软件工程

第8章 软件测试

@第8章.教材

本章内容

- 测试概述
- 测试策略
- 测试技术
- 测试过程
- 自动化测试
- 软件可靠性（补充）

1、测试概述

- 测试（*testing*）的目的
  - 发现软件的错误，从而保证软件质量
- 成功的测试
  - 发现了未曾发现的错误
- 与调试（*debugging*）的不同在哪？
  - 定位和纠正错误
  - 保证程序的可靠运行

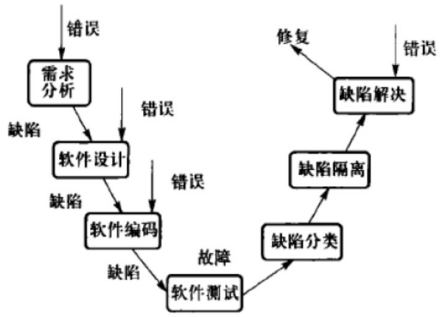


图 8－1 软件错误的引入阶段

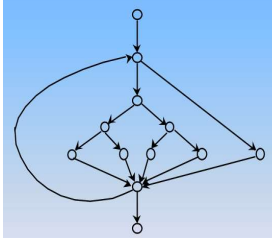
测试准则

- 所有的软件测试应追溯到用户的需求
- 穷举测试是不可能的
- 根据软件错误的聚集性规律，对存在错误的程序段应进行重点测试
- 尽早地和不断地进行软件测试
- 避免测试自己的程序
- 制定测试计划，避免测试的随意性
- 测试应该从小到大

黑盒测试不可能实现穷尽测试：

假设有一个很简单的小程序，输入量只有两个：A和B，输出量只有一个：C。如果计算机的字长为32位，A和B的数据类型都只是整数类型。利用黑盒法进行测试时，将A和B的可能取值进行排列组合，输入数据的可能性有：$2^{32}\times 2^{32}=2^{64}$种。假设这个程序执行一次需要1毫秒，要完成所有的测试，计算机需要连续工作5亿年。显然，这是不能容忍的，而且，设计测试用例时，不仅要有合法的输入，而且还应该有非法的输入，在这个例子中，输入还应该包括实数、字符串等，这样，输入数据的可能性就更多了。所以说，穷尽测试是不可能实现的。

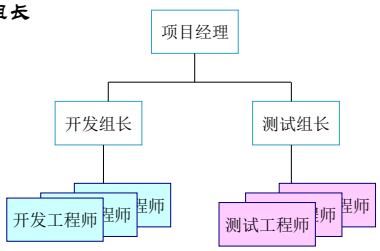
白盒测试也不能实现为穷测试(Exhaustive testing)：



左图所示的一个小程序的控制流程，其中每个圆圈代表一段源程序（或语句块），图中的曲线代表执行次数不超过20的循环，循环体中共有5条通路。这样，可能执行的路径有5^{20}条，近似为10^{14}条可能的路径。如果完成一个路径的测试需要1毫秒，那么整个测试过程需要3170年。显然，这也是不能接受的。

测试组的组成

- 测试经理/测试组长
- 测试工程师
  - 测试开发工程师
  - 测试工程师



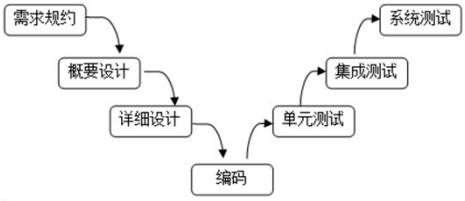
测试工程师和开发工程师的能力要求和工作习惯是不同的

2、测试策略

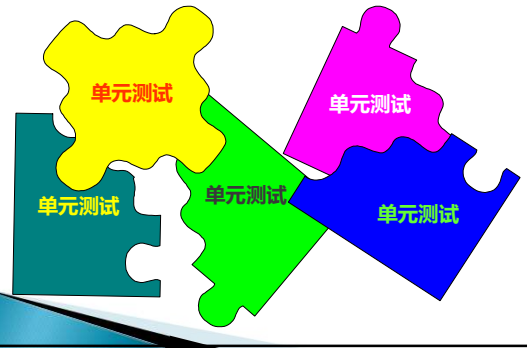
- 按测试层次分类
  - 单元测试、集成测试、系统测试
- 按软件质量属性分类
  - 功能性测试、可靠性测试、易用性测试、性能测试、可移植性测试、可维护性测试
- 其他测试策略
  - 验收测试、α测试、β测试、安装测试、回归测试

