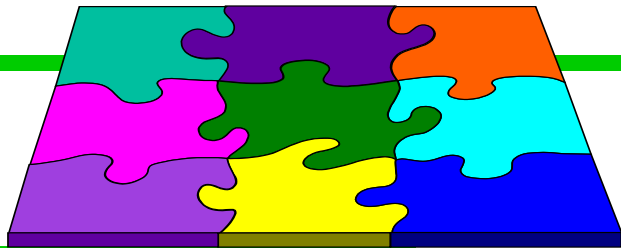


单元测试的驱动模块/桩模块



桩模块: 只做少量的数据操作

2 集成测试



- 每个模块都能单独工作→集成在一起却不能工作

Why?

——模块通过接口相互调用时会引入很多新问题...

- **集成测试(Integration Testing)**

- 在单元测试的基础上，将所有模块按照总体设计的要求组装成为子系统或系统进行的测试。
- 集成测试是构造软件体系结构的系统化技术，同时也是进行一些旨在发现与接口相关的错误的测试。
- 结构化集成测试针对调用关系测试，面向对象针对依赖关系测试

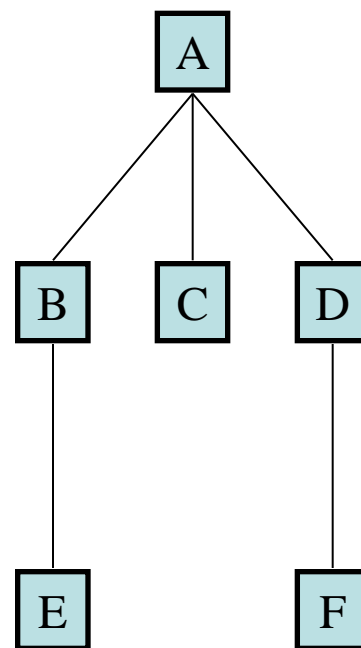
集成测试的方法：整体集成

■ 整体集成方式(非增量式集成)

- 把所有模块按设计要求一次全部组装起来，然后进行整体测试

■ 例如：

- Test A (with stubs for B, C, D)
- Test B (with driver for A and stub for E)
- Test C (with driver for A)
- Test D (with driver for A and stub for F)
- Test E (with driver for B)
- Test F (with driver for D)
- Test (A, B, C, D, E, F)



集成测试的方法：整体集成

■ 优点：

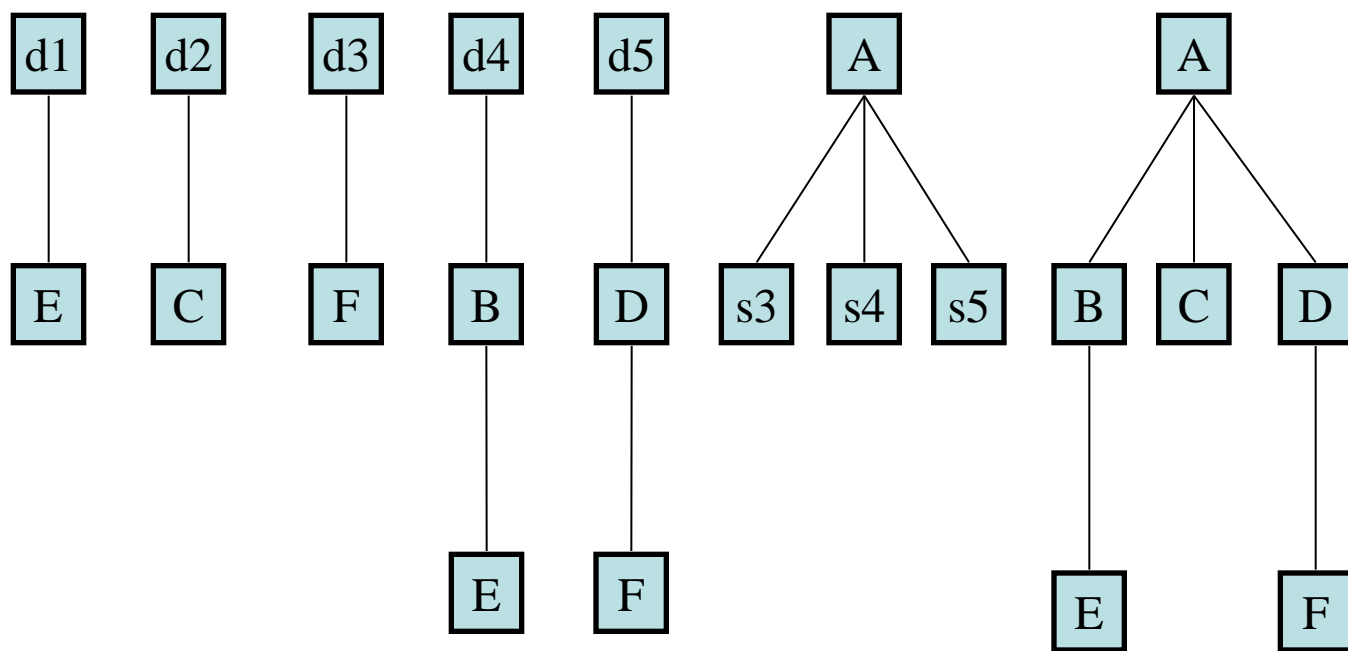
- 效率高，所需人力资源少；
- 测试用例数目少，工作量低；
- 简单，易行；

■ 缺点：

- 可能发现大量的错误，难以进行错误定位和修改；
- 即使测试通过，也会遗漏很多错误；
- 测试和修改过程中，新旧错误混杂，带来调试困难；

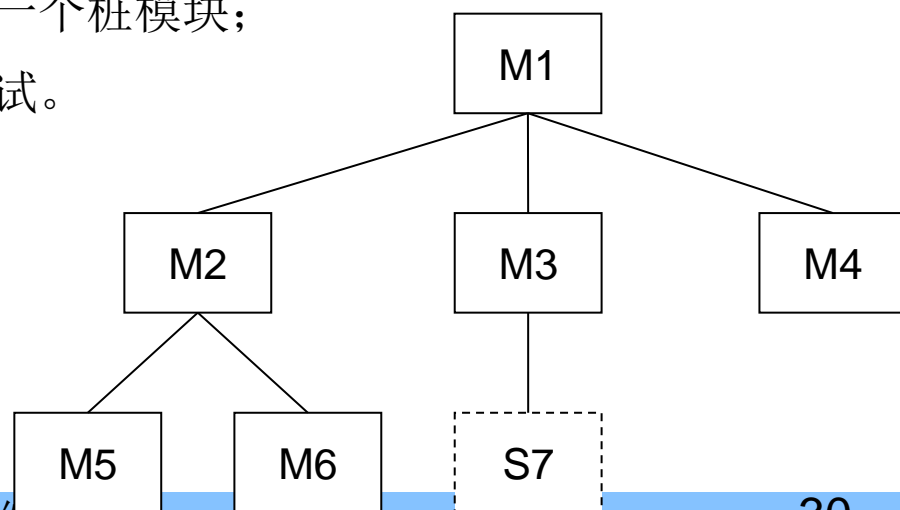
集成测试的方法：增量式集成

- 增量式集成测试方法：
 - 逐步将新模块加入并测试



(1) 自顶向下的增量集成

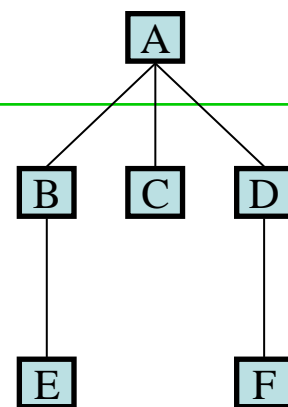
- **自顶向下的集成测试：**从主控模块开始，按软件的控制层次结构，以深度优先或广度优先的策略，逐步把各个模块集成在一起。
- **具体步骤：**
 - 以主控模块作为测试驱动模块，把对主控模块进行单元测试时所引入的所有桩模块用实际模块代替；
 - 依据所选的集成策略(深度优先、广度优先)，每次只替代一个桩模块；
 - 每集成一个模块立即测试一遍；
 - 只有每组测试完成后，才着手替换下一个桩模块；
 - 为避免引入新错误，不断进行回归测试。



(1) 自顶向下的增量集成

■ 自顶向下集成

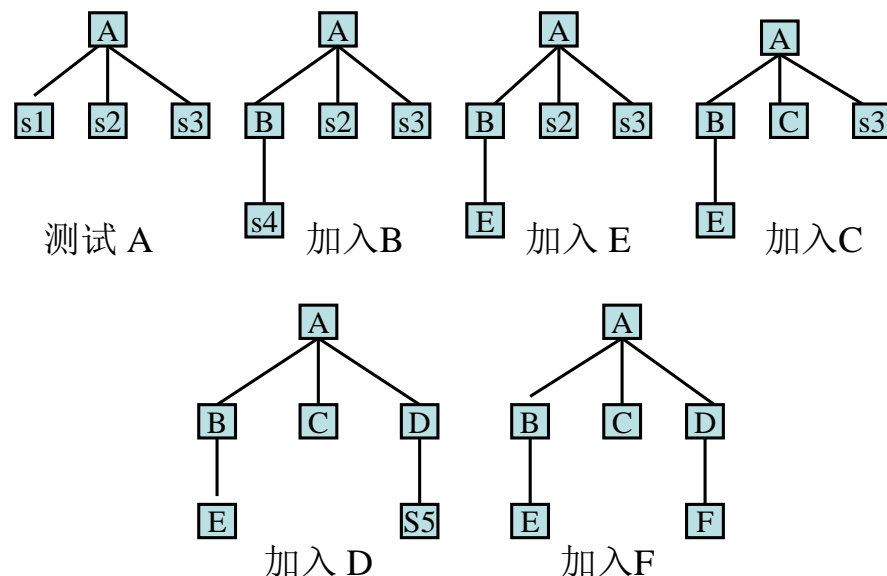
- 深度优先：A、B、E、C、D、F
- 广度优先：A、B、C、D、E、F



■ 广度优先的测试过程:

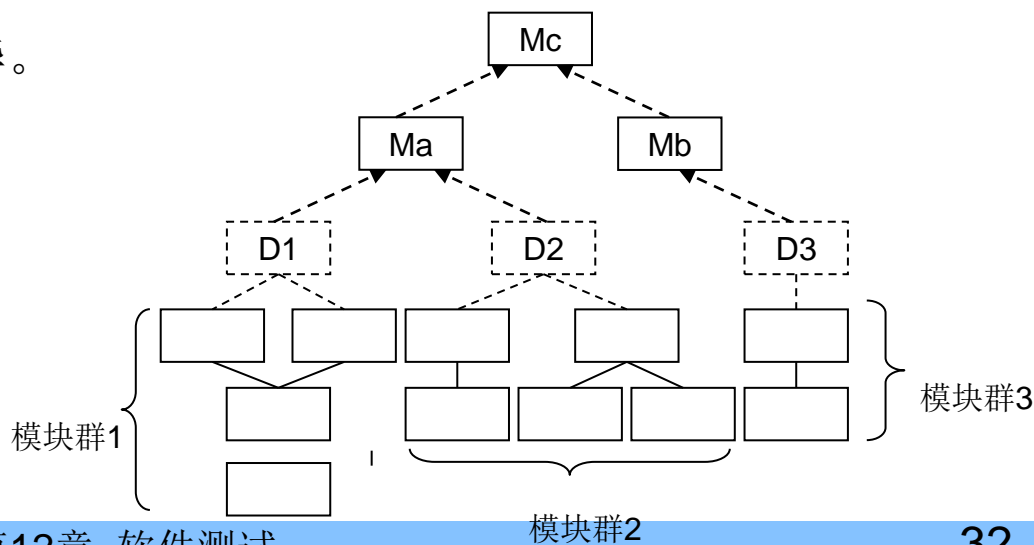
- Test A (with stubs for B,C,D)
- Test A;B (with stubs for E,C,D)
- Test A;B;C (with stubs for E,D)
- Test A;B;C;D (with stubs for E,F)
- Test A;B;C;D;E (with stubs for F)
- Test A;B;C;D;E;F

