# 目录

# P1：Bug 历史

# P2：PIE 模型

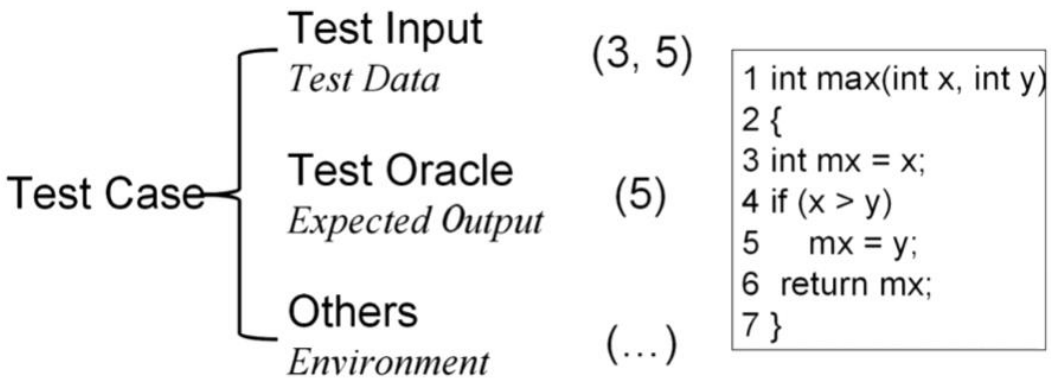
- Software Fault : A **static** defect in the software (i.e., defect)
- Software Error : An incorrect **internal** state that is the manifestation of some faults
- Software Failure : **External**, incorrect behavior with respect to the requirements or other description of the expected behavior

Fault：代码的问题
Error：运行中出现的问题
Failure：传播到外部可观测的结果有问题

PIE:
1. 执行 Execution：存在 fault 的位置被执行到了
2. 感染 Infection：程序状态不对
3. 传播 Propagation：错误的状态导致了输出错误的结果

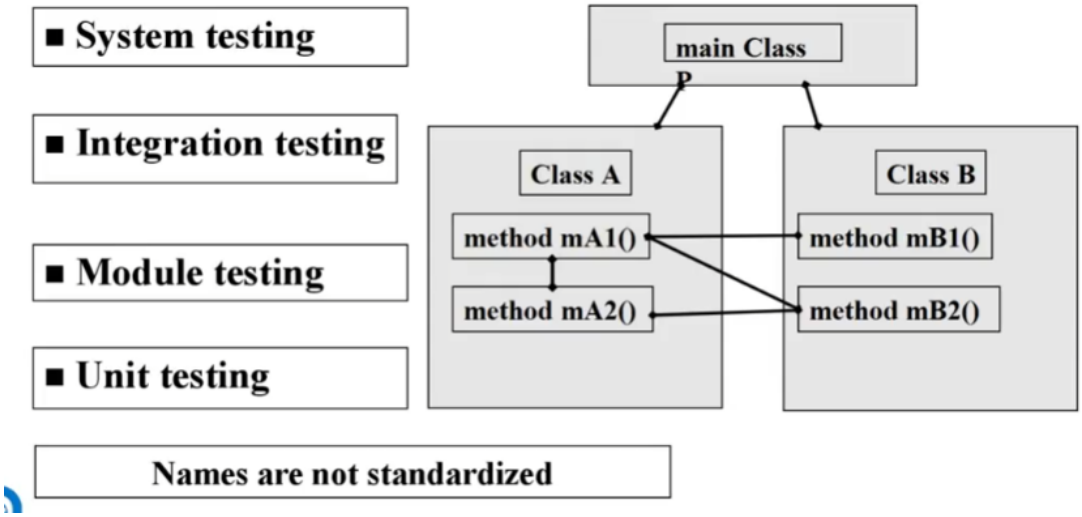
测试不一定能执行到 fault 位置，即使执行到了也不一定触发 Error，有 Error 也可能没 failure