测试层次

- 不同层次的测试：
  - 单元测试 (Unit testing)
  - 集成测试 (Integration testing)
  - 系统测试 (System testing)

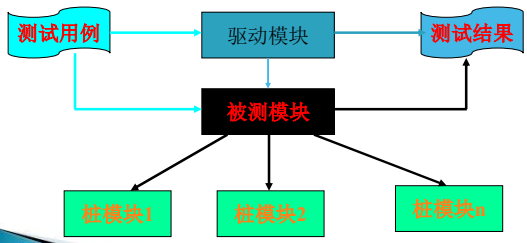


单元测试



单元测试

- 单元测试（*unit testing*），又称为模块测试，是针对软件结构中独立的基本单元（如函数、子过程、类）进行的测试。



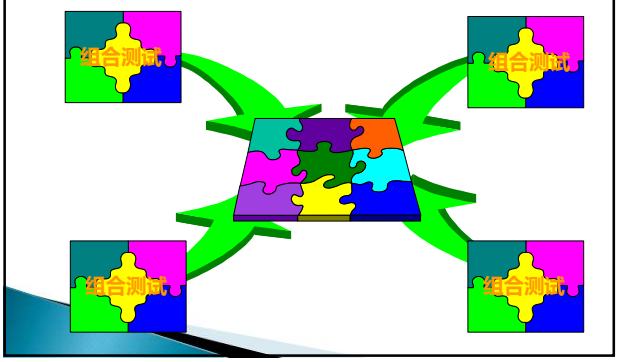
单元测试——测什么？

- 单元测试是针对每个基本单元，重点关注5个方面：
  - 模块接口
  - 局部数据结构
  - 边界条件
  - 独立的路径
  - 错误处理路径

单元测试——何时测试

- 一般地，该基本单元的编码完成后就可以对其进行单元测试。
- 也可以提前，即测试驱动开发（*test driven development*），在详细设计的时候就编写测试用例，然后再编写程序代码来满足这些测试用例。

集成测试



集成测试

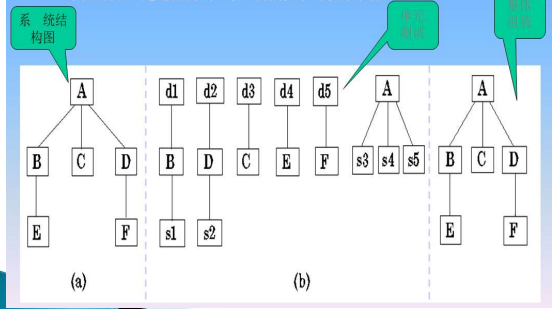
- 集成测试（*integration testing*），又称组装测试，它根据设计将软件模块组装起来，进行有序的、递增的测试，并通过测试评价它们之间的交互。
- 集成测试一般由项目经理组织软件测试工程师或由独立的测试部门进行。
- 集成测试重点关注：
  - 在把各个软件单元连接起来的时候，穿越单元接口的数据是否会丢失；
  - 一个软件单元的功能是否会对另一个软件单元的功能产生不利的影响；
  - 各个子功能组合起来，能否达到预期要求的父功能；
  - 全局数据结构是否有问题；
  - 单个软件单元的误差累积起来，是否会放大，从而达到不能接受的程度。

软件集成策略

- 增量式集成
  - 自顶向下集成
  - 由底向上集成
  - 混合方式集成
    - 对软件中上层使用自顶向下集成，对软件的中下层采用自底向上集成。
- 一次性集成
  - 缺点：接口错误发现晚，错误定位困难
  - 优点：可以并行测试和调试所有软件单元

1.一次性组装方式

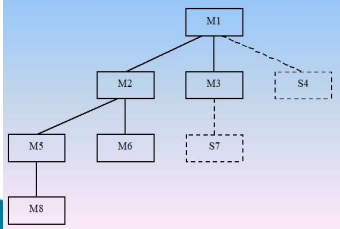
- 它是一种非渐增式组装方式。也叫做整体拼装。
- 使用这种方式，首先对每个模块分别进行模块测试，然后再把所有模块组装在一起进行测试，最终得到要求的软件系统。