深度优先的测试过程



(2) 自底向上的增量集成

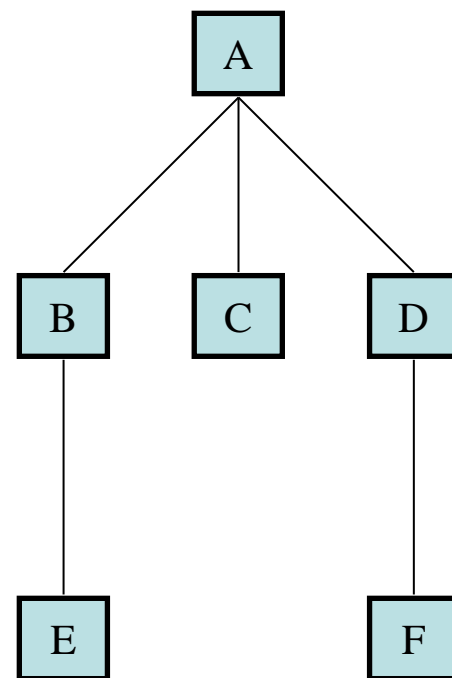
- 自底向上的集成测试：从软件结构最底层的模块开始组装测试。
- 具体步骤：
 - 把底层模块组织成实现某个子功能的模块群(cluster)；
 - 开发一个测试驱动模块，控制测试数据的输入和测试结果的输出；
 - 对每个模块群进行测试；
 - 删除测试使用的驱动模块，用较高层模块把模块去组织成为完成更大功能的新模块；
 - 循环，直到整个程序测试完毕。



(2) 自底向上的增量集成

■ 过程:

- Test E (with driver for B)
- Test C (with driver for A)
- Test F (with driver for D)
- Test B;E (with driver for A)
- Test D;F (with driver for A)
- Test (A;B;C;D;E;F)



两种集成测试的优缺点

■ 自顶向下集成：

- 优点：能尽早地对程序的主要控制和决策机制进行检验，因此较早地发现错误；较少需要驱动模块；
- 缺点：所需的桩模块数量巨大；在测试较高层模块时，低层处理采用桩模块替代，不能反映真实情况，重要数据不能及时回送到上层模块，因此测试并不充分；

■ 自底向上集成：

- 优点：不用桩模块，测试用例的设计亦相对简单；
- 缺点：程序最后一个模块加入时才具有整体形象，难以尽早建立信心。

(3) 三明治集成

- 三明治集成：一种混合增量式集成策略，综合了自顶向下和自底向上两种方法的优点。
- 步骤：
 - 确定以哪一层为界来决定使用三明治集成侧路额；
 - 对该层次及其下面的所有各层使用自底向上的集成策略；
 - 对该层次之上的所有各层使用自顶向下的集成策略；
 - 把该层次各模块同相应的下层集成；
 - 对系统进行整体测试。

(3) 三明治集成

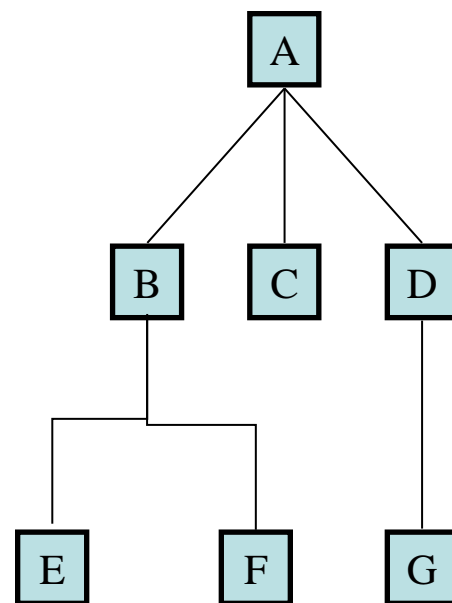
- 选定模块**B**所在的中间层，过程如下：

- Test A (with stubs for B,C,D)
- Test B (with driver for A and stubs for E,F)
- Test C (with driver for A)
- Test D (with driver for A and stub for G)
- Test A;B (with stubs for E,F,C,D)
- Test A;B;C (with stubs for E,F,D)
- Test A;B;C;D (with stubs for E,F,G)

自顶向下

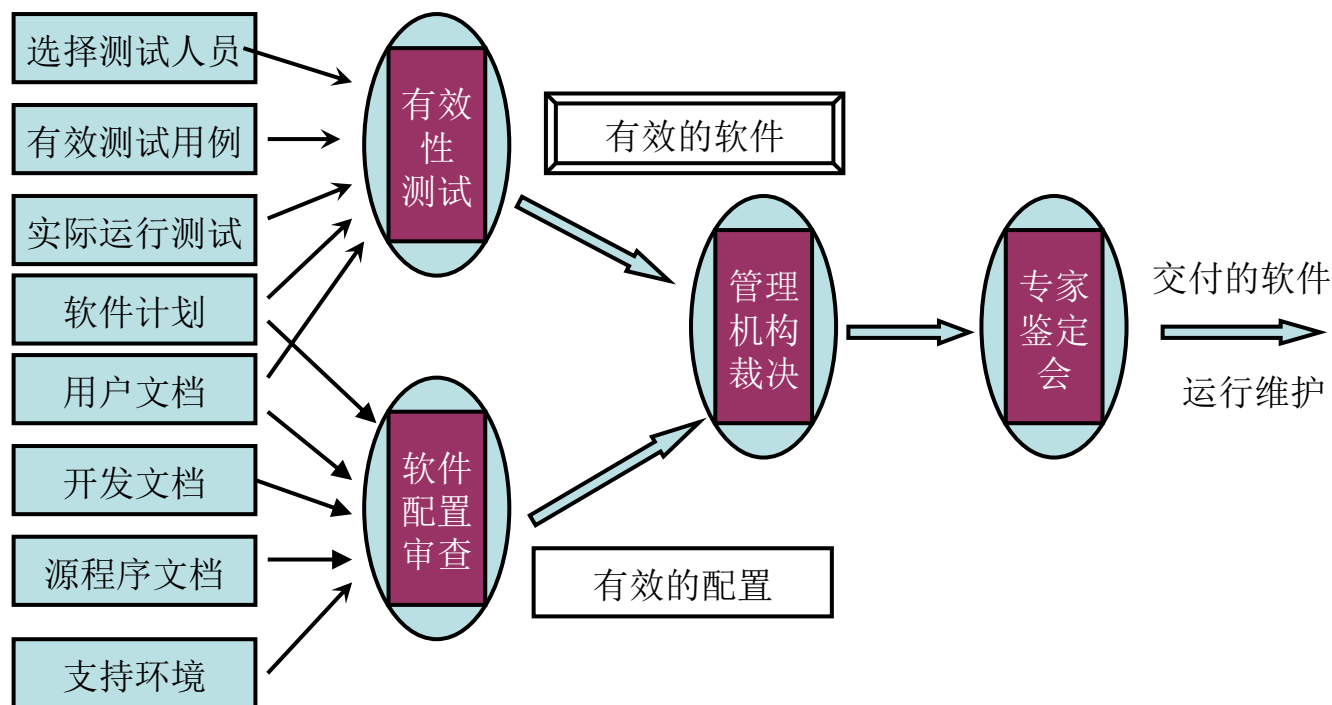
- Test E (with driver for B)
- Test F (with driver for B)
- Test B;E;F (with driver for A)
- Test G (with driver for D)
- Test D;G (with driver for A)
- Test A;B;C;D;E;F;G

自底向上



3 确认测试

- 软件确认是通过一系列表明已符合软件需求的测试而获得的
- **确认测试(Validation Testing):** 检查软件能否按合同要求进行工作，即是否满足软件需求说明书中的确认标准。



3 确认测试

■ 确认测试准则

- 测试计划和测试用例用于确保满足所有的功能需求
- 符合需求或存在偏差，创建缺陷列表，与用户协商

■ 配置评审

- 管理软件的变更

■ α 测试与 β 测试

- α 测试与 β 测试是产品在正式发布前经常进行的两种测试
 - α 测试是由用户在开发环境下进行的测试；
 - 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。

4 系统测试

■ 系统测试(System Testing)

- 软件是基于计算机的大系统的一部分
- 软件要与其他系统部分（硬件、人和信息）相结合，并执行一系列的集成测试和确认测试
- 系统测试是对整个基于计算机的系统进行一系列不同考验的测试（主要测试非功能性需求）

■ 系统测试方法

- 恢复测试——容错性测试
- 安全测试——非法侵入测试
- 压力测试——破坏性测试
- 性能测试——环境运行性能测试

(1) 系统测试：恢复测试

■ 恢复测试(Recovery Testing)

- 恢复测试是检验系统从软件或者硬件失败中恢复的能力，即采用各种人工干预方式使软件出错，而不能正常工作，从而检验系统的恢复能力。
- 恢复性测试的例子
 - 当供电出现问题时的恢复
 - 恢复程序的执行
 - 对选择的文件和数据进行恢复
 - 恢复处理日志方面的能力
 - 通过切换到一个并行系统来进行恢复

(2) 系统测试：安全性测试

■ 安全性测试(Security Testing)

- 安全性测试检查系统对非法侵入的防范能力。
- 安全性测试期间，测试人员假扮非法入侵者，采用各种办法试图突破防线。
- 安全性测试的例子
 - 想方设法截取或破译口令
 - 专门定做软件破坏系统的保护机制
 - 故意导致系统失败，企图趁恢复之机非法进入
 - 试图通过浏览非保密数据，推导所需信息

(3) 系统测试：压力测试

■ 压力测试(Press Testing)

- 压力测试是检查系统在资源超负荷情况下的表现，特别是对系统的处理时间有什么影响。
- 压力测试的例子
 - 对于一个固定输入速率(如每分钟120 个单词)的单词处理响应时间
 - 在一个非常短的时间内引入超负荷的数据容量
 - 成千上万的用户在同一时间从网上登录到系统
 - 引入需要大量内存资源的操作
- 压力测试采用边界值和错误猜测方法，且需要工具的支持。

(4) 系统测试：性能测试

- **性能测试(Performance Testing)**
 - 在实际应用的环境下系统性能的表现
 - 常与压力测试一起进行

主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.2.1 软件测试策略

12.2.2 软件测试步骤

12.2.3 软件调试

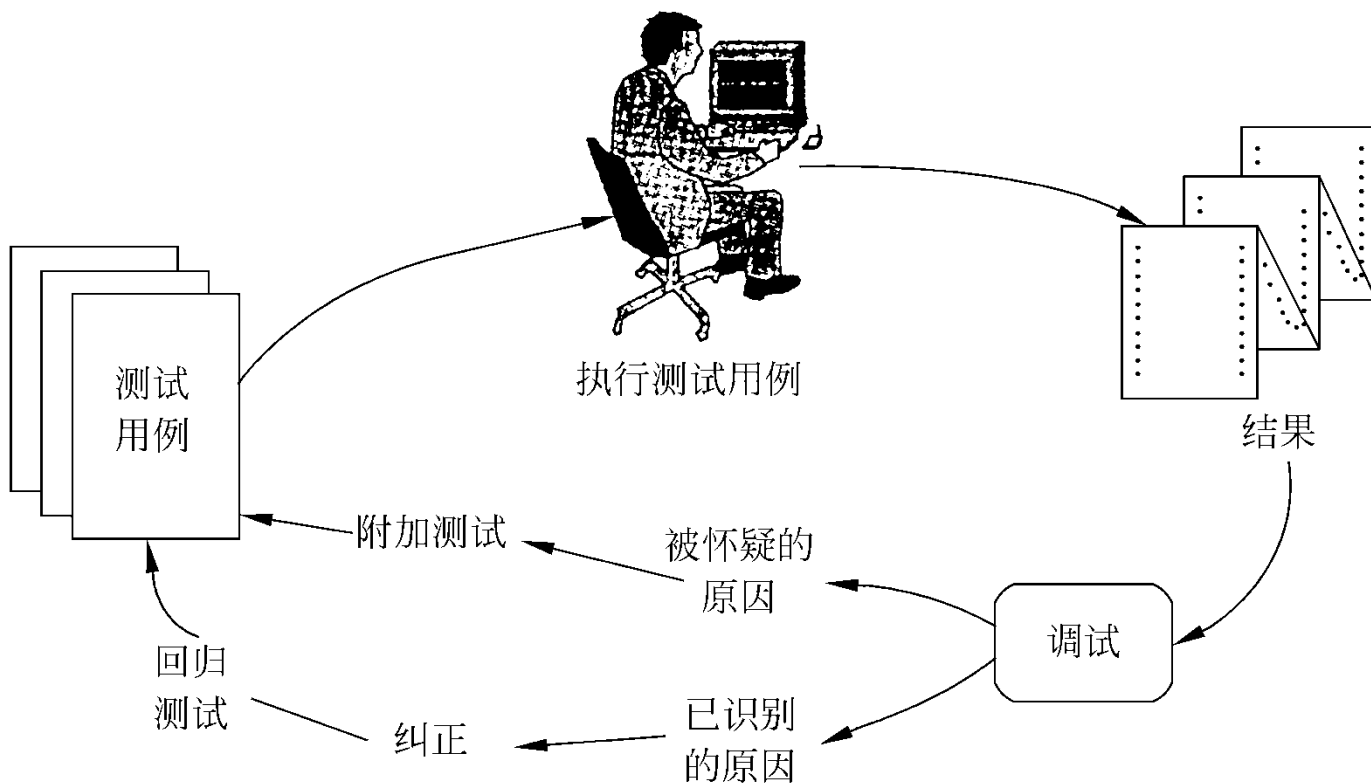
12.3 软件测试战术

调试(Debug)

