上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

软件工程原理与实践
SOFTWARE ENGINEERING

软件测试

沈备军

饮水思源 · 爱国荣校

大 纲

01-软件测试概述

02-软件测试技术

03-软件测试策略

04-软件测试过程

05-自动化测试

@第8章.教材

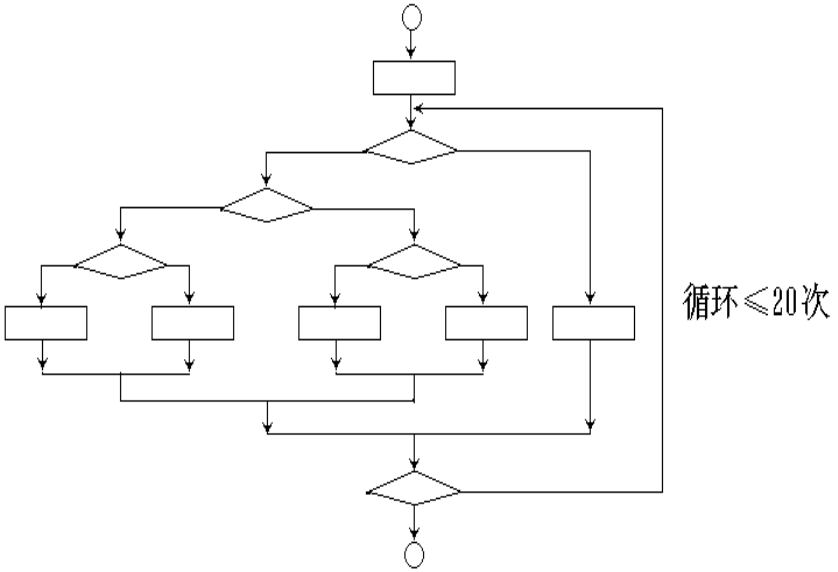
测试（testing）的目的

- 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

有关软件测试的错误观点

- “软件测试是为了证明程序是正确的，即测试能发现程序中所有的错误”。
  - 事实上这是不可能的。要通过测试发现程序中的所有错误，就要穷举所有可能的输入数据。
  - 举例：对于一个输入三个16位字长的整型数据的程序，输入数据的所有组合情况有$2^{48} \approx 3*10^{14}$，如果测试一个数据需1ms，则即使一年365天一天24小时不停地测试，也需要约1万年。

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。例如一个小程序的流程图，它包括了一个执行20次的循环，其循环体有五个分支。这个循环的不同执行路径数达520条，如果对每一条路径进行测试需要1毫秒，那么即使一年工作365 ×24小时，要想把所有路径测试完，大约需3170年。



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
- 制定测试计划，避免测试的随意性
- 测试应该从小到大

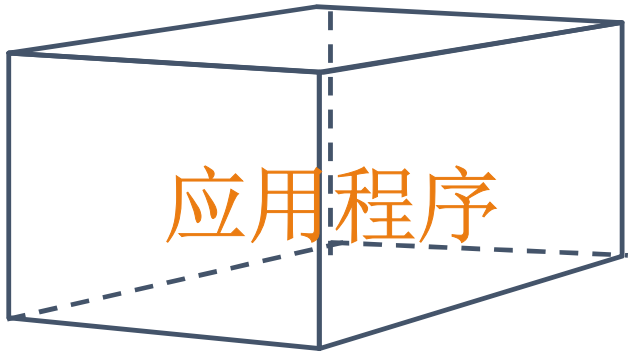
大 纲

软件测试技术

	白盒测试	黑盒测试
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 (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。



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

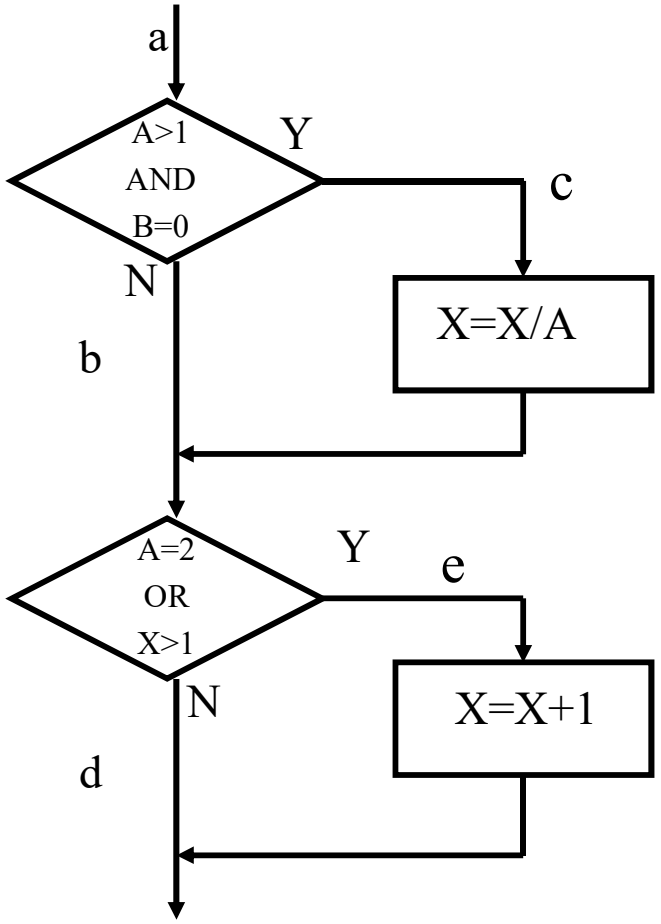
- 举例：对下列子程序进行测试

```
procedure example(y,z:real;var x:real);
begin
    if (y>1) and (z=0) then x:=x/y;
    if (y=2) or (x>1) then x:=x+1;
end;
```

- 测试用例：

A=2，B=0，X=3

程序流程图



2）判定覆盖（分支）

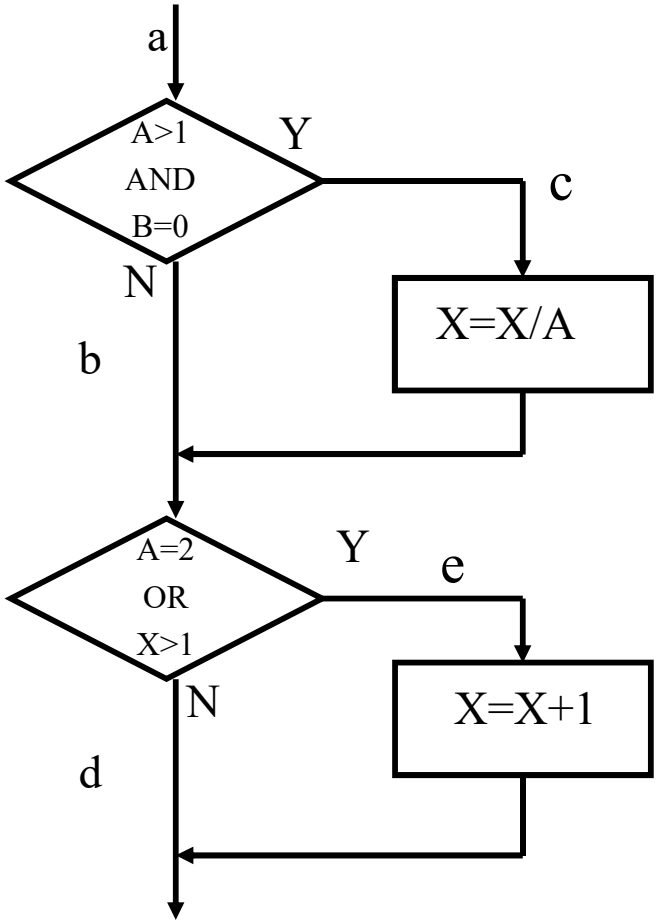
- 使得程序中每一个分支至少被遍历一次
- 测试用例

1. A=2, B=0, X=1

（沿路径 ace）

2. A=1, B=0, X=0

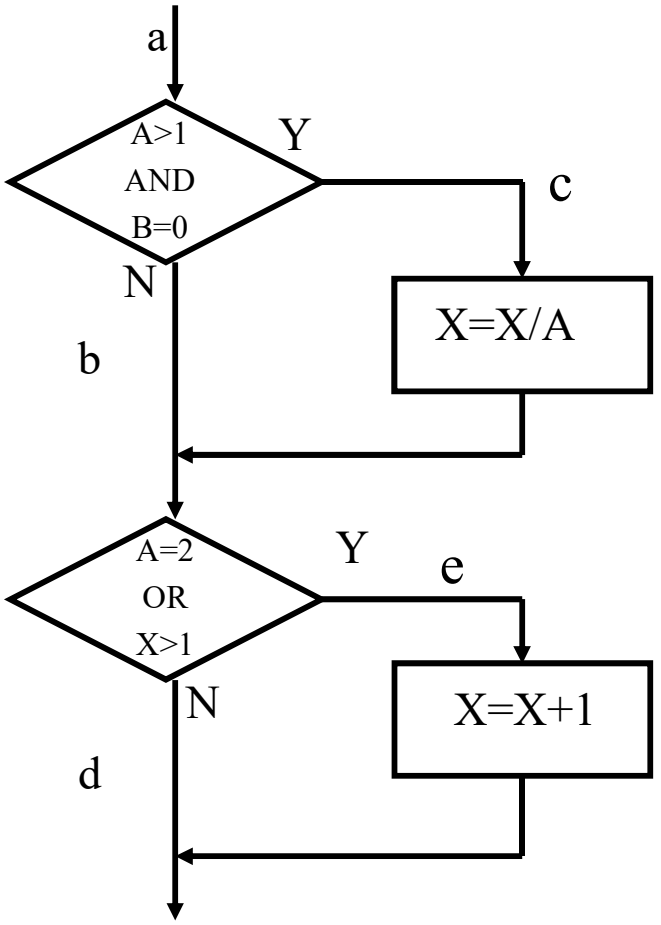
（沿路径abd）



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

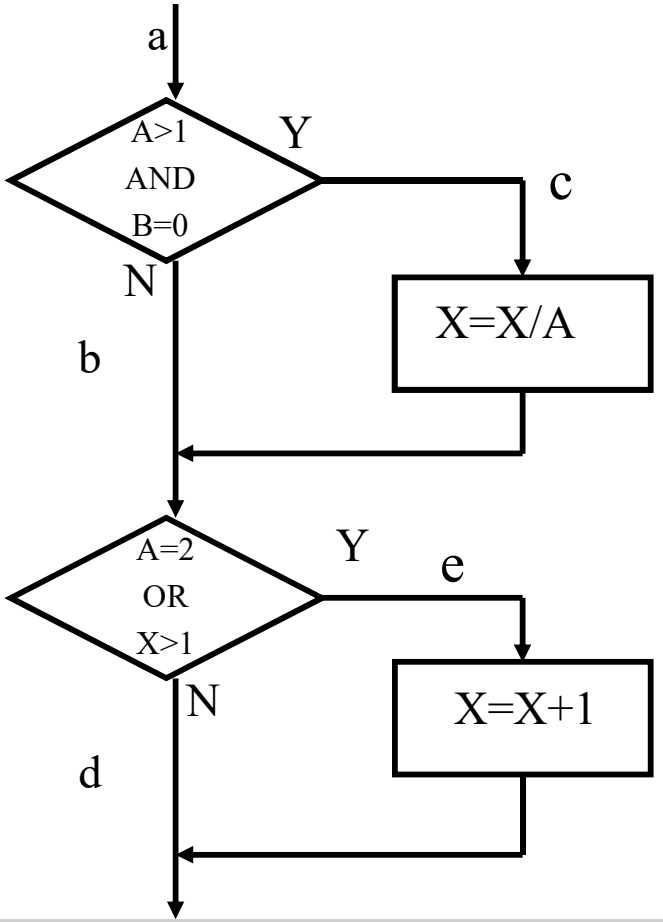
1. A=2, B=0, X=4
   (沿路径ace)
2. A=1, B=1, X=1
   (沿路径abd)