在使用渐增式组装方式时，常用的有自顶向下和自底向上两种方法。

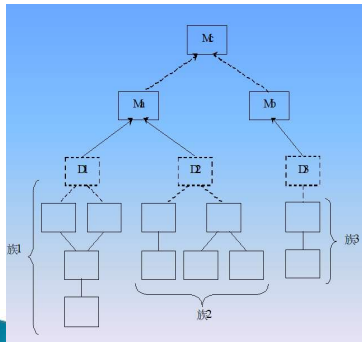
1、自顶向下结合Top-Down Intergration

采用这种组装方式时，是从主控制模块开始，沿着软件的控制层次向下移动，从而逐渐把各个模块都结合起来。



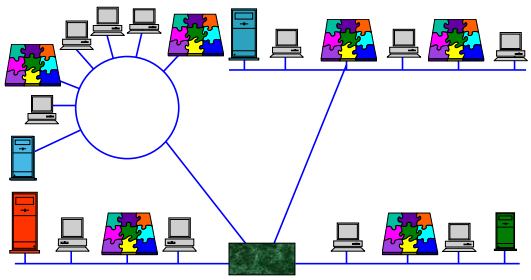
左图是一个树形结构，主控制模块是M1，在把主控制模块M1所属的那些模块都组装起来时可以采取两种方法：深度优先策略或者宽度优先策略 (depth-frist or breadtn-frist manner)。

2、自底向上结合Bottom-Up Intergration



图中自底向上的结合过程：首先把模块组合成族1、族2和族3，然后设计相应的驱动模块D1、D2和D3，并对每个子功能族进行测试；族1和族2下属于模块Ma，去掉驱动模块D1和D2，把这两个族直接与Ma结合，同样地，在族3与模块Mb结合之前将D3去掉；最后Ma和Mb与Mc结合起来。

系统测试



系统测试

- 软件集成及集成测试完成后，对整个软件系统进行的一系列测试，称为系统测试（*system testing*）。
- 系统测试的目的是为了验证系统是否满足需求规约。
- 测试内容包括功能测试和非功能测试，其中非功能测试常常是系统测试的重点，例如：可靠性测试、性能测试、易用性测试、可维护性测试、可移植性测试等。
- 如果该软件只是一个大的计算机系统的一个组成部分，此时应将软件与计算机系统的其他元素集成起来，检验它能否与计算机系统的其他元素协调地工作。
- 系统测试一般由与开发无直接责任关系的独立方负责，例如项目组的软件测试工程师、测试部门、第三方评测机构、客户等。

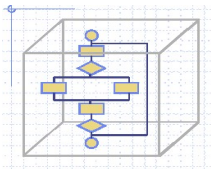
软件质量属性的测试

- **功能性测试**
  又称正确性测试或一致性测试，包括适应性、准确性、互操作性、安全性、功能依存性测试。
- **可靠性测试**
  成熟性、容错性、已恢复性、可靠性依从性测试。
- **性能测试**
  时间特性、资源利用性、性能依存性测试，常用压力测试方法。
- **易用性测试**
  易理解性、易学性、易操作性、吸引性、依从性测试。
- **可移植性测试**
  适应性、易安装、易替换、可移植性依从测试。
- **可维护性测试**
  被修改的能力，包括易分析、易改变、稳定性、易测试、维护依从性测试。

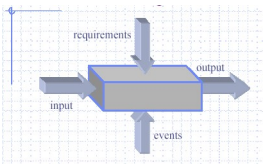
其他测试策略

- **验收测试**
  由用户主导的，根据合同、需求规约或验收计划对软件成品进行验收测试。
- **α测试和β测试**
  α测试：由开发者主导，在受控环境中进行
  β测试：在用户环境中进行，开发者不在现场
- **安装测试**
  测试各种允许安装平台能否成功安装。
- **回归测试**
  对软件系统或部件重新测试，测试改动没有引入新的错误。

3、软件测试技术



白盒测试



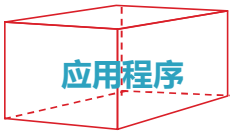
黑盒测试

软件测试技术

		白盒测试	黑盒测试
通用测试技术	基于直觉和经验	即兴测试*	
		探索式测试*	
	基于代码	控制流测试*	
		基本路径测试*	
		数据流测试	
	基于规约		等价类划分*
			边界值分析*
			随机测试*
	基于错误	错误猜测*	
		变异测试	
	基于模型		因果图/判定表
			基于有限状态机的测试
			基于形式化规约的测试
专用测试技术	专用测试技术（即基于应用类型）	面向对象的测试	
		基于构件的测试	
		并发程序的测试	
			基于Web的测试
			图形用户界面的测试
			协议一致性的测试
			实时系统的测试

