中南大学
CENTRAL SOUTH UNIVERSITY

软件工程

Software Engineering

龙 军

jlong@csu.edu.cn 18673197878

计算机学院 | 大数据研究院

Review: 软件质量事故

- 1981年，由计算机程序改变而导致1/67的时间偏差，使航天飞机上的5台计算机不能同步运行。这个错误导致了航天飞机发射失败。
- 1986年，1台Therac25机器泄露致命剂量的辐射，致使两名医院病人死亡。原因是一个软件出现了问题，导致这台机器忽略了数据校验。
- 2016年，区块链业界最大的众筹项目TheDAO遭到攻击，导致300多万以太币资产被盗，原因是其智能合约中splitDAO函数有漏洞。
- 2018年，印尼狮航一架波音737 MAX 8客机途中坠落，189人罹难，失事原因为软件设计缺陷，飞机的迎角传感器“数据错误”触发“防失速”自动操作，导致机头不断下压，最终坠海。
- 2018年至今，无人车在自动驾驶时出现了多次交通事故，导致车毁人亡。原因是机器学习模型未能及时正确识别出前方的情况并做出决策。





软件缺陷与质量问题

- 不可避免的软件缺陷
  - 软件是一种复杂的人造物
  - 不可避免会因为人在认知和思维方式上的缺陷（例如理解和掌握复杂问题的局限性、偶然的疏忽等）而引入各种缺陷和质量问题
- 由此导致了各种质量事故
  - 软件的作用越大，软件缺陷越可能造成灾难性的后果

软件测试是保证软件质量的最重要手段

第九章 软件测试

大 纲

01-软件测试概述

02-软件测试方法

03-软件测试层次

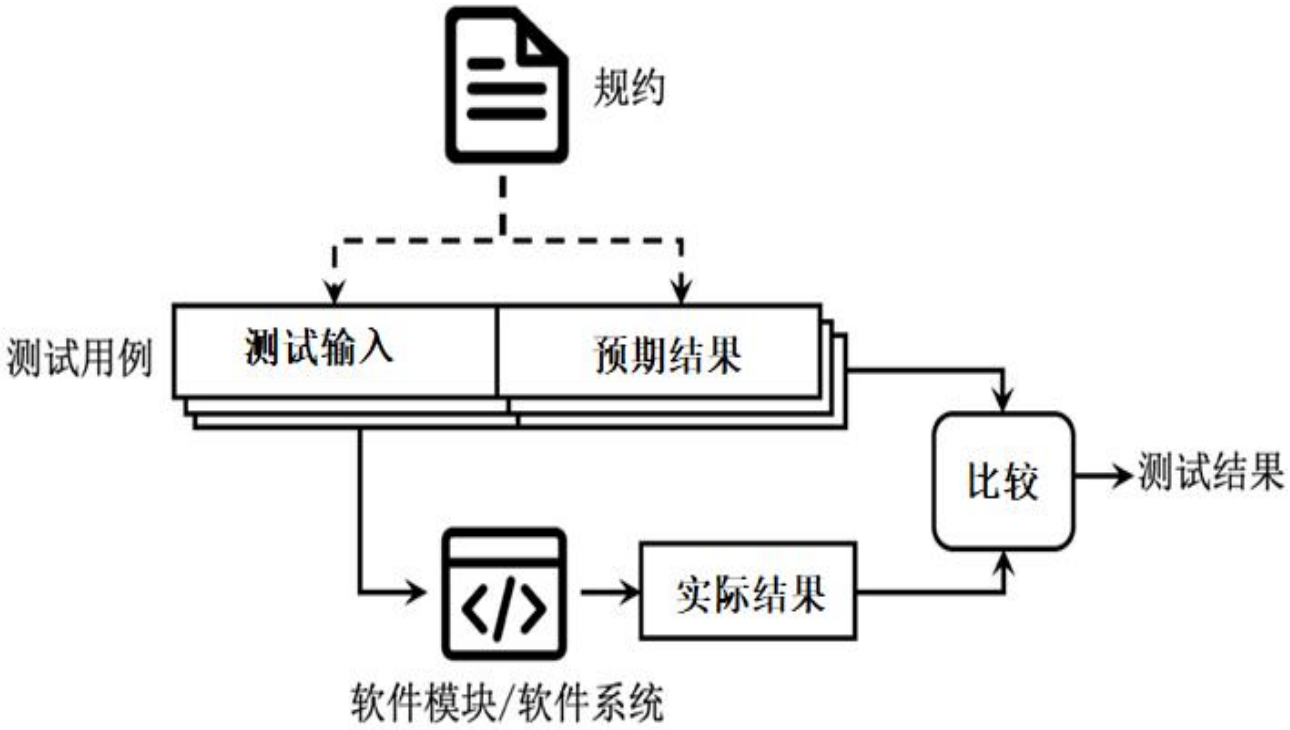
04-系统测试技术

05-软件测试过程

@第8章.教材

什么是软件测试

- IEEE SWEBOK 给出的定义：一个动态的过程，它基于一组有限的测试用例执行待测程序，目的是验证程序是否提供了预期的行为



测试（testing）的目的

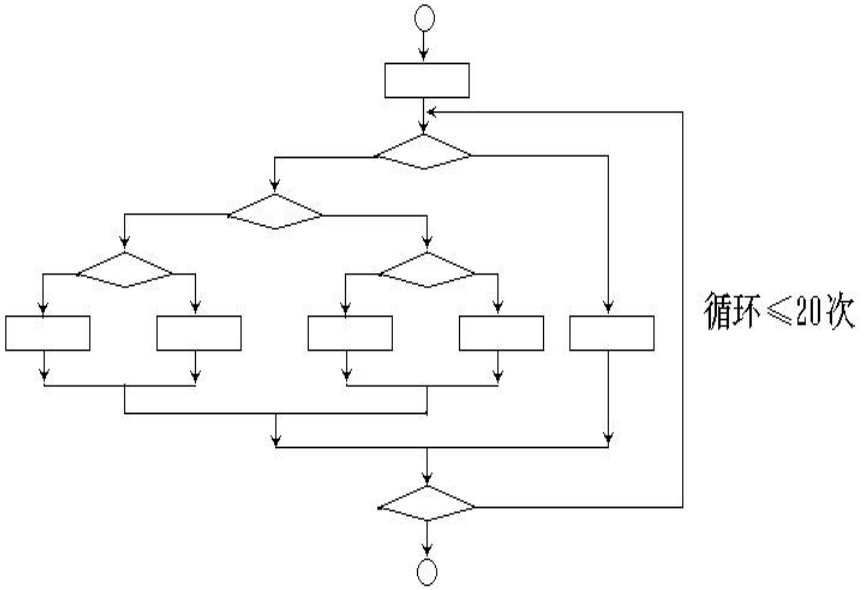
- 测试（testing）的目的
  - 发现软件的错误，从而保证软件质量
- 成功的测试
  - 发现了未曾发现的错误
- 与调试（debugging）的不同在哪?
  - 定位和纠正错误
  - 保证程序的可靠运行

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
  - 如何采用尽可能少的测试用例尽可能多地发现缺陷?
- 缺陷经常是聚集分布的
  - 80%的缺陷集中在20%的软件模块中，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
  - 每个迭代都安排测试
- 测试中的杀虫剂悖论
  - 软件经受的测试越多，对于测试人员的测试就具有越高的免疫力
- 测试应该从小到大
  - 模块 -- 子系统 -- 系统
- 制定测试计划，避免测试的随意性