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）



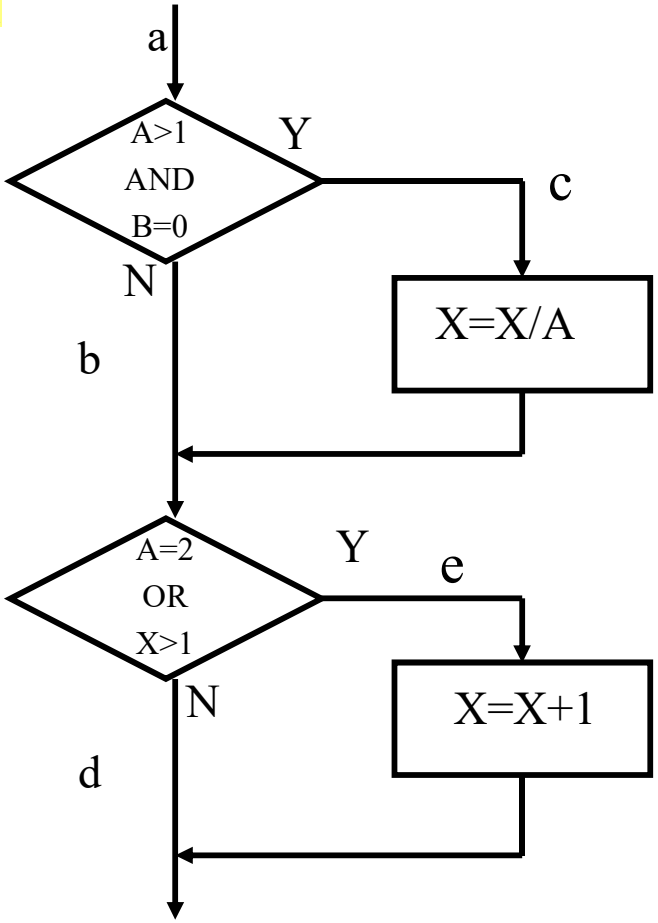
5）条件组合覆盖

- 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

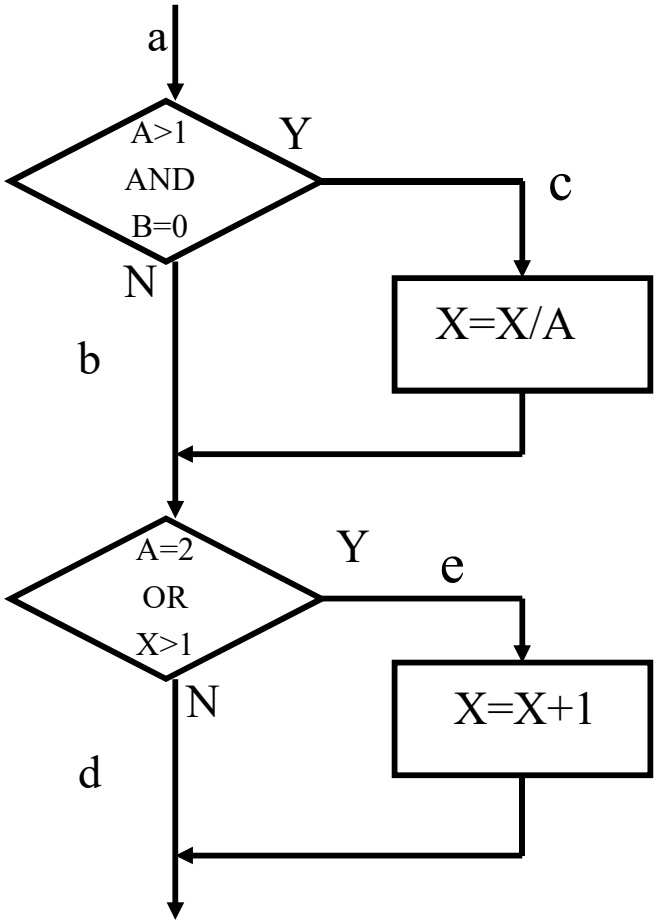


6）路径覆盖

- 覆盖程序中所有可能的路径
- 如果程序中包含环路，则要求每条环路至少经过一次

测试用例：

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



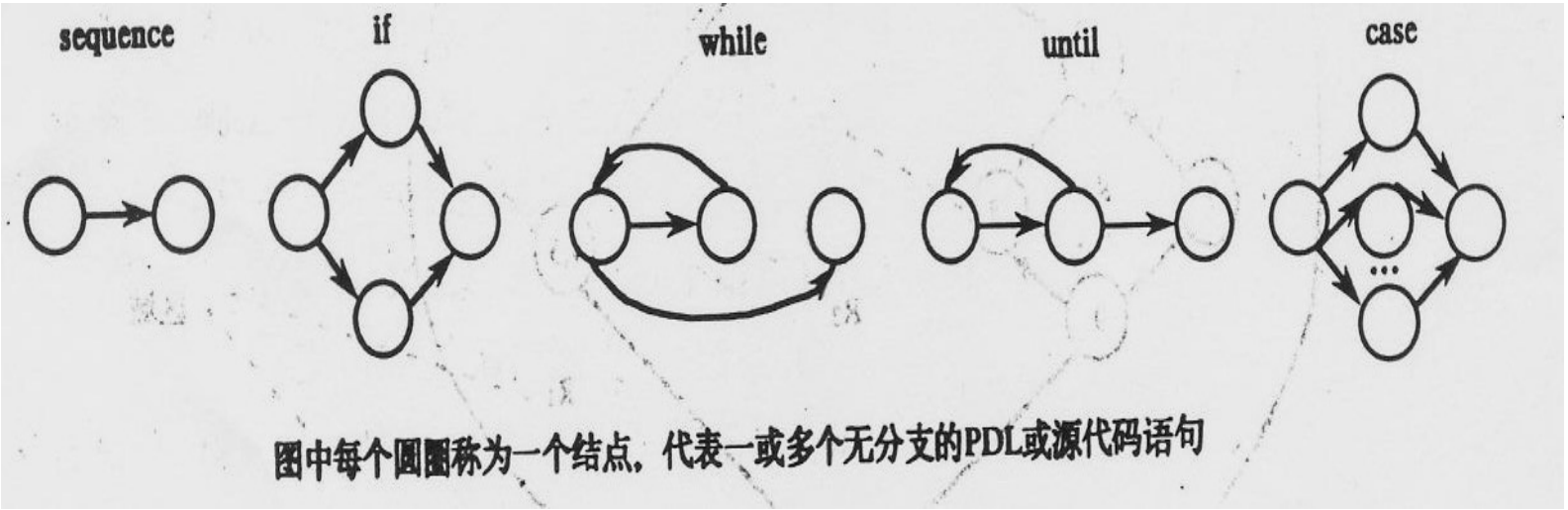
基本路径测试

- 在实际问题中，一个不太复杂的程序，特别是包含循环的程序，其路径数可能非常大。因此测试常常难以做到覆盖程序中的所有路径，为此，我们希望把测试的程序路径数压缩到一定的范围内。

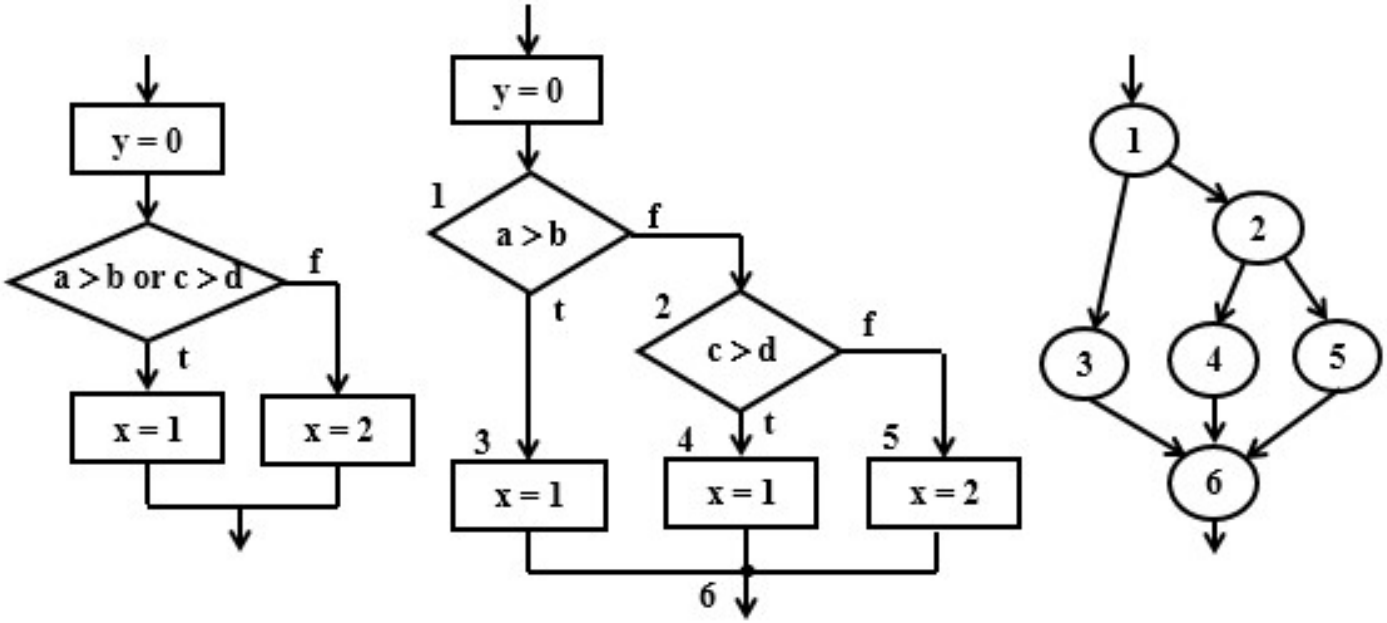
- 基本路径测试是Tom McCabe提出的一种白盒测试技术，这种方法首先根据程序或程序流程图画出控制流图（flow graph），并计算其区域数，然后确定一组独立的程序执行路径（称为基本路径），最后为每一条基本路径设计一个测试用例。

程序的控制流图

- 控制流图由结点和边组成，分别用圆和箭头表示。程序流程图中一个连续的处理框（对应于程序中的顺序语句）序列和一个判定框（对应于程序中的条件控制语句）映射成控制流图中的一个结点，程序流程图中的箭头（对应于程序中的控制转向）映射成控制流图中的一条边。对于程序流程图中多个箭头的交汇点可以映射成控制流图中的一个结点（空结点）。



- 上述映射的前提是程序流程图的判定中不包含复合条件。如果判定中包含了复合条件，那么必须先将其转换成等价的简单条件的程序流程图。

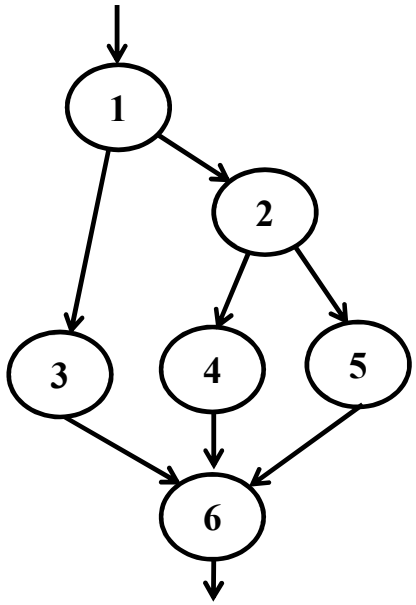


a）含复合条件的程序流程图

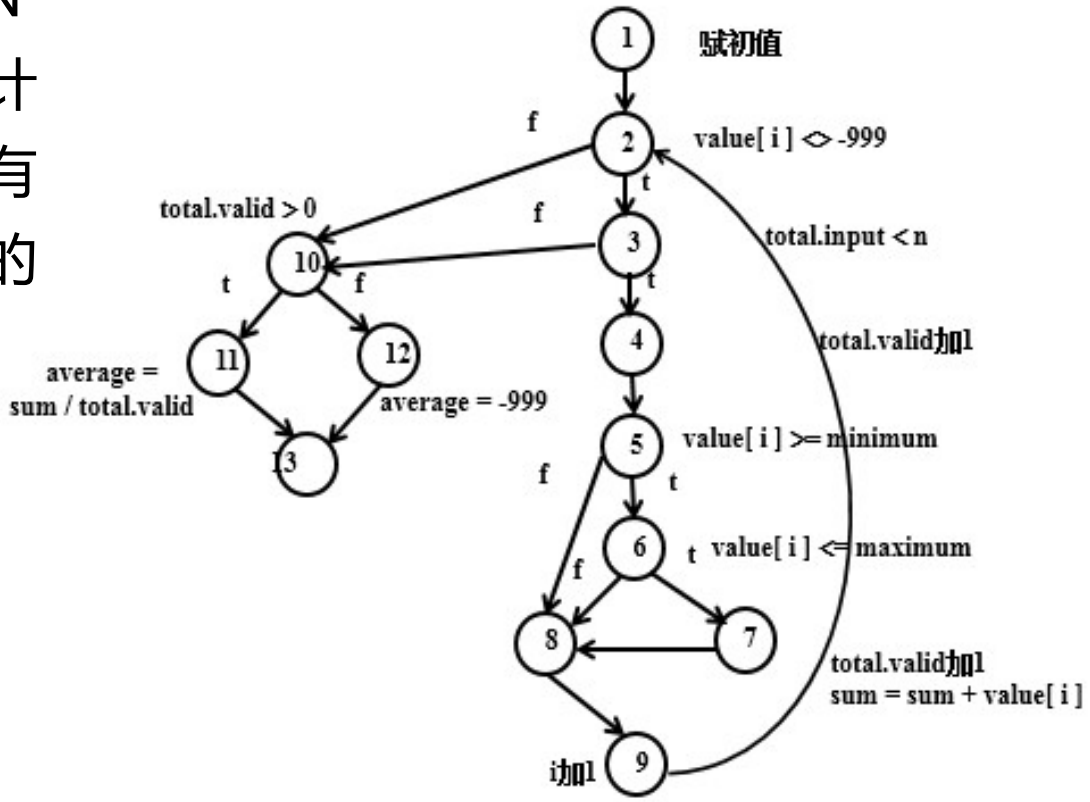
b）只含简单条件的程序流程图

c）对应的控制流图

- 我们把流图中由结点和边组成的闭合部分称为一个区域（region），在计算区域数时，图的外部部分也作为一个区域。例如，右图所示的流图的区域数为3。
- 独立路径是指程序中至少引进一个新的处理语句序列或一个新条件的任一路径，在流图中，独立路径至少包含一条在定义该路径之前未曾用到过的边。在基本路径测试时，独立路径的数目就是流图的区域数。



- 例如，对一个PDL程序进行基本路径测试，该程序的功能是：最多输入N个值（以-999为输入结束标志），计算位于给定范围内的那些值（称为有效输入值）的平均值，以及输入值的个数和有效值的个数。



其区域数为6，我们选取独立路径如下：

- 路径1：1-2-10-11-13
- 路径2：1-2-10-12-13
- 路径3：1-2-3-10-11-13
- 路径4：1-2-3-4-5-8-9-2-10-12-13
- 路径5：1-2-3-4-5-6-8-9-2-10-12-13
- 路径6：1-2-3-4-5-6-7-8-9-2-10-11-13

为每一条独立路径设计测试用例。

假设：n = 5；minimum = 0；maximum = 100。

路径1： 1-2-10-11-13

- 测试数据：value = [90，-999，0，0，0]
- 预期结果：Average = 90，total.input = 1，total.valid = 1