白盒测试

- 白盒测试把被测软件看作一个透明的白盒子，测试人员可以完全了解软件的设计或代码，按照软件内部逻辑进行测试。
- 白盒测试又称玻璃盒测试，常常应用在单元测试中。

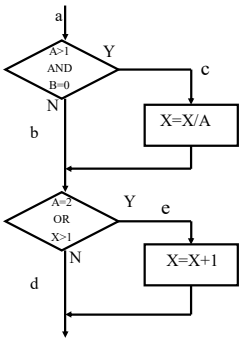


控制流测试（属白盒测试）

1）语句覆盖法
2）判定覆盖（分支）
3）条件覆盖
4）判定/条件覆盖
5）条件组合覆盖
6）路径覆盖

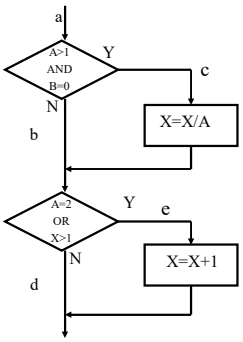
1）语句覆盖法

- 使得程序中的每一个语句至少被遍历一次
- 测试用例：
  A=2，B=0，X=3



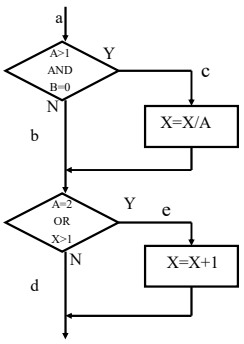
2）判定覆盖（分支）

- 使得程序中每一个分支至少被遍历一次
- 测试用例
  1. A=2, B=0, X=1
     （沿路径 ace）
  2. A=1, B=0, X=0
     （沿路径abd）



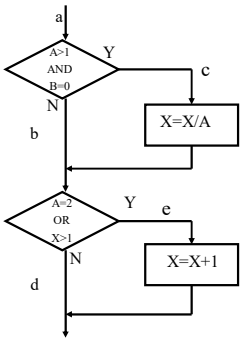
3）条件覆盖

- 使得每个判定的条件获取各种可能的结果
- 在a点, A>1, A≤1, B=0, B≠0
- 在b点, A=2, A≠2, X>1, X≤1
- 测试用例
  1. A=2, B=0, X=4 （沿路径ace）
  2. A=1, B=1, X=1 （沿路径abd）



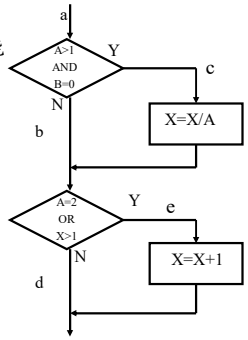
4）判定/条件覆盖

- 使得判定中的条件取得各种可能的值，并使得每个判定取得各种可能的结果
- 测试用例
  1. A=2, B=0, X=4 （沿路径ace）
  2. A=1, B=1, X=1 （沿路径abd）



5）条件组合覆盖

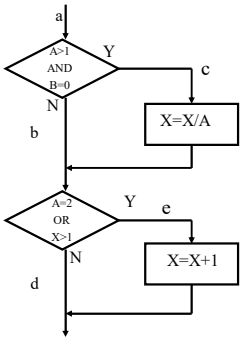
- 使得每个判定条件的各种可能组合都至少出现一次
- 要求
  1. A>1, B= 0　5. A=2, X>1
  2. A>1, B≠0　6. A=2, X≤1
  3. A≤1, B=0　7. A≠2, X>1
  4. A≤1, B≠0　8. A≠2, X≤1
- 测试用例
  1.A=2, B=0, X=4
  2.A=2, B=1, X=1
  3.A=1, B=0, X=2
  4.A=1, B=1, X=1



6）路径覆盖

- 覆盖程序中所有可能的路径

A	B	X	覆盖路径	
2	0	3	a c e	L_1
1	0	1	a b d	L_2
2	1	1	a b e	L_3
3	0	1	a c d	L_4



黑盒测试

- 黑盒测试把程序看成一个黑盒子，完全不考虑程序内部结构和处理过程。
- 黑盒测试是在程序接口进行测试，它只是检查程序功能是否按照需求规约正常使用。
- 黑盒测试又称功能测试、行为测试，在软件开发后期执行。