- **调试(Debug):** 在测试发现错误之后排除错误的过程。
- 调试目前更多的被看作为一项“技巧”，依赖于调试者的编程水平和发现错误的能力。
- 软件工程师在评估测试结果时，往往仅面对着软件错误的症状。但是，软件错误的外部表现和它的内在原因之间可能并没有明显的联系。
- 调试就是把症状和原因联系起来的智力过程。

调试过程

- 调试≠测试
- 二者的关系为：



调试途径

- 调试的目标：寻找缺陷和修正缺陷。
- 寻找缺陷的方法：
 1. 将错误状态稳定下来
 2. 确定错误的来源
 - a. 收集产生缺陷的相关数据
 - b. 分析所收集的数据，并构造对缺陷的假设
 - c. 确定怎样去证实或证伪这个假设，可以对程序进行测试或是通过检查代码
 - d. 按照上一步确定的方法对假设做出最终结论
 3. 修补缺陷
 4. 对所修补的地方进行测试
 5. 查找是否还有类似的错误

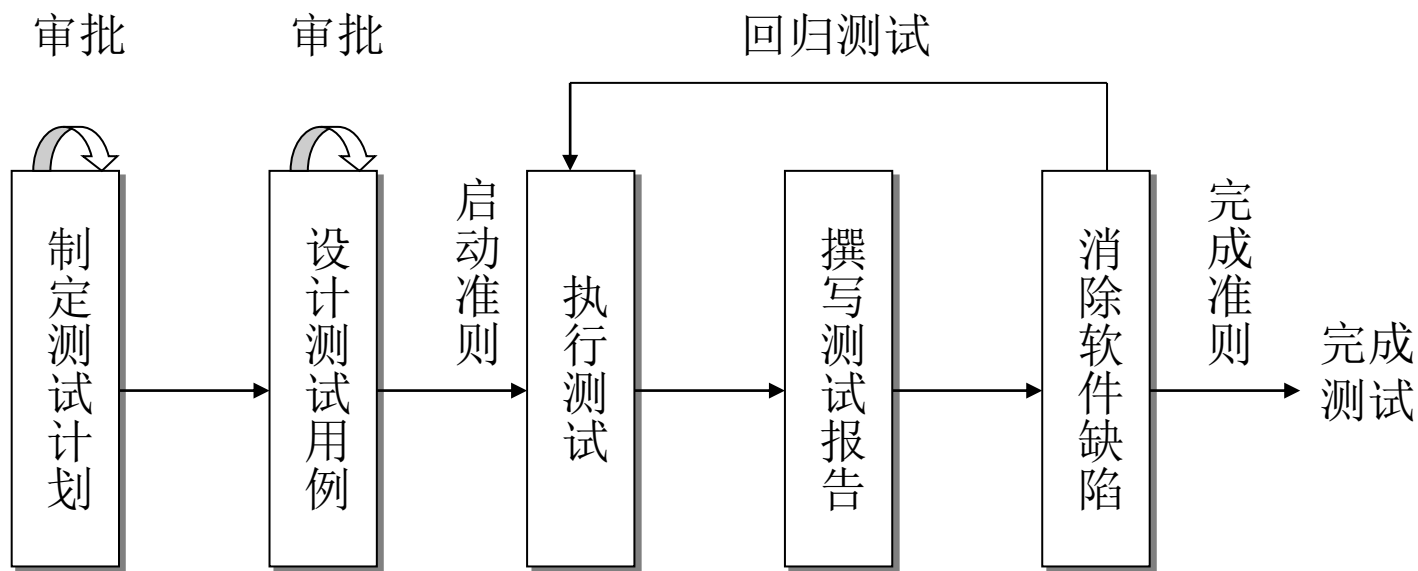
调试途径

■ 修正缺陷

- 在动手之前先要理解问题
- 理解程序本身，而不仅仅是问题
- 验证对错误的分析
- 放松一下
- 保存最初的源代码
- 治本，而不是治标
- 修改代码时一定要有恰当的理由
- 一次只做一个改动
- 检查自己的改动
- 增加能暴露问题的单元测试
- 搜索类似的缺陷

回归测试

- 在软件生命周期中的任何一个阶段，只要软件发生了改变，就可能给该软件带来问题。
- 为了验证修改的正确性及其影响就需要进行回归测试。



主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.3 软件测试战术

12.3.1 软件测试技术

12.3.2 黑盒测试

12.3.3 白盒测试

典型的软件测试技术

- 测试的技术就是设计一组测试用例
 - 执行每个软件构件的内部逻辑和接口
 - 测试程序的输入和输出域以发现程序功能、行为和性能方面的错误
- 利用“白盒”测试用例设计技术执行程序内部逻辑
- 利用“黑盒”测试用例设计技术确认软件需求

测试用例的定义与特征

■ 测试用例(testing case):

- 测试用例是为特定的目的而设计的一组测试输入、执行条件和预期的结果。
- 测试用例是执行的最小测试实体。
- 测试用例就是设计一个场景，使软件程序在这种场景下，必须能够正常运行并且达到程序所设计的执行结果。

■ 测试用例的特征:

- 最有可能抓住错误的;
- 不是重复的、多余的;
- 一组相似测试用例中最有效的;
- 既不是太简单，也不是太复杂。

测试用例的设计原则

■ 测试用例的代表性：

- 能够代表并覆盖各种合理的和不合理的、合法的和非法的、边界的和越界的以及极限的输入数据、操作和环境设置等

■ 测试结果的可判定性：

- 测试执行结果的正确性是可判定的，每一个测试用例都应有相应的期望结果。

■ 测试结果的可再现性：

- 对同样的测试用例，系统的执行结果应当是相同的。

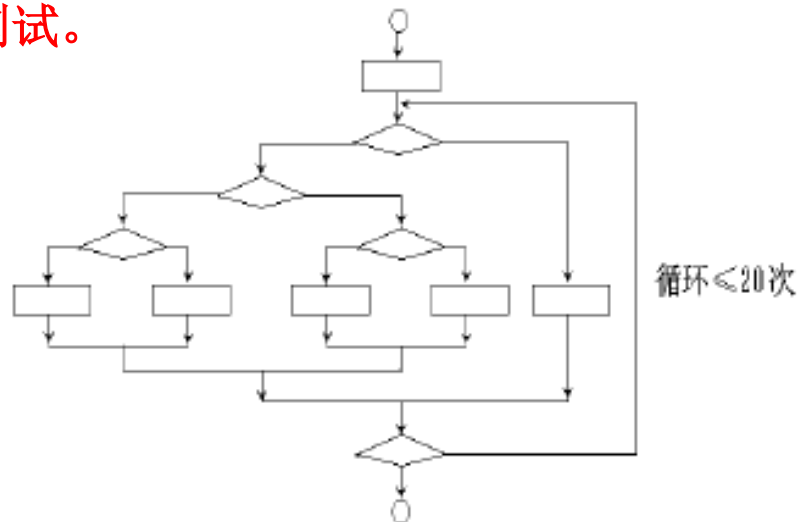
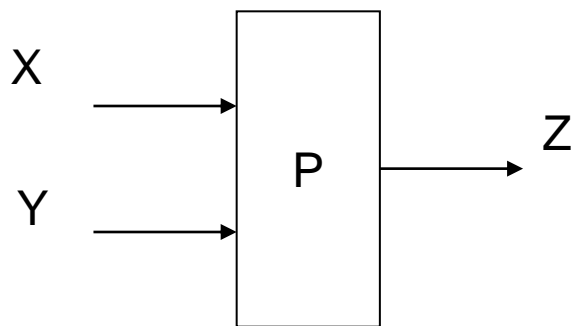
典型的软件测试技术

■ 黑盒测试：又称“功能测试”或“数据驱动测试”

- 它将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。**主要用于集成测试、确认测试和系统测试。**

■ 白盒测试：又称“结构测试”或“逻辑驱动测试”

- 它把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。**主要用于单元测试和集成测试。**



主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.3 软件测试战术

12.3.1 软件测试技术

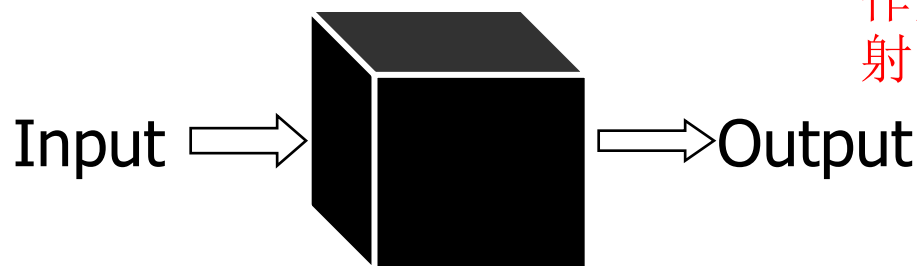
12.3.2 黑盒测试

12.3.3 白盒测试

1 黑盒测试概念

■ 黑盒测试(black-box testing):

- 又称“功能测试”、“数据驱动测试”或“基于规格说明书的测试”，是一种从用户观点出发的测试。
- 将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。
- 通常在软件接口处进行。



原理：任何程序都可以看作是将输入定义域取值映射到输出值域的函数

2 黑盒测试能发现的错误

- 是否有不正确或遗漏的功能
- 接口错误
- 数据结构或外部数据库访问信息？
- 行为或性能错误
- 初始化或终止错误

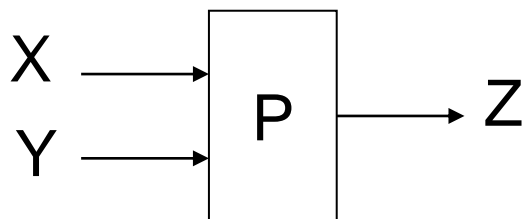
3 黑盒测试中的“穷举”

- 用黑盒测试发现程序中的错误，必须在所有可能的输入条件和输出条件中确定测试数据，来检查程序是否都能产生正确的输出，但这是不可能的。

——因为穷举测试数量太大，无法完成。

黑盒测试中的“穷举”

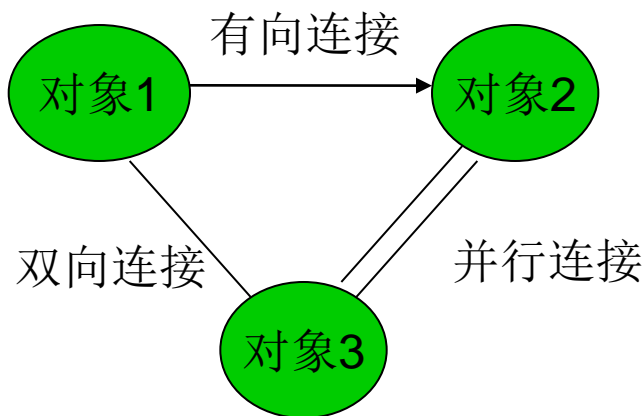
- 举例：程序P有输入整数X和Y及输出量Z，在字长为32位的计算机上运行。
 - 可能采用的测试数据组： $2^{32} \times 2^{32} = 264$
 - 如果测试一组数据需要1毫秒，一年工作 365×24 小时，完成所有测试需5亿年。



- 因此，测试人员只能在大量可能的数据中，选取其中一部分作为测试用例。

4 基于图的测试方法

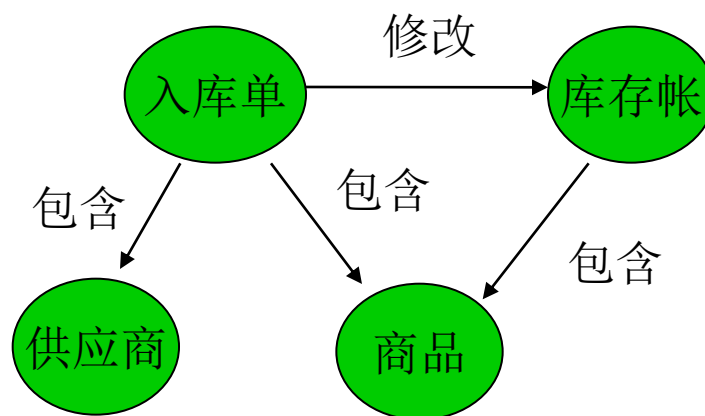
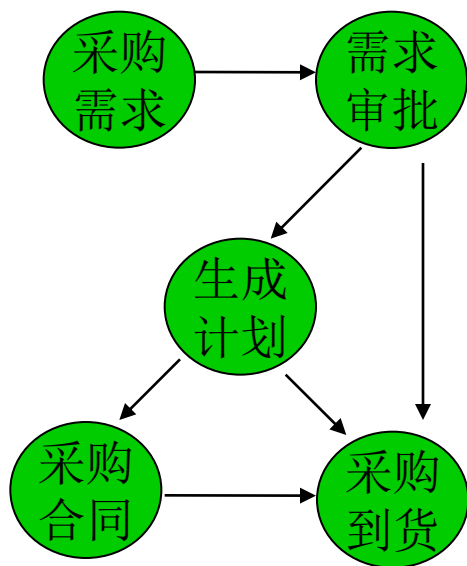
- 黑盒测试要首先理解软件中建模的对象及这些对象的关系
- 创建重要对象及其关系图
- 设计测试用例以检查对象关系并发现错误