第九章 软件测试

大 纲

中南大學
CENTRAL SOUTH UNIVERSITY

01-软件测试概述

02-软件测试方法

- 白盒测试方法
- 黑盒测试方法

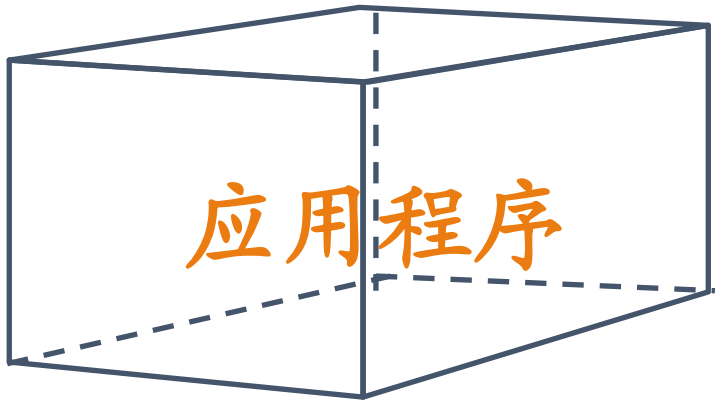
03-软件测试层次

04-系统测试技术

05-软件测试过程

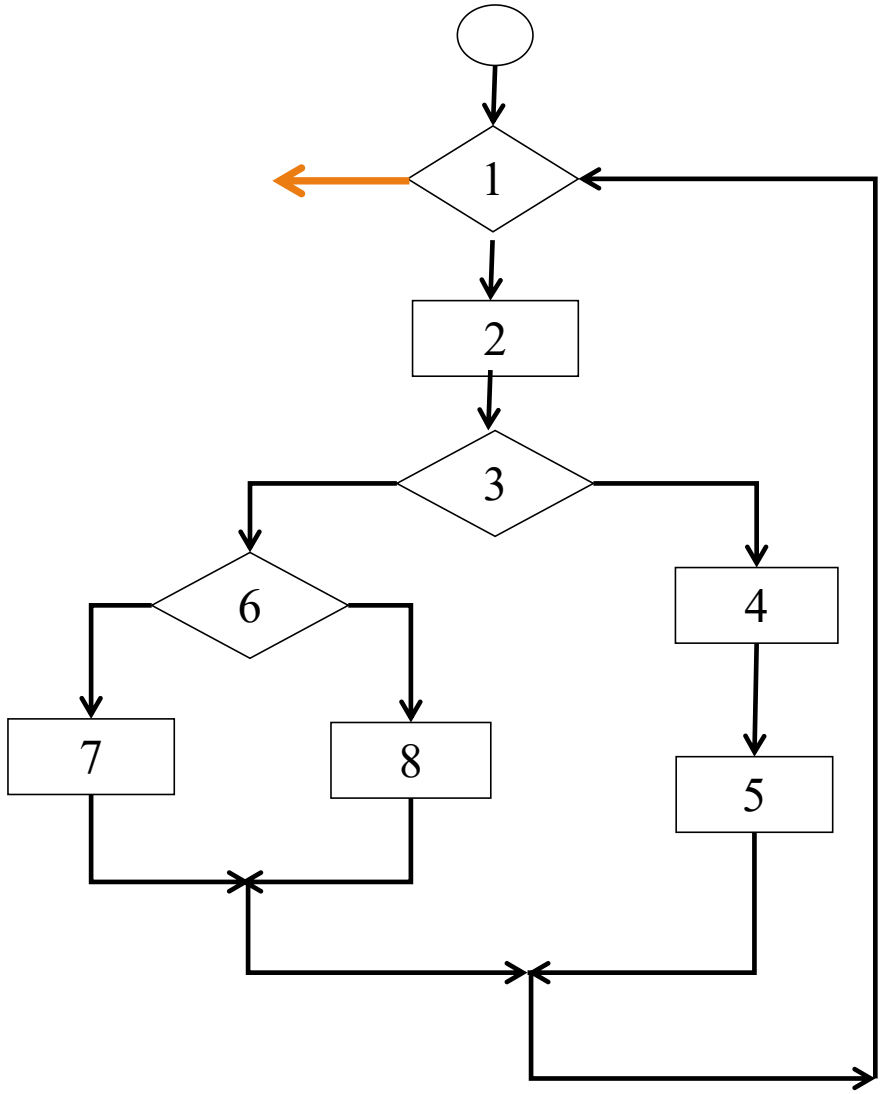
白盒测试

- 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，是一种基于代码的测试。



控制流测试--白盒测试的主要技术

1）语句覆盖
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

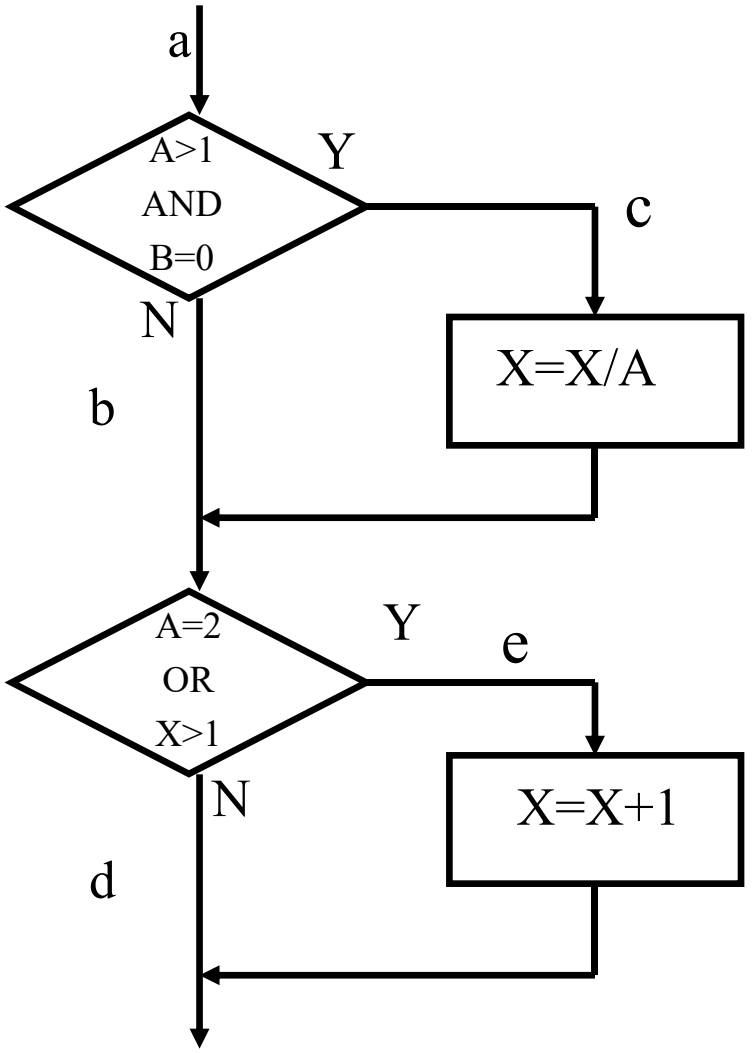
- 使得程序中的每一个语句至少被遍历一次
- 举例：对下列子程序进行测试

```
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）

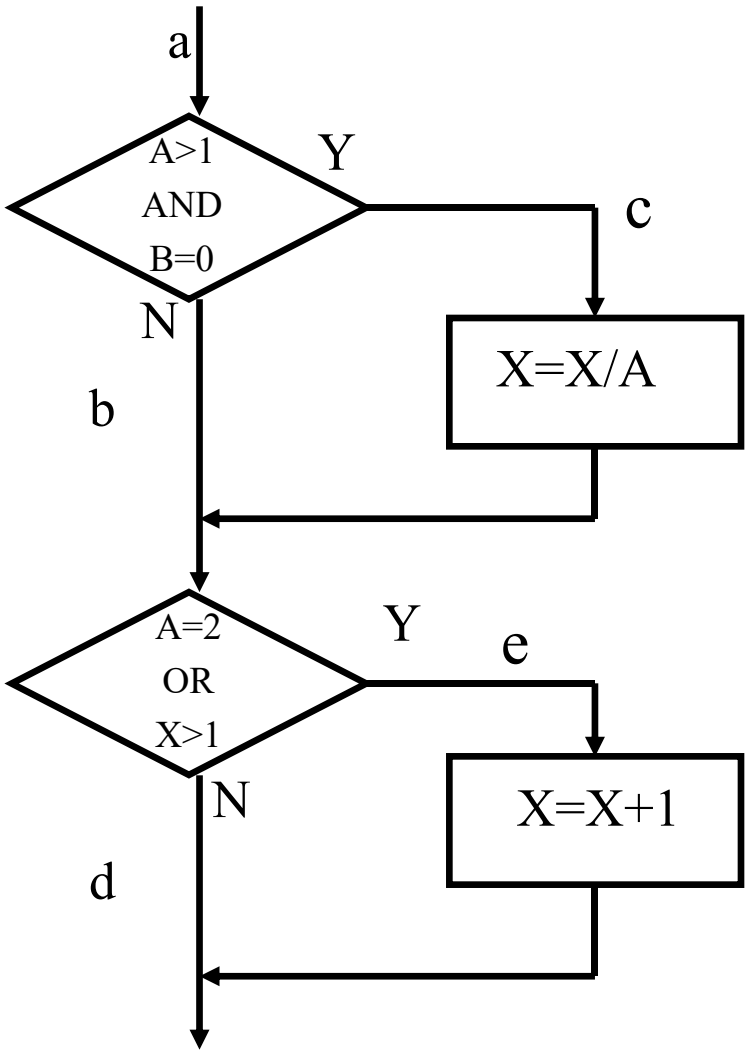
- 使得程序中每一个分支至少被遍历一次
- 测试用例

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

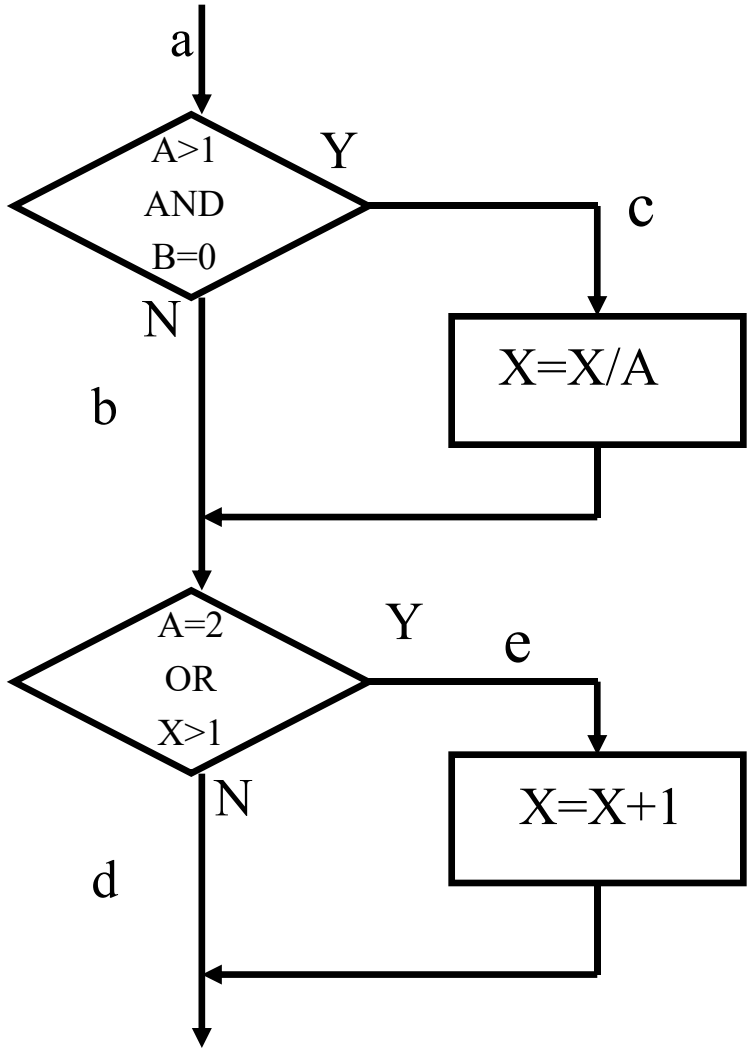
- 使得每个判定的条件获取各种可能的结果
- 在a点 A>1, A≤1, B=0, B≠0
- 在b点 A=2, A≠2, X>1, X≤1
- 测试用例

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

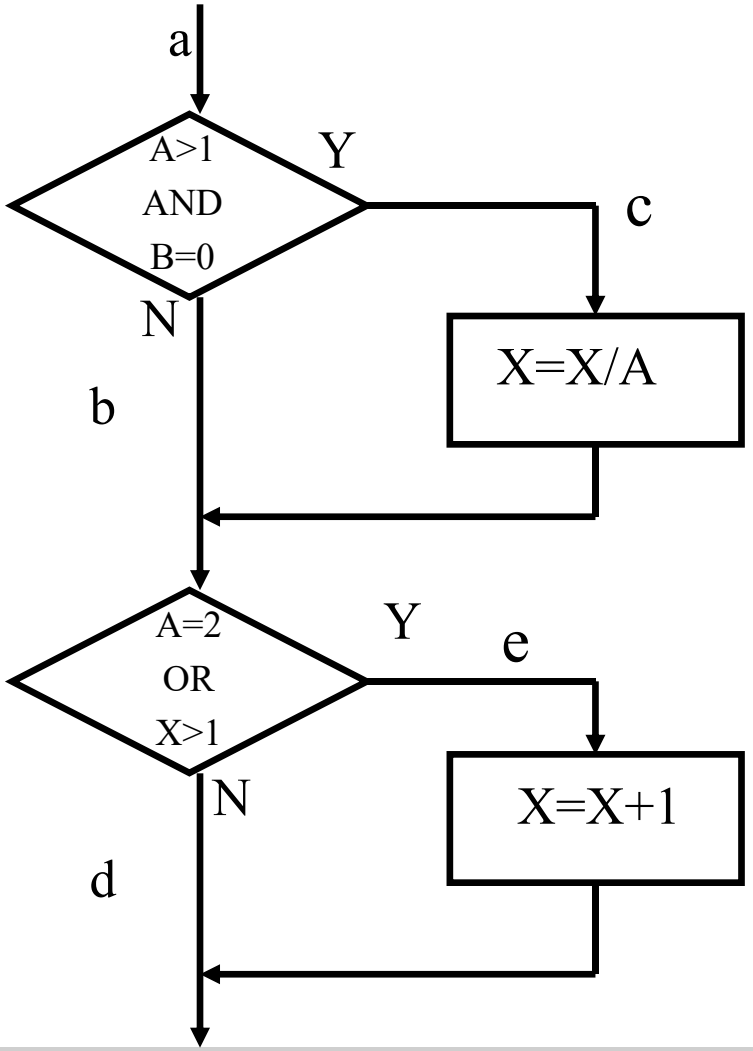
- 使得判定中的条件取得各种可能的值，并使得每个判定取得各种可能的结果
- 测试用例

1. A=2, B=0, X=4

   （沿路径ace）

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

   （沿路径abd）



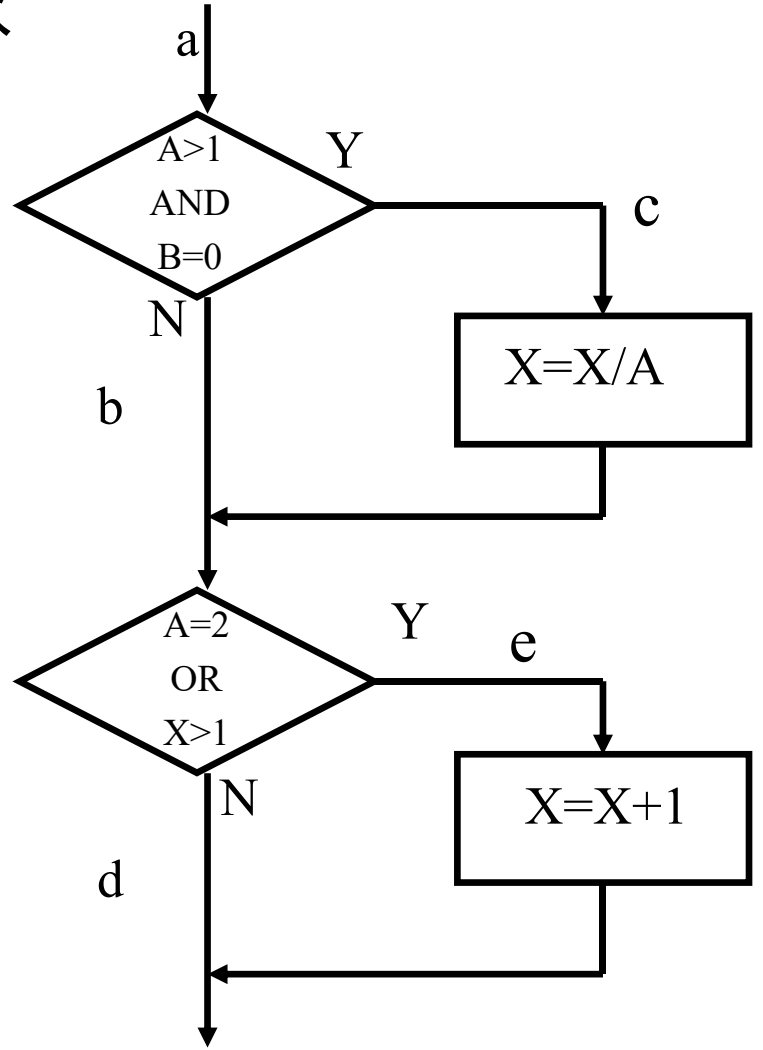
5）条件组合覆盖

- 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

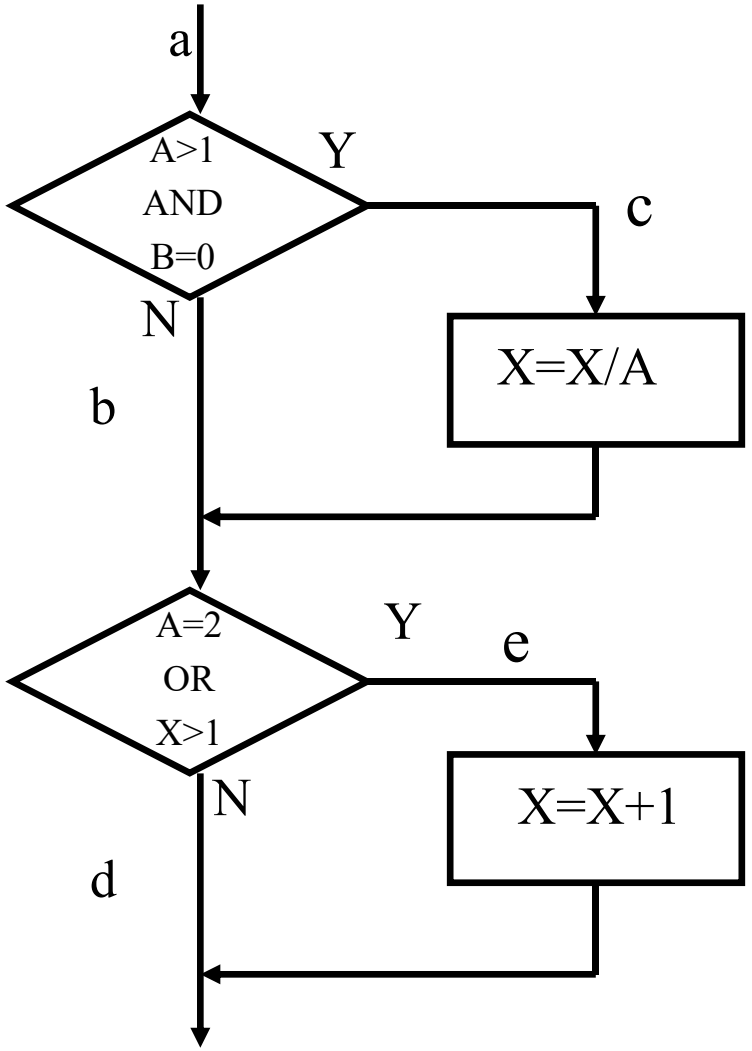


6）路径覆盖

- 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



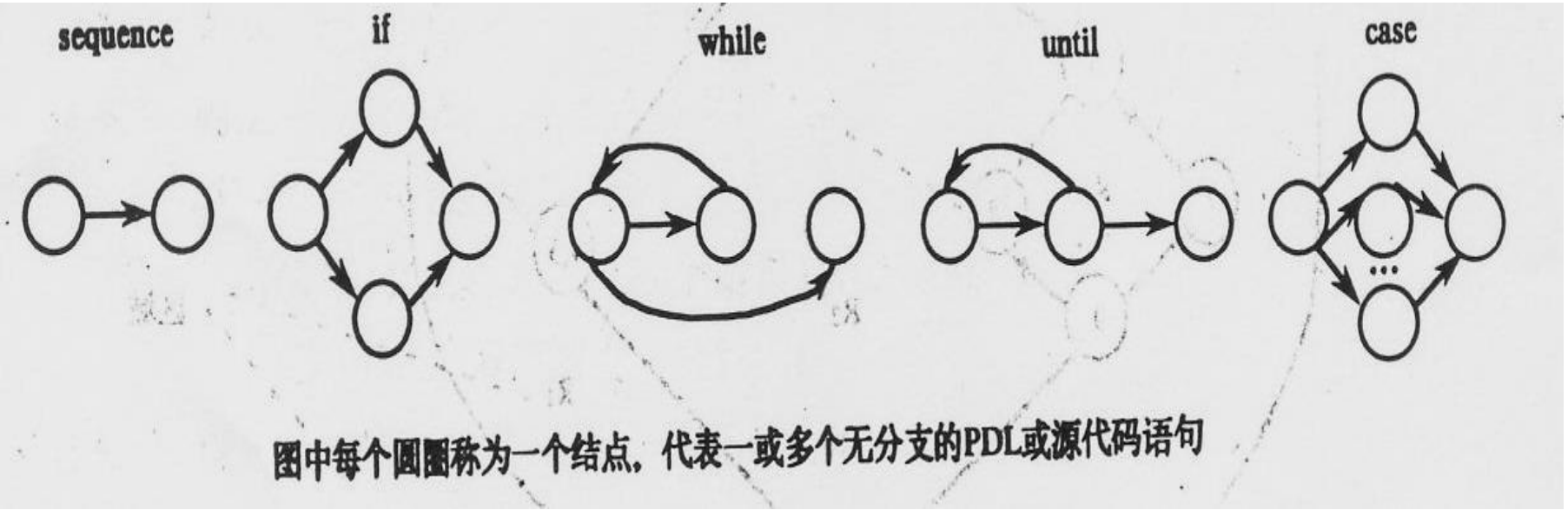
基本路径测试

- 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。

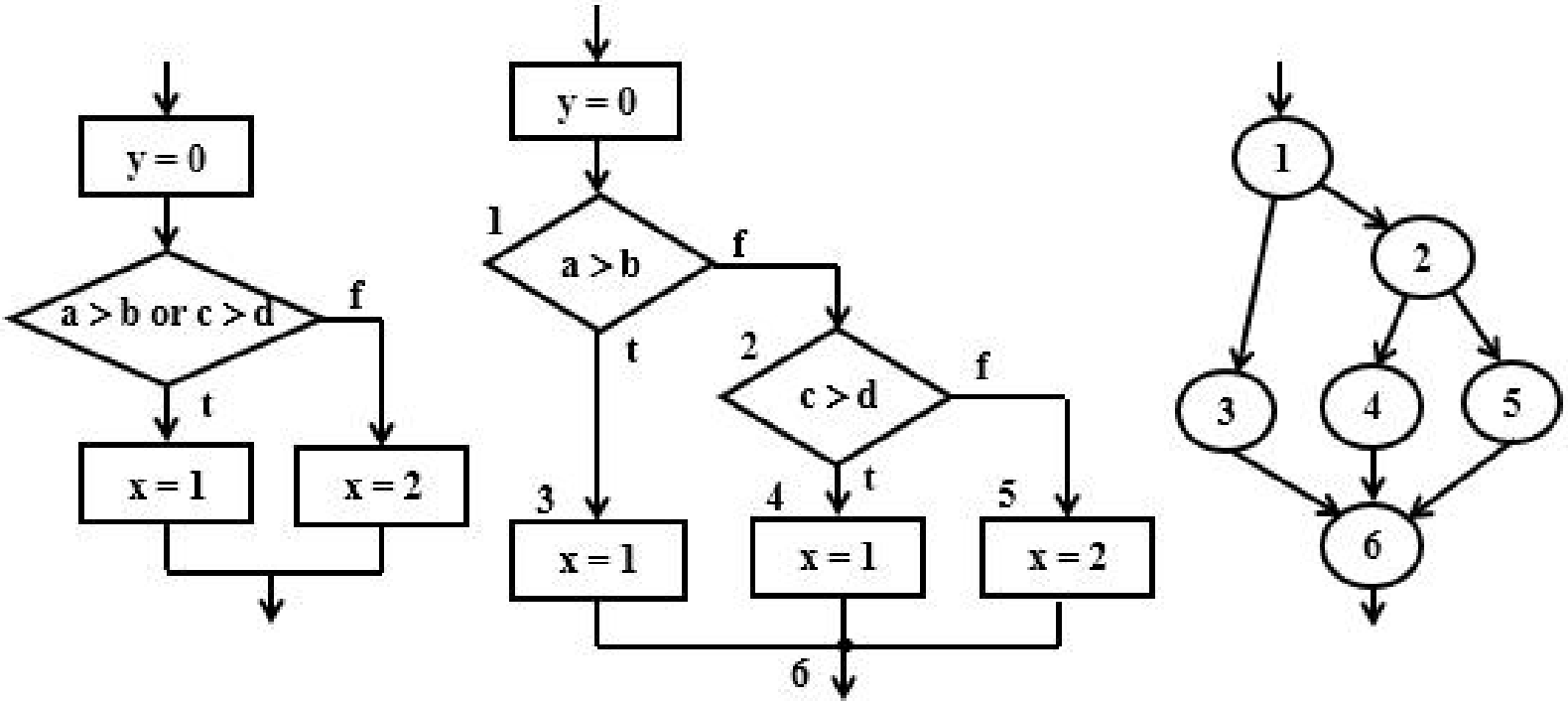
- 基本路径测试是Tom McCabe提出的一种白盒测试技术，这种方法首先根据程序或程序流程图画出控制流图（flow graph），并计算其区域数，然后确定一组独立的程序执行路径（称为基本路径），最后为每一条基本路径设计一个测试用例。

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。