基于规约的测试（黑盒）

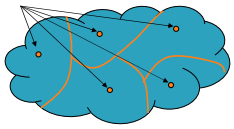
- **等价类划分**
  - 将所有可能的输入数据划分成若干个等价类，然后在每个等价类中选取一组（通常是一个）代表性的数据作为测试用例。
- **边界值分析**
  - 通常是等价类划分技术的一种补充，在等价类划分技术中，一个等价类中的任一输入数据都可作为该等价类的代表用作测试用例，而边界值分析技术则是专门挑选那些位于输入或输出范围边界附近的数据用作测试用例。
- **随机测试**

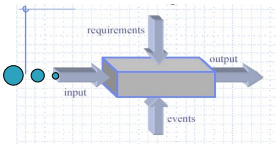
  在软件输入域上随机选择输入数据来测试软件的技术。

设计测试方案——黑盒测试技术

测试人员将程序看成是一个“黑盒”，即不关心程序内部是什么，只要检查程序是符合它的“功能说明”。

- N个等价类
- 每个等价类中的一组具代表性的测试数据





2 等价划分(1)

- 等价分类法是将输入数据的可能值分成若干“等价类”，每一类以一个代表性的测试数据进行测试，这个数据就等价于这一类中的其它数据。
- 该法的关键在于如何将输入数据分类。
- 例如：输入的数据范围是1~999，我们可以划分三类：x<1，1≤x≤999，x>999

1、划分等价类
Equivalence classes Partitioning

等价类的划分在很大程度上依靠的是测试人员的经验，下面给出几条基本原则：

（1）输入规定了取值范围，则可划分出一个有效的等价类（输入值在此范围内）和两个无效的等价类（输入值小于最小值、输入值大于最大值）。

（2）输入规定了输入数据的个数，则可相应地划分出一个有效的等价类（输入数据的个数等于给定的个数要求）和两个无效的等价类（输入数据的个数少于给定的个数要求、输入数据的个数多于给定的个数要求）。

（3）输入规定了输入数据的一组可能的值，而且程序对这组可能的值做相同的处理，则可将这组可能的值划分为一个有效的等价类，而这些值以外的值划分成无效的等价类。

（4）输入规定了输入数据的一组可能的值，但是程序对不同的输入值做不同的处理，则每个输入值是一个有效的等价类，此外还有一个无效的等价类（所有不允许值的集合）。

（5）输入规定了输入数据必须遵循的规则，则可以划分一个有效的等价类（符合规则）和若干个无效的等价类（从各种角度违反规则）。

确定测试用例
Test case design for Equivalence Partitioning

划分出等价类后，根据以下原则设计测试用例：

（1）为每个等价类编号。

（2）设计一个新的测试用例，使它能包含尽可能多的尚未被覆盖的有效等价类。重复这一过程，直到所有的有效等价类都被覆盖。

（3）设计一个新的测试用例，使它包含一个尚未被覆盖的无效等价类。重复这一过程，直到所有的无效等价类都被覆盖。

等价划分(例)

等价类说明	测试数据	预期输出	测试结果	备注
1-6个数字的数字串	1	1		
最高位是零的数字串	000001	1		
最高位数字左邻是负号的数字串	-00001	-1		
最高位是零的数字串	000000	0		
太小的负整数	-47561	错误—无效输入（负数）		
太大的正整数	132767	错误—无效输入（正数）		
空字符串---6个空格		错误-没有数字		
字符串左边字符既不是空格也不是零	*+kgh1	错误—填充错		
最高位数字后面有空格	1 2	错误—无效输入		
最高位数字后面有其他字符	1****2	错误—无效输入		
负号和最高位数字之间有空格	- 12	错误—负号位置错		

2、边界值分析
Boundary Value Analysis

人们在长期的测试中发现，程序往往在处理边界值的时候容易出错。

通常输入等价类和输出等价类的边界，就是应该着重测试的程序边界情况。

边界值分析也属于黑盒测试，可以看作是对等价类划分的一个补充。

在设计测试用例时，往往联合等价类划分和边界值分析这两种方法。

边界值分析(例)

经验表明：处理边界情况时程序最容易发生错误；
下标、数据结构、循环等边界。

- 对等价划分法中的不同等价类的边界情况进行重点测试

等价类说明	测试数据	预期输出	测试结果	备注
使输出刚好等于最小的负整数	-32768	-32768		
使输出刚好等于最大的正整数	32767	32767		
使输出刚刚小于最小的负整数	-32769	错误—无效输入（负数）		
使输出刚刚大于最大的正整数	32768	错误—无效输入（正数）		