对象可以是构件、数据或操作

4 基于图的测试方法

- 有限状态建模
- 事物流建模
- 数据流建模
- 时间建模

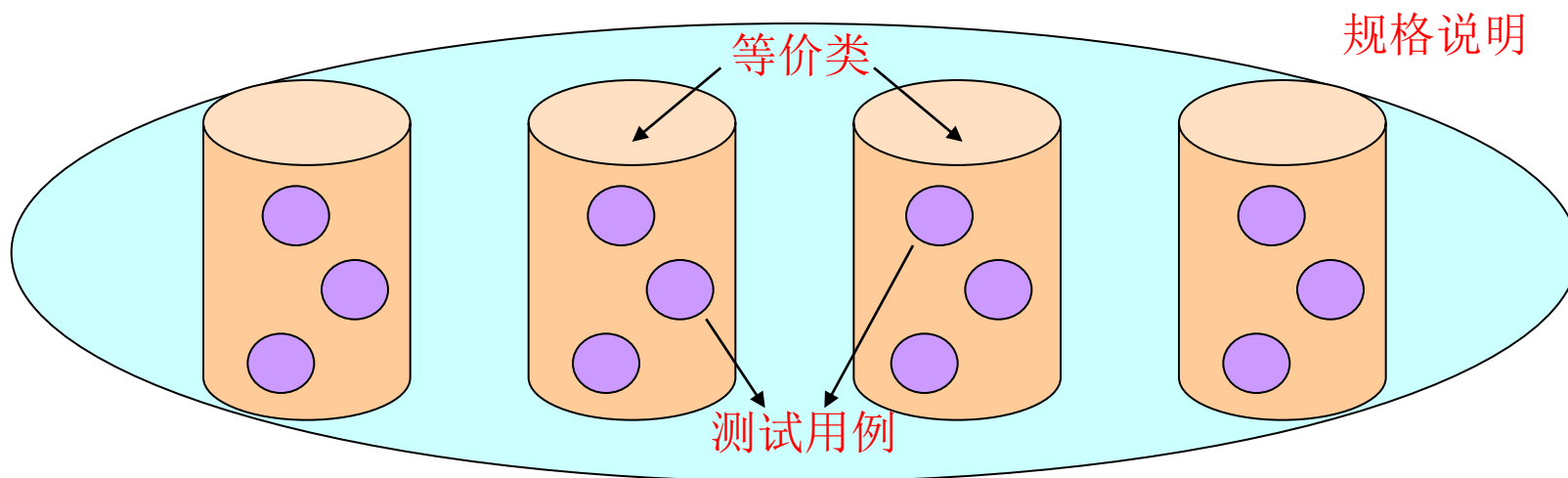


5 测试用例的设计技术

- 等价类划分
- 边界值分析
- 正交数组测试
- 错误推测法
- 随机测试

(1) 等价类划分

- **等价类**：将程序的输入划分为若干个数据类，从中生成测试用例。并合理地假定“测试某等价类的代表值就等于对这一类其它值的测试”。



- 在每一个等价类中选取少量有代表性的数据作为测试的输入条件，就可以用少量代表性的测试数据，并取得较好的测试结果。

等价类划分(Equivalence partitioning)

- 关键步骤：确定等价类和选择测试用例
- 基本原则：
 - 每个可能的输入属于某一个等价类
 - 任何输入都不会属于多个等价类
 - 用等价类的某个成员作为输入时，如果证明执行存在误差，那么用该类的任何其他成员作为输入，也能检查到同样的误差。

有效/无效等价类

■ 有效等价类

- 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。
- 利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。

■ 无效等价类

- 对程序的规格说明是不合理的或无意义的输入数据所构成的集合。
- 无效等价类至少应有一个，也可能有多个。

■ 设计测试用例时，要同时考虑这两种等价类。

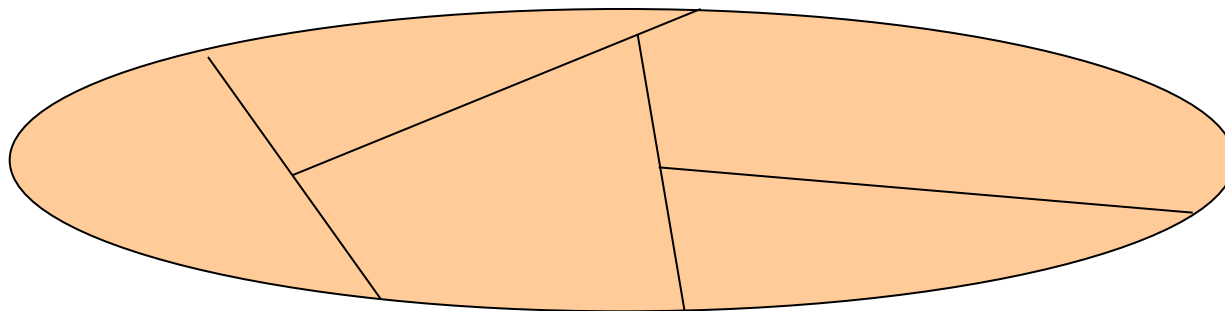
- 软件不仅要能接收合理的数据，也要能经受意外的考验，这样的测试才能确保软件具有更高的可靠性。

划分等价类的标准

- 划分等价类的标准：“完备测试、避免冗余”
- 将输入数据的集合(P)划分为一组子集(E_1, E_2, \dots, E_n), 并尽可能满足:

$$E_1 \cup E_2 \cup \dots \cup E_n = P$$

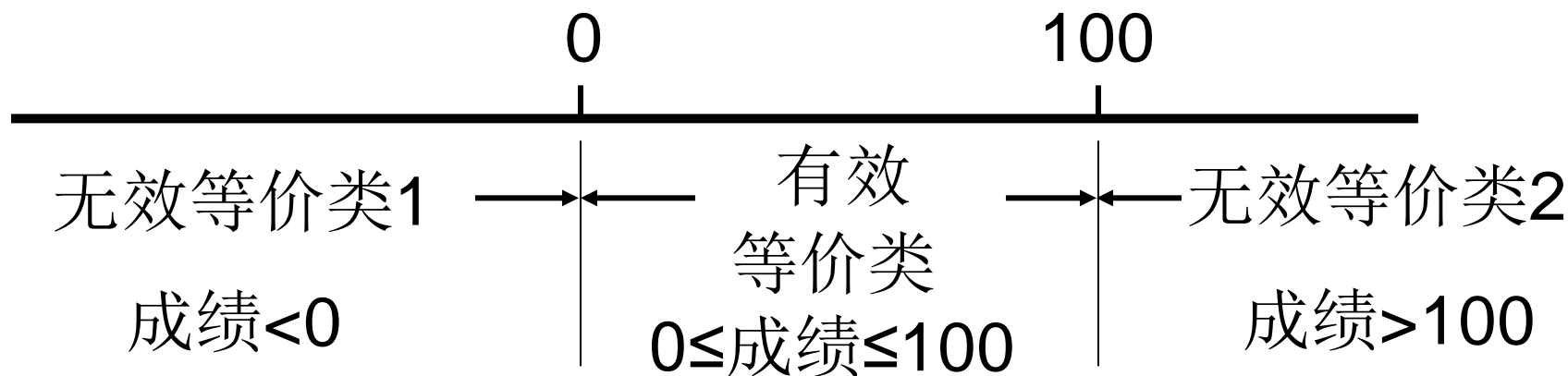
$$E_i \cap E_j = \emptyset$$



确定等价类的四大原则

- 原则1：在输入条件指定一个范围，则可以定义一个有效和两个无效等价类。

- 例如：输入值是学生成绩，范围是0~100



确定等价类的四大原则

- 原则2：若输入条件需要特定的值，则可以定义一个有效和两个无效的等价类。
- 原则3：若输入条件指定集合的某个元素，则可以定义一个有效和一个无效等价类。
- 原则4：若输入条件为布尔值，则可以定义一个有效和一个无效的等价类。

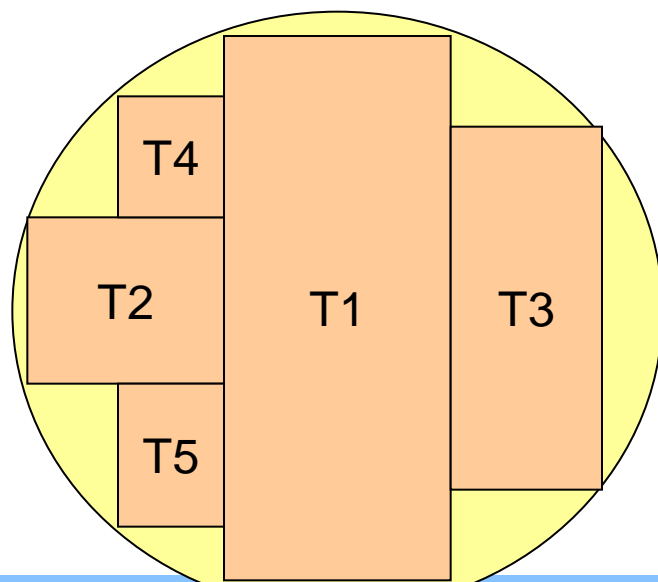
设计测试用例

- 测试用例 = {测试数据+期望结果}
- 测试结果 = {测试数据+期望结果+实际结果}
- 在确立了等价类后，可建立等价类表，列出所有划分出的等价类输入数据+期望结果：

输入条件	有效等价类	无效等价类
...
...

设计测试用例

- 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
- 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



例1：数据录入问题

- **[例1]某一程序要求输入数据满足以下条件：**
 - 可输入1个或多个数据，每个数据由1-8个字母或数字构成，且第一个字符必须为字母；
 - 如果满足上述条件，则判断“合法数据”；否则，判断“非法数据”；
- **用等价类划分方法为该程序进行测试用例设计。**

例1：输入变量问题

■ 划分等价类：

输入条件	有效等价类	号码	无效等价类	号码
标识符个数	1个	(1)	0个	(6)
	多个	(2)		
标识符字符数	1~8个	(3)	0个	(7)
			>8个	(8)
标识符组成	数字与字母	(4)	含有非“字母或数字”	(9)
第一个标识符	字母	(5)	非字母	(10)

例1：输入变量问题

- 设计测试用例：

测试用例编号	测试用例内容	覆盖的等价类
1	a2ku83t	(1)(3)(4)(5)
2	a2ku83t, fta, b2	(2)(3)(4)(5)
3		(6)
4	a2ku83t, , b2	(7)
5	a2ku83t29, fta	(8)
6	a2ku8\$t	(9)
7	92ku83t	(10)

例2：日期检查

- **[例2]** 档案管理系统，要求用户输入以年月表示的日期。假设日期限定在**1990年1月~2049年12月**，并规定日期由**6位数字字符**组成，前**4位**表示年，后**2位**表示月。
- 用等价类划分法设计测试用例，来测试程序的“日期检查功能”。

例2：日期检查

■ 划分等价类：

输入等价类	有效等价类	无效等价类
日期的类型及长度	①6位数字字符	②有非数字字符 ③少于6位数字字符 ④多于6位数字字符
年份范围	⑤在1990~2049之间	⑥小于1990 ⑦大于2049
月份范围	⑧在01~12之间	⑨等于00 ⑩大于12

例2：日期检查

- 设计有效等价类的测试用例：

测试用例编号	测试用例内容	覆盖的等价类
1	200711	(1)(5)(8)

- 设计无效等价类的测试用例：

测试用例编号	测试用例内容	覆盖的等价类
1	07June	(2)
2	20076	(3)
3	2007011	(4)
4	198912	(6)
5	205401	(7)
6	200700	(8)
7	200713	(10)