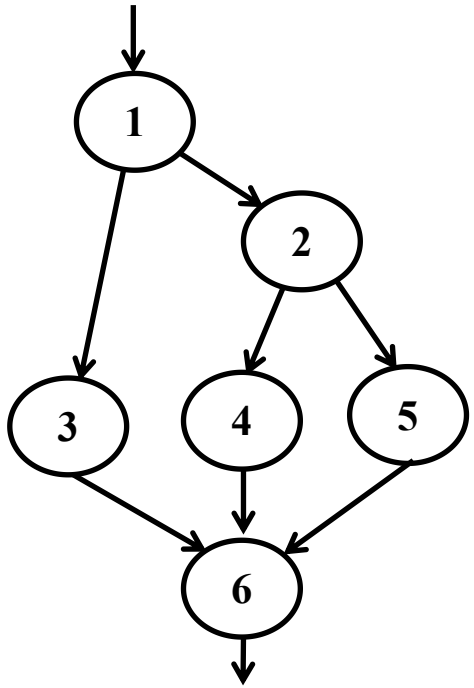


a）含复合条件的程序流程图

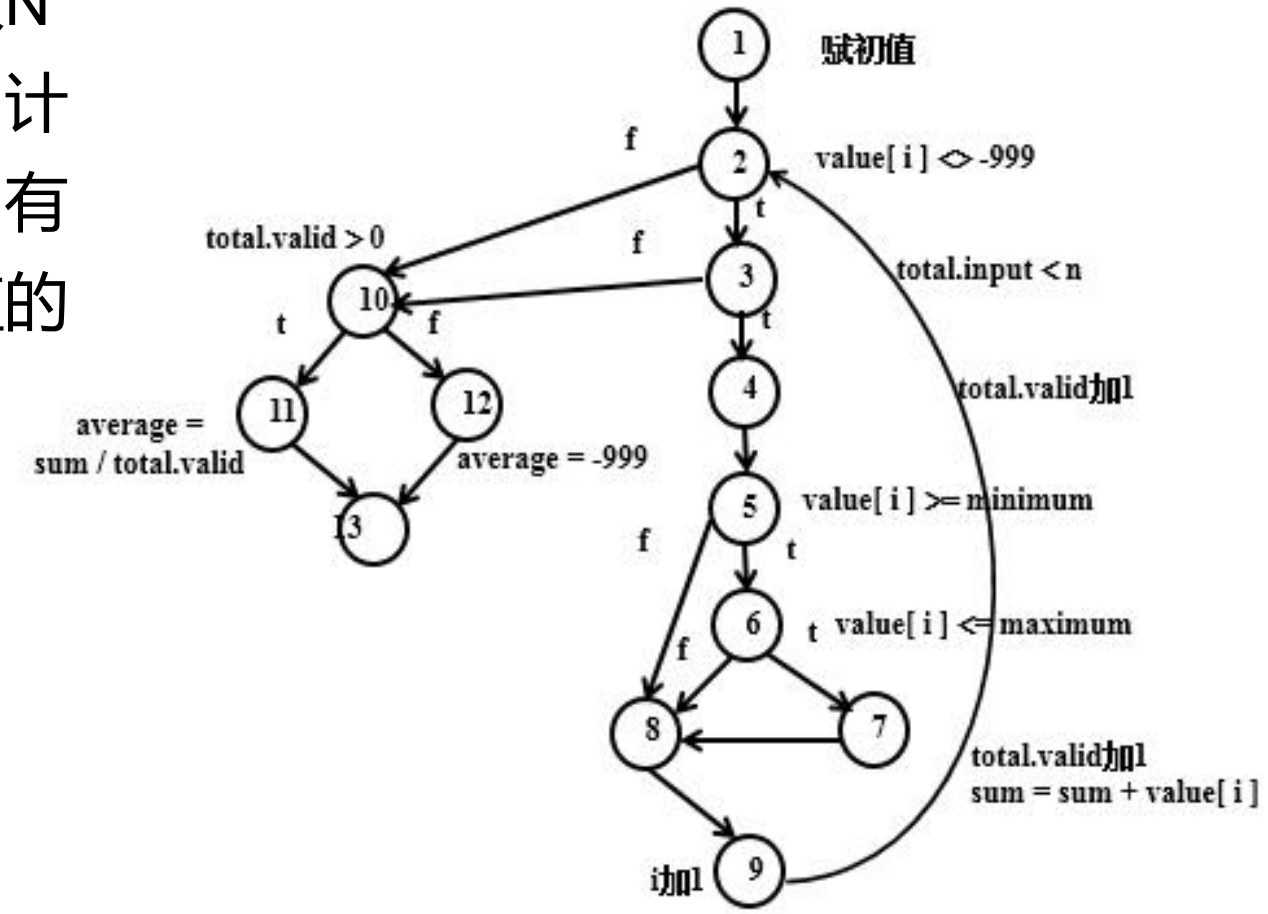
b）只含简单条件的程序流程图

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

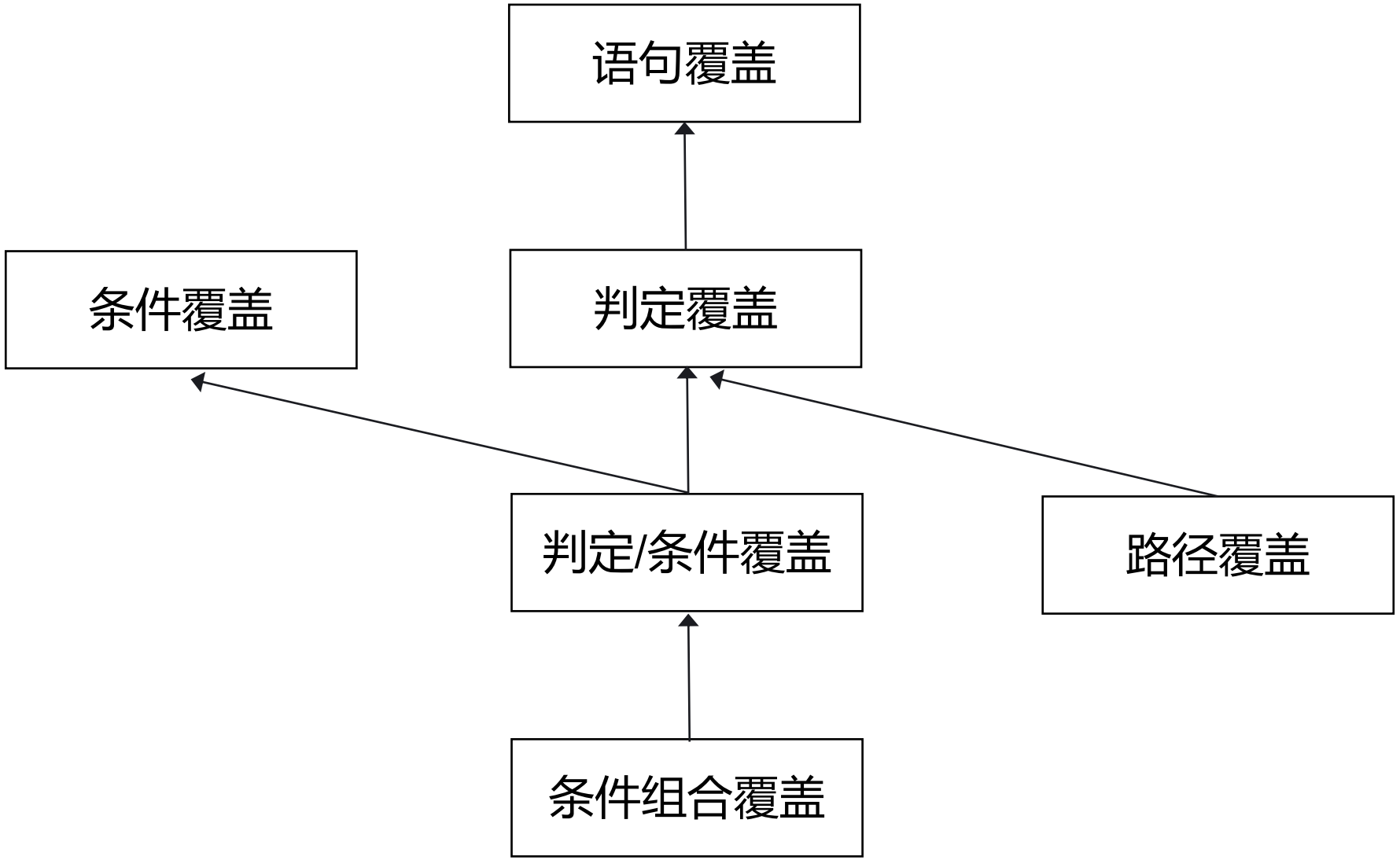
路径5： 1-2-3-4-5-6-8-9-2-10-12-13

- 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

各种覆盖准则之间的包含关系



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，是一种基于需求规约的测试，在软件开发后期执行。

黑盒测试技术

- 1）等价类划分法
- 2）边界值分析法
- 3）判定表法
- 4）错误推测法

1）等价类划分法(equivalence partitioning)

- 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。
- 输入的数据划分为有效等价类和无效等价类
- 输出也同样可以划分

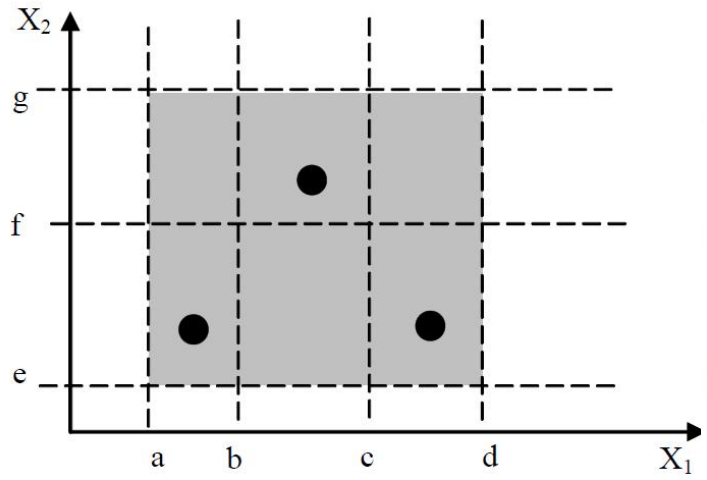
✓ 例1：每个学生可以选取修1至3门课程
  - 有效等价类：选修1至3门课程
  - 无效等价类：没选修课程；超过3门课程

✓ 例2：录入百分数成绩，输出五级：A、B、C、D、F
  - 有效等价类为: A、B、C、D、F级的成绩
  - 无效等价类为: 负分；大于100分

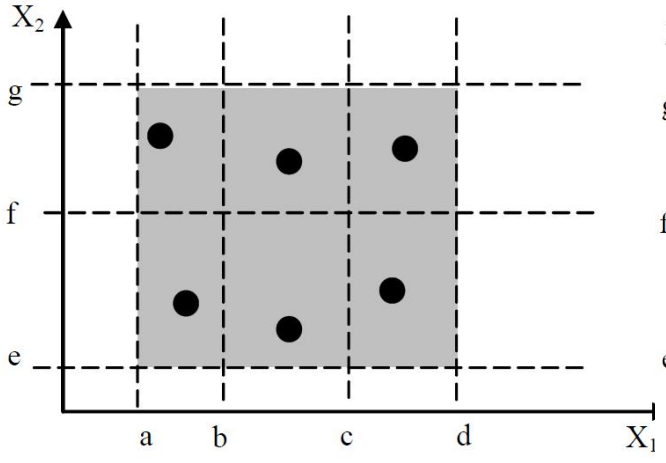
基于等价类的测试用例设计

- 根据是否考虑无效等价类，可划分为一般和健壮
- 根据测试时基于单缺陷还是多缺陷，可划分弱和强

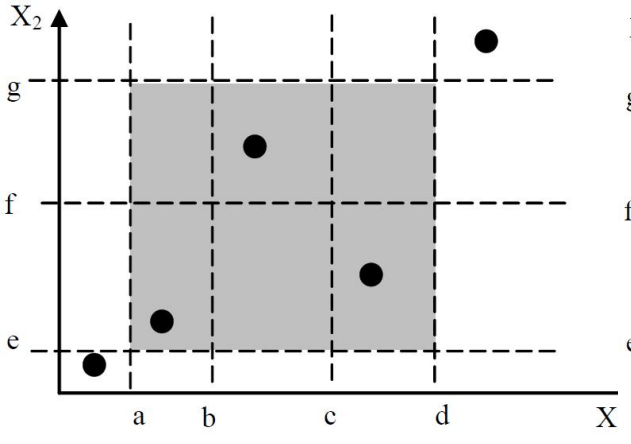
	弱（单缺陷假设）	强（多缺陷假设）
一般	弱一般等价类测试	强一般等价类测试
健壮	弱健壮等价类测试	强健壮等价类测试



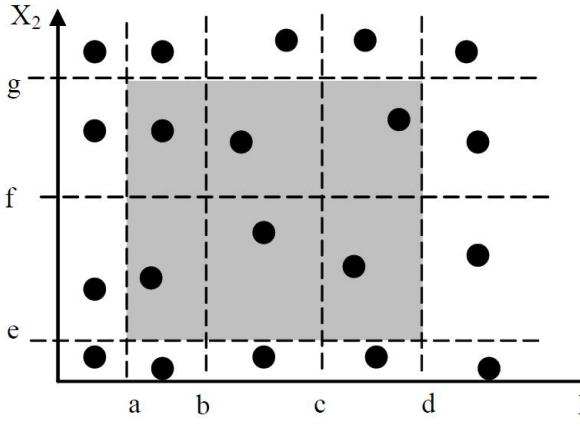
弱一般等价类测试



强一般等价类测试



弱健壮等价类测试



强健壮等价类测试

示例

输入货品信息，最后给出货品存放指示

- 编号必须为英文字母与数字的组合，由字母开头，包含6个字符，且不能有特殊字符
- 货品的登记数量在10到500之间（包含10和500）
- 货品的类型是设备、零件、耗材中的一种
- 货品的尺寸是大型、中型、小型中的一种，大型货品存放在室外堆场，中型货品存放在专用仓库，小型货品存放在室内货架
- 违反以上要求的登记信息被视为无效输入

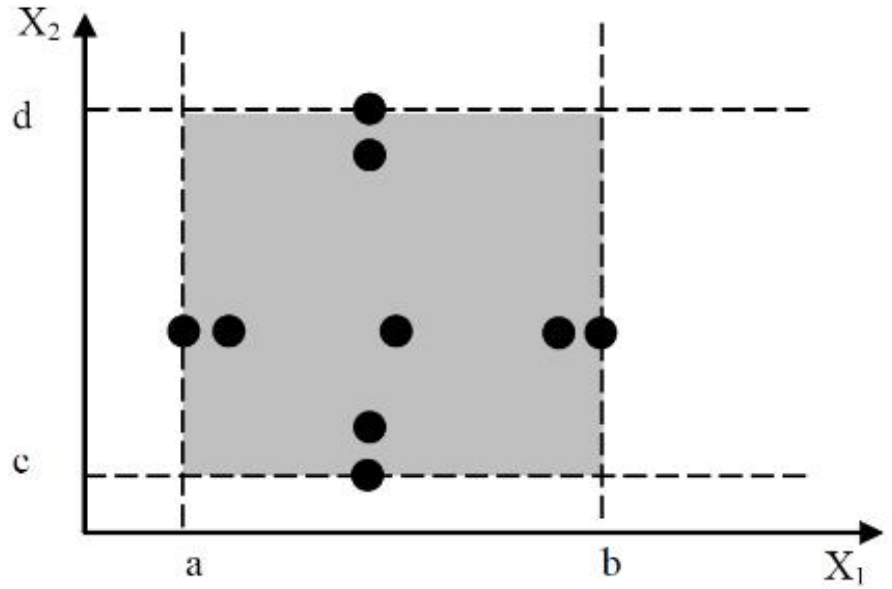
采用弱健壮等价类测试法：

输入数据	有效等价类	无效等价类
货品编号	（1）符合规则的编号	（7）编号长度不为6 （8）编号有特殊字符 （9）编号不以字母开头
登记数量	（2）10≤数量≤500	（10）数量＜10 （11）数量＞500
货品类型	（3）{设备，零件、耗材}	（12）非设备、零件、耗材中的一种
货品尺寸	（4）大型 （5）中型 （6）小型	（13）非大型、中型、小型中的一种

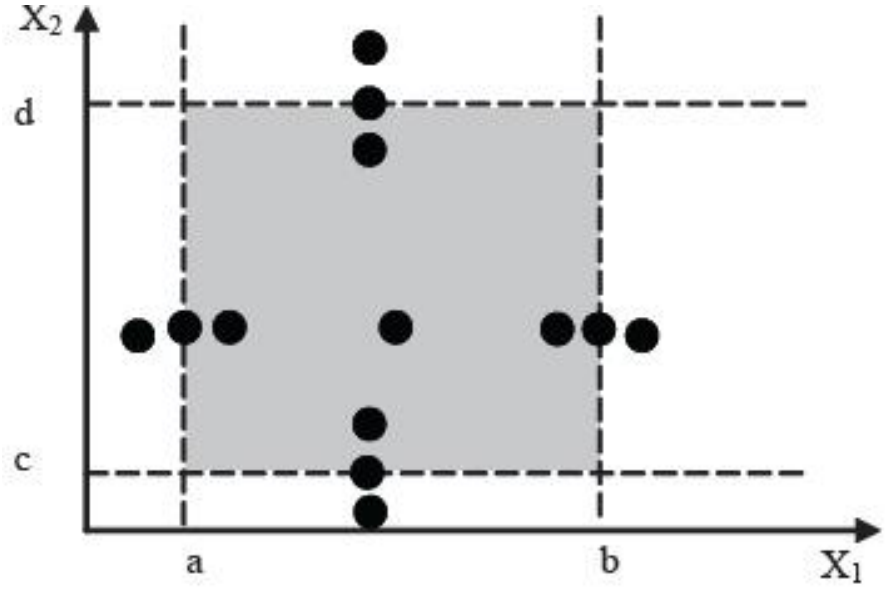