# P3：术语

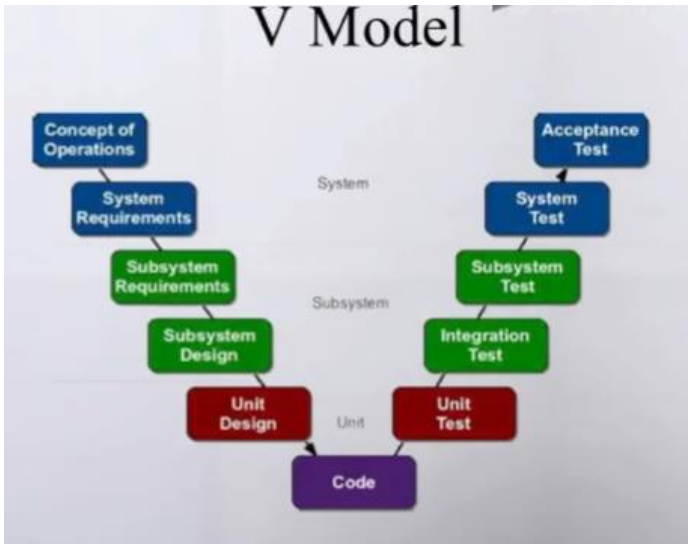


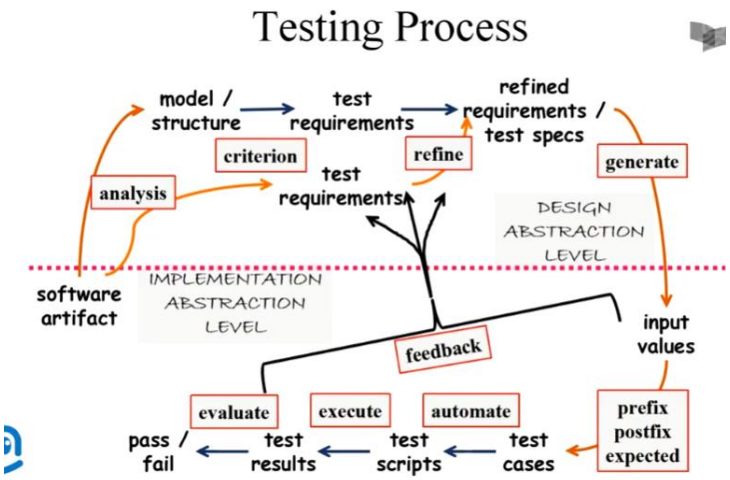

Test 是发现 bug（执行测试，观察 failure），debug 是修复 bug（定位，理解，改正）
Verification&Validation
Verification： 文档和最终实现是否一致

Validation：确认文档是用户想要的
静态测试（严格不能算）、动态测试（本课程重点）
黑盒（不用知道代码），白盒（运行并分析内部结构）
灰盒：其他软件制品、反编译获得部分软件结构进行测试




单元：最小，测试函数方法等
模块：整个模块级的输入输出
集成：模块组合
系统：所有的加起来测试



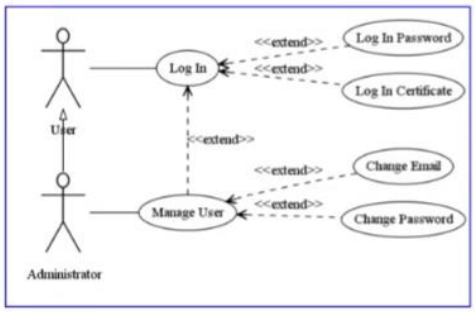



## P4：Fault Revisit

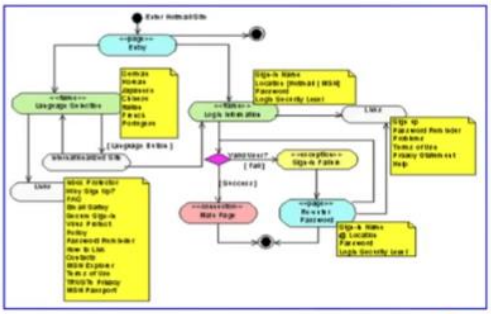
Fault 被修复所定义，修复方式不同，fault 定位也不同，倾向最小修复是 fault
难判定 fault 个数
Fault 之间有干扰
没人知道到底什么是 fault