路径2： 1-2-10-12-13

- 测试数据：value = [-999 ，0，0，0，0]
- 预期结果：Average = -999，total.input = 0，total.valid = 0

路径3： 1-2-3-10-11-13

- 测试数据：value = [-1，90，70，-1，80]
- 预期结果：Average = 80，total.input = 5，total.valid = 3

路径4： 1-2-3-4-5-8-9-2-10-12-13

- 测试数据：value = [-1，-2，-3，-4，-999]
- 预期结果：Average = -999，total.input = 4，total.valid = 0

路径5： 1-2-3-4-5-6-8-9-2-10-12-13

- 测试数据：value = [120，110，101，-999，0]
- 预期结果：Average = -999，total.input = 3，total.valid = 0

路径6： 1-2-3-4-5-6-7-8-9-2-10-11-13

- 测试数据：value = [95，90，70，65，-999]
- 预期结果：Average = 80，total.input = 4，total.valid = 4

黑盒测试

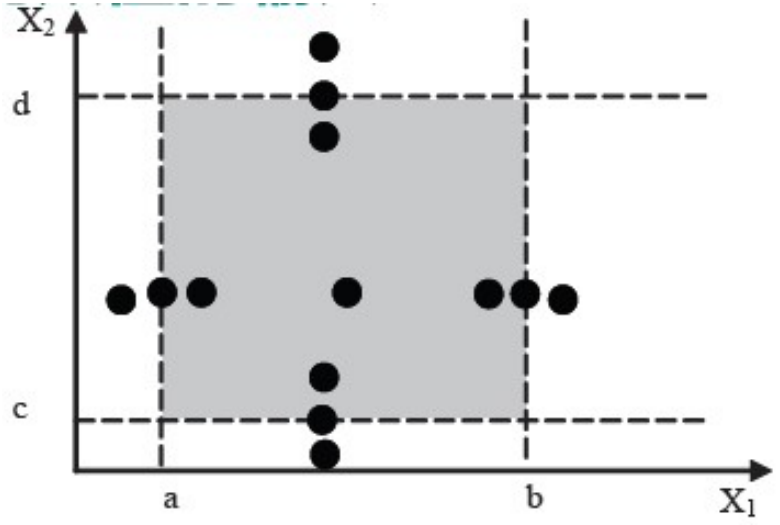
- 黑盒测试把程序看成一个黑盒子，完全不考虑程序内部结构和处理过程。
- 黑盒测试是在程序接口进行测试，它只是检查程序功能是否按照需求规约正常使用。
- 黑盒测试又称功能测试、行为测试，在软件开发后期执行。

几种黑盒测试技术

- 1）边界值划分法
- 2）等价类划分法
- 3）错误推测法
- 4）因果图法

1）边界值划分(boundary value analysis)

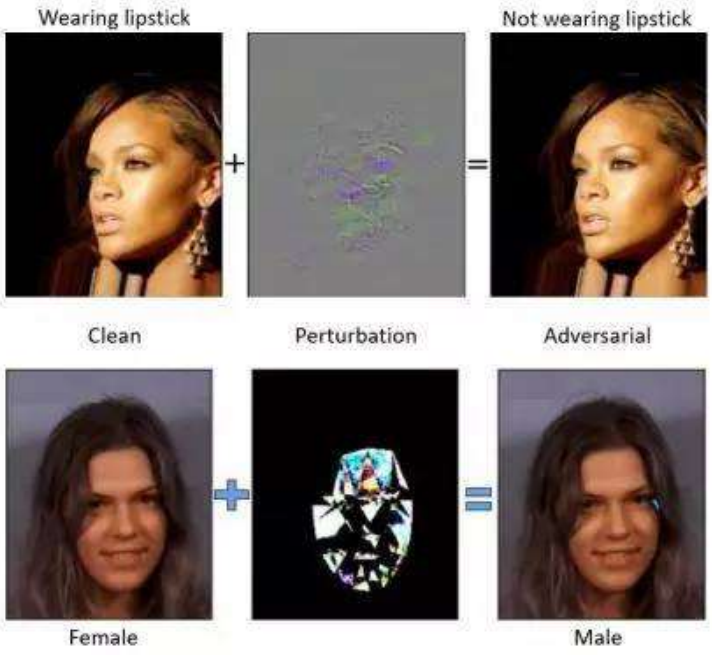
- 基本概念
  - 边界值划分法使被测程序在边界值及其附近运行，从而更有效地暴露程序中潜藏的错误
  - 不仅根据输入条件，它还根据输出情况设计测试用例

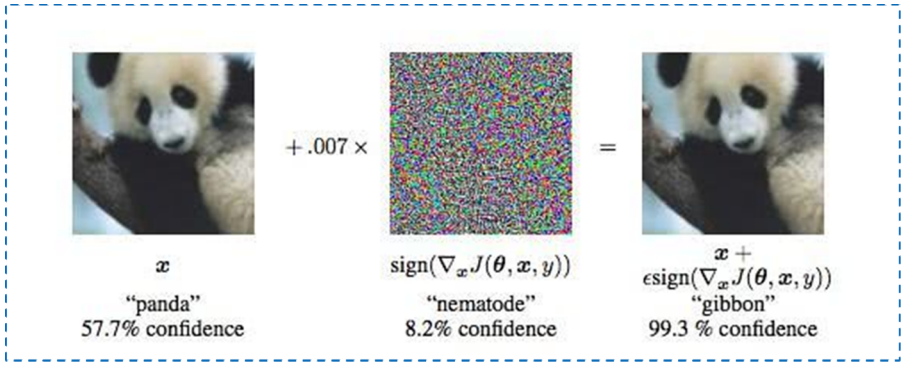


举例

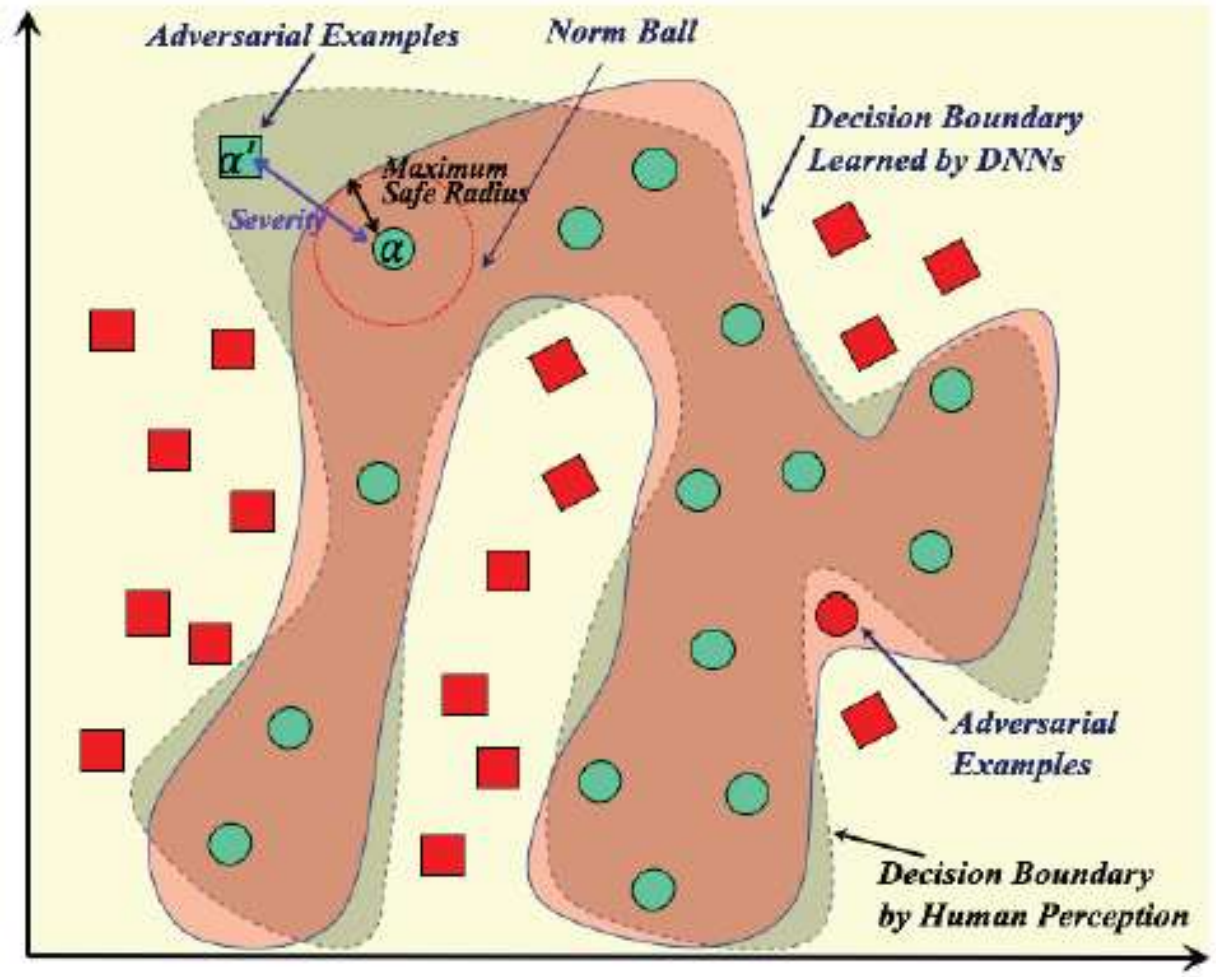
- 例1：输入条件-1.0到1.0
  - 选择-1.0，1.0，-1.001和1.001等
- 例2：重量在10公斤至50公斤范围内的邮件，其邮费计算公式为……
  - 应取10及50，还应取10.01, 49.99, 9.99, 50.01等
- 例3： 某输入文件可包含1至255个记录，……
  - 可取1和255，还应取0及256等
- 例4: 每月保险金扣除额为0至1165.25元
  - 可取0．00及1165.2、还可取 -0.01及1165.26等
- 例5: 情报检索系统要求每次“最多显示1～4条情报摘要”
  - 可取1和4，还应取0和5等。

举例：深度学习系统的对抗攻击/测试



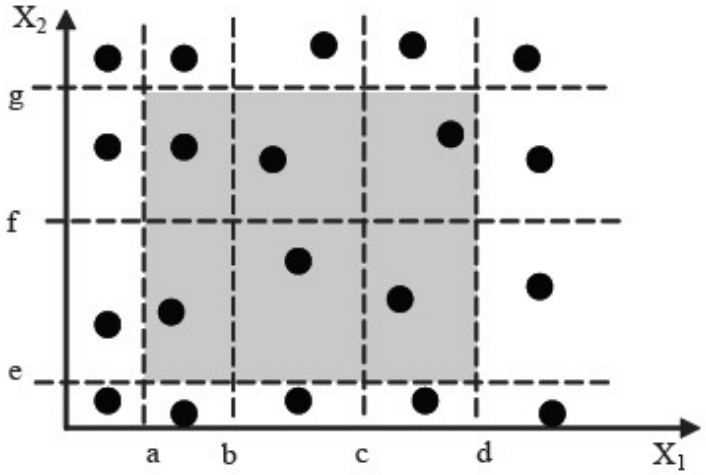


对抗样本


Adversarial Examples
Norm Ball
Decision Boundary
Learned by DNNs
α'
Maximum
Safe Radius
Severity
α
Adversarial
Examples
Decision Boundary
by Human Perception

2）等价类划分(equivalence partitioning)

- 基本概念
  - 试遍所有输入数据是不可能的
  - 等价类划分的办法是把程序的输入域划分成若干部分，然后从每个部分中选取少数代表性数据当作测试用例。
  - 输入的数据划分为合理等价类和不合理等价类



举例

- 例1：每个学生可以选取修1至3门课程
  - 有效等价类：选修1至3门课程
  - 无效等价类：没选修课程以及超过3门课程
- 例2：…项数可以从1到999…
  - 有效等价类为　　“1≤ 项数≤ 999”
  - 无效等价类为　　“项数<1”
    “项数>999”
- 例3：学生选课允许2门至4门
  - 有效等价类：　　选课2至4门
  - 无效等价类：　　只选一门课或未选课
    选课超过4门
- 例4：必须以标识符以字母开头的字符串
  - 有效等价类：　　以字母开头的字符串
  - 无效等价类：　　以非字母开头的字符串

3）错误推测法(error guessing)

- 基本概念
  - 猜测被测程序在哪些地方容易出错
  - 针对可能的薄弱环节来设计测试用例
- 例子1：对一个排序程序，可测试：
  - 输入表为空
  - 输入表中只有一行
  - 输入表中所有的值具有相同的值
  - 输入表已经是排序的
- 例子2：测试二分法检索子程序，可考虑：
  - 表中只有一个元素
  - 表长为2n
  - 表长为2n-1
  - 表长为2n+1

4）因果图法（Cause - Effect Graphics）

- When
  - 检查输入条件的各种组合情况
  - 在等价类划分方法和边界值方法中未考虑输入条件的各种组合，当输入条件比较多时，输入条件组合的数目会相当大
- How
  - 分析需求规约，找出因（输入条件）和果（输出或程序状态的修改）
  - 画出因果图
  - 通过因果图功能说明转换成一张判定表，然后为判定表的每一例设计测试用例