输入数据	预期输出	覆盖的等价类
A=EQ0101，B=30，C=设备，D=大型	合法登记信息，存放地为室外堆场	（1）（2）（3）（4）
A=CM0202，B=100，C=零件，D=中型	合法登记信息，存放地为专用仓库	（1）（2）（3）（5）
A=MT0303，B=400，C=耗材，D=小型	合法登记信息，存放地为室内货架	（1）（2）（3）（6）
A=EQ01023，B=30，C=设备，D=大型	非法登记信息	（7）
A=EQ0102#，B=30，C=设备，D=大型	非法登记信息	（8）
A=0102EQ，B=30，C=设备，D=大型	非法登记信息	（9）
A=EQ0101，B=0，C=设备，D=大型	非法登记信息	（10）
A=EQ0101，B=600，C=设备，D=大型	非法登记信息	（11）
A=EQ0101，B=30，C=装置，D=大型	非法登记信息	（12）
A=MT0202，B=100，C=耗材，D=中小型	非法登记信息	（13）

2）边界值分析法(boundary value analysis)

- 边界值分析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，是等价类划分方法的一种补充
- 不仅根据输入条件，它还根据输出情况设计测试用例

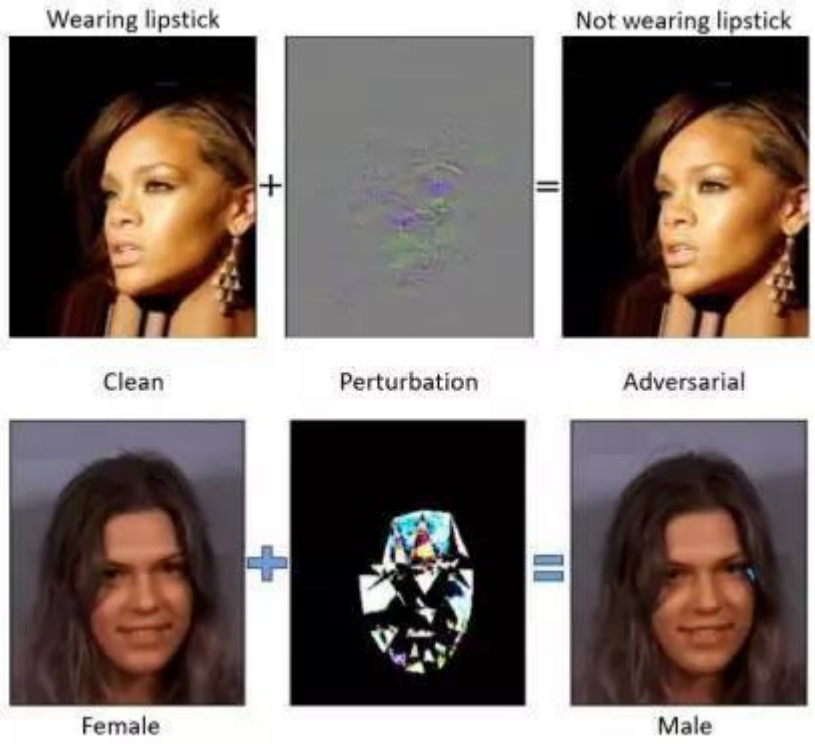


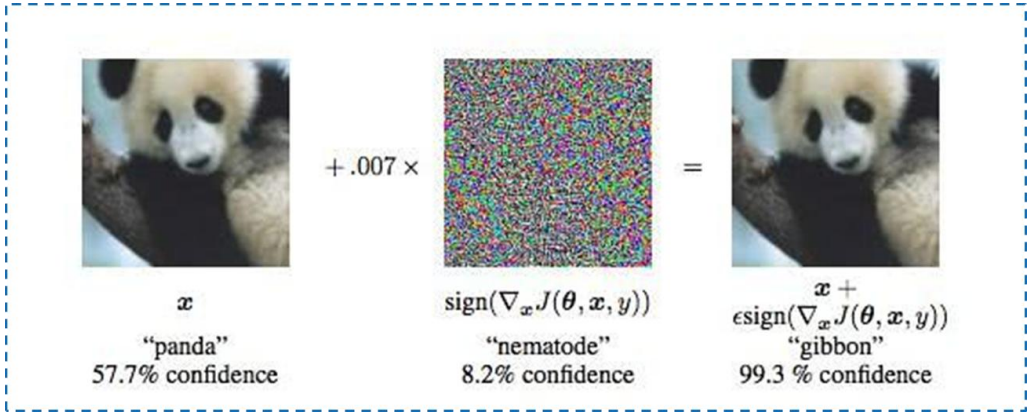
一般边界值测试



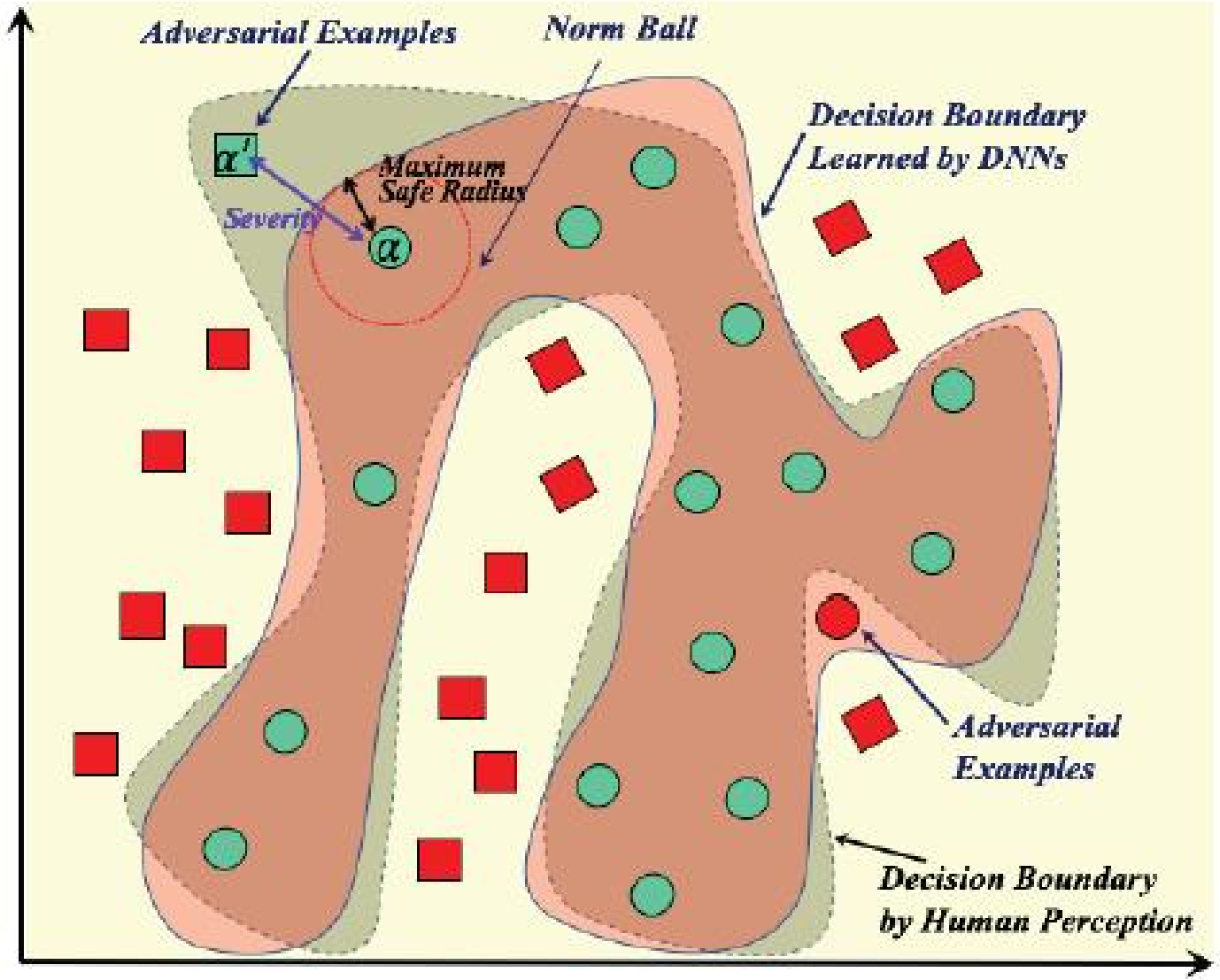
健壮边界值测试

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
x'
Maximum
Safe Radius
Severity
x
Adversarial
Examples
Decision Boundary
by Human Perception

3）判定表法

- 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
  - 画出判定表，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。

画出判定表

条件桩	条件项										
	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
投入1元5角硬币 投入2元硬币 按“可乐”按钮 按“雪碧”按钮 按“红茶”按钮	1 0 1 0 0	1 0 0 1 0	1 0 0 0 1	1 0 0 0 0	0 1 1 0 0	0 1 0 1 0	0 1 0 0 1	0 1 0 0 0	0 0 1 0 0	0 0 0 1 0	0 0 0 0 1
动作桩	**动作项**										
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	√	√	√		√ √	√ √	√ √				

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	规则	输入	预期输出
1	R1	投入1元5角，按“可乐”	送出“可乐”饮料
2	R2	投入1元5角，按“雪碧”	送出“雪碧”饮料
3	R3	投入1元5角，按“红茶”	送出“红茶”饮料
4	R5	投入2元，按“可乐”	找5角，送出“可乐”
5	R6	投入2元，按“雪碧”	找5角，送出“雪碧”
6	R7	投入2元，按“红茶”	找5角，送出“红茶”