(2) 边界值分析

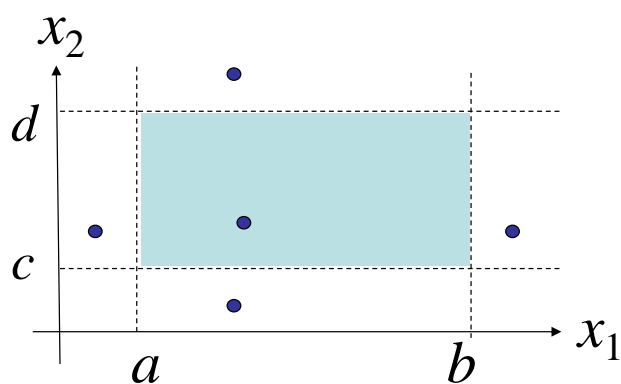
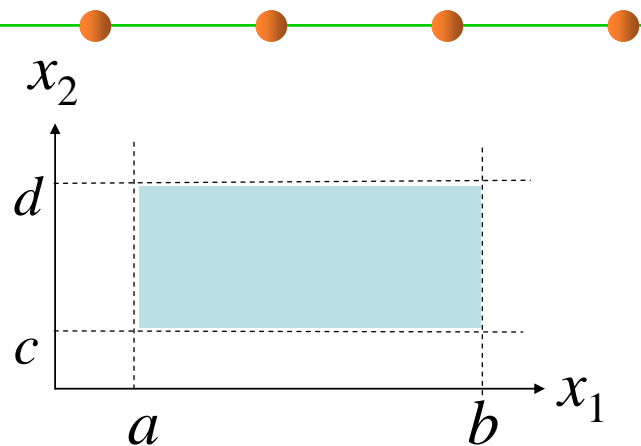
- 边界值分析是等价类测试的补充，主要是考虑等价类的边界条件，在等价类的“边缘”选择元素。
- 长期的测试经验表明：大量的错误是发生在输入或输出范围的边界上，而不是发生在输入输出范围的内部。
- 因此针对各种边界情况设计测试用例，可以查出更多的错误。

确定边界

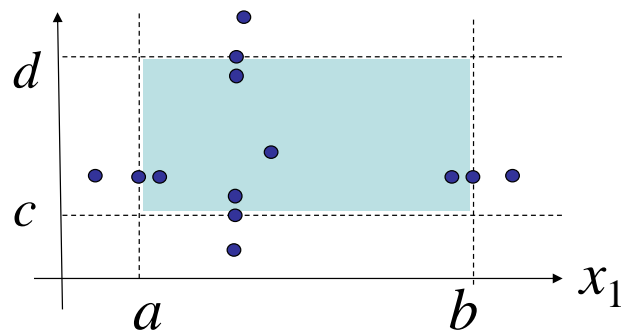
- 使用边界值分析方法设计测试用例，首先应确定边界情况：
 - 通常输入和输出等价类的边界，就是应着重测试的边界情况。
 - 选取正好等于、刚刚大于、刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值作为测试数据。
- 例：在 $[R_1, R_2]$ 的取值区间中，应如何选择？



根据边界确定测试用例



等价类划分
5个测试用例：
小于最小值（2）；
正常值（1）；
大于最大值（2）



在5个测试用例的基础上增如：
边界值（4）
略低于最大值（2）；
略高于最小值（2）；

常见的边界值

- 通常情况下，软件测试所包含的边界检验有几种类型：数字、字符、位置、重量、大小、速度、方位、尺寸、空间等。
- 相应地，以上类型的边界值应该在：最大/最小、首位/末位、上/下、最快/最慢、最高/最低、最短/最长、空/满等情况下。
 - 对16-bit 的整数而言，32767和-32768是边界
 - 屏幕上光标在最左上、最右下位置
 - 报表的第一行和最后一行
 - 数组元素的第一个和最后一个
 - 循环的第0次、第1次和倒数第2次、最后1次

边界值分析的原则

- **原则1：若输入条件指定为以a和b为边界的范围，则测试用例应该包括a、b，略大于a和略小于b**
- **例如：如果程序的规格说明中规定“重量在10公斤至50公斤范围内的邮件，其邮费计算公式为……”。**
 - 作为测试用例，应取10、50、10.01、49.99

边界值分析的原则

- **原则2：**若输入条件指定为一组值，则测试用例应当执行其中的最大值和最小值，以及略大于最小值和略小于最大值的值
- 比如，一个输入文件应包括**1~255**个记录，则测试用例可取**1**和**255**，还应取**2**及**254**等。

边界值分析的原则

- **原则3：原则1和2也适用于输出条件**
- 例如，某程序的规格说明要求计算出“每月保险金扣除额为0至1165.25元”，创建测试用例使输出为0.00、1165.25和0.01、1165.24等。
- **原则4：若内部程序数据结构有预定义的边界值（如：数组有100项），要在其边界处测试数据结构**

主要内容

12.1 什么是软件测试

12.2 软件测试策略

12.3 软件测试战术

12.3.1 软件测试技术

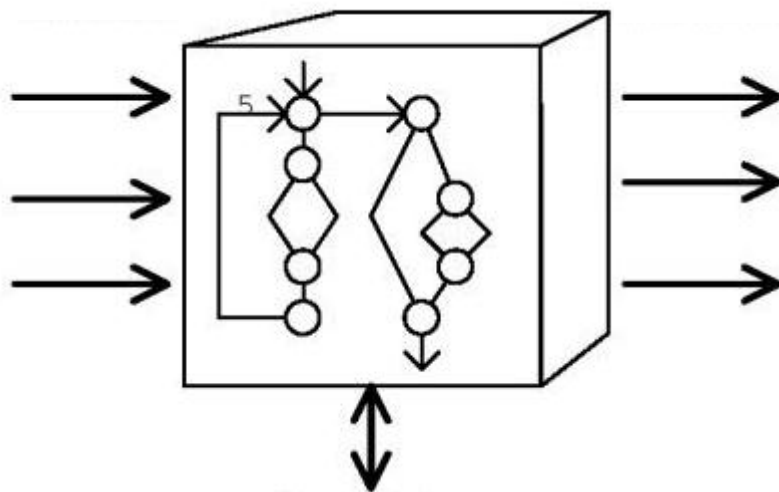
12.3.2 黑盒测试

12.3.3 白盒测试

1 白盒测试的概念

■ 白盒测试(又称为“结构测试”或“逻辑驱动测试”)

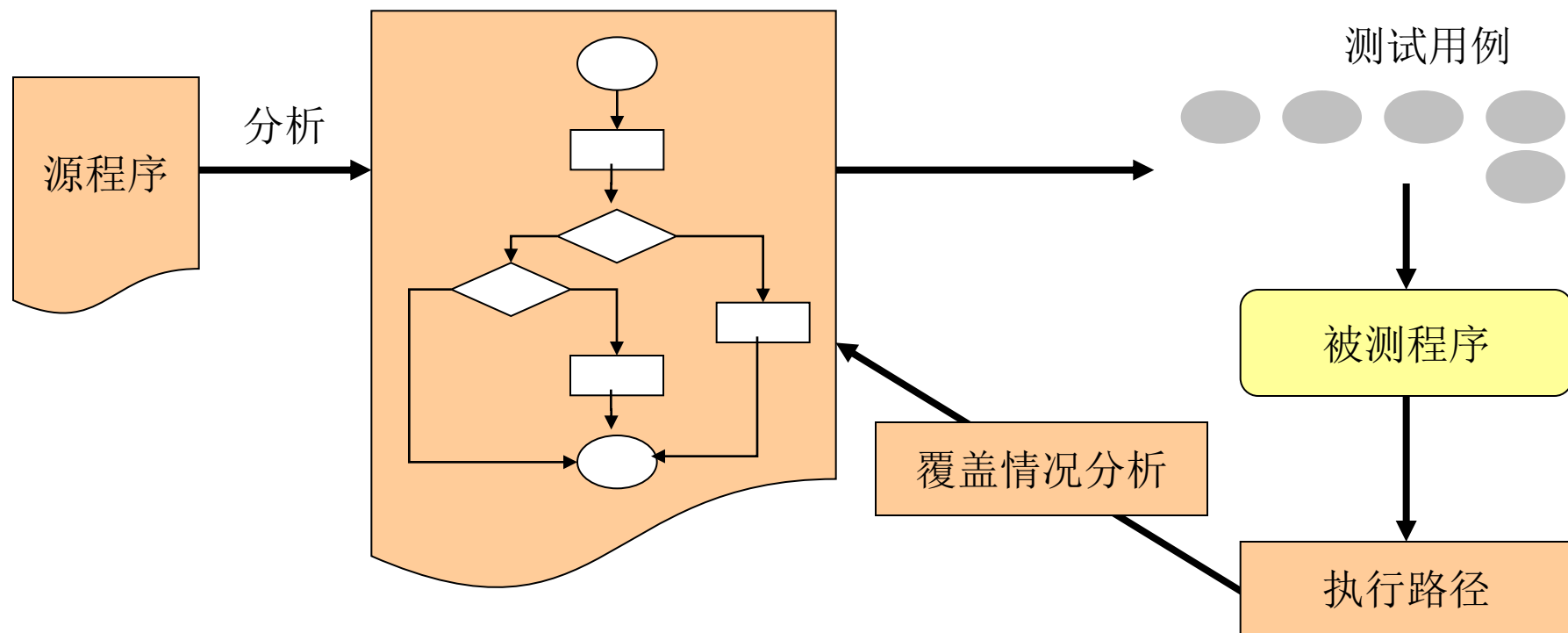
- 把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。



2 白盒测试的目的

- 白盒测试主要对程序模块进行如下的检查：
 - 对模块的每一个独立的执行路径至少测试一次；
 - 对所有的逻辑判定的每一个分支(真与假)都至少测试一次；
 - 在循环的边界和运行界限内执行循环体；
 - 测试内部数据结构的有效性；
- 错误隐藏在角落里，聚集在边界处

白盒测试的目的



白盒测试用例中的输入数据从程序结构导出，
但期望输出务必从需求规格中导出。

3 测试覆盖标准

- **白盒测试的特点：**

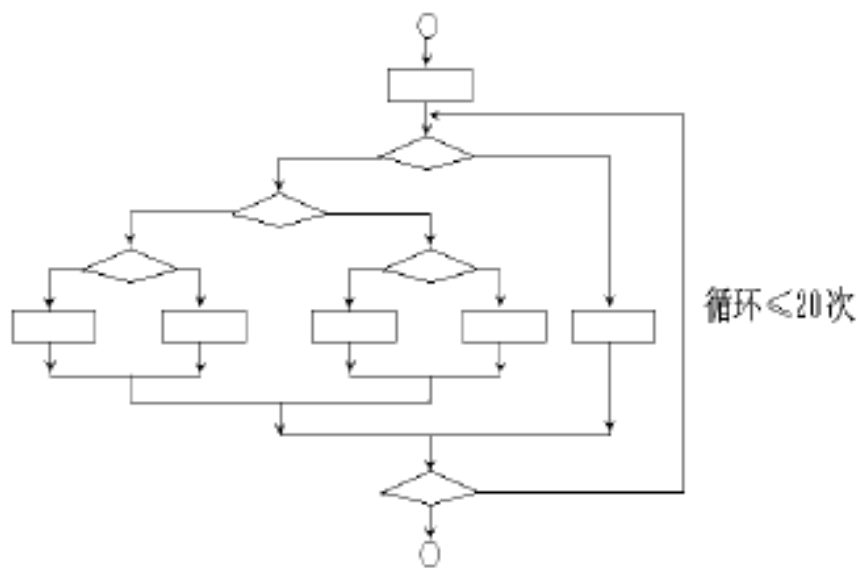
- 以程序的内部逻辑为基础设计测试用例，又称逻辑覆盖法。
- 应用白盒法时，手头必须有程序的规格说明以及程序清单。

- **白盒测试考虑测试用例对程序内部逻辑的覆盖程度：**

- 最彻底的白盒法是覆盖程序中的每一条路径，但是由于程序中一般含有循环，所以路径的数目极大，要执行每一条路径是不可能的，只能希望覆盖的程度尽可能高些。

测试覆盖标准

- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。
- 举例：某个小程序的流程图，包括了一个执行**20次**的循环。
 - 包含的不同执行路径数达**520**条，对每一条路径进行测试需要1 毫秒，假定一年工作 365×24 小时，要把所有路径测试完，需**3170**年。



测试覆盖标准

- 为了衡量测试的覆盖程度，需要建立一些标准，目前常用的一些覆盖标准从低到高分别是：
 - **逻辑覆盖：**
 - 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
 - **控制结构覆盖：**
 - 基本路径测试
 - 循环测试
 - 条件测试
 - 数据流测试
- 请同学自己阅读教材，理解上述五种逻辑覆盖