## P5：图

代码：控制流图、数据流图
规格文档：有限状态机




*Use Case Diagram*



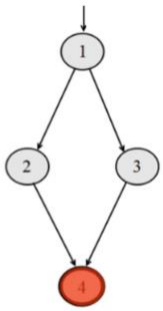
*Activity Diagram*

图：（离散数学中的概念）

## Formal Definition of Graphs

A graph $G$ is defined as follows: $G=(V,E)$

- $V$: a **finite, nonempty** set of vertices
  $V=\{v_1,v_2,v_3,v_4\}$
- $E$: *a set of edges (pairs of vertices)*
  $E=\{(v_1,v_2), (v_1,v_3), (v_2,v_3), (v_3,v_4)\}$
- $V_0$: a set of initial vertices, $V_0=\{v_1\}$
- $V_f$: a set of initial vertices, $V_f=\{v_4\}$



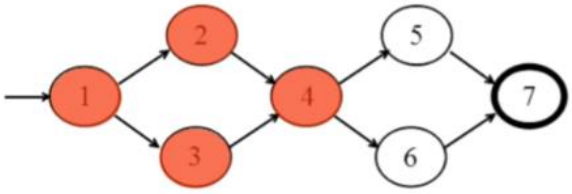

Quiz:

1. E 能否为空集？可以。
2. E 是否无限集？可以是。、

多个初始节点：添加一个亚节点，使之作为初始节点
终结节点同。

路径：路，点和边组成的序列。

- Path : A sequence of vertices – $[v_1,v_2, \ldots,v_n]$
  - Each pair of vertices is an edge




$[V_1,V_2,V_4,V_6]$ ✓

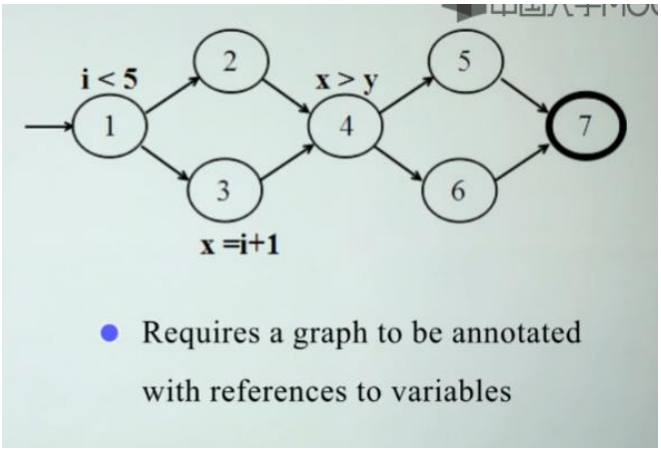
$[V_1,V_2,V_3,V_4]$ ✗

边的数目就是路的长度，单点长度 0.
子路径：子序列
测试路径：初始节点到终结节点的路径
表达测试的执行
相同路径可能会被不同数据多次测试；有的路径没法被任何数据测试（不可证明判定）。
本课程规定一条测试只能执行一条测试路径

- path ($t$) : The test path executed by test $t$
- path ($T$) : The set of test paths executed by the set of tests $T$

# P6：图覆盖准则

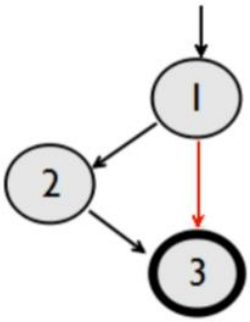
可达
语法可达：图里有路
语义可达：测试可以执行的路
V 在测试路径内，则 v 被覆盖
测试的步骤：
1. 把软件变成图结构
2. 用测试用例覆盖路、子路
结构覆盖：
只关注图的点和边
数据流覆盖：




测试准则 TR：
测试需求：描述测试路径的性质
测试准则：描述测试需求的规则
测试集 T 满足测试准则 C 指对 TR 中的每一个测试要求，T 中都有一个测试用例满足