4）错误猜测法(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

综合运用多种黑盒测试用例设计策略

- 对输入和输出划分有效的和无效等价类；
- 使用边界值分析法进行补充；
- 如果规范中含有输入条件的组合，采用判定表；
- 使用错误“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

中南大學
CENTRAL SOUTH UNIVERSITY

第九章 软件测试

01-软件测试概述

02-软件测试方法

03-软件测试层次

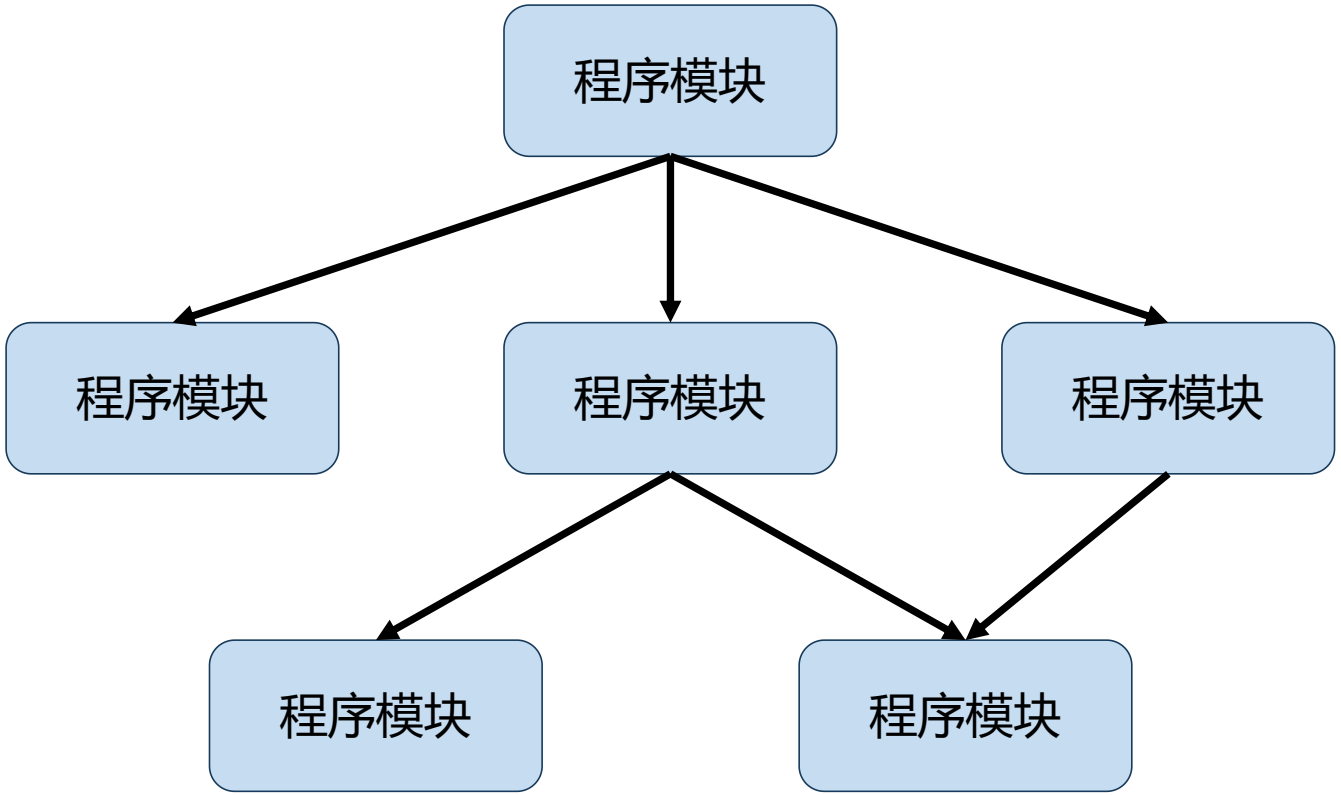
单元测试
集成测试
系统测试

04-系统测试技术

05-软件测试过程

缺陷的潜在位置

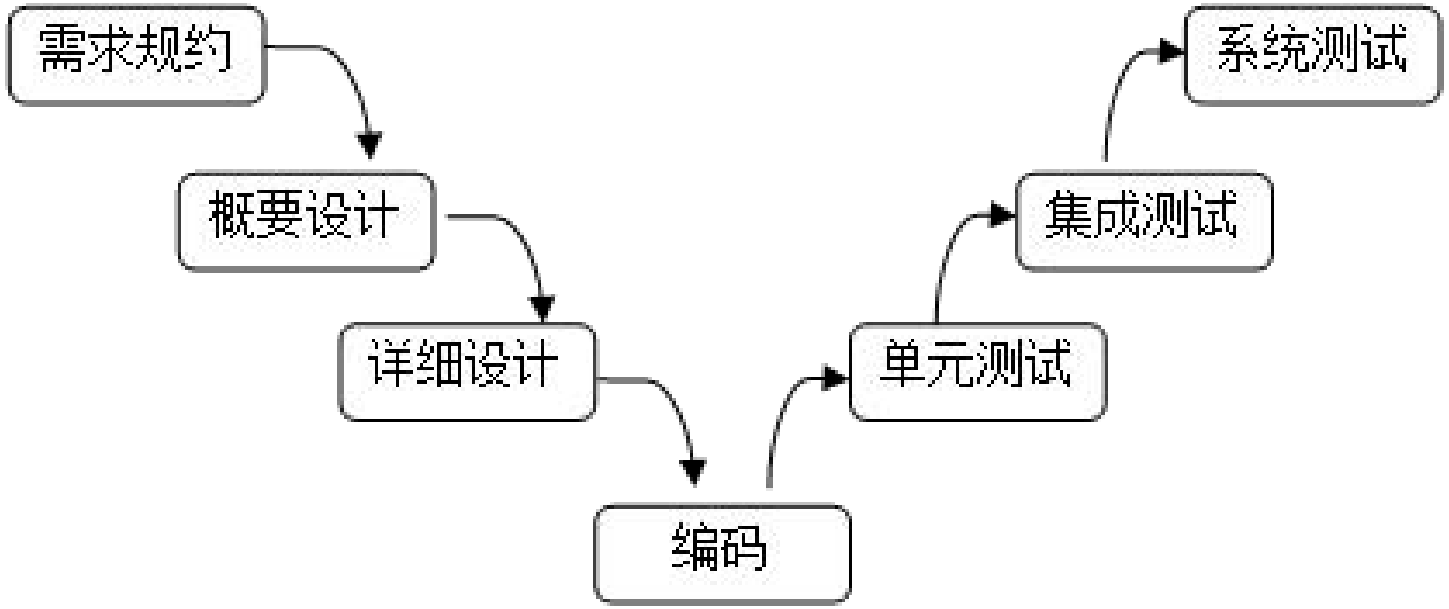
- 思考：软件缺陷会“潜伏”在程序代码的哪些地方？



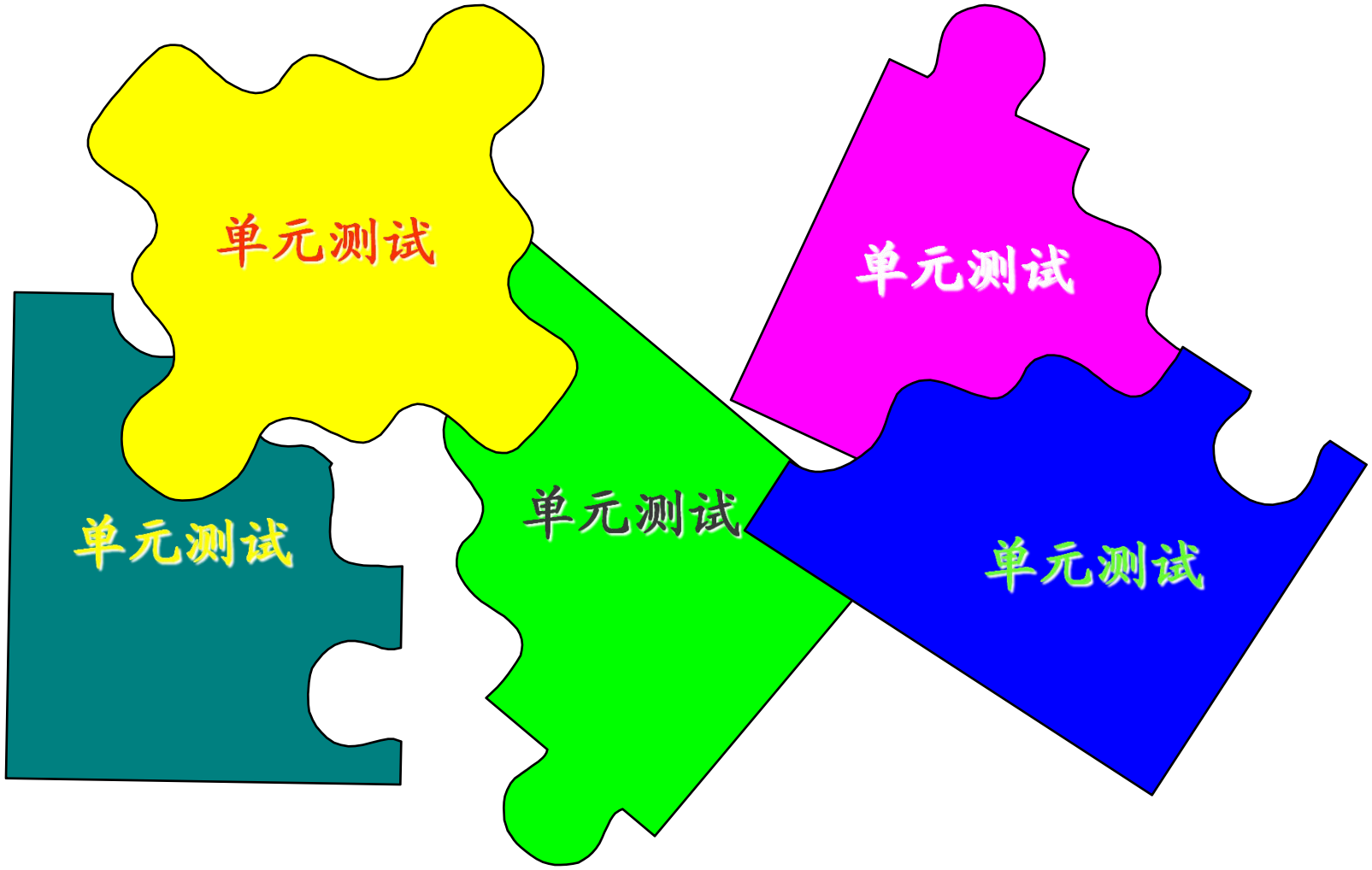
- ✓ 模块内部
- ✓ 模块接口与交互
- ✓ 整个系统

测试层次

- 不同层次的测试：
  - 单元测试 (Unit testing) – 采用白盒测试方法
  - 集成测试 (Integration testing) – 采用白盒+黑盒测试方法
  - 系统测试 (System testing) – 采用黑盒测试方法

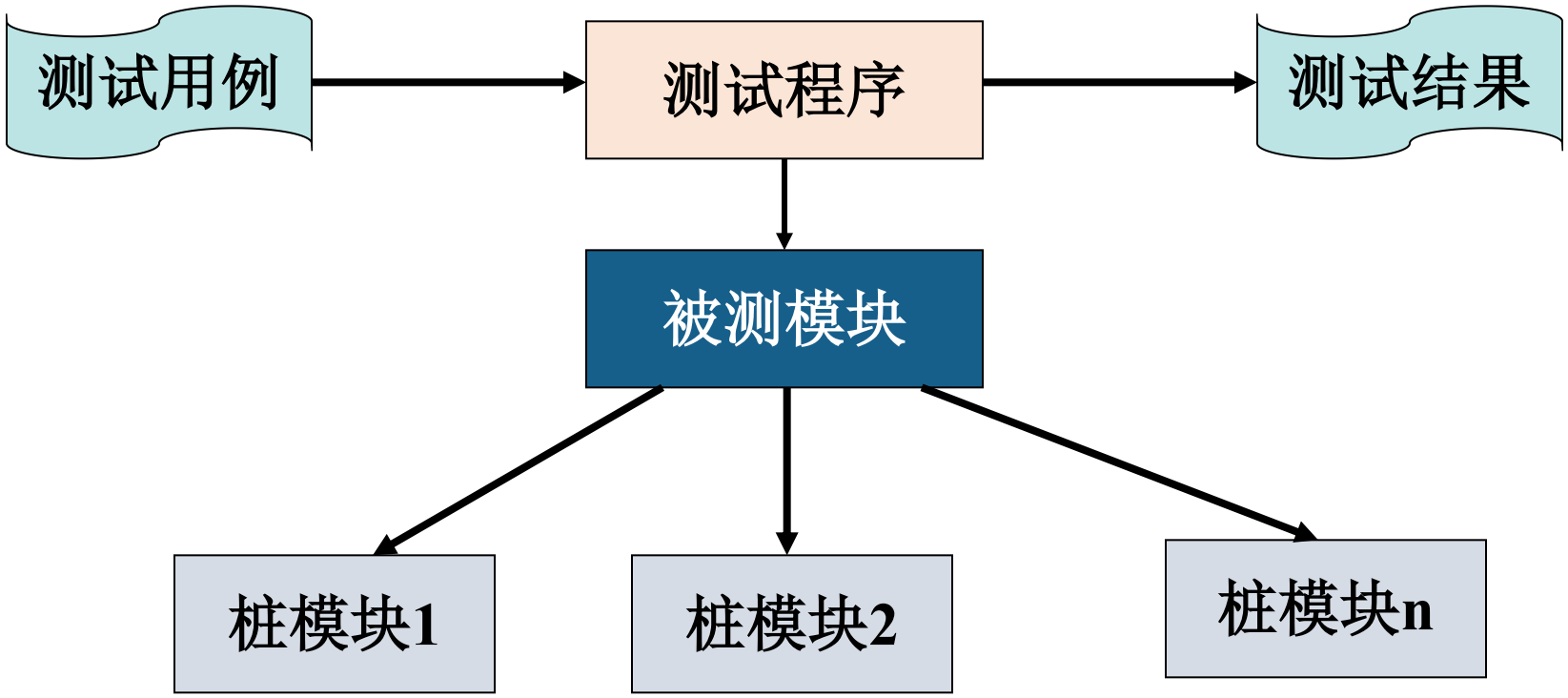


单元测试



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模块单元（如函数、子过程、类）进行的测试。

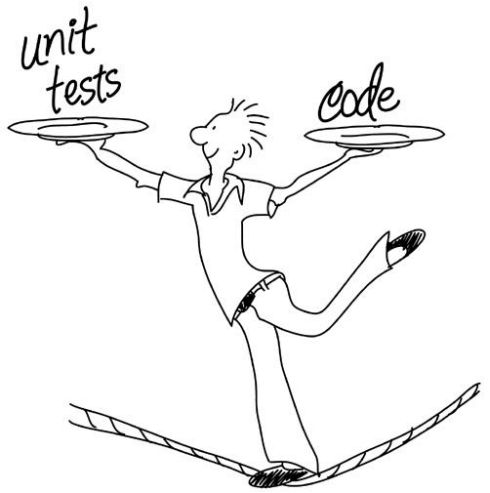


单元测试 - - 测什么?

- 单元测试是针对每个基本模块，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

单元测试 - - 何时测试和由谁测?

- When
  - 该基本单元的编码完成后就可以对其进行单元测试。
  - 推荐提前测试，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。
  - 测试小组可以对单元测试的代码作一定程度的审核。

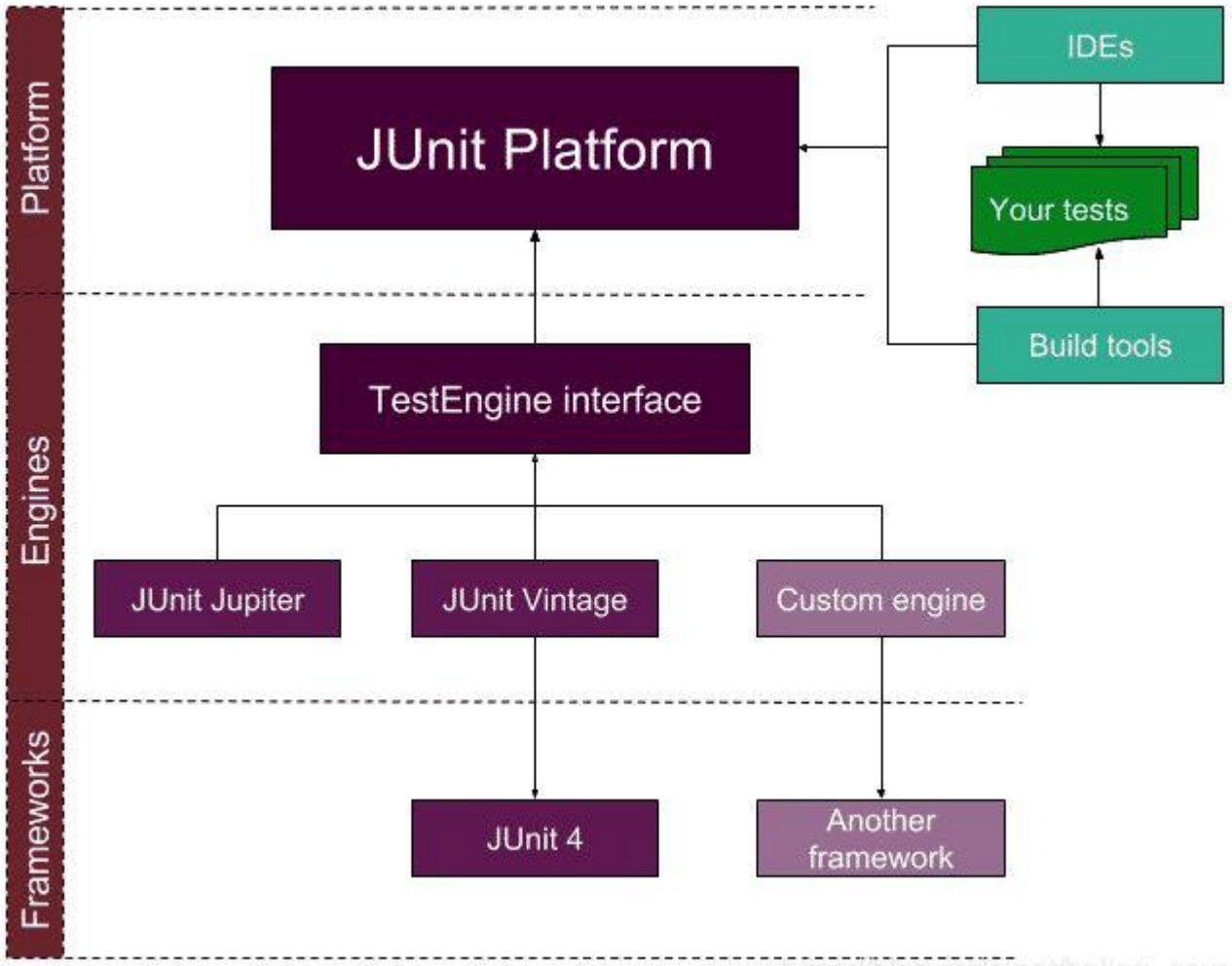


单元测试 - - 如何测试?

- 采用白盒测试方法
  - 基于代码的测试，进行控制流测试，实现语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合、或者路径覆盖。
- 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
  - JUnit（Java），CppUnit（C++），Cunit (C)，DUnit（Delphi），NUnit（.net），PhpUnit（PHP），PyUnit/unittest/pytest (Python)，Jest (Javascript) 等等。

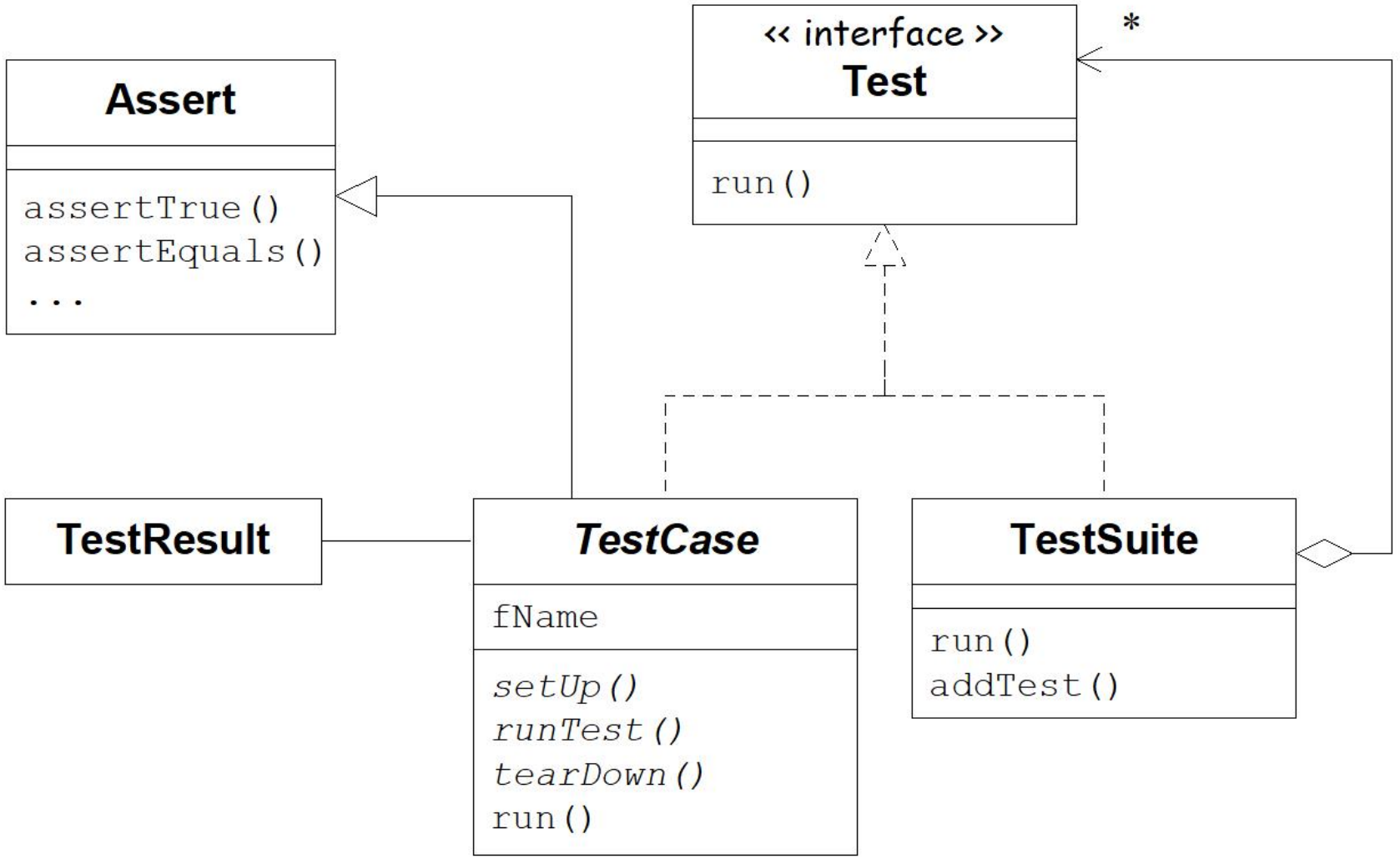
Junit 5 框架

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));
    }
}
```