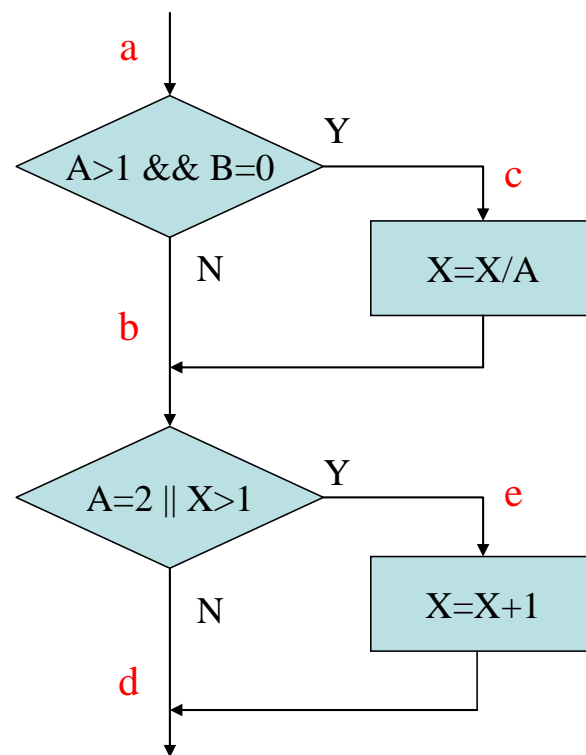
4 白盒测试技术

- 基本路径测试
- 控制结构测试

(1) 路径测试

- **路径测试：**设计足够多的测试用例，覆盖被测试对象中的所有可能路径。
- 对于左例，下面的测试用例则可对程序进行全部的路径覆盖。

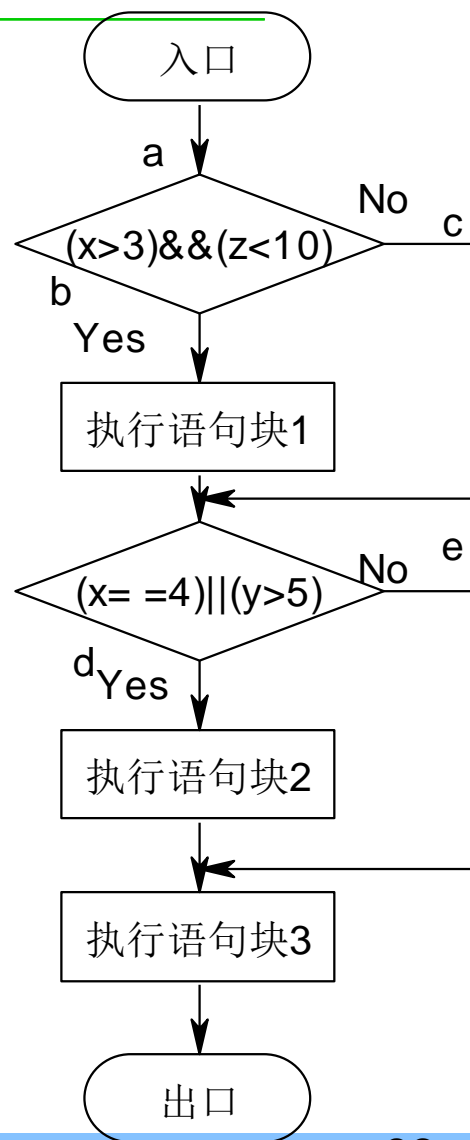
测试用例	通过路径
A=2、B=0、X=3	ace
A=1、B=0、X=1	abd
A=2、B=1、X=1	abe
A=3、B=0、X=1	acd



路径测试

- 对于左例，下面的测试用例则可对程序进行全部的路径覆盖。

测试用例	通过路径	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



基本路径测试

- 在实际中，即使一个不太复杂的程序，其路径都是一个庞大的数字，要在测试中覆盖所有的路径是不现实的。为了解决这一难题，只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行一次。
- 基本路径测试：
 - 在程序控制图的基础上，通过分析控制构造的环行复杂性，导出基本可执行路径集合，从而设计测试用例。
 - 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执行一次。

基本路径测试

■ 前提条件

- 测试进入的前提条件是在测试人员已经对被测试对象有了一定的了解，基本上明确了被测试软件的逻辑结构。

■ 测试过程

- 过程是通过针对程序逻辑结构设计和加载测试用例，驱动程序执行，以对程序路径进行测试。测试结果是分析实际的测试结果与预期的结果是否一致。

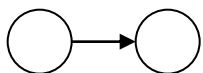
基本路径测试

- 在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。包括以下4个步骤：
 1. 以设计或源代码为基础，画出相应的流图
 2. 确定所得流图的环复杂度
 3. 确定线性独立路径的基本集合
 4. 准备测试用例，执行基本集合中每条路径

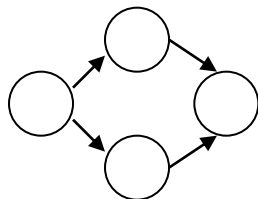
控制流图的符号

■ 流图只有2种图形符号

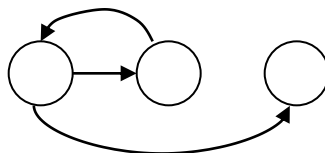
- 图中的每一个圆称为流图的结点，代表一条或多条语句。
- 流图中的箭头称为边或连接，代表控制流。



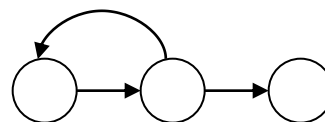
顺序结构



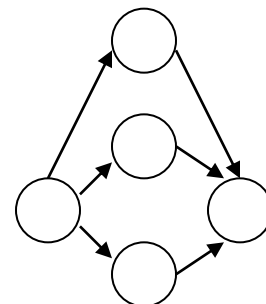
if 结构



while 结构

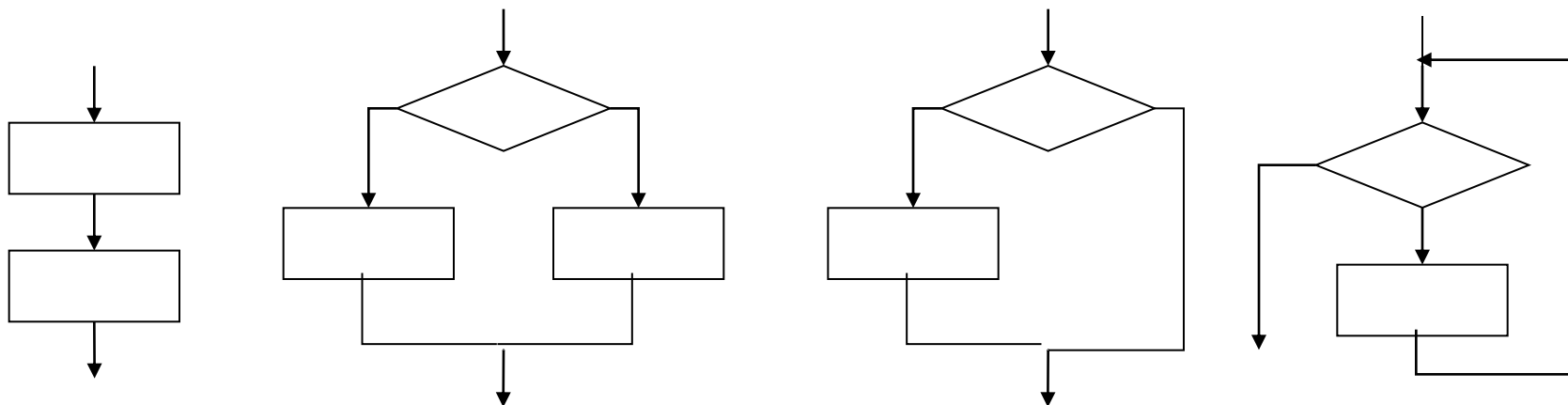


until 结构

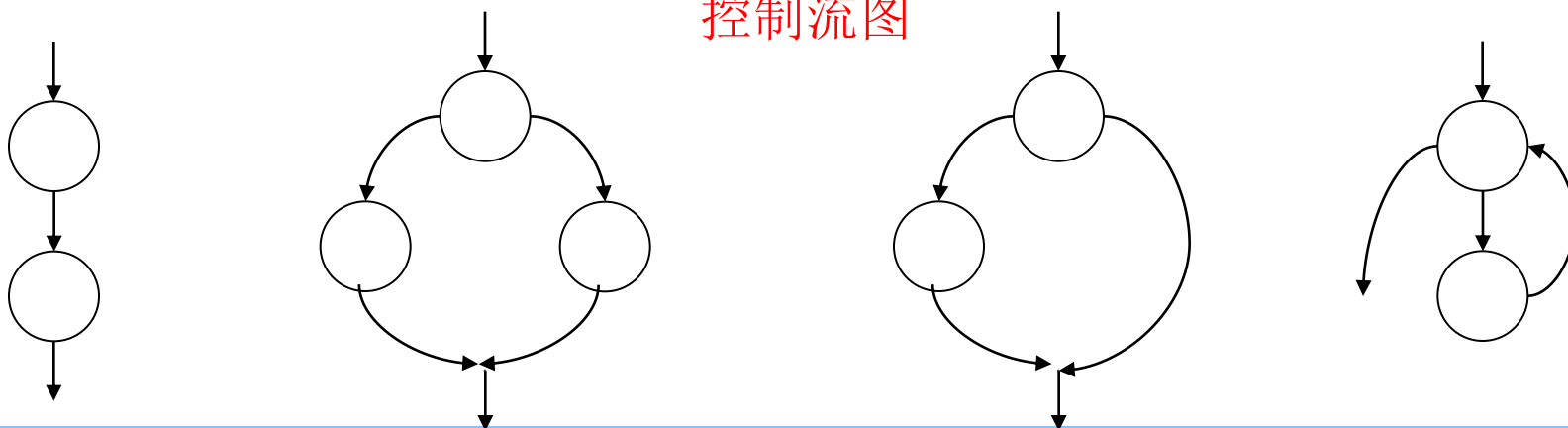


case 结构

程序流程图→控制流图



程序流程图



控制流图

控制流图

- 如果判断中的条件表达式是由一个或多个逻辑运算符 (**OR**, **AND**, **NAND**, **NOR**) 连接的复合条件表达式, 则需要改为一系列只有单条件的嵌套的判断。

- 例如:

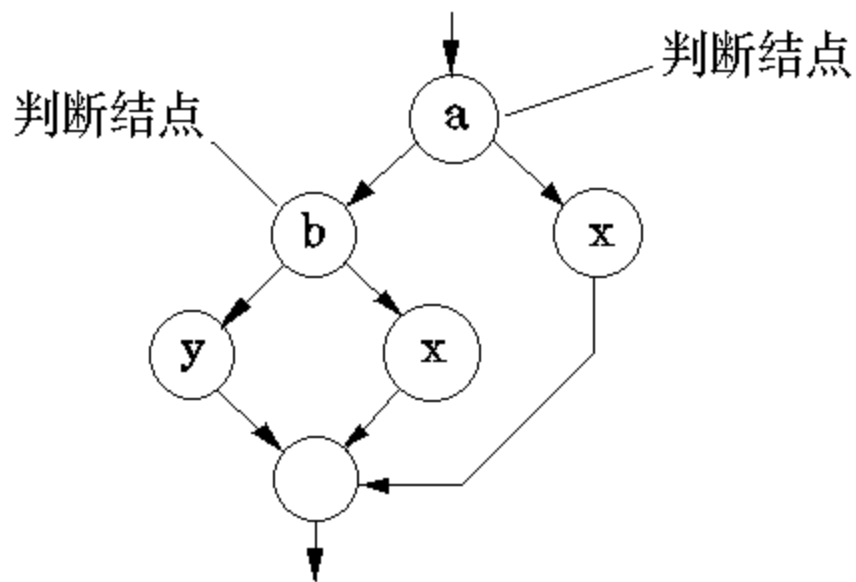
1 if a or b

2 x

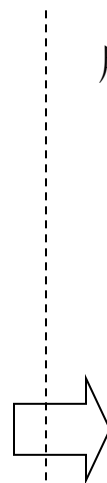
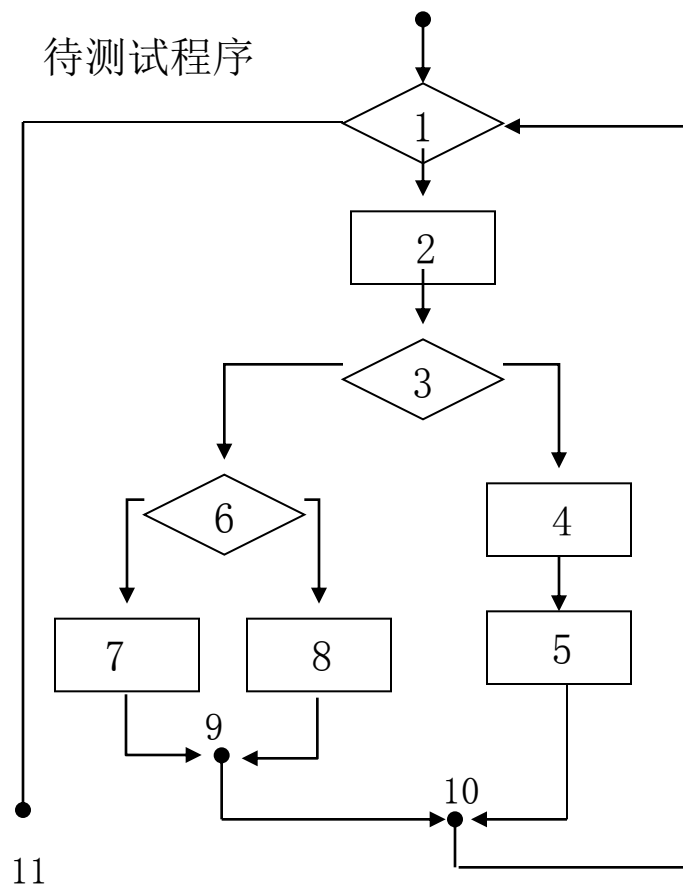
3 else