# P7：结构化覆盖

顶点覆盖：TR：包括每个可达顶点。因此对于所有顶点，测试集中都存在用例能够覆盖
边覆盖：TR：每个测试需求都是一条可达边。对每个可达边都存在测试用例覆盖，这个 T 满足边覆盖。
两者关系：满足 EC 定满足 VC，反过来不一定




Vertex Coverage : $TR = \{ v_1, v_2, v_3 \}$
Test Path = $[v_1, v_2, v_3]$

Edge Coverage : $TR = \{ (v_1, v_2), (v_1, v_3), (v_2, v_3) \}$
Test Paths = $[v_1, v_2, v_3]$
$[v_1, v_3]$

边对覆盖 EPC：覆盖所有的可达边对，即包括所有长度为 2 的可达路
边对：相邻的两条边
CPC 全路径覆盖
N 路径覆盖 nPC：

n-Path Coverage (nPC) : TR contains each reachable path of length up to n, inclusive, in G.

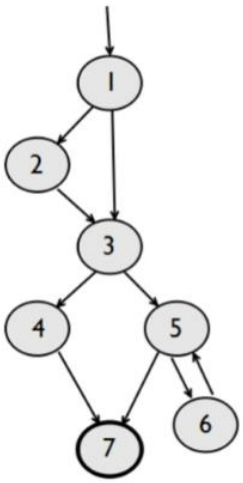
- VC($n$=0), EC($n$=1), EPC($n$=2), CPC($n$=∞)

单点图，边覆盖指覆盖长度<=1 的路径
两个点的图，边对覆盖指覆盖长度<=2 的子路径
nPC 同，<=n 即可

蕴含 subsume
测试准则 C1 蕴含 C2，写作 C1>=C2
指对于任何测试用例集，如果满足 C1，就一定满足 C2
N1PC>=n2PC, if n1 >= n2，但这不表示 C1 检测能力强于 C2

*Structural Coverage Example*






Vertex Coverage
TR = { 1, 2, 3, 4, 5, 6, 7 }
Test Paths: [ 1, 2, 3, 4, 7 ] [ 1, 2, 3, 5, 6, 7 ]

Edge Coverage
TR = { (1,2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 7), (5, 6), (5, 7), (6, 5) }
Test Paths: [ 1, 2, 3, 4, 7 ] [1, 3, 5, 6, 5, 7 ]

Edge-Pair Coverage
TR = {[1,2,3], [1,3,4], [1,3,5], [2,3,4], [2,3,5], [3,4,7], [3,5,6], [3,5,7], [5,6,5], [6,5,6], [6,5,7] }
Test Paths: [ 1, 2, 3, 4, 7 ] [ 1, 2, 3, 5, 7 ] [ 1, 3, 4, 7 ] [ 1, 3, 5, 6, 5, 6, 5, 7 ]

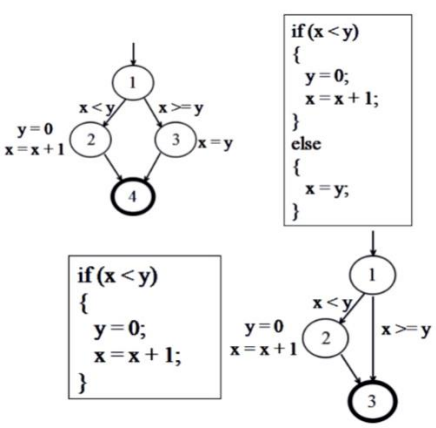
Complete Path Coverage
Test Paths: [ 1, 2, 3, 4, 7 ] [ 1, 2, 3, 5, 7 ] [ 1, 2, 3, 5, 6, 5, 6, 5, 7 ] [ 1, 2, 3, 5, 6, 5, 6, 5, 6, 5, 7 ] …