举例

- 例如，有一个处理单价为5角钱的饮料自动售货机软件，其需求规约如下：

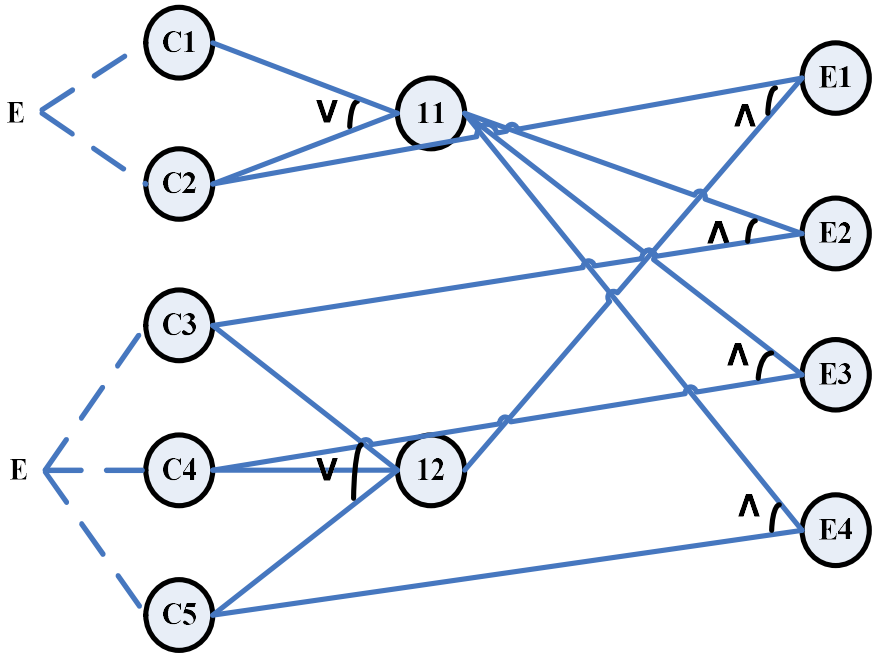
- 有一个处理单价为1元5角的盒装饮料的自动售货机软件。若投入1元5角硬币，按下“可乐”，“雪碧”或“红茶”按钮，相应的饮料就送出来。若投入的是两元硬币，在送出饮料的同时退还5角硬币。

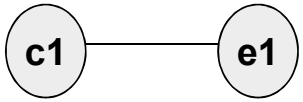
步骤一 分析需求规约，列出原因和结果

原因	c1:投入1元5角硬币； c2:投入2元硬币； c3:按“可乐”按钮； c4:按“雪碧”按钮； c5:按“红茶”按钮；
中间状态	11：已投币 12：已按钮
结果	E1:退还5角硬币； E2:送出“可乐”饮料； E3:送出“雪碧”饮料； E4:送出“红茶”饮料；

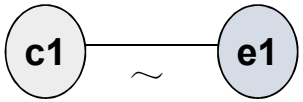
步骤二 画出因果图

所有原因结点列在左边，所有结果结点列在右边。
由于原因C1与C2，C3、C4与C5不能同时发生，分别加上互斥约束E。

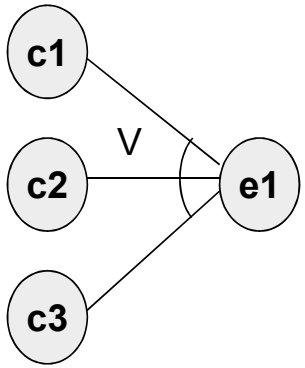




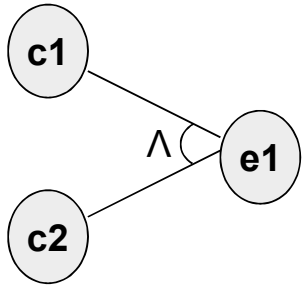
（a）恒等



（b）非



（c）或



（d）与

步骤三 根据因果图画出判定表

	1	2	3	4	5	6	7	8	9	10	11
c1:投入1元5角硬币	1	1	1	1	0	0	0	0	0	0	0
c2:投入2元硬币	0	0	0	0	1	1	1	1	0	0	0
c3:按“可乐”按钮	1	0	0	0	1	0	0	0	1	0	0
c4:按“雪碧”按钮	0	1	0	0	0	1	0	0	0	1	0
c5:按“红茶”按钮	0	0	1	0	0	0	1	0	0	0	1
11：已投币	1	1	1	1	1	1	1	1	0	0	0
12：已按钮	1	1	1	0	1	1	1	0	1	1	1
E1:退还5角硬币					√	√	√				
E2:送出“可乐”饮料	√				√						
E3:送出“雪碧”饮料		√				√					
E4:送出“红茶”饮料			√				√				

步骤四 根据判定表设计测试用例

为判定表的每个有意义的列设计一个测试用例

用例编号	测试用例	预期输出
1	投入1元5角，按“可乐”	送出“可乐”饮料
2	投入1元5角，按“雪碧”	送出“雪碧”饮料
3	投入1元5角，按“红茶”	送出“红茶”饮料
4	投入2元，按“可乐”	找5角，送出“可乐”
5	投入2元，按“雪碧”	找5角，送出“雪碧”
6	投入2元，按“红茶”	找5角，送出“红茶”

综合运用多种黑盒测试用例设计策略

- 不管情况怎样，都使用边值分析方法；
- 对输入和输出划分合格的和不合格的两个等价类，作为补充；
- 如果规范中含有输入条件的组合，便从因果图开始：

  原因：输入条件或输入条件的等价类；

  结果：输出条件或系统变换；

  因果图：连接原因与结果的布尔图；
- 使用“猜测”技巧，增加一些测试用例

讨论：三角形问题

- 从键盘上输入三个整数（最大值为255），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设计测试用例。

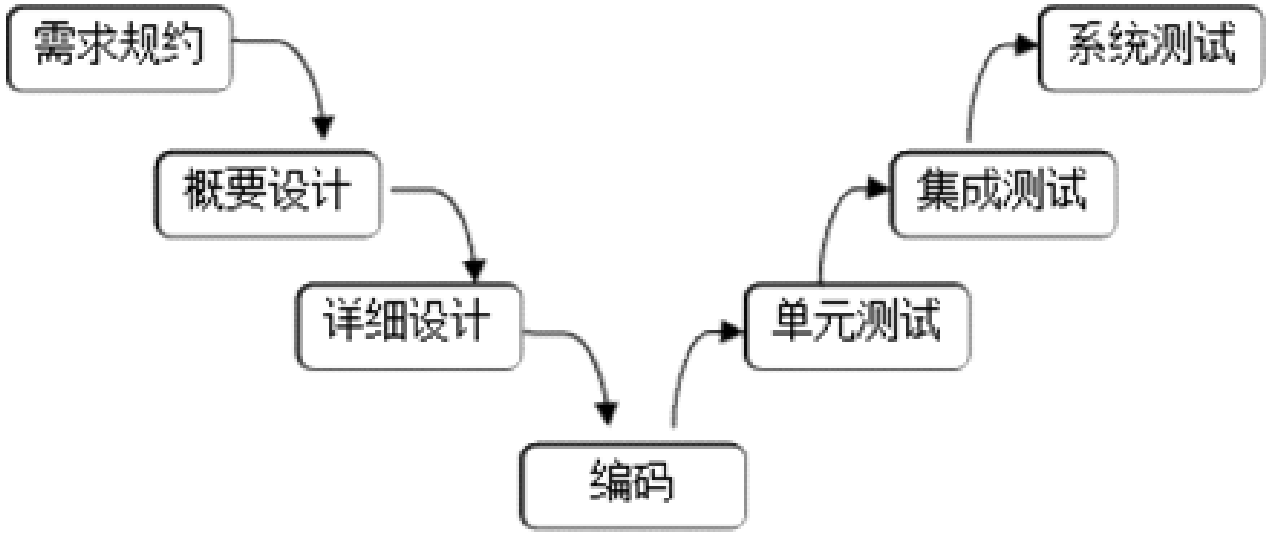
大 纲

测试策略

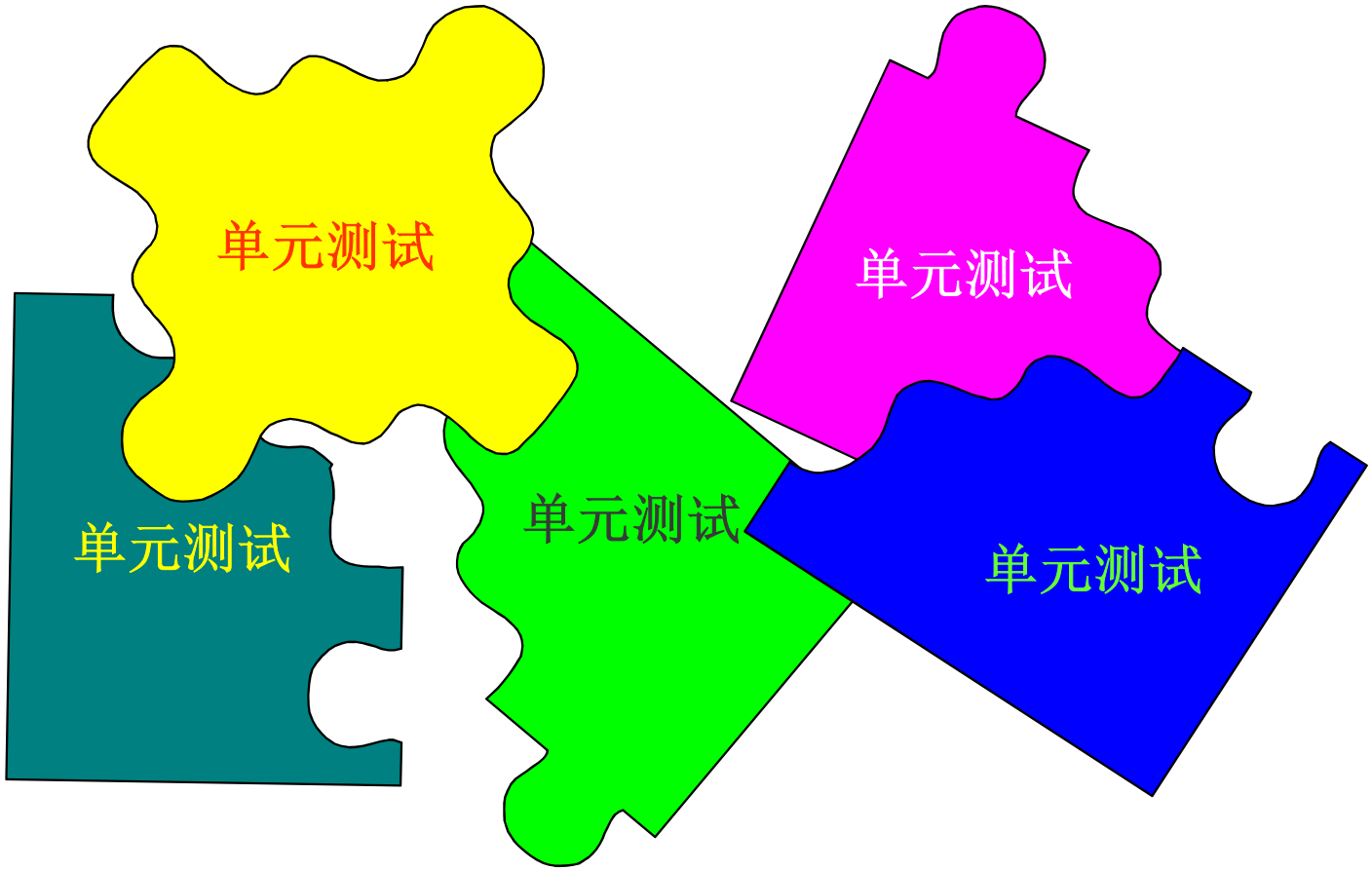
- 按测试层次分类
  - 单元测试、集成测试、系统测试
- 按软件质量属性分类
  - 功能性测试、可靠性测试、性能测试、易用性测试、可移植性测试、可维护性测试
- 其他测试策略
  - 验收测试、α测试、β测试、安装测试、回归测试、AB测试、众测

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)

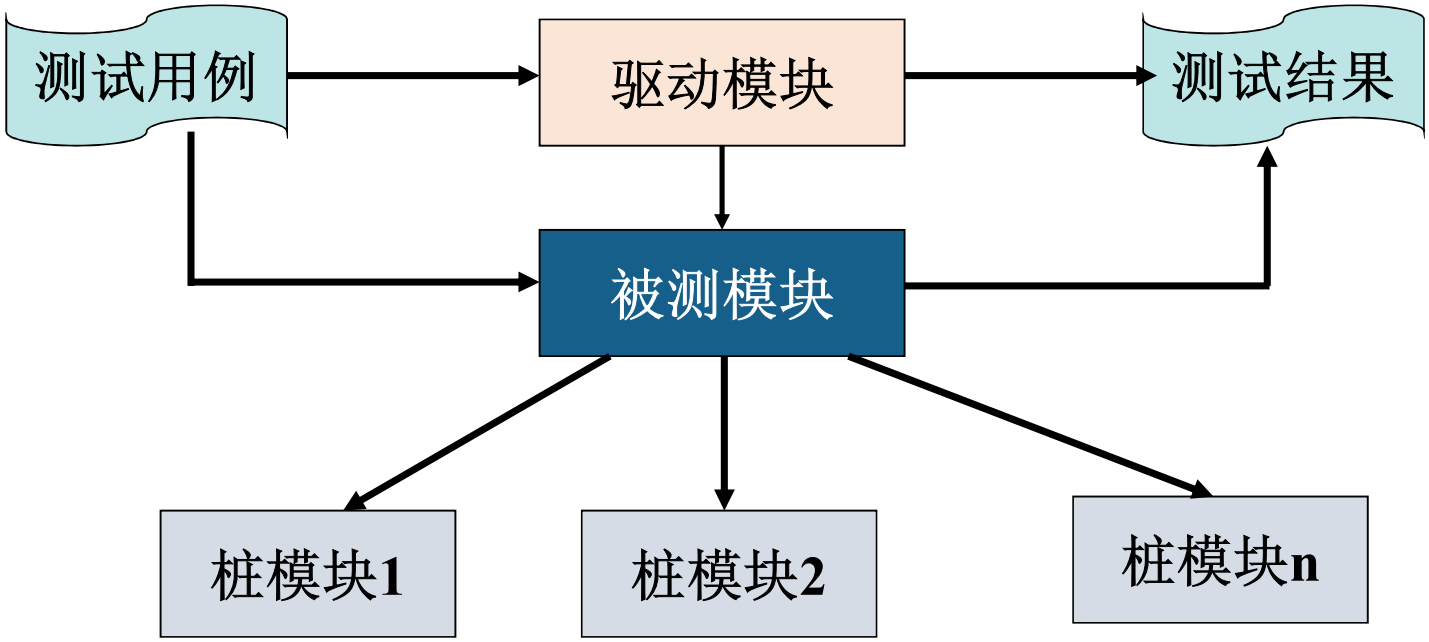


单元测试



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。



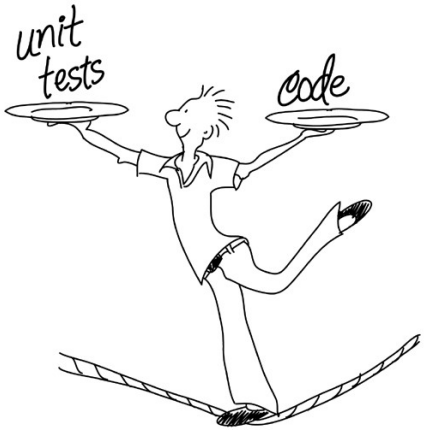
单元测试 - - 测什么?