4 y

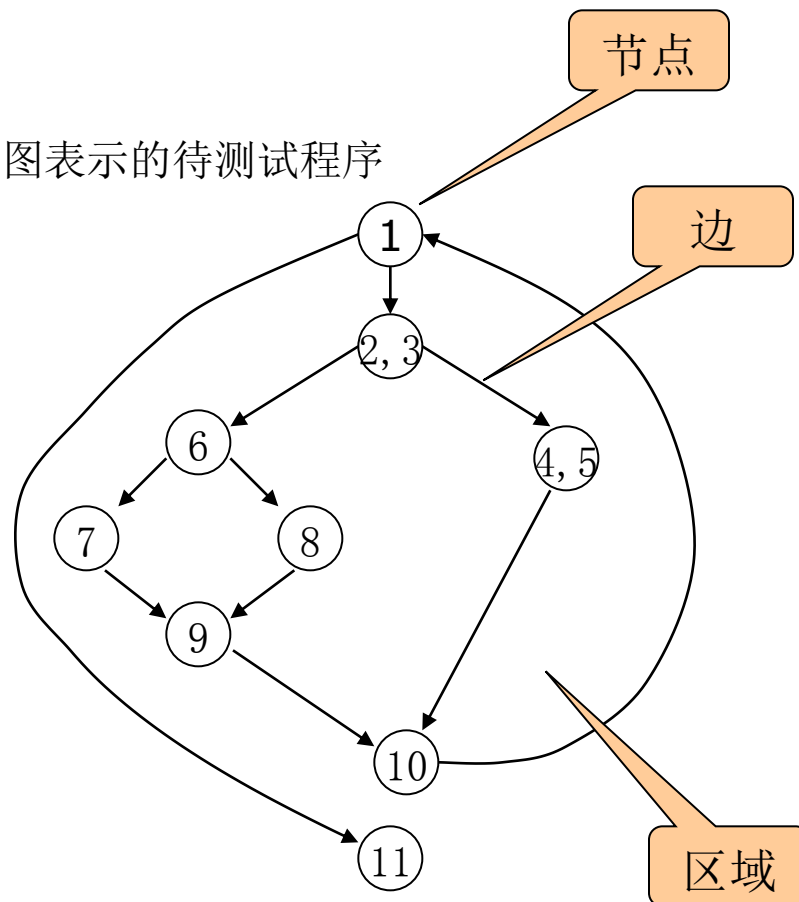
对应的逻辑为:



控制流图



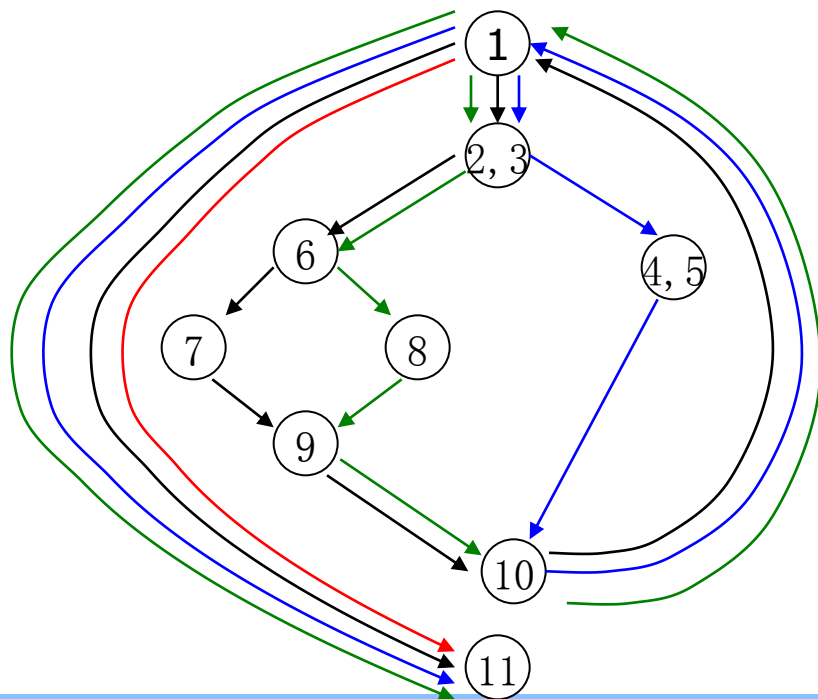
用流图表示的待测试程序



区域：由边和节点封闭起来的区域
计算区域：不要忘记区域外的部分

独立路径

- 独立路径：一条路径，至少包含一条在定义该路径之前不曾用过的边(至少引入程序的一个新处理语句集合或一个新条件)。



路径1：1-11

路径2：1-2-3-4-5-10-1-11

路径3：1-2-3-6-8-9-10-1-11

路径4：1-2-3-6-7-9-10-1-11

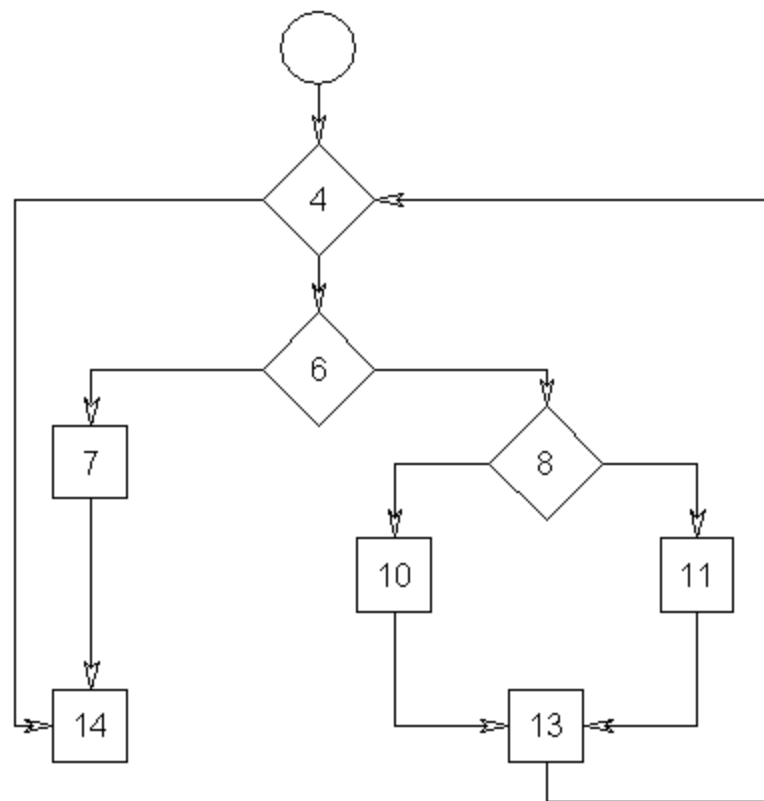
对以上路径的遍历，就是至少一次地执行了程序中的所有语句。

第一步：画出控制流图

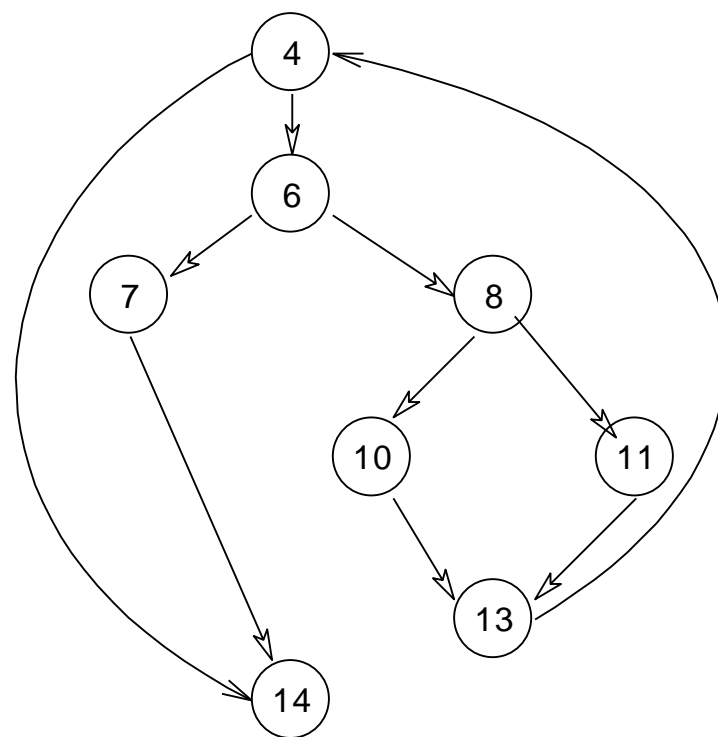
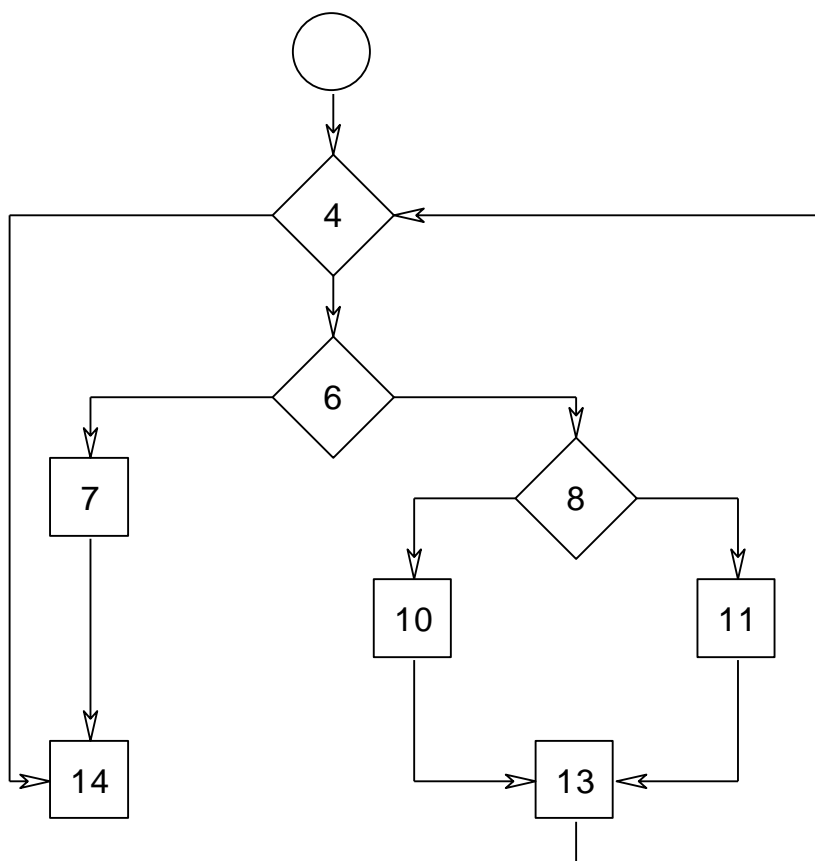
例4： 有下面的C函数，用基本路径测试法进行测试

```
void Sort(int iRecordNum,int iType)
```

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



第一步：画出控制流图



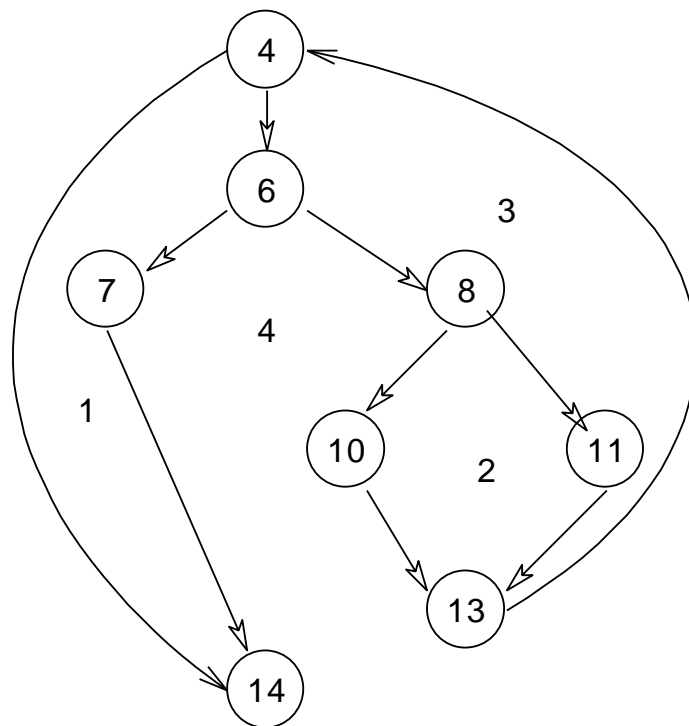
第二步：确定所得流图的环境复杂度

- 环境复杂度是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于计算程序的基本的独立路径数目，为确保所有语句至少执行一次的测试数量的上界。
- 独立路径必须包含一条在定义之前不曾用到的边。
- 有以下三种方法计算环境复杂度：
 1. 流图中区域的数量；
 2. 给定流图G的圈复杂度V(G)，定义为 $V(G)=E-N+2$ ，E是流图中边的数量，N是流图中结点的数量；
 3. 给定流图G的圈复杂度V(G)，定义为 $V(G)=P+1$ ，P是流图G中判定结点的数量。