边对覆盖>=边覆盖>=点覆盖

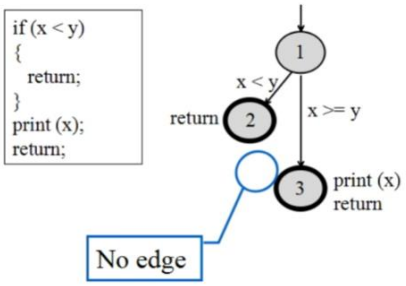
# P8：控制流图

程序执行的流转过程 CFG
顶点：statement，block，function，module
边：flow，jump，call

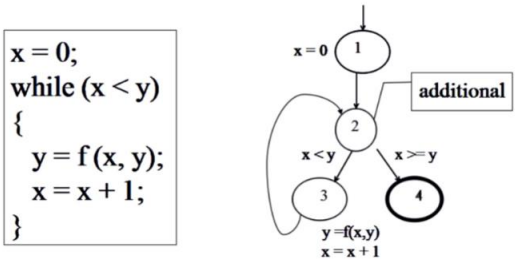
CFG: if




CFG: if-return

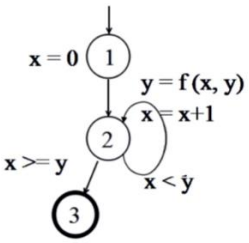



CFG: while

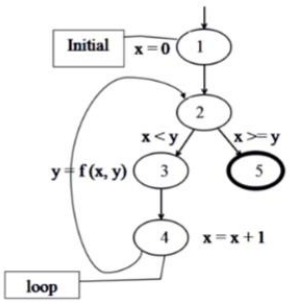


CFG : do

```
x = 0;
do
{
   y = f (x, y);
   x = x + 1;
} while (x < y);
println (y)
```

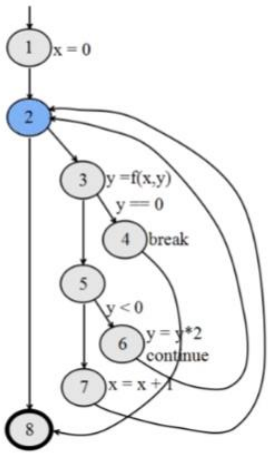



```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

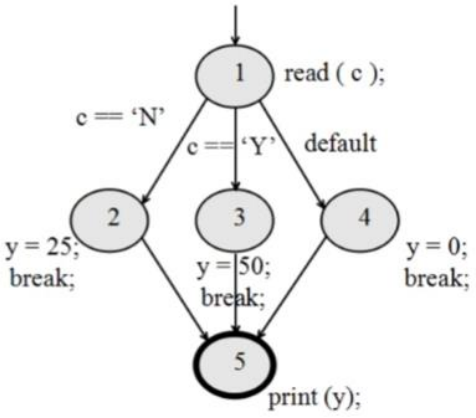



# CFG : break and continue

```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```

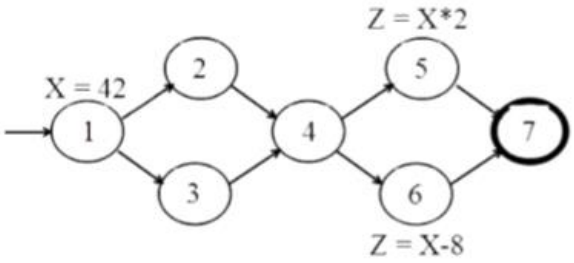


```
read ( c ) ;
switch ( c )
{
  case 'N':
    y = 25;
    break;
  case 'Y':
    y = 50;
    break;
  default:
    y = 0;
    break;
}
print (y);
```



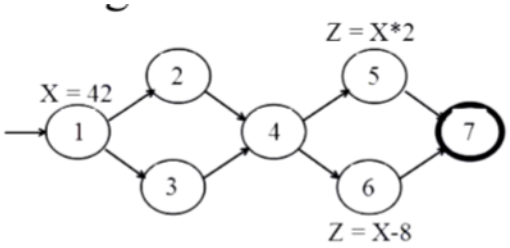

# P9：数据流覆盖

超过结构，关心数据计算使用是否正确
数据流：
定义操作：变量值塞到内存中的地址中（赋值、初始化）
使用：某地址的变量的值被访问（分支、判断、循环等）




Defs: def (1) = {X}
def (5) = {Z}
def (6) = {Z}
Uses: use (5) = {X}
use (6) = {X}