- 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
    - 保证被测基本单元的信息能够正常地流入和流出
  - 局部数据结构
    - 确保临时存储的数据在算法的整个执行过程中能维持其完整性
  - 边界条件
    - 在到达边界值的极限或受限处理的情形下仍能正确执行
  - 独立的路径
    - 执行控制结构中的所有独立路径以确保基本单元中的所有语句至少执行一次
  - 错误处理路径
    - 对所有的错误处理路径进行测试

主要采用白盒测试技术

单元测试 - - 何时测试和由谁测?

- When
  - 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。
- Who
  - 单元测试可看作是编码工作的一部分，应该由程序员完成，也就是说，经过了单元测试的代码才是已完成的代码，提交产品代码时也要同时提交测试代码。
  - 测试小组可以对此作一定程度的审核。



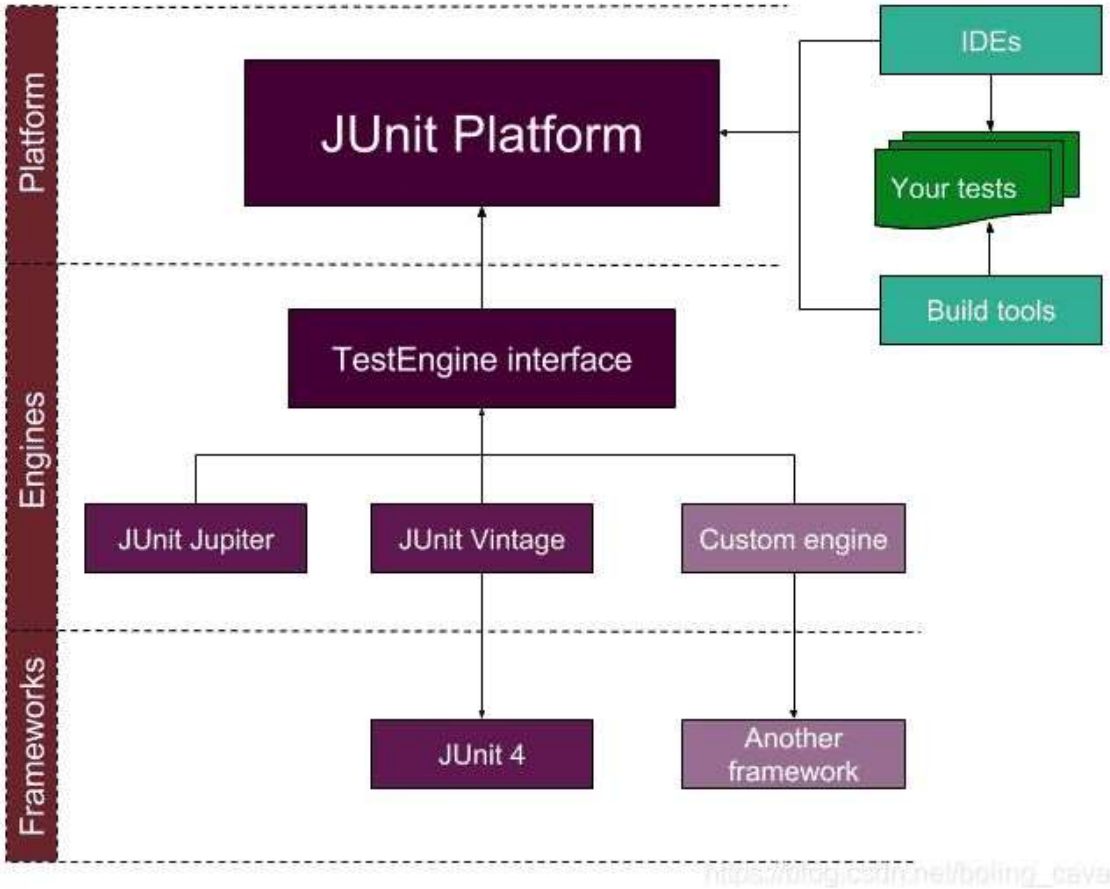
自动化的单元测试

- 只有用代码编写的UT，才能够重现，才能真正节约未来手工测试的时间。
- 只有用代码编写的UT ,才能做到自动化，才能在软件开发的任何时候都能快速，简单的大批量执行，保证能准确地定位错误，保证不会因为修改而引入新的错误。在系统开发的后期尤为明显。
- 目前最流行的单元测试工具是xUnit系列框架，常用的根据语言不同分为JUnit（java），CppUnit（C++），DUnit （Delphi ），NUnit（.net），PhpUnit（Php ）等等。

测试驱动的开发（TDD）

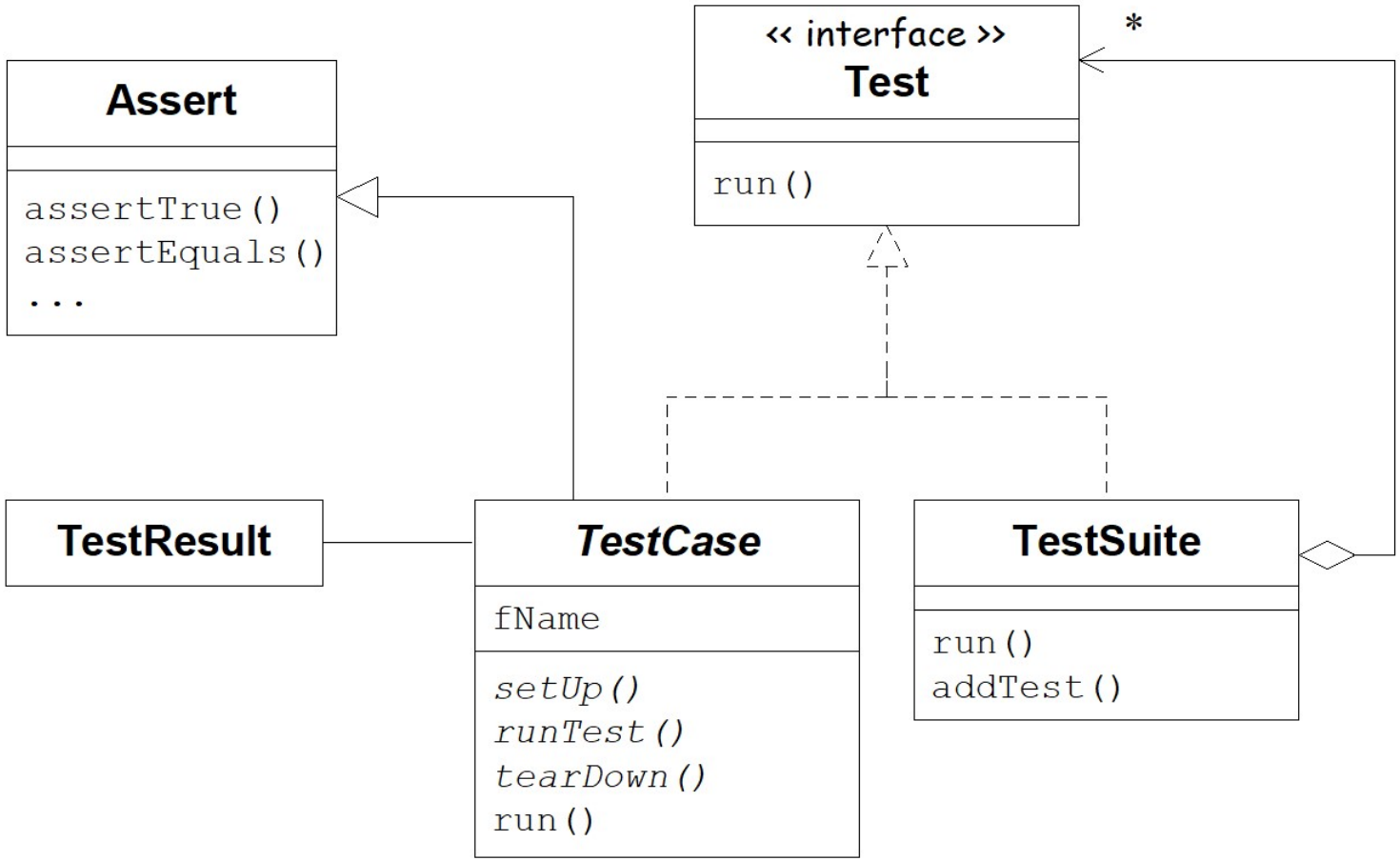
Junit 5 框架

https://junit.org/



- Platform：JVM上执行单元测试的基础平台，对接了各种IDE（例如IDEA、eclipse），并且还与引擎层对接，定义了引擎层对接的API；
- Jupiter：位于引擎层，支持Junit 5的编程模型和扩展模型；
- Vintage：位于引擎层，用于执行低版本的测试用例。

Junit Jupiter



核心类

- Test接口用来测试和收集测试的结果, 采用了 Composite设计模式,它是单独的测试用例,聚合的测试模式以及测试扩展的共同接口。
- TestCase抽象类用来定义测试中的固定方法, Testcase是Test接口的抽象实现，由于TestCase是一个抽象类,因此不能被实例化,只能被继承。其构造函数可以根据输入的测试名称来创建一个测试用例,提供测试名的目的在于方便测试失败时查找失败的测试用例。
- TestSuite是由几个TestCase或其他的TestSuite构成的。可以很容易构成一个树形测试,每个测试都由持有另外一些测试的TestSuite来构成。被加入到 Test Suite中的测测试在一个线程上依次被执行。
- Assert类用来验证实际结果和期望结果是否一致, 若不一致时就抛出异常。
- TestResult负责收集TestCase所执行的结果,它将结果分类,分为客户可预测错误和没有预测的错误,它还将测试结果转发到 TestListener处理。

JUnit 测试用例（Test Case）

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

每个@Test方法负责对某种情况测试，测试结果为true/false

JUnit assertXXX( )

Assert Method Summary	
Method	Description
assertEquals( )	进行等值比较
assertFalse( )	进行boolean值比较
assertTrue( )	进行boolean值比较
assertNull( )	比较对象是否为空
assertNotNull( )	比较对象是否不为空
assertSame( )	对2个对象应用的内存地址进行比较.
assertNotSame	对2个对象应用的内存地址进行比较
fail( )	引发当前测试失败，通常用于异常处理

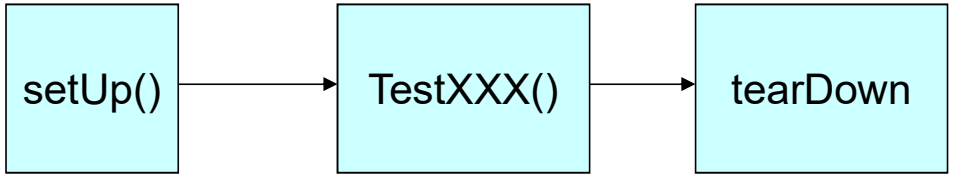
使用一系列的assertXXX方法来判断执行结果是否和预期相符，不符则执行失败，不会再继续case（当前方法）的余下部分

JUnit Annotation

Annotation	Description
@Test	Denotes that a method is a test method.
@ParameterizedTest	Denotes that a method is a parameterized test.
@RepeatedTest	Denotes that a method is a test template for a repeated test.
@TestFactory	Denotes that a method is a test factory for dynamic tests.
@BeforeEach	Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class.
@AfterEach	Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class.
@BeforeAll	Denotes that the annotated method should be executed before all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class.
@AfterAll	Denotes that the annotated method should be executed after all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class.