- 不同类型不同特点的程序通常有一些特殊的容易出错的情况；
- 有时测试数据的组合数量也是非常多，难于覆盖所有情况；
- 经验数据

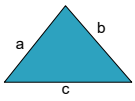
错误猜测（黑盒和白盒）

- **错误猜测（*error guessing*）是一种凭经验、知识和直觉推测某些可能存在的错误，从而针对这些可能存在的错误设计测试用例的技术。**
- 基本思想：列举出程序中所有可能的错误和容易发生错误的特殊情况，然后根据这些猜测设计测试用例。
- 例如，测试一个排序子程序，可考虑如下情况：
  - 输入表为空
  - 输入表只有一个元素
  - 输入表的所有元素都相同
  - 输入表已排好序

讨论：三角形测试

- 从键盘上输入三个整数，这三个数值表示三角形三条边的长度。然后，输出信息，以表明这个三角形是等腰、等边或是一般三角形，或不能构成三角形。
- 请采用黑盒测试方法例举有多少种测试用例？

例：程序Triangle读入三个整数值，这三个整数代表同一个三角形三条边的长度，程序根据这三个值判断三角形属于不等边、等腰或等边三角形中的那一种。



黑盒测试（等价划分）

正常的三角形 （a,b,c）	
不等边三角形	(8,10,12); (10,8,12); (10,12,8)
等边三角形	(10,10,10)
等腰三角形	(10,10,17); (10,17,10); (17,10,10)

退化的三角形 （a,b,c）	
不等边三角形	(10,6,4)
等边三角形	(0,0,0)
等腰三角形	(10,5,5); (5,10,5); (10,5,5)

不能构成三角形的非法数据 (a,b,c)	
a+b<c	(10,10,21)
b+c<a	(21,10,10)
c+a<b	(10,21,10)

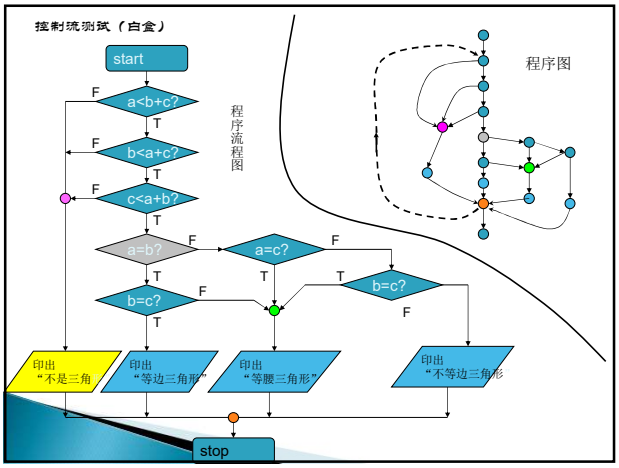
黑盒测试（边界值分析）

一条边长度为零的情况	（0，10，12）；（10，0，12）；（10，12，0）
两条边的长度为零的情况	（0，0，17）；（0，17，0）；（17，0，0）
三条边的长度为零的情况	（0，0，0）

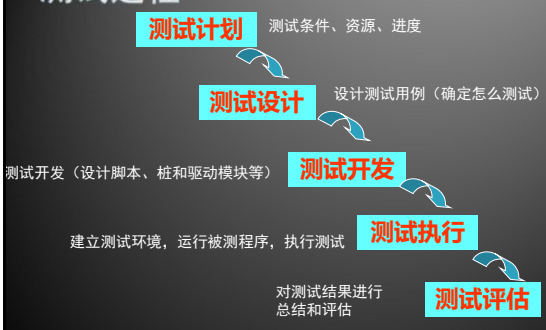
黑盒测试（错误推测）

输入数据中包含负整数	（-10，-10，-10）……
输入数据不全（不足三个正整数）	（10，-，-）……
输入数据中包含非整数型的数据	(a,b,c) (1.2,6e-4,7.8) ……

控制流测试（白盒）



测试过程



- 测试计划、测试设计和测试开发在软件开发完成前进行
- 测试执行只能在软件开发完成后进行

测试计划

- 测试目的
- 测试对象
- 测试范围
- 文档的检验
- 测试策略和测试技术
- 测试过程
- 进度安排
- 资源
- 测试开始、结束准则
- 测试文档和测试记录