第二步：计算环复杂度

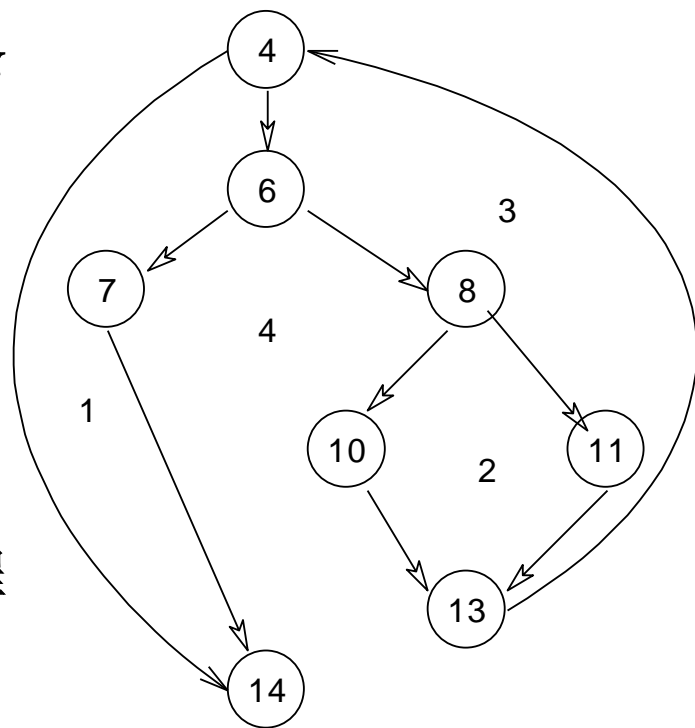
■ 对应上图中的环复杂度，计算如下：

- 流图中有4个区域；
- $V(G)=10$ 条边-8结点+2=4；
- $V(G)=3$ 个判定结点+1=4。



第三步：确定线性独立路径的基本集合

- 根据上面的计算方法，可得出四个独立的路径。
(一条独立路径是指，和其他的独立路径相比，至少引入一个新处理语句或一个新判断的程序通路。 $V(G)$ 值正好等于该程序的独立路径的条数。)
- 路径1：4-14
- 路径2：4-6-7-14
- 路径3：4-6-8-10-13-4-14
- 路径4：4-6-8-11-13-4-14
- 根据上面的独立路径，去设计输入数据，使程序分别执行到上面四条路径。



第四步：准备测试用例

- 为了确保基本路径集中的每一条路径的执行，根据判断结点给出的条件，选择适当的数据以保证某一条路径可以被测试到。

第四步：准备测试用例

路径1: 4-14

输入数据: iRecordNum=0, 或者取
iRecordNum<0的某一个值

预期结果: x=0

路径2: 4-6-7-14

输入数据: iRecordNum=1,iType=0

预期结果: x=2

路径3: 4-6-8-10-13-4-14

输入数据: iRecordNum=1,iType=1

预期结果: x=10

路径4: 4-6-8-11-13-4-14

输入数据: iRecordNum=1,iType=2

预期结果: x=20

```
void Sort(int iRecordNum,int iType)
```

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

(2) 控制结构测试

- 基本路径测试是控制结构测试技术之一，尽管基本路径测试是简单且有效的，但其本身并不充分。
- 控制结构测试提高了白盒测试的质量
 - 条件测试
 - 数据流测试
 - 循环测试

条件测试

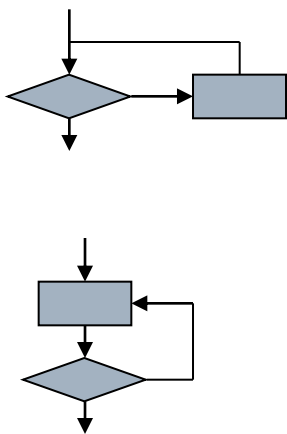
- 组成条件的元素：布尔算符、布尔变量、括号、关系算符、算术表达式等
- 条件测试通过检查程序模块中包含的逻辑条件进行测试用例设计
- 条件测试方法侧重于测试程序中的每个条件以确保其不包含错误
- 例：if **a>b or a+b>100** then

数据流测试

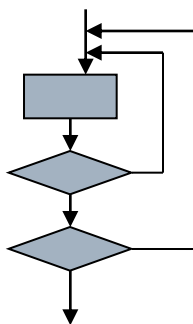
- 根据变量的定义和使用位置来选择程序测试路径的方法

循环测试

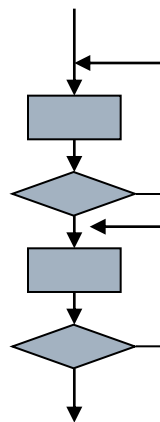
- 循环测试是一种白盒测试技术，注重于循环构造的有效性。
- 四种循环：简单循环，串接(连锁)循环，嵌套循环和不规则循环



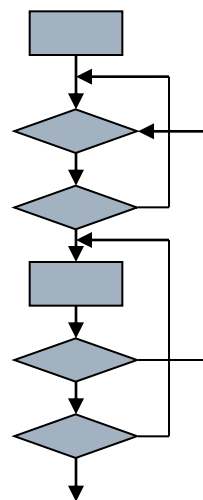
简单循环



嵌套循环



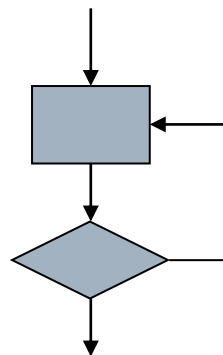
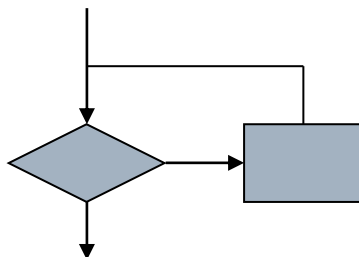
串接循环



无结构循环

简单循环

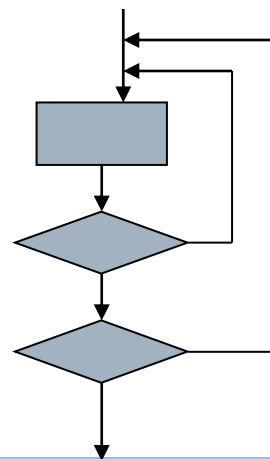
- 对于简单循环，测试应包括以下几种，其中的 n 表示循环允许的最大次数。
 - 零次循环：从循环入口直接跳到循环出口。
 - 一次循环：查找循环初始值方面的错误。
 - 二次循环：检查在多次循环时才能暴露的错误。
 - m 次循环：此时的 $m < n$ ，也是检查在多次循环时才能暴露的错误。
 - n (最大)次数循环、 $n+1$ (比最大次数多一)次的循环、 $n-1$ (比最大次数少一)次的循环。



嵌套循环

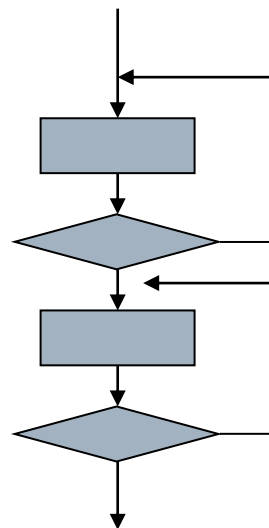
■ 对于嵌套循环：

- 从最内层循环开始，设置所有其他层的循环为最小值；
- 对最内层循环做简单循环的全部测试。测试时保持所有外层循环的循环变量为最小值。另外，对越界值和非法值做类似的测试。
- 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。
- 反复进行，直到所有各层循环测试完毕。
- 对全部各层循环同时取最小循环次数，或者同时取最大循环次数。对于后一种测试，由于测试量太大，需人为指定最大循环次数。



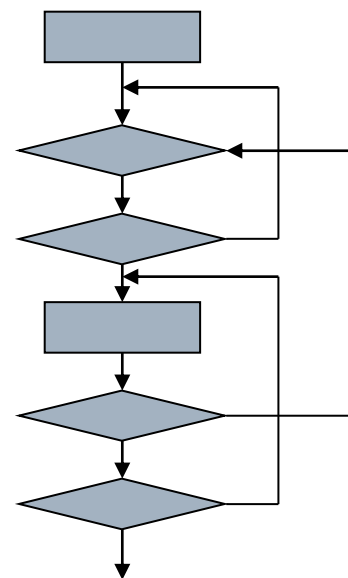
串接循环

- 对于串接循环，要区别两种情况。
 - 如果各个循环互相独立，则串接循环可以用与简单循环相同的方法进行测试。
 - 如果有两个循环处于串接状态，而前一个循环的循环变量的值是后一个循环的初值。则这几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。



非结构循环

- 对于非结构循环，不能测试，应重新设计循环结构，使之成为其它循环方式，然后再进行测试。





黑盒测试和白盒测试对比分析

黑盒测试的优点

- 对于较大的代码单元来说(子系统甚至系统级)，黑盒测试比白盒测试效率要高；
- 测试人员不需要了解实现的细节，包括特定的编程语言；
- 测试人员和编码人员是彼此独立的；
- 从用户的视角进行测试，很容易被理解和接受；
- 有助于暴露任何规格不一致或有歧义的问题；
- 测试用例可以在规格完成之后马上进行。

黑盒测试的缺点

- 只有一小部分可能的输入被测试到，要测试每个可能的输入流几乎是不可能的；
- 没有清晰的和简明的规格，测试用例是很难设计的；
- 如果测试人员不被告知开发人员已经执行过的用例，在测试数据上会存在不必要的重复；
- 会有很多程序路径没有被测试到；
- 不能直接针对特定程序段测试，这些程序段可能很复杂(因此可能隐藏更多的问题)；
- 大部分和研究相关的测试都是直接针对白盒测试的。

白盒测试的优缺点

■ 优点

- 迫使测试人员去仔细思考软件的实现;
- 可以检测代码中的每条分支和路径;
- 揭示隐藏在代码中的错误;
- 对代码的测试比较彻底;
- 最优化。

■ 缺点

- 昂贵;
- 无法检测代码中遗漏的路径和数据敏感性错误;
- 不验证规格的正确性。

有了黑盒测试为何还需要白盒测试

- 黑盒测试只能观察软件的外部表现，即使软件的输入输出都是正确的，却并不能说明软件就是正确的。因为程序有可能用错误的运算方式得出正确的结果，例如“负负得正，错错得对”，只有白盒测试才能发现真正的原因。
- 白盒测试能发现程序里的隐患，例如内存泄漏、误差累计问题。在这方面，黑盒测试存在严重的不足。

有了白盒测试为何还需要黑盒测试

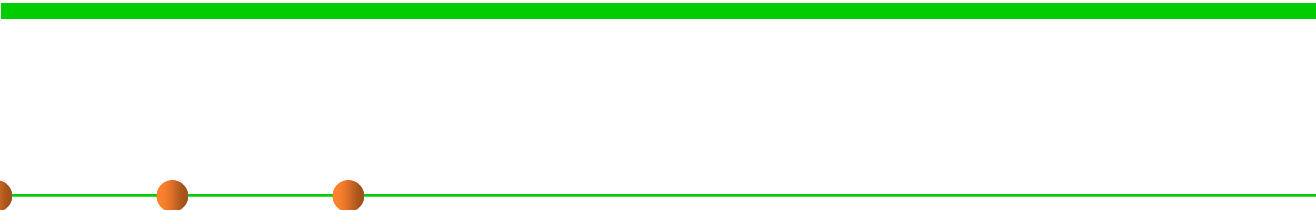
- 通过了白盒测试只能说明程序代码符合设计需求，并不能说明程序的功能符合用户的需求。如果程序的系统设计偏离了用户需求，即使**100%**正确编码的程序也不是用户所要的。

测试工具

- 黑盒测试（功能测试）：**winRunner**
- 白盒测试（单元测试）：**xUnit(JUnit/NUnit/CppUnit/PhpUnit)**

软件测试报告





结束

2011年5月25日