JUnit Set Up and Tear Down

- setUp 提供初始化方法
- tearDown提供清除方法
- 在执行每个测试方法TestXXX之前和之后都会调用



常见JUnit测试代码结构

```
mport static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }
}
```

JUnit Repeating Tests

```
@RepeatedTest(10)
void repeatedTest() {
// ...
}
```

如果想重复的运行某个测试若干次，可用此法来制造模拟数据或性能测试

JUnit 参数化测试

- 参数化测试允许开发人员使用不同的值反复运行同一个测试。

```java
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

```java
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

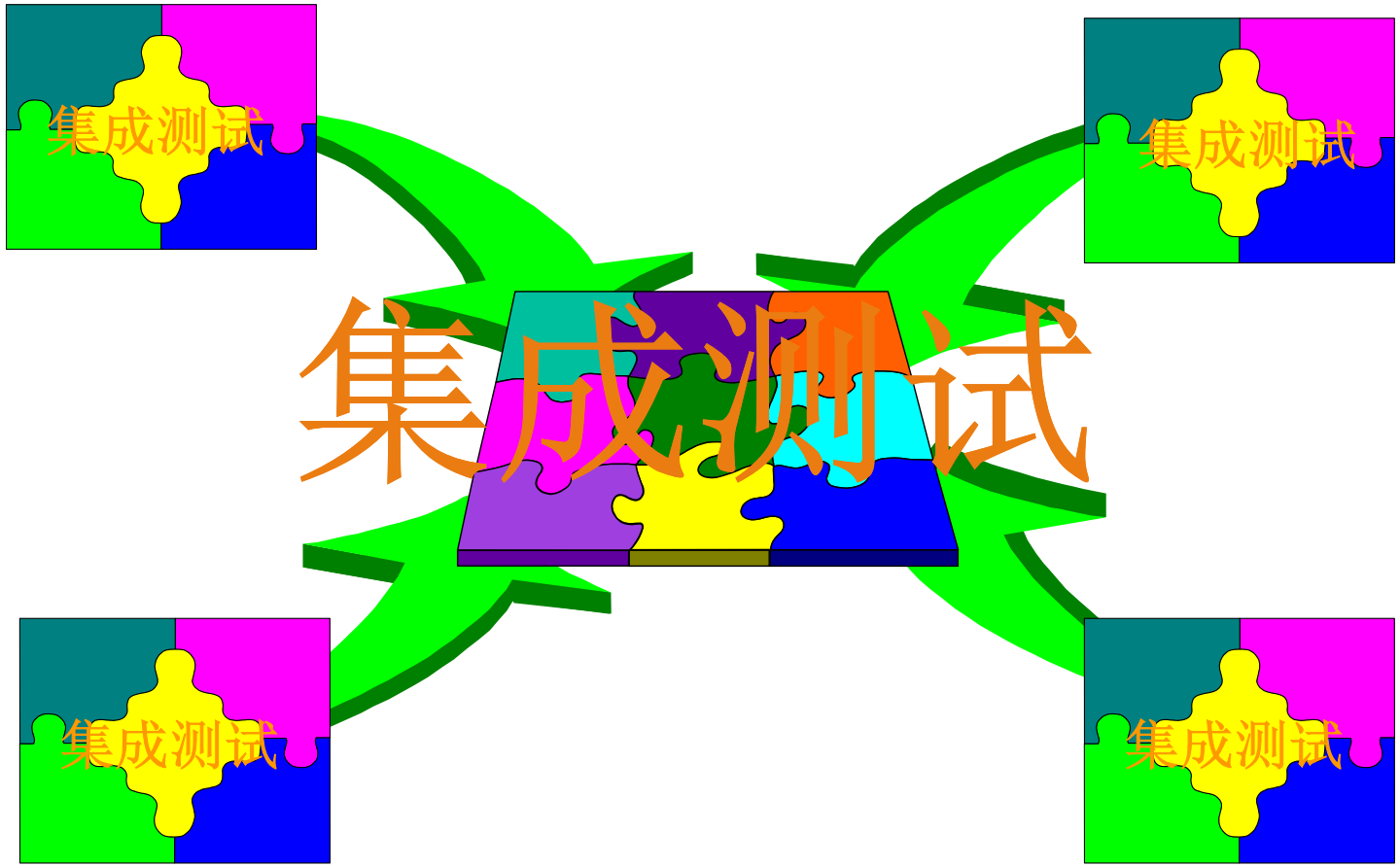
two-column.csv

```
Country, reference
Sweden, 1
Poland, 2
"United States of America", 3
```

单元测试经验

- 测试驱动开发
  - 编写单元测试用例促进解除模块之间的耦合。先编写测试用例，强迫自己从利于调用者的角度来设计单元，关注单元的接口。为了便于调用和独立测试，必须降低单元和周边环境的耦合程度，单元的可测试性得到加强，模块化程度得到提高。这样单元的可重用性也容易被考虑和提高。

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。

常采用白盒测试+黑盒测试技术

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

集成测试示例

- 自顶向下深度优先测试次序：

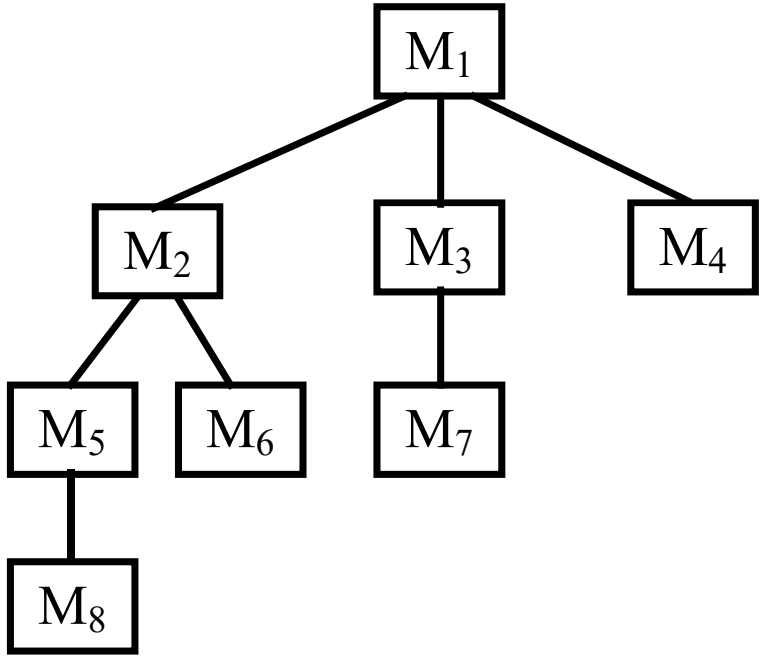
  M1、M2、M5、M8、M6、M3、M7、M4

- 自顶向下广度优先测试次序：

  M1、M2、M3、M4、M5、M6、M7、M8

- 自底向上集成测试次序：

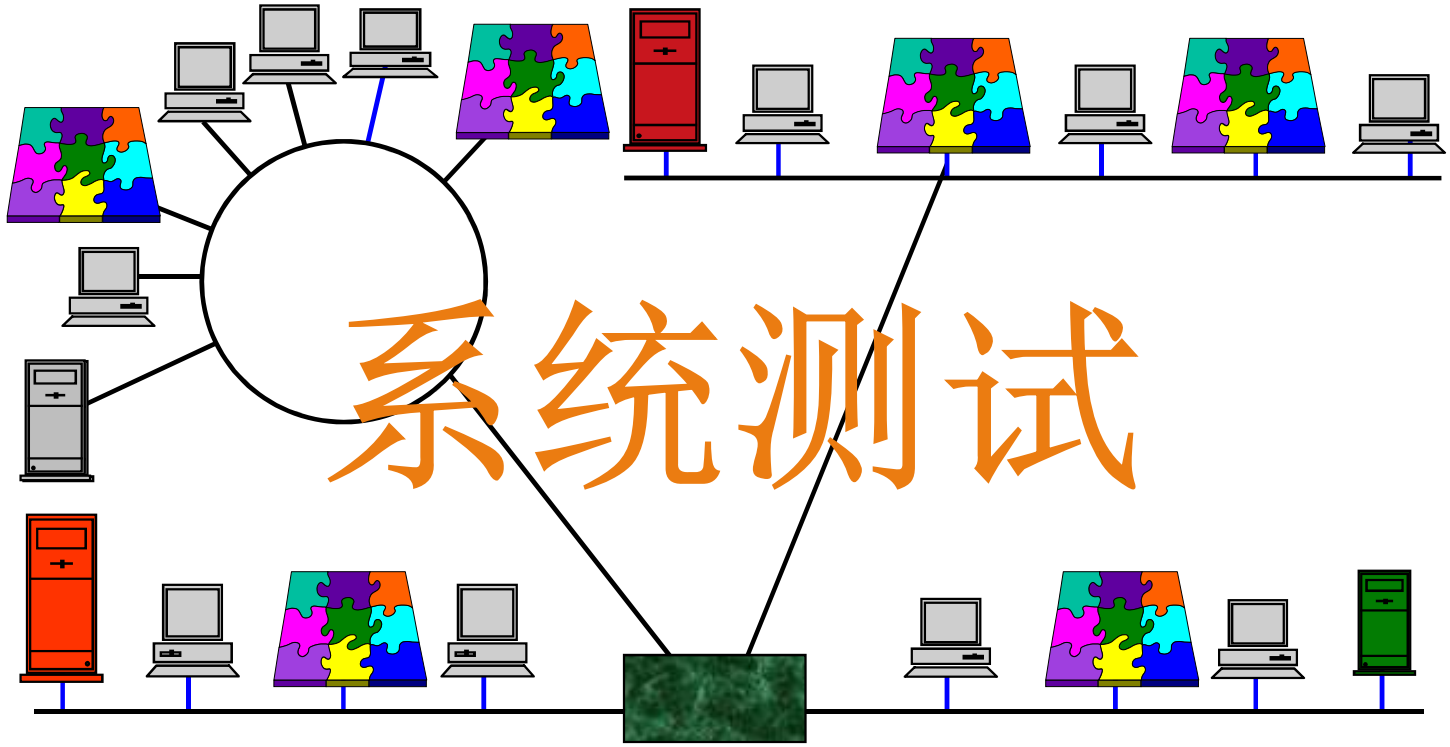
  M8、M5、M6、M7、M2、M3、M4、M1



集成测试 - - 何时测试和由谁测?

- When
  - 根据集成测试策略，和单元测试并行或之后进行。
- Who
  - 由程序员或软件测试工程师测试。
- Tool
  - postman、 SoapUI等API测试工具
  - Spring后端应用的集成测试工具SpringBootTest
  - React Native集成测试工具Cavy等

系统测试



系统测试 （system testing）

常采用黑盒测试技术

- When
  - 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。
- Why
  - 系统测试的目的是为了验证系统是否满足需求规约。
- What
  - 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。
- Who
  - 系统测试可以由程序员、软件测试工程师、第三方评测机构、客户等

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、性能测试、易用性测试、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、AB测试、众测

功能性测试

- 功能性测试（functionality testing），又称为正确性测试或一致性测试，其目的是用以确认软件在指定条件下使用时，软件产品提供满足明确和隐含要求的功能的能力。
  - 适用性测试，测试软件为指定的任务和用户目标提供一组合适的功能的能力。
  - 准确性测试，测试软件提供具有所需精度的正确或相符的结果或效果的能力。
  - 互操作测试，测试软件与一个或更多的规定系统进行交互的能力。
  - 安全性测试，测试软件保护信息和数据的能力，以使未授权的人员或系统不能阅读或修改这些信息和数据，而不拒绝授权人员或系统对它们的访问。
  - 功能依从性测试，测试规约中所有的功能都应实现，而且应是正确的。
- 采用人工测试或开源Selenium等测试工具。

- 例如，4S系统的系统测试时
  - 按其用例模型进行功能依从性测试，覆盖每个用例的各个基本流和备选流；
  - 在系统安全性测试时，测试者扮演一个试图攻击系统的角色，采用各种方式攻击系统：
    - 截获或破译4S系统的用户名和密码；借助于某种软件攻击4S系统；
    - “制服”4S系统，使得别人无法访问；故意引发系统错误，期望在系统恢复过程中侵入系统；
    - 通过浏览非保密的数据，从中找到进入4S系统的钥匙等等。

可靠性测试

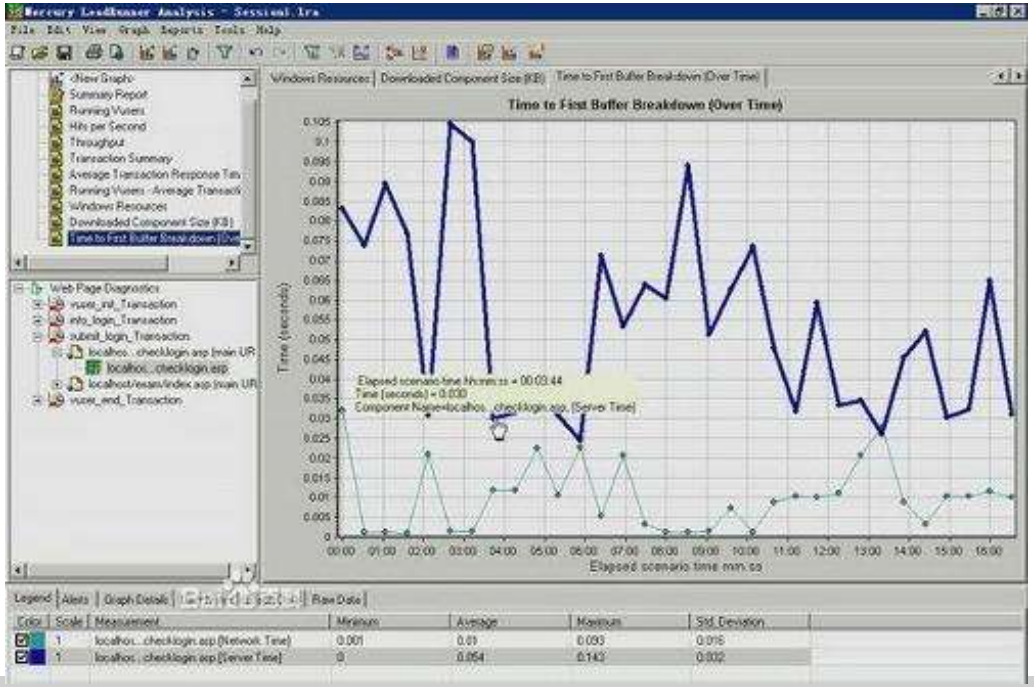
- 软件可靠性测试（reliability testing）用以测试在故障发生时，软件产品维持规定的绩效级别的能力。
  - 成熟性测试，测试软件为避免由软件中故障而导致失效的能力。
  - 容错性测试，测试在软件出现故障或者违反其指定接口的情况下，软件维持规定的性能级别的能力。
  - 易恢复性测试，测试在失效发生的情况下，软件重建规定的性能级别并恢复受直接影响的数据的能力。
  - 可靠性的依从性测试，测试软件遵循与可靠性相关的规约、标准或法规的能力。