Def(n)节点 n 上所有的定义变量，def(e)边上的定义变量
Use 同
DU pair（li, lj）：v 在 li 定义，在 lj 使用
定义清晰：在 DU pair 中 li 到 lj 中 v 没有被重新定义过
可达：有一条定义清晰的路径从 li 到 lj
DU Path：定义清晰的简单的子路径
Du(ni, nj, v)关于变量 v 的 ni 到 nj 的 DU Path 集合
Du(ni, v)ni 定义，任意地方使用
定义覆盖（ADC）：所有定义的地方都至少覆盖了一次（S=du(n,v)）
引用覆盖（UDC）：所有引用的地方都至少覆盖了一次（S=du(ni, nj, v)）
所有定义引用覆盖（ADUPC）：所有的 DU Path 都覆盖（UDC 只用覆盖一条满足 S 的路径，这个是全部）




| All-defs for $X$ |
|---|
| [ 1, 2, 4, 5 ] |

| All-uses for $X$ |
|---|
| [ 1, 2, 4, 5 ] |
| [ 1, 2, 4, 6 ] |

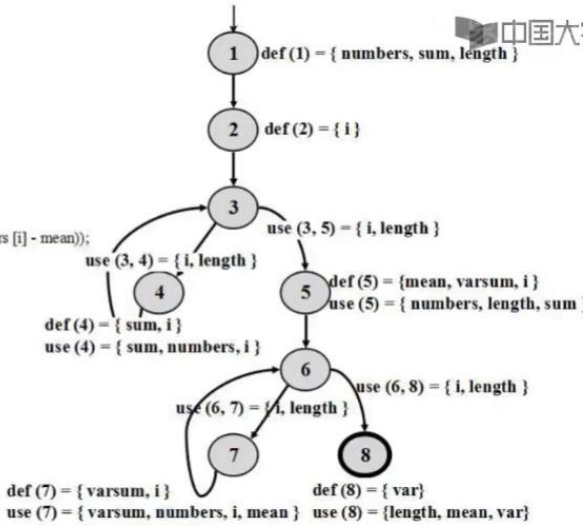
| All-du-paths for $X$ |
|---|
| [ 1, 2, 4, 5 ] |
| [ 1, 3, 4, 5 ] |
| [ 1, 2, 4, 6 ] |
| [ 1, 3, 4, 6 ] |

```
public static void CSta (int [ ] numbers)
{
1  int length = numbers.length;
   double var, mean, sum, varsum;

2  sum = 0.0;
3  for (int i = 0; i < length; i++)
   {
4    sum += numbers [ i ];
   }
   mean = sum / (double) length;

5  varsum = 0.0;
6  for (int i = 0; i < length; i++)
   {
7    varsum = varsum + ((numbers [i] - mean)*(numbers [i] - mean));
   }
   var = varsum / ( length - 1.0 );

8  System.out.println ("length:    " + length);
   System.out.println ("mean:      " + mean);
   System.out.println ("variance:  " + var);
}
```




| Node | Def | Use |
|---|---|---|
| 1 | { numbers, sum, length } | { numbers } |
| 2 | { i } | |
| 3 | | |
| 4 | { sum, i } | { numbers, i, sum } |
| 5 | {mean, varsum, i } | { numbers, length, sum } |
| 6 | | |
| 7 | { varsum, i } | { varsum, numbers, i, mean } |
| 8 | {var} | {length, mean, var} |

| Edge | Use |
|---|---|
| (1, 2) | |
| (2, 3) | |
| (3, 4) | { i, length } |
| (4, 3) | |
| (3, 5) | { i, length } |
| (5, 6) | |
| (6, 7) | { i, length } |
| (7, 6) | |
| (6, 8) | { i, length } |