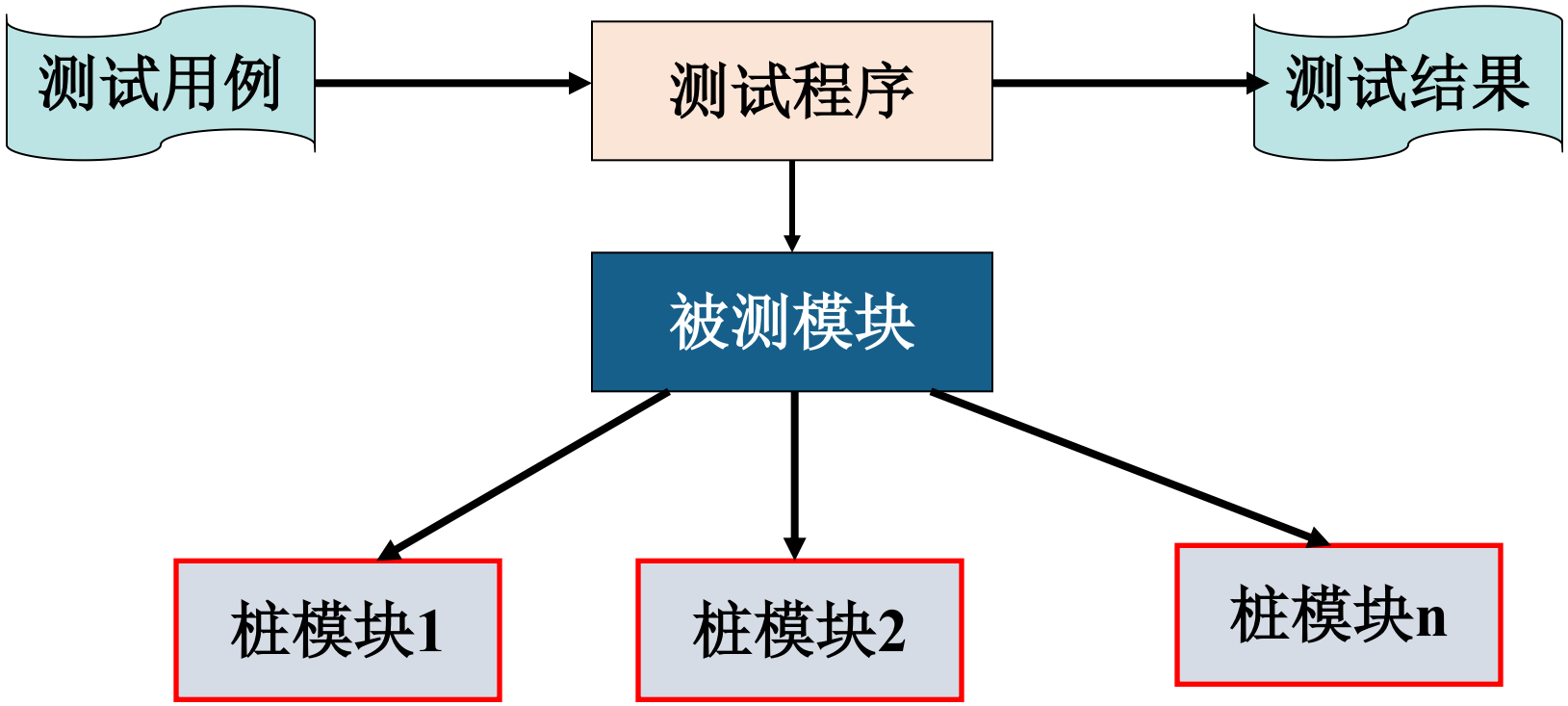
测试类

测试方法，即测试用例

每个@Test方法负责对某种情况测试，测试结果为true/false

单元测试的桩模块

- 测试的桩模块采用Mock工具来实现
  - JMockit 是一款针对Java类、接口、对象的Mock工具，被广泛应用于Java单元测试。
  - 其他常见的开源Mock工具有JMock、EasyMock、Mockito等。



带桩模块的单元测试用例代码示例

```
1       public class testCalFee {
2           @Test
3           public void testCalFeeByCity() {
4               MockUp<IGetPostcode> stub = new MockUp<IGetPostcode>() {
5                   @Mock
6                   public String getPCodeByCity(String city) {
7                       if ("Shanghai".equals(city)) {
8                           return "200000";
9                       }
10                      else {
11                          return "000000";
12                      }
13                  }
14              };
15              IGetPostcode mockInstance = stub.getMockInstance();
16              CalFee calFee = new CalFee(mockInstance);
17              assertEquals(20,calFee.calFeeByCity("Shanghai"));
18          }
19      }
```

定义并构造一个代表IGetPostcode接口的测试桩sub

实例化测试桩
实例化被测方法的实例化对象
利用断言对返回结果进行判断

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，不会再继续执行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.

常见JUnit测试代码结构

```
import 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);
}
```