测试用例设计文档

- 测试用例（*test case*）是按一定顺序执行的与测试目标相关的一系列测试。其主要内容包括：
  - 测试输入
  - 测试操作
  - 期望结果

编号	标题	步骤	期望结果	结果
1	系统设置模块			
1-1	用户管理		添加，修改，删除用户，设置权限。	
1-1-1	添加用户	1. 点击菜单中的用户管理菜单项进入用户管理窗口。 2. 点击〈新建〉命令按钮。 3. 然后，分别输入用户信息。 4. 最后，点击〈保存〉命令按钮	正确添加用户	通过

缺陷报告

- 内容包括：缺陷名称、分类、等级、发现时间，发现人，所执行的测试用例、现象等
- 缺陷等级
  - 5级：灾难性的--系统崩溃、数据被破坏
  - 4级：很严重的--数据被破坏
  - 3级：严重的--特性不能运行，无法替代
  - 2级：中等的--特性不能运行，可替代
  - 1级：烦恼的--提示不正确，报警不确切
  - 0级：轻微的--表面化的错误，拼写错等
- 缺陷报告通常保存在缺陷跟踪系统

测试报告

- 被测软件的名称和标识
- 测试环境
- 测试对象
- 测试起止日期
- 测试人员
- 测试过程
- 测试结果
- 缺陷清单
- 等

开始和终止测试的标准

- 在测试计划中规定开始和终止测试的标准
- 开始测试的常用标准
  - 通过“冒烟” 测试
- 终止测试的常用标准
  - 所有严重的缺陷都已纠正，剩余的缺陷密度少于0.01%
  - 100%测试覆盖度
  - 缺陷数收敛了

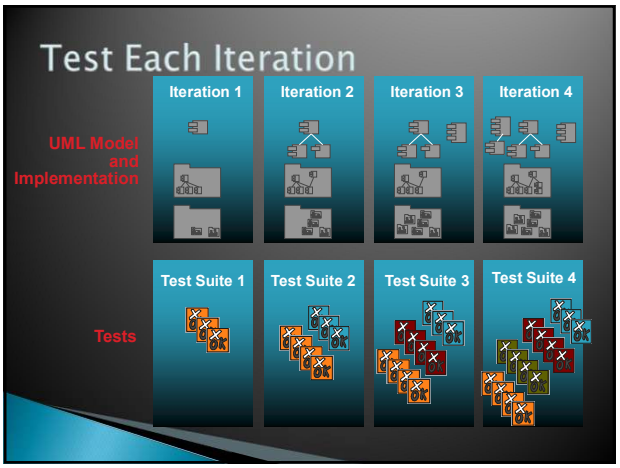
自动化测试

- 工具类型
  - 单元测试工具，即白盒测试工具
  - 性能测试工具
  - 功能测试工具，即回归测试工具
  - 缺陷跟踪工具
  - 测试数据生成工具
  - 测试管理工具
  - 等
- 工具产品
  - HP Mercury： WinRunner, LoadRunner …
  - IBM Rational
  - Compuware： QA Run, QA Load, QA Director …
  - Freeware：JUnit, Bugzilla, Mantis …
  - ……

什么情况下适合用自动测试？

- 产品型项目
- 增量式开发、持续集成项目
- 能够自动编译、自动发布的系统
- 回归测试
- 多次重复的机械性动作，如性能测试
- 需要频繁运行的测试
- 将烦琐的任务转化为自动化测试