- 例如，4S系统的系统测试时，
  - 连续运行4S系统100小时，计算系统正常运行时间占总时间的比率，以及故障恢复的平均用时；
  - 主动制造故障的方法来验证系统的容灾和恢复能力
    - 通过强制系统重启、拔网线，以及制造数据库读写失败和缓存读写失败等各种手段，让4S系统发生故障，然后观察系统是否还能降级运行，是否能及时地恢复运行，数据库中的数据是否因此留下了“脏”数据等。

性能测试

- 性能测试（performance testing）用来测试软件在规定条件下，相对于所用资源的数量，可提供适当性能的能力。
  - 时间特性测试：测试在规定条件下，软件执行其功能时，提供适当的响应和处理时间以及吞吐率的能力。
  - 资源利用性测试：测试在规定条件下，软件执行其功能时，使用合适数量和类别的资源的能力。这些资源包括CPU、内存、网络等。
  - 性能依从性测试：测试软件遵循性能相关的规约、标准或法规的能力。
- 压力测试（stress testing），又称强度测试，是一种超常情况下的性能测试。它需要在超常数量、频率或资源的方式下执行系统，以获得系统对非正常情况下（如大数据量的输入、处理和输出，大并发数等）的承受程度。
- 自动化的性能/压力测试工具：HP的Loadrunner，开源的Jmeter等。

- 例如，4S系统的系统测试时，
  - 采用Loadrunner进行自动化性能测试，模拟不同数量的并发用户，在不同的数据量情况下，观察系统的响应时间和资源使用情况。



易用性测试

- 易用性测试（usability testing）用以评价用户学习和使用软件（包括用户文档）的难易程度、支持用户任务的有效程度、从用户的错误中恢复的能力。
  - 易理解性测试，测试软件使用户能理解软件是否合适以及如何能将软件用于特定的任务和使用条件的能力。
  - 易学性测试，测试软件使用户能学习其应用的能力。
  - 易操作性测试，测试软件使用户能操作和控制它的能力。
  - 吸引性测试，测试软件吸引用户的能力。
  - 易用依从性测试，测试软件遵循易用相关的规约、标准、风格指南或法规的能力。
- 易用性测试可以采用模拟用户的方式进行，也可以通过观察用户的操作行为来执行。

- 例如，4S系统的易用性测试时，
  - 测量用户所需的培训时间
  - 观察销售人员完成下订单的任务平均需要多少步骤
  - 测试各界面的风格是否一致
  - 信息显示和反馈是否准确
  - 是否提供在线的支持帮助等等。

可移植性测试

- 可移植性测试（portability testing）用以测试软件从一种环境迁移到另外一种环境的能力。
  - 适应性测试，测试软件毋需采用额外的活动或手段就可适应不同指定环境的能力。
  - 易安装性测试，测试软件在指定环境中被安装的能力。
  - 共存性测试，测试软件在公共环境中同与其分享公共资源的其他独立软件共存的能力。
  - 易替换性测试，测试软件在同样环境下，替代另一个相同用途的指定软件产品的能力。
  - 可移植性的依从性测试，测试软件遵循可移植性相关的规约、标准或法规的能力。

- 例如，4S系统的可移植性测试时，
  - 在支持的MS Window和Linux两种操作系统的规定版本上，可以方便地进行4S系统的安装。
  - 用IE、Firefox等多个版本的浏览器都能正常访问。
  - 能和已安装的MS Office等软件共存。
  - 同时检查4S系统提供的安装文档和初始化数据。

可维护性测试

- 可维护性测试（maintainability testing）用以测试软件可被修改的能力，包括纠正、改进或软件对环境、需求变化的适应。
  - 易分析性测试，测试软件诊断缺陷或失效原因或识别待修改部分的能力。
  - 易改变性测试，测试软件使指定的修改可以被实现的能力。
  - 稳定性测试，测试软件避免由于软件修改而造成意外结果的能力。
  - 易测试性测试，测试软件使已修改软件能被确认的能力。
  - 维护依从性测试，测试软件遵循可维护性相关的规约、标准或法规的能力。

- 一般，软件可维护性更多的是通过度量来进行评估。
  - 常用可维护性度量：软件的复杂性、规模、可重用性、耦合度和内聚度、注解代码率、缺陷的平均修复成本、变更的平均实现成本等。
  - 软件越复杂、规模越大、可重用性越低、耦合度越大、内聚度越小、注解代码行数所占比率越小、缺陷的平均修复成本或变更的平均实现成本越高，则软件可维护性越差。

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、性能测试、易用性测试、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、AB测试、众测**

验收测试

- 针对应用型软件
- 所谓验收测试（acceptance testing），是确定一个开发完成的软件系统是否符合其验收准则，使用户/客户或其他授权实体能确定是否接受此软件系统的正式测试。
- 验收测试是以用户为主的测试。验收测试原则上在用户所在地进行，但如经用户同意也可以在开发机构内模拟用户环境下进行。
- 项目经理负责组织验收组进行最终验收测试。验收组应由用户代表、相关专家、项目组成员等组成。验收组根据合同、《需求规约》或《验收测试计划》对软件成品进行验收测试。

α测试和β测试

- 针对通用产品
- α测试是邀请小规模的、有代表性的潜在用户，在开发环境中，由开发者“指导”下进行的测试（试用），开发者负责记录使用中出现的问题和软件的缺陷，因此α测试是在一个受控的环境中进行的。
- β测试是由用户在一个或多个用户环境下进行的测试，是产品正式发布前的系统测试形式。一组有代表性的用户和消费者在典型操作条件下尝试做常规使用，由用户记录下测试中发现的问题或任何希望改进的建议，报告给开发者。
- β测试与α测试不同的是，β测试时开发者通常不在测试现场，由用户去使用，软件在一个开发者不能控制的环境中的“活的”试用。

安装测试

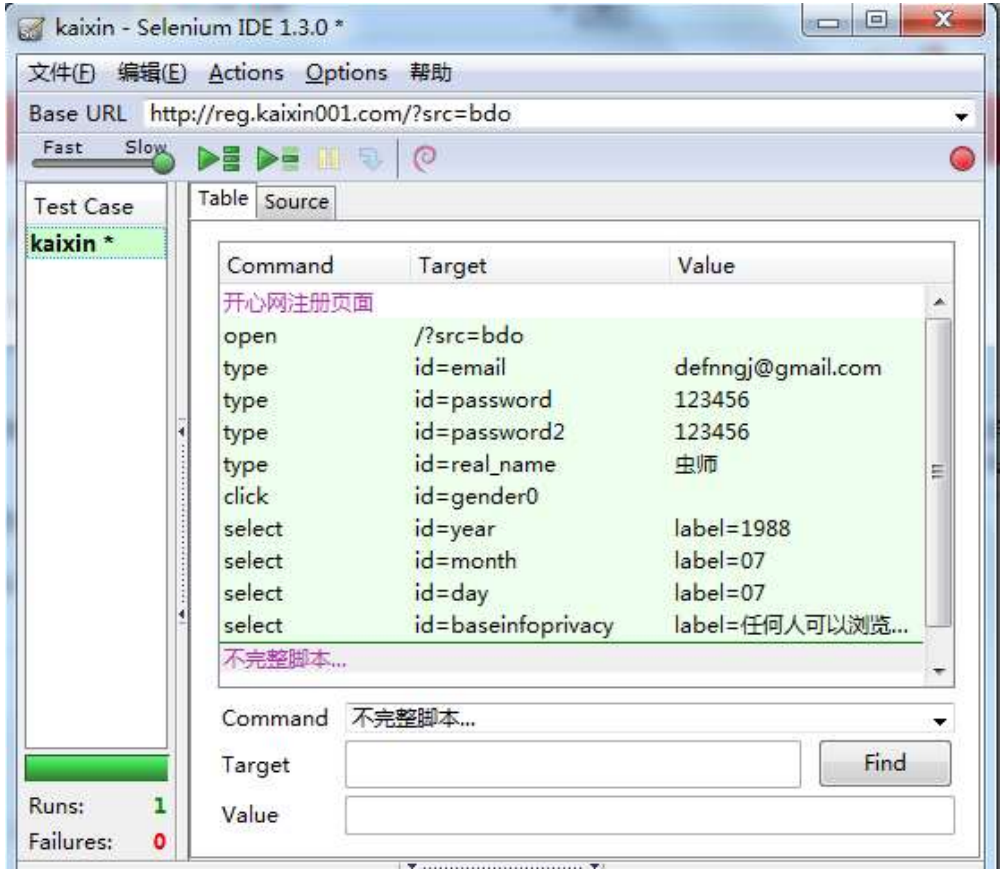
- 安装测试（installation testing）用以确保该软件在各种支持的平台下，各种允许的安装情况下，都能成功安装。
  - *首次安装、升级、完整的或自定义的安装等*
- 除了安装软件，常常还需要安装用户文档以及初始化数据，这些同样需要进行检查。

回归测试（regression testing）

- 为了保证软件返工时没有引进新的错误，要全部或部分地重复以前做过的测试。
- 在集成测试、缺陷纠正后的重新测试、迭代开发的后续迭代测试中常常用到回归测试。
- 回归测试可以手工执行，也可以使用自动化的回归测试工具（又称功能测试工具）。回归测试工具使得软件工程师能够捕获到执行过的测试，然后进行回放和比较。
- 回归测试应重新执行所有执行过的测试，或者对受影响的软件部分进行局部回归测试。仅仅对修改的软件部分进行重新测试常常不够的。

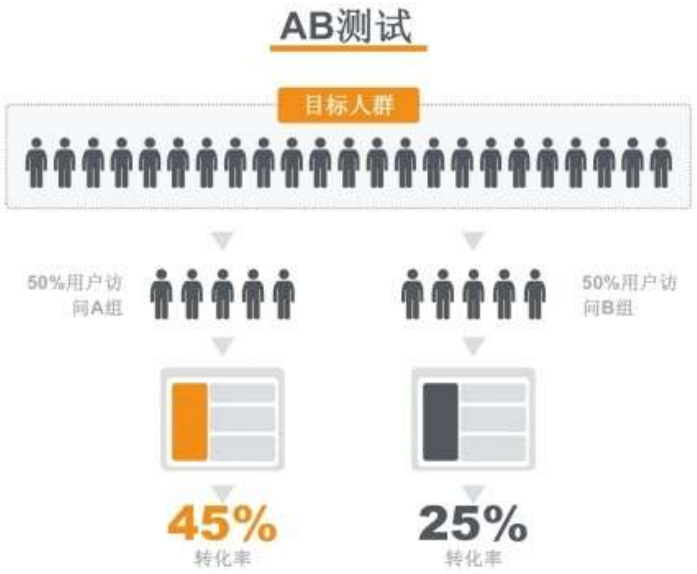
自动化的回归测试

- HP的QTP
- 开源的selenium



AB测试

- 为Web系统或App软件制作两个（A/B）或多个（A/B/n）版本，在同一时间维度，分别让组成成分相同（相似）的访客群组（目标人群）随机的访问这些版本，收集各群组的用户体验数据和业务数据，最后分析、评估出最好版本，正式采用。