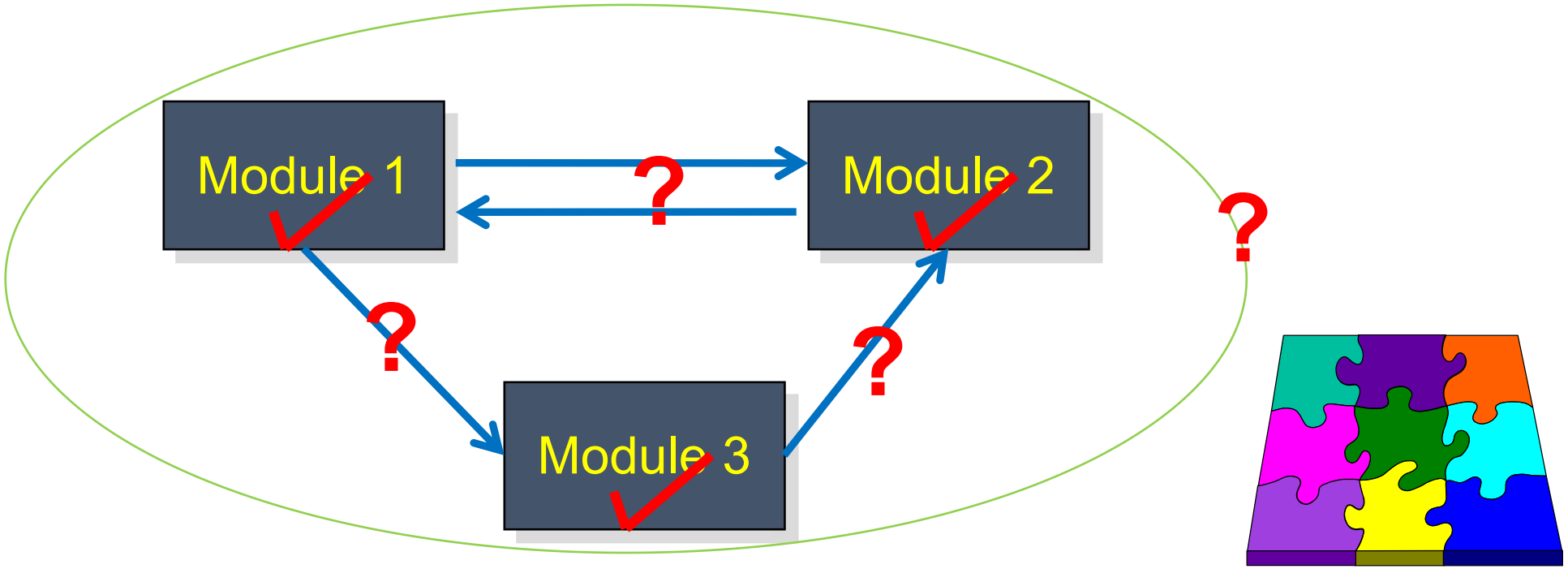
two-column.csv

```
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（integration testing）：又称组装测试， 根据软件架构构造完整系统并在此过程中进行测试，以发现与接口和交互相关的问题
- 目标：在通过单元测试的软件模块基础上根据软件架构构造完整的软件系统

集成测试 - - 测什么?

- 在把各个软件模块连接起来的时候，穿越单元接口的数据是否会丢失;
- 一个软件模块的功能是否会对另一个软件模块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;
- 单个软件模块的误差累积起来，是否会放大，从而达到不能接受的程度。

集成测试针对的主要错误来源

- 数据在穿越软件模块之间的接口传递时发生丢失或明显延迟（例如通过网络传输）
- 不同软件模块对参数或值存在不一致的理解，这种问题不会导致软件模块之间调用或通信失败，但可能导致不一致的处理逻辑
- 软件模块由于共享资源或其他原因而存在相互影响和副作用
- 软件模块交互后进行计算的误差累计达到了不能接受的程度，或者接口参数取值超出取值范围或者容量
- 全局数据结构出现错误，使得不同软件模块之间无法按照统一的标准进行计算
- 软件模块使用未在接口中明确声明的资源时，参数或资源造成边界效应

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模块

集成测试示例

- 自顶向下深度优先测试次序：

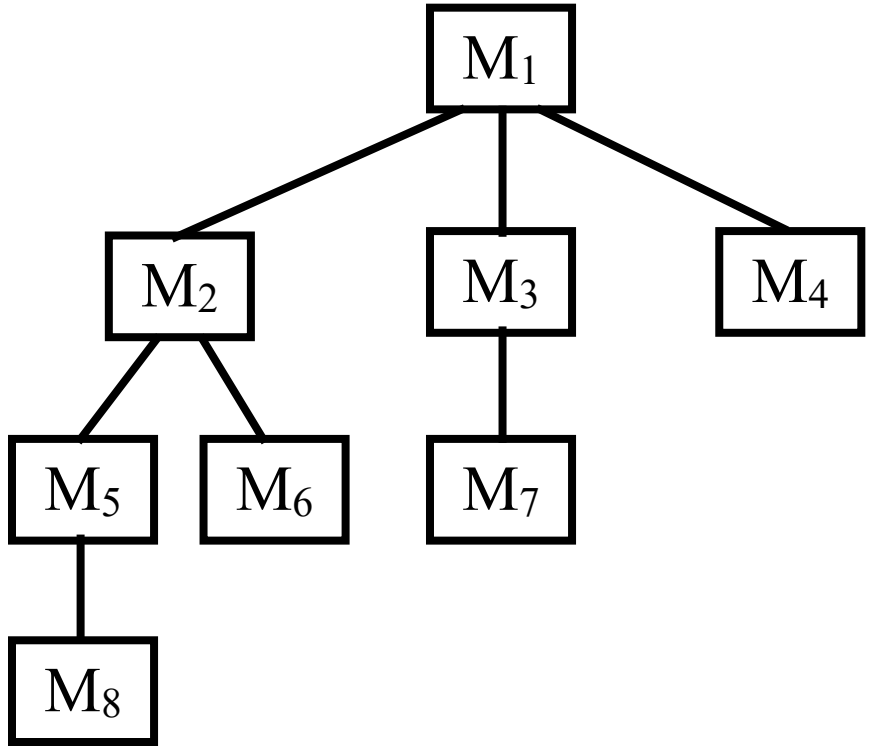
  M1、M2、M5、M8、M6、M3、M7、M4

- 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

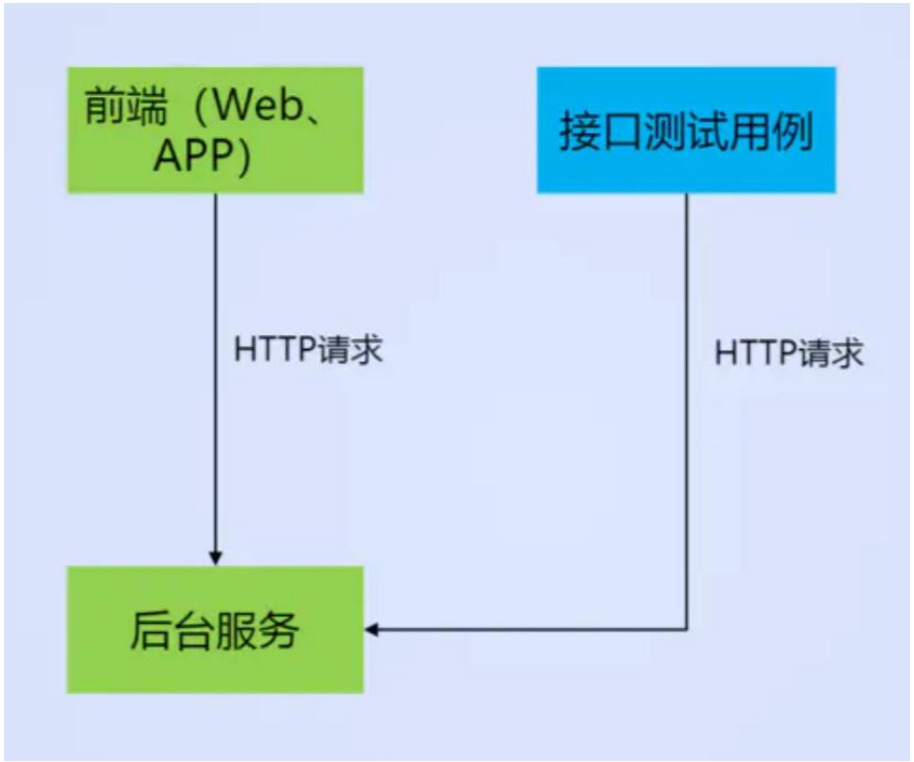


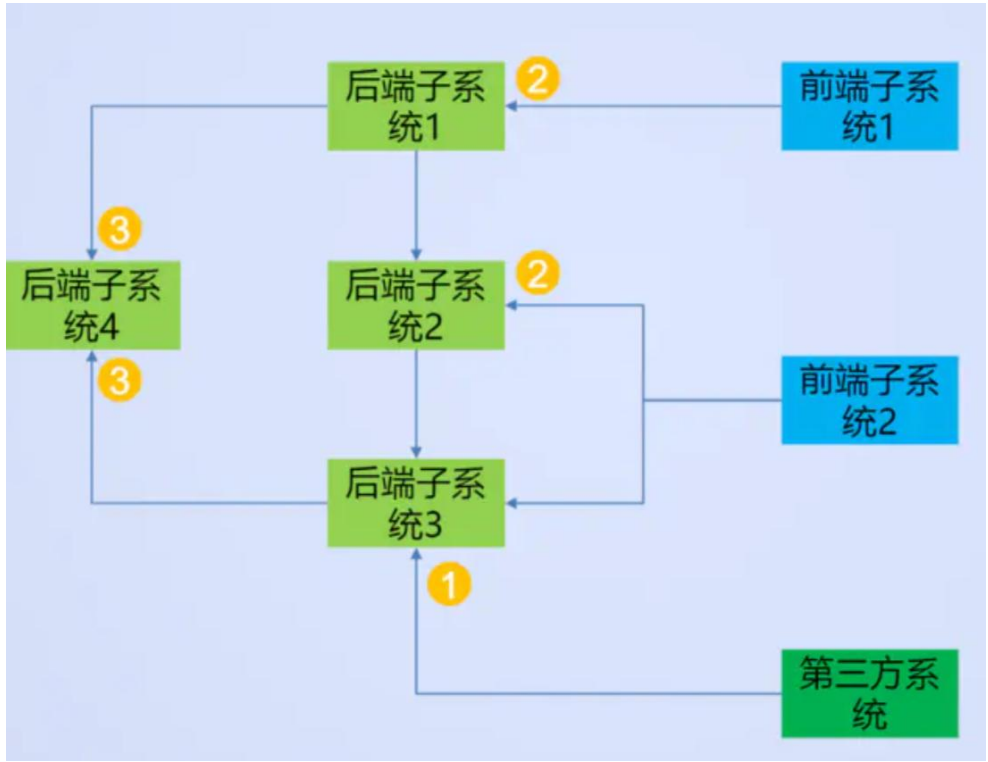
集成测试 - - 何时测试和由谁测?

- 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。
- How
  - 采用黑盒+白盒测试方法，进行基于设计的测试
- 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

API测试

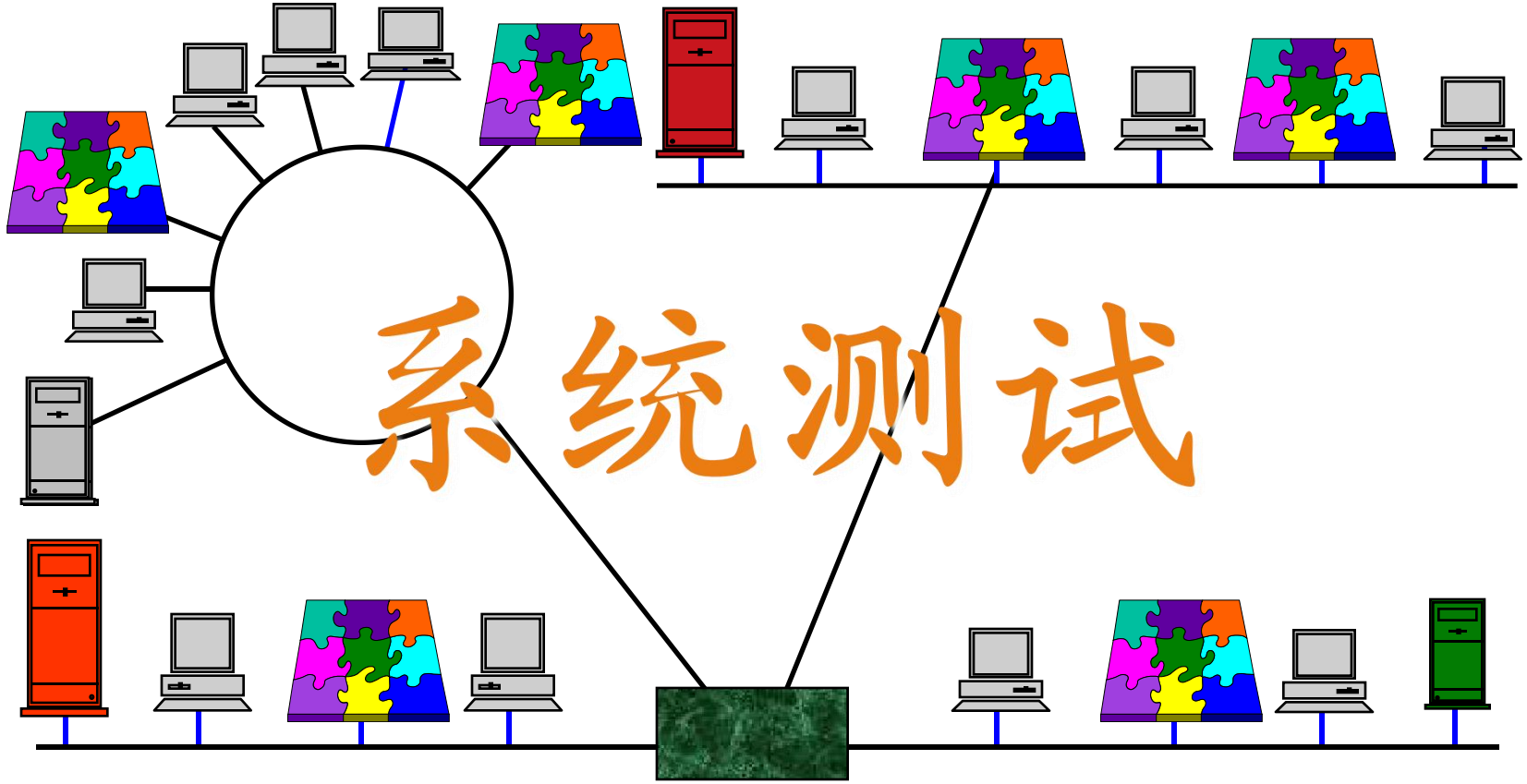
- Best Practice: 基于接口的编程，基于接口的集成测试
- API测试工具: postman、Apifox、SoapUI等
- 示例：C/S系统的API测试





1. 对外为第三方提供的接口
2. 前端调用的后端接口
3. 后端系统内部调用的关键接口
4. 后端系统内部调用的非关键接口

系统测试（system testing）



系统测试 - - 何时测试和由谁测?

- 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。
- 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

系统测试 - - 测什么和如何测?

- 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
    - 功能测试、可靠性测试、性能测试、易用性测试、兼容性测试、信息安全测试等
  - 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。

- How
  - 黑盒测试，进行基于需求规约的测试

第九章 软件测试

大 纲

01-软件测试概述

02-软件测试方法

03-软件测试层次

04-系统测试技术

05-软件测试过程

功能测试（functionality testing）

- 又称为正确性测试或一致性测试
  - 功能正确性测试，测试规约中所有的功能都应实现，而且应是正确的。
    - 采用场景法测试所有用例的所有事件流
    - 采用等价类划分、边界值分析、判定表法、错误猜测法等进行输入输出的测试
  - 准确性测试，测试功能的准确性，例如AI预测的准确性
  - 互操作测试，测试与外界系统进行交互的能力
  - 信息安全测试，对于安全性有较高需求的软件，则须做专门的信息安全测试
- 采用人工测试或自动化测试方法
- 自动化的功能测试工具，又称为回归测试工具，或UI测试工具，例如
  - Web应用的开源Selenium工具
  - App应用的开源Appium工具

示例

- 例如，4S系统的系统测试时
  - 采用人工测试方式进行功能测试
  - 场景法：按其用例模型为每个用例的各个基本流和备选流分别设计了测试用例。
  - 输入输出的测试: 等价类划分、边界值分析、判定表法、错误猜测法

基本流 Basic Flow

1. Customer logs on
2. Customer selects ‘Get Quote’ function
3. Customer selects stock trading symbol
4. Get desired quote from Quote System
5. Display quote
6. Customer gets other quotes
7. Customer logs off

备选流 Alternative Flows

A1. Unidentified Trading Customer
A2. Quote System Unavailable
A3. Quit

场景法：4个测试用例

信息安全测试

- 权限控制测试
  - 测试系统的认证、授权、鉴权和权限控制，确保用户在所授的权限内进行功能的操作和数据的存取。
- 漏洞检测
  - 基于漏洞数据库，通过漏洞扫描工具采用静态和动态手段对系统的安全脆弱性进行检测，发现漏洞。
- 安全攻击
  - 进行威胁建模，找出可以实施渗透攻击的攻击点，进行验证。

信息安全测试

常见的面向Web应用的信息安全测试的测试内容

类型	测试内容
服务器信息测试	运行账号测试；web服务器端口版本测试；HTTP方法测试
文件目录测试	目录遍历测试；文件归档测试；目录列表测试
认证测试	验证码测试；认证错误测试；找回、修改密码测试
会话管理测试	会话超时测试；会话固定测试；会话标识随机性测试
授权管理测试	横向越权测试；纵向越权测试；跨站伪造请求测试
文件上传下载测试	文件上传测试；文件下载测试
信息泄露测试	数据库账号密码测试；客户端源代码测试；异常处理测试
输入数据测试	SQL注入测试；XML注入测试；LDAP注入测试
跨站脚本攻击测试	反射型测试；存储型跨站测试；DOM型跨站测试
逻辑测试	上下文逻辑测试；算术逻辑测试
Webservice测试	Webservice接口测试；Webservice完整性、机密性测试
HTML5测试	CORS测试；Web客户端存储测试；WebWorker安全测试
FLASH安全配置测试	全局配置文件安全测试；浏览器端安全测试
其他测试	Struts2测试；Web部署管理测试；日志审计测试

示例

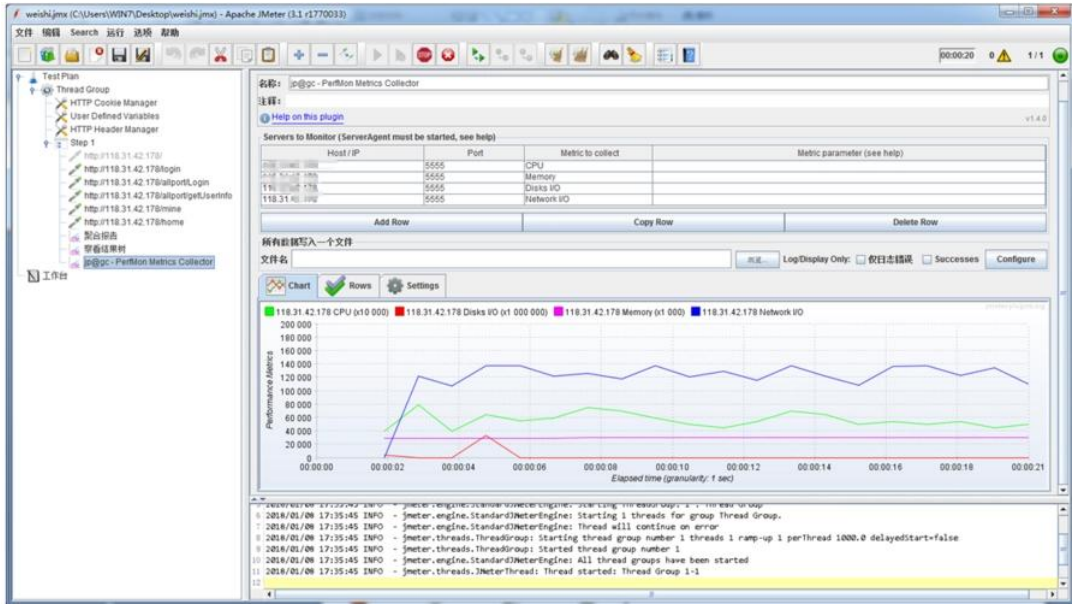
- 例如，4S系统的系统测试时
  - 在功能测试同时检查所有功能是否符合权限控制的要求，即所有授权的动作是否都能做，所有非授权的信息是否都无法看到。
  - 测试者扮演一个试图攻击系统的角色，采用各种方式攻击系统：
    - 截获或破译4S系统的用户名和密码；借助于某种软件攻击4S系统；
    - “制服”4S系统，使得别人无法访问；故意引发系统错误，期望在系统恢复过程中侵入系统；
    - 通过浏览非保密的数据，从中找到进入4S系统的钥匙等等。

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。

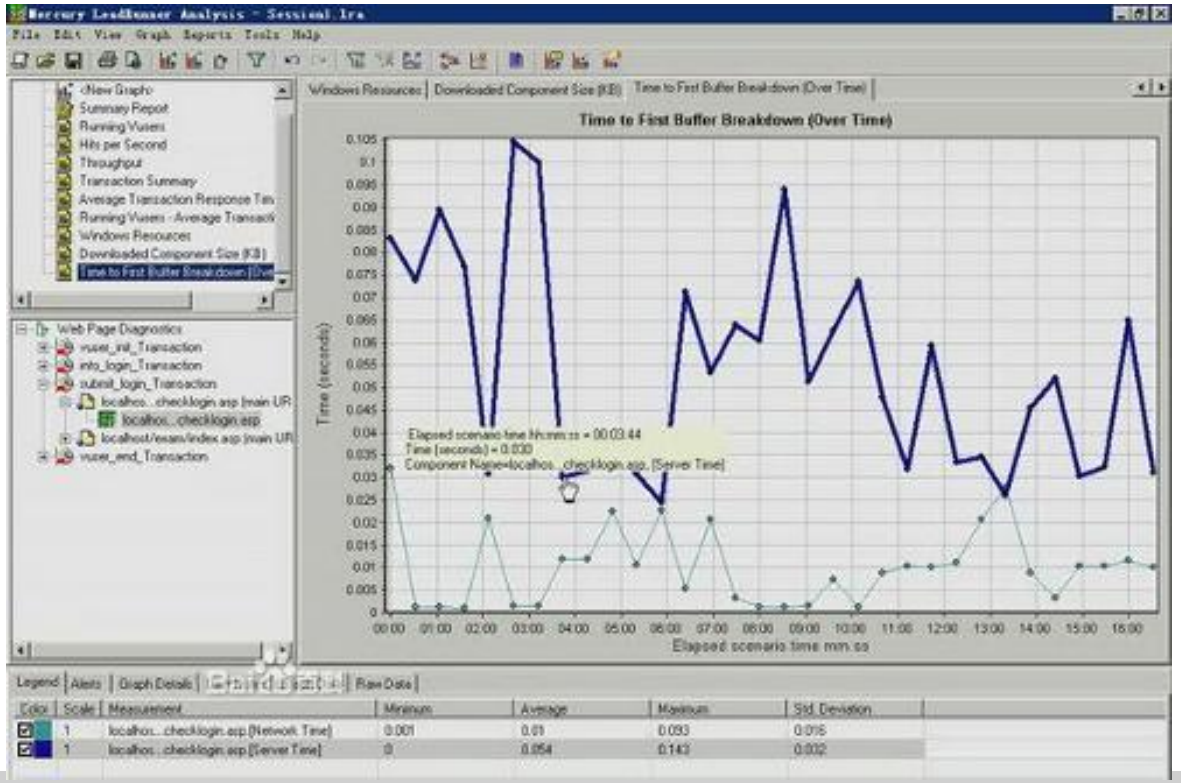
JMeter