讨论：迭代开发中测试有什么特点



软件可靠性

Definition of Software Reliability

软件可靠性是软件可靠性是程序在给定的时间间隔内，按照规格说明书的规定成功地运行的概率。

术语“错误”的含义是由开发人员造成的软件差错（*bug*），而术语“故障”的含义是由错误引起的软件的不正确行为。

软件可用性是程序在给定的时间点，按照规格说明书的规定，成功地运行的概率。

平均无故障时间*MTTF*是系统按规格说明书规定成功地运行的平均时间。

如果在一段时间内，软件系统故障停机时间分别为t_{d1}，t_{d2}，…，正常运行时间分别为t_{u1}，t_{u2}，…，则系统的稳态可用性为：

$$A_{ss}=T_{up}/(T_{up}+T_{down}) \quad (1)$$

其中$T_{up}=\Sigma t_{ui}$，$T_{down}=\Sigma t_{di}$

如果引入系统平均无故障时间MTTF和平均维修时间MTTR的概念，则(1)式可以变成

$$A_{ss}=MTTF/(MTTF+MTTR)$$

平均无故障时间MTTF它主要取决于系统中潜伏的错误的数目，因此和测试的关系十分密切。

1.估算平均无故障时间的方法

软件的平均无故障时间*MTTF*是一个重要的质量指标，往往作为对软件的一项要求，由用户提出来。为了估算*MTTF*，首先引入一些有关的量。

1. 符号

在估算MTTF的过程中使用下述符号表示有关的数量：

E_T——测试之前程序中错误总数；

I_T——程序长度(机器指令总数)；

τ——测试(包括调试)时间；

$E_d(\tau)$——在0至τ期间发现的错误数；

$E_c(\tau)$——在0至τ期间改正的错误数。

2. 基本假定

根据经验数据，可以作出下述假定。

(1) 在类似的程序中，单位长度里的错误数E_T/I_T近似为常数。美国的一些统计数字表明，通常$0.5\times10^{-2}\leqslant E_T/I_T\leqslant2\times10^{-2}$也就是说，在测试之前每1000条指令中大约有5～20个错误。

(2) 失效率正比于软件中剩余的(潜藏的)错误数，而平均无故障时间MTTF与剩余的错误数成反比。

(3) 此外，为了简化讨论，假设发现的每一个错误都立即正确地改正了(即，调试过程没有引入新的错误)。因此

$E_c(\tau)=E_d(\tau)$

剩余的错误数为

$E_r(\tau)=E_T-E_c(\tau)$

单位长度程序中剩余的错误数为

$\varepsilon_r(\tau)=E_T/I_T-E_c(\tau)/I_T$

3. 估算平均无故障时间

经验表明，平均无故障时间与单位长度程序中剩余的错误数成反比，即

$MTTF=1/[K(E_T/I_T-E_c(\tau)/I_T)]$

其中K为常数，它的值应该根据经验选取。美国的一些统计数字表明，K的典型值是200。

估算平均无故障时间的公式，可以评价软件测试的进展情况。此外，由上式可得

$E_c=E_T-I_T/(K\times MTTF)$

因此也可以根据对软件平均无故障时间的要求，估计需要改正多少个错误之后，测试工作才能结束。

4. 估计错误总数的方法

程序中潜藏的错误的数目是一个十分重要的量，它既直接标志软件的可靠程度，又是计算软件平均无故障时间的重要参数。

程序中的错误总数估计E_T的两个方法。

(1) 植入错误法

在测试之前由专人在程序中随机地植入一些错误，测试之后，根据测试小组发现的错误中原有的和植入的两种错误的比例，来估计程序中原有错误的总数E_T。

假设人为地植入的错误数为N_s,经过一段时间的测试之后发现n_s个植入的错误，此外还发现了n个原有的错误。如果可以认为测试方案发现植入错误和发现原有错误的能力相同，则能够估计出程序中原有错误的总数为

$N=(n/n_s)\times Ns$

其中N即是错误总数E_T的估计值。

(2) 分别测试法

植入错误法的基本假定是所用的测试方案发现植入错误和发现原有错误的概率相同。但是，人为地植入的错误和程序中原有的错误可能性质很不相同，发现它们的难易程度自然也不相同，因此，上述基本假定可能有时和事实不完全一致。

如果有办法随机地把程序中一部分原有的错误加上标记，然后根据测试过程中发现的有标记错误和无标记错误的比例，估计程序中的错误总数，则这样得出的结果比用植入错误法得到的结果更可信一些。

为了随机地给一部分错误加标记，分别测试法使用两个测试员(或测试小组)，彼此独立地测试同一个程序的两个副本，把其中一个测试员发现的错误作为有标记的错误。具体做法是，在测试过程的早期阶段，由测试员甲和测试员乙分别测试同一个程序的两个副本，由另一名分析员分析他们的测试结果。用τ表示测试时间，假设

$\tau=0$时错误总数为B_0;

$\tau=\tau_1$时测试员甲发现的错误数为B_1;

$\tau=\tau_1$时测试员乙发现的错误数为B_2;

$\tau=\tau_1$时两个测试员发现的相同错误数为b_c。

即程序中有标记的错误总数为B_1,则测试员乙发现的B_2个错误中有b_c个是有标记的。假定测试员乙发现有标记错误和发现无标记错误的概率相同，则可以估计出测试前程序中的错误总数为

$$B_0 = (B_1/b_c)B_2$$

作业：设计下列伪码程序的语句覆盖和路径覆盖测试用例：

```
START
INPUT (A,B,C)
IF A>5
        THEN X=10
        ELSE X=1
END IF
IF B>10
        THEN Y=20
        ELSE Y=2
END IF
IF C>15
        THEN Z=30
        ELSE Z=3
END IF
PRINT (X,Y,Z)
STOP
```