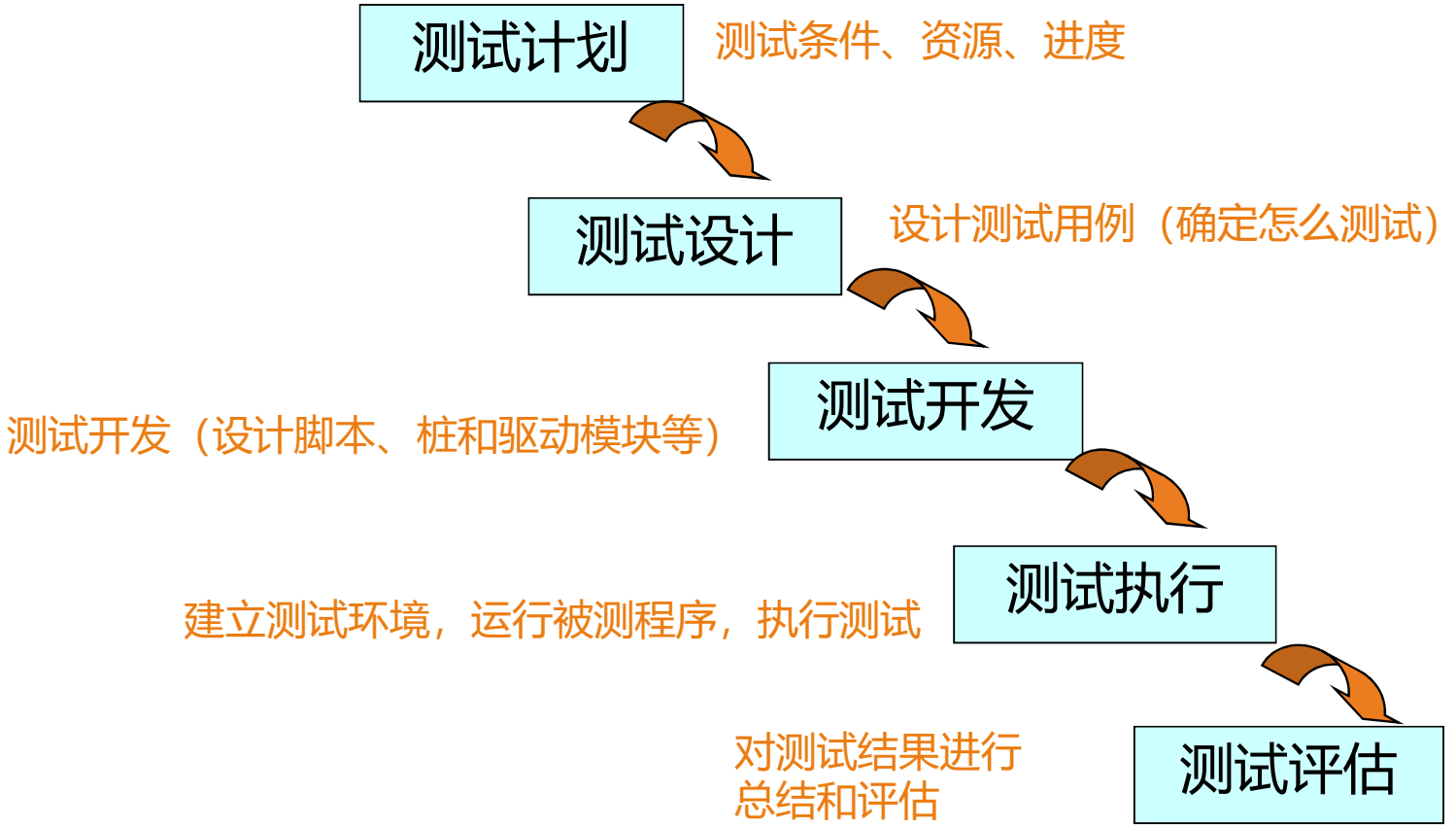


众测

- 即众包测试，利用大众的测试能力和测试资源，在短时间内完成大工作量的产品体验，并能够保证质量，第一时间将体验结果反馈上来，这样开发人员就能从用户角度出发，改善产品质量。

大 纲

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

1）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、结束标准
- 测试文档和测试记录

开始测试的标准

- 在测试计划中规定开始测试的标准
- 开始测试的常用标准
  - 通过冒烟测试

2）测试设计

- 任务：设计测试用例
- 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：
  - 前置条件（Pre-conditions）Optional
  - 测试输入（Test input)
  - 观察点　(Observation Points) Optional
  - 控制点　(Control Points) Optional
  - 期望结果（Expected Results)
  - 后置条件（Post-conditions) Optional

示例：测试用例/测试记录表

××××公司×××项目测试用例及测试实施记录 记录编号：YDDZ2000-SYDAS2000TP02
项目编号：SYDAS2000 项目名称：XXXX配网自动化系统测试规范版本号：2.0
开发负责人：xx 开发部门：配网 测试规范最新更新时间：2001-03-01

编号	标题	步骤	期望结果	现象及记录	结果	开发人员反馈
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。	添加用户			

3）测试开发

- 任务：开发测试脚本、桩和驱动模块
- 测试脚本（test script）是具有正规语法的数据和指令的集合，在测试执行自动工具使用中，通常以文件形式保存

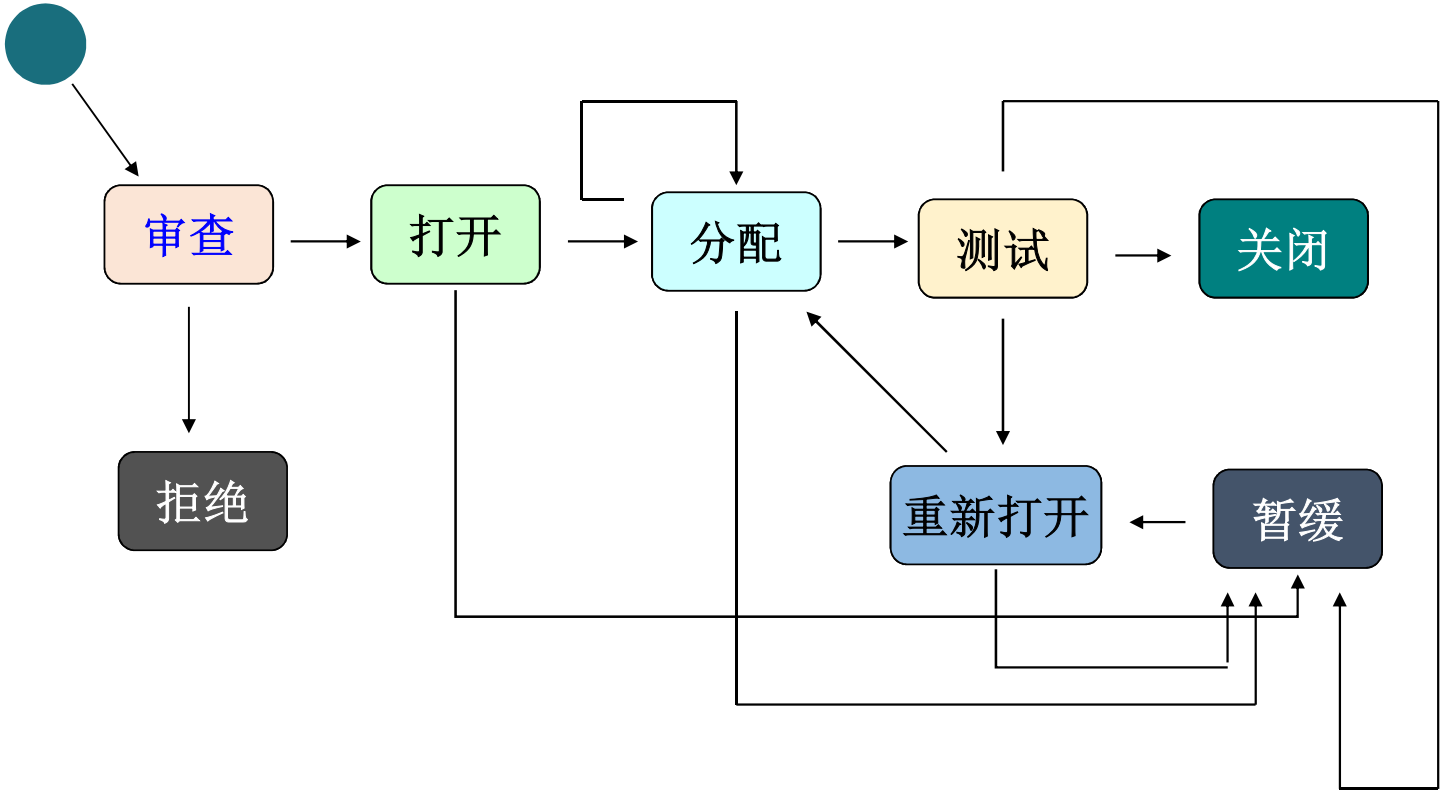
4）测试执行

- 任务：执行测试用例
- 对于手动测试：
  - 测试者按事先准备好的手工过程进行测试，测试者输入数据、观察输出、记录发现的问题。
- 对于自动测试：
  - 可能只需要启动测试工具，并告诉工具执行哪些测试用例。
- 文档：测试记录、缺陷报告

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

缺陷跟踪系统



开源缺陷跟踪系统：bugzilla和Mantis等

5）测试评估

- 通过评估测试的步骤是否按计划进行，以发现是否存在测试中的随意性，以及分析没有按照测试计划执行的原因。
- 通过评估测试的覆盖率情况、测试用例的通过率、测试的结果与测试的目标一致性，来评估测试的有效性，确定是否需要补充测试和复测或回归测试。
- 通过分析软件缺陷的严重性和缺陷的分布情况，向委托客户提供咨询意见或建议。

终止测试的标准

- 在测试计划中规定终止测试的标准
- 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

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

测试度量

- 如果不做正式的、定量的度量，就只能对测试过程有效性作出定性的描述
- 原始测试数据
  - 缺陷的数量
  - 软件的规模（KLOC、功能点）
  - 测试成本、测试工作量
  - 测试时间

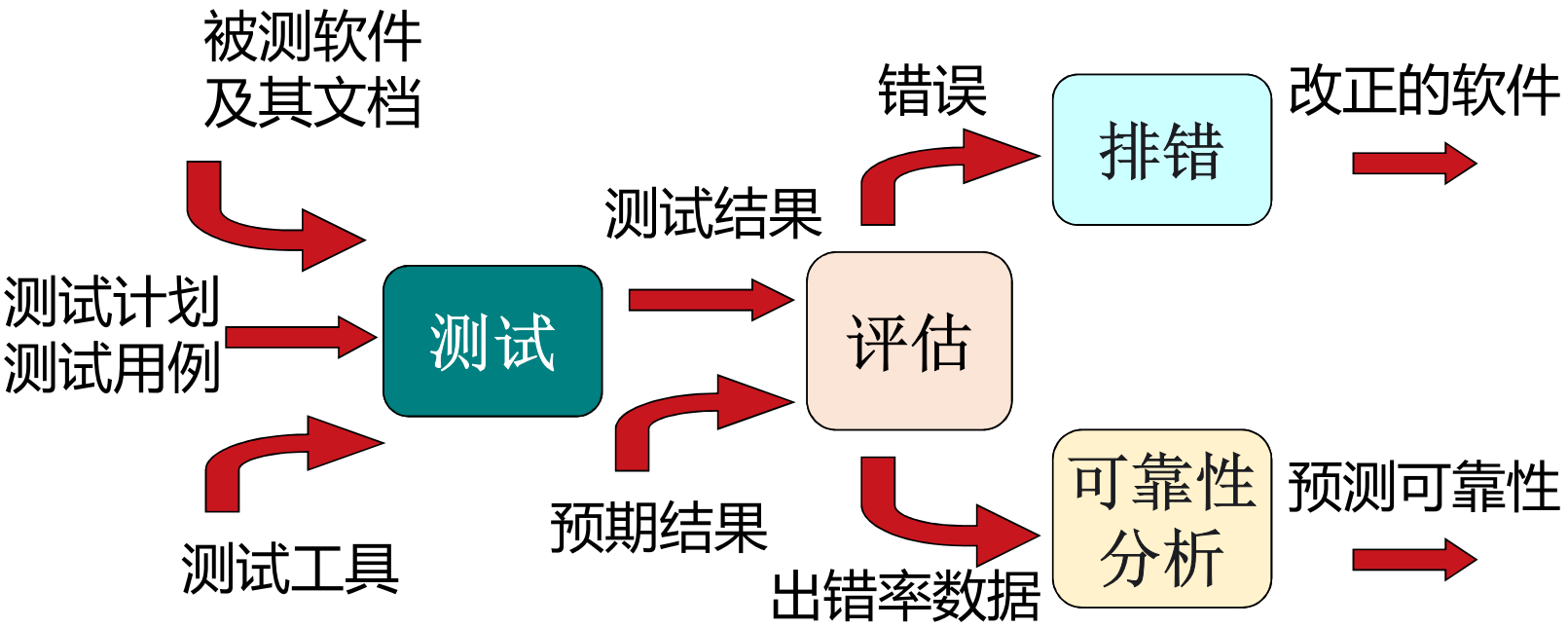
主要测试度量指标

- 测试中发现的全部缺陷数
- 客户发现的全部缺陷数
- 缺陷检测有效性
  - 测试中发现的全部缺陷数 / （测试中发现的全部缺陷数+客户发现的全部缺陷数）
- 缺陷排除有效性
  - 测试中改正的全部缺陷数 / 测试中发现的全部缺陷数
- 测试用例设计效率
  - 测试中发现的全部缺陷数 / 运行的测试用例数
- 缺陷密度 = 缺陷数 / 软件规模
- 覆盖率
  - 测试需求覆盖率
  - 测试执行覆盖率
  - 用例通过覆盖率
  - 代码覆盖率
  - 功能覆盖率
- 缺陷分布
  - 按测试时间（趋势）
  - 按测试一定周期
  - 按软件开发组
  - 按缺陷严重性
  - 按缺陷状态

测试的文档

- 测试计划 – 在测试计划阶段
- 测试用例文档 – 在测试设计阶段
- 缺陷报告、测试记录 – 在测试执行阶段
- 测试报告 – 在测试完成后

测试的数据流


被测软件
及其文档
测试计划
测试用例
测试工具
测试
测试结果
预期结果
评估
错误
排错
改正的软件
出错率数据
可靠性
分析
预测可靠性

大 纲

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, selenium，Jmeter, Bugzilla …
  - ……

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

什么情况下不适合用自动测试

- 定制型项目（一次性的）
- 项目周期很短的项目
- 业务规则复杂的对象
- 美观、声音、易用性测试
- 测试很少运行
- 软件不稳定
- 涉及物理交互

自动化测试的误区

- 期望自动化测试能取代手工测试
  - 不能期望自动化测试来取代手工测试， 测试主要还是要靠人工的；
- 期望自动测试发现大量新的缺陷
  - 事实证明新缺陷越多，自动化测试失败的几率就越大。发现更多的新缺陷应该是手工测试的主要目的。测试专家James Bach总结得 85%的缺陷靠手工发现，而自动化测试只能发现15%的缺陷。自动化测试能够很好的发现老缺陷。
- 工具本身不具有想象力
  - 工具毕竟是工具，出现一些需要思考、体验、界面美观方面的测试，自动化测试工具无能为力
- 技术问题、组织问题、脚本维护
  - 自动化测试的推行，有很多阻力，比如组织是否重视， 是否成立这样的测试团队，是否有这样的技术水平，对于测试脚本的维护工作量也挺大的，是否值得维护等等问题都必须考虑。