- Apache基于java开发的一款开源的性能测试软件（https://jmeter.apache.org/），支持多协议（HTTP/HTTPS、SOAP / REST Webservices、JDBC、TCP等）。它不仅做性能测试，还可以做接口的自动化测试。
- 基本元件
  - 线程组：模拟用户
  - 配置元件：进行测试环境和测试数据初始化
  - 前置处理器：对要发送请求进行预处理
  - 取样器：往服务器发送请求
  - 后置处理器：对收到服务器的响应进行数据提取
  - 断言：将收到响应结果又预期结果做对比
  - 监听器：查看测试脚本运行后结果和日志
  - 定时器：等待一段时间
  - 测试片段：封装基本功能，不单独执行，需要通过脚本的调用才能执行



示例

- 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（50、100、200、500）的并发用户，在正常和超量的数据量情况下，观察系统的响应时间和资源使用情况。



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。
- 可靠性测试关注故障的避免、预防、容错和恢复，它通过自然方式和故障注入方式来触发和激活系统中的故障来测试。

可靠性和可用性度量

- 可靠性使用MTBF度量
  - MTBF（Mean Time Between Failure）是平均故障间隔时间，或称为平均无故障工作时间，指相邻两次故障之间的平均工作时长
- 易恢复性使用MTTR度量
  - MTTR（Mean Time To Recover）是平均故障修复时间
  - 越短的MTTR意味着在规定条件和规定时间内，按规定的程序和方法维修时，系统保持和恢复到规定状态的能力越强
- 可用性计算： MTBF/(MTBF+MTTR)
  - 可用性不仅取决于可靠性，还取决于易恢复性

广义的可靠性包括可靠性和可用性

示例

- 例如，4S系统的系统测试时，
  - 连续运行4S系统100小时，计算系统平均故障间隔时间，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。

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。
- 易用性测试可采用模拟用户的方式进行，也可通过观察用户的操作行为来执行。

示例

- 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。

兼容性测试

- 兼容性测试用以验证软件系统与其所处的上下文环境的兼容情况，即系统在不同环境下，其功能和非功能质量都能够符合要求。
- 面向以下三个兼容性维度
  - 系统内部兼容：系统内部各部件之间的兼容性，包括软件和软件(与其他软件、浏览器、操作系统、数据库系统等)、软件和硬件、硬件和硬件之间的兼容性
  - 系统间兼容：系统与其他系统存在接口互连、功能交互等情况下的配合能力
  - 系统自身兼容：系统的新老版本间需要保证的功能、操作体验等方面的一致性，包括前向兼容和后向兼容

示例

- 4S系统的兼容性测试如下：
  - 在MS Window和Linux两种操作系统的服务器上，进行系统的首次安装、升级、完整的或自定义的安装，测试是否都能成功安装
  - 检查是否能和已安装的杀毒软件等共存
  - 在客户端检查是否用Chrome和Firefox浏览器在规定的分辨率下都能正常使用
  - 检查是否通过人事管理系统能顺利得到员工信息
  - 检查是否能成功卸载

其他测试技术

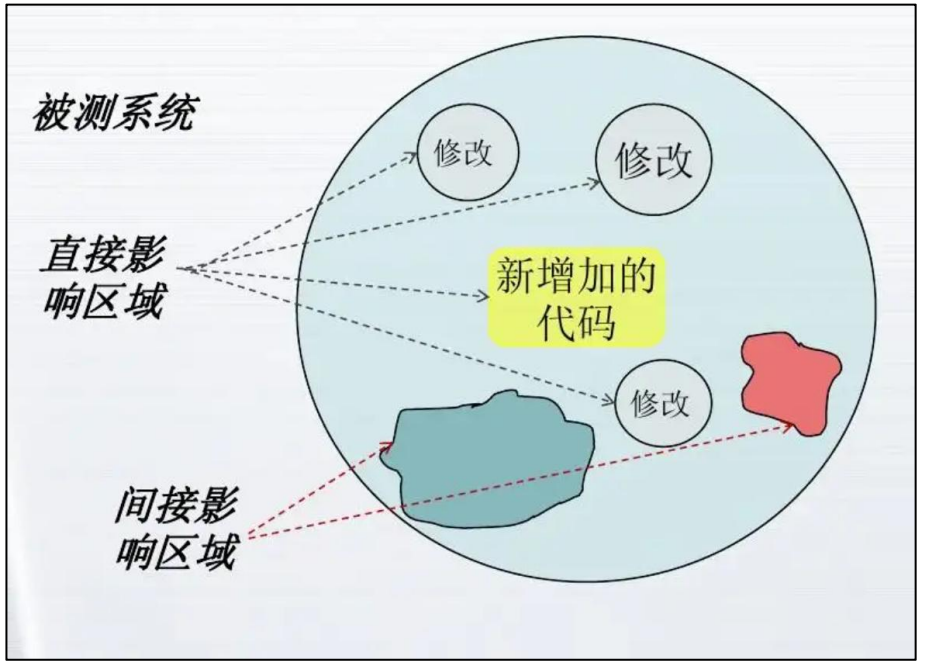
- α测试和β测试
- 回归测试
- AB测试
- 众测

α测试和β测试

- 针对通用软件产品，在企业内部测试通过后，进行α测试和β测试。
- 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。

回归测试（regression testing）

- **目的**：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。
- **时机**：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时
- **方法**：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。
- **工具**：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）。

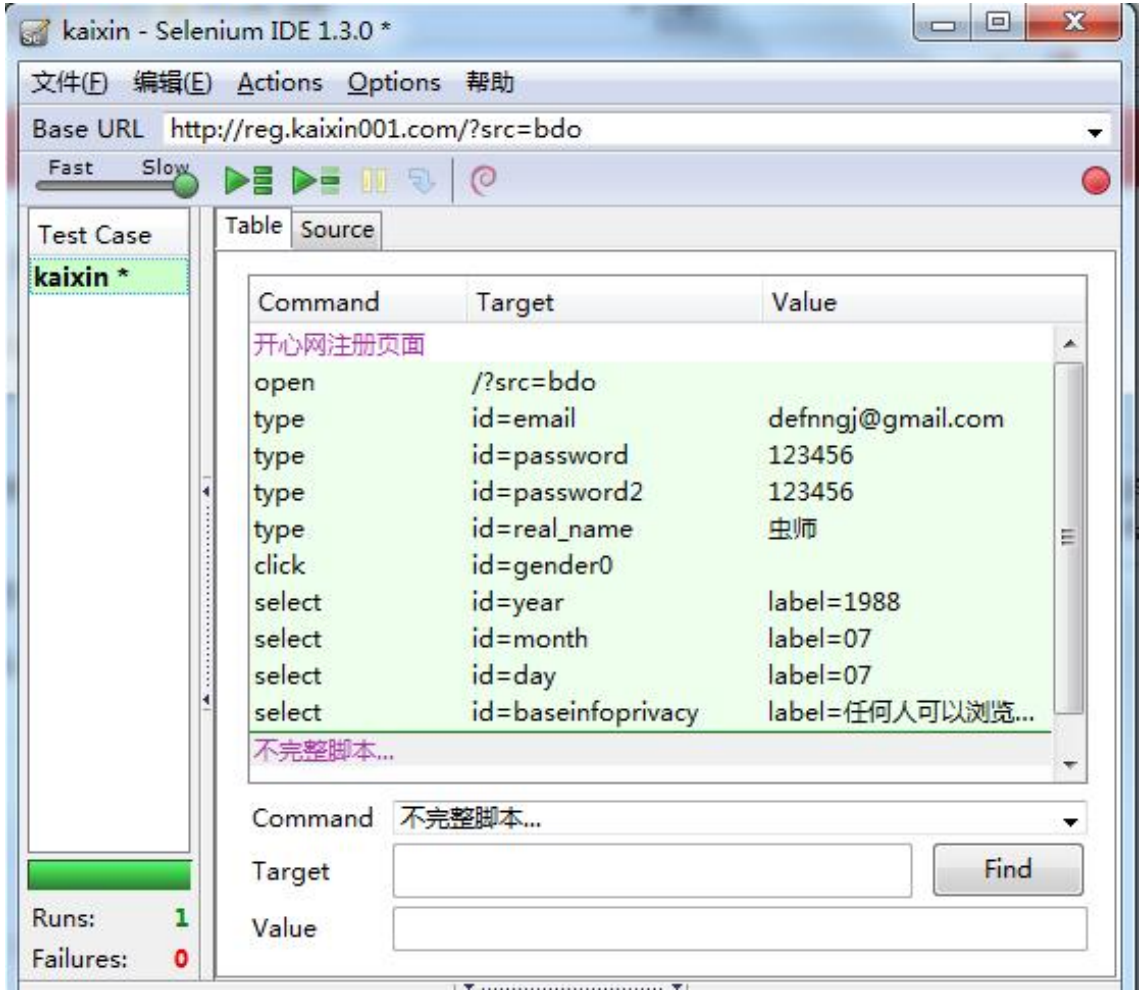


Version 1	Version 2
1. Develop *P*	4. Modify *P* to *P′*
2. Test *P*	5. Test *P′* for new functionality
3. Release *P*	6. Perform regression testing on *P′* to ensure that the code carried over from *P* behaves correctly
	7. Release *P′*

自动化的回归测试

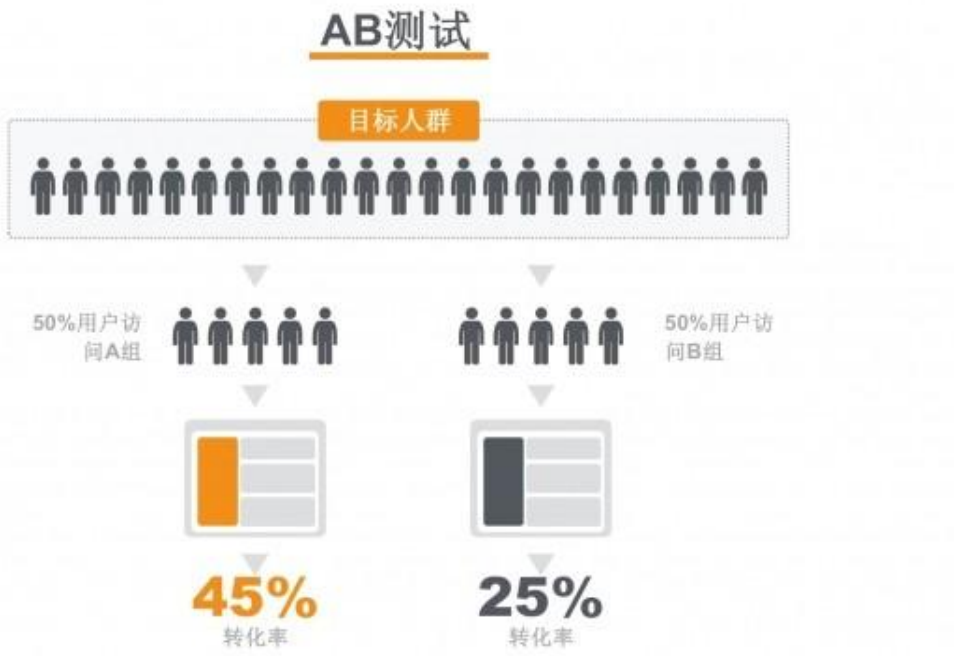
- 回归测试工具使得软件工程师能够捕获到执行过的测试，然后进行回放和比较。

- 工具示例：
  - 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。

第九章 软件测试

大 纲

01-软件测试概述

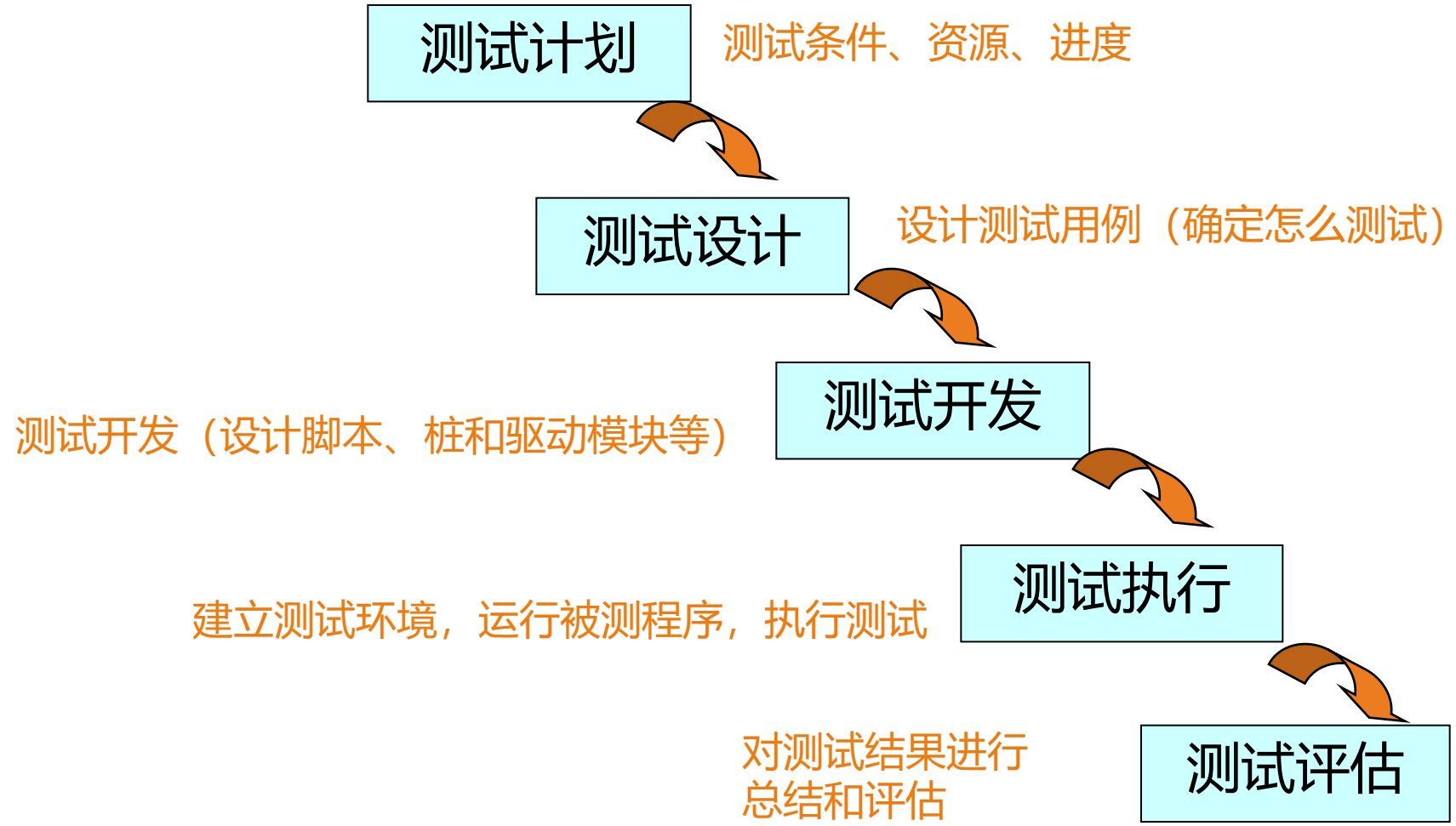
02-软件测试方法

03-软件测试层次

04-系统测试技术

05-软件测试过程

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

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，即通过快速的基本功能测试来表明它已稳定到足以进行后续的正式测试

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

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

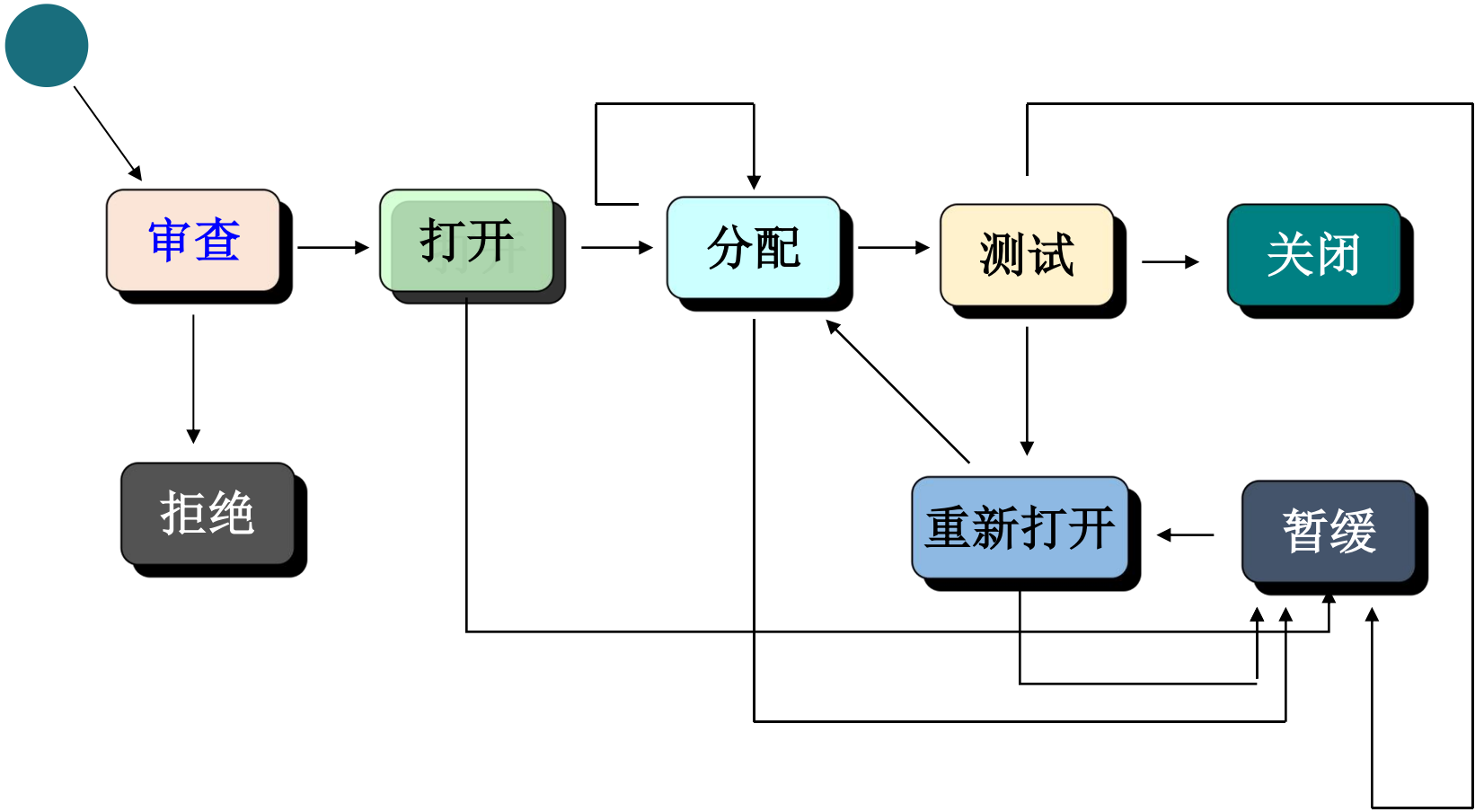
4）测试执行

- 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



缺陷跟踪系统：bugzilla和Jira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。

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

软件测试的度量

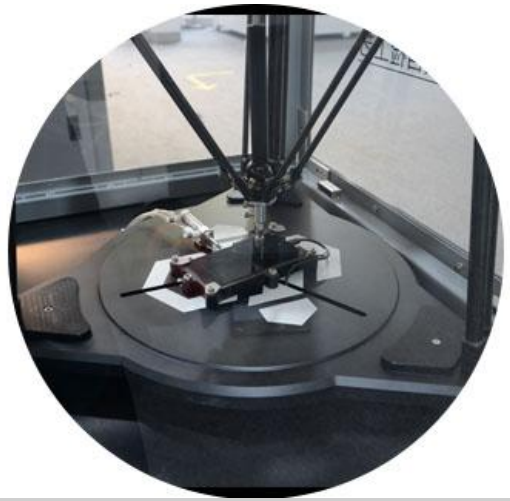
- 软件测试的质量度量，用以评估测试的有效性和、完备性和充分性
  - 例如缺陷逃逸率、测试覆盖率等。
- 被测试产品的质量度量
  - 例如系统的平均响应时间、缺陷密度、测试通过率、缺陷分布等。
- 软件测试过程的效能度量，用以评估测试过程的效率和能力
  - 例如缺陷平均发现成本、测试执行率、缺陷发现率等。
- 缺陷修复过程的效能度量，用以评估的缺陷修复过程的准确性和效率
  - 例如缺陷修复率、缺陷平均修正成本、缺陷重现率等。

总结：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

人工测试 Vs. 自动化测试

- 不能期望自动化测试来取代手工测试，两者须结合执行
- 测试专家James Bach研究得出：85%的缺陷靠人工测试发现，自动化测试能够很好地发现老缺陷。
- 针对复杂的、需要体验、界面美观方面的测试，以人工测试为主。
- 针对简单的、反复执行的、人工成本过高的测试，以自动化测试为主。
- 自动化测试面临技术、组织、脚本维护的问题。



软件测试的后续工作