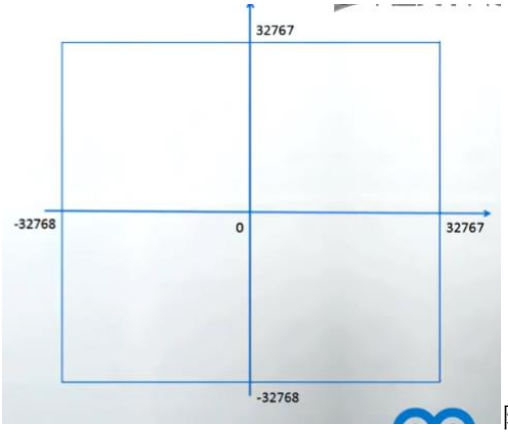
# P10：Junit

白盒测试使用

IDEA、Eclipse

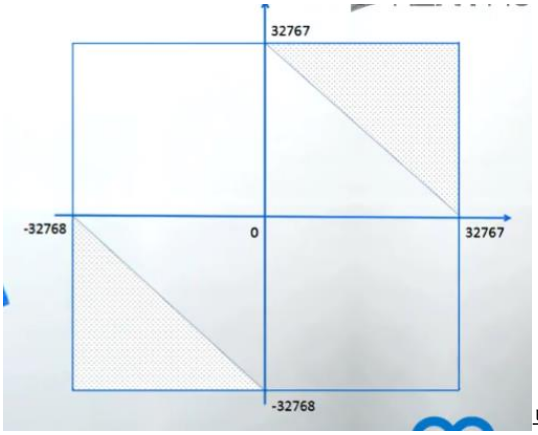
# P11：随机测试

黑盒测试
测试示例完全随机生成，定义输入域，在域中随机选择点，适合自动化



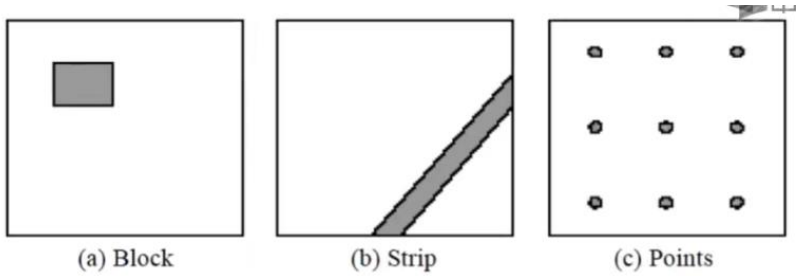
随机选点

定义输入域：分析文档，选择输入域
随机数生成：平方取中等伪随机数，random.org
模糊测试：随机测试的一种，用于软件安全领域

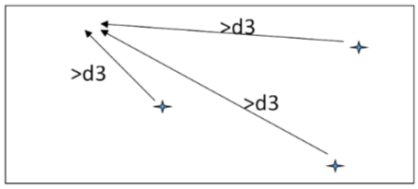


导致程序出错的用例可能聚集

导致程序出错的用例：




因此随机测试应该选较远的（自适应）




FSCS-ART 固定候选集的自适应测试
随机生成一条数据，测试终止条件没满足，那么下一条测试首先生成 k 个候选用例，然后每个用例计算与前一条测试的距离，选距离最大的那条用例，直到测试终止
问题：定义距离，考虑分布（不均匀）
考虑扩大输入域的自适应随机，先人为扩大输入域，再用自适应，最后剪切掉扩大的部分
Anti-Random Testing
离散的输入：先随机选第一个用例，后面算新用例与已有用例的海明距离之和，取和最大的用例

# P12：等价类划分

按照一定标准对输入域进行划分，划分为不同子集，每个子集选代表测试用例
划分时的原则：对不同类型数据的处理进行划分，根据不同数据拥有的控制流、数据流来划分，根据数据合法非法划分
一般考虑合法、非法输入（测试功能和异常）
完备性原则：指输入域中任何点或区域都应该属于某一类，不能有遗漏，输入域中不能有任何点，它不属于任何等价类
无冗余性原则：任意两个等价类间不存在交集
划分方法：
输入域范围：一个有效，两个无效

E. g.: Input is integer (10<x<100), two invalid classes are x≤10 and x≥100。

不同数据的不同处理情况：n 种不同的数据有 n 种不同方法来处理，则有 n 个有效，1 个无效

E.g. The value of input x could be {1,3,7,15}. Program process four values with four different methods. Valid classes include x=1, x=3, x=7, x=15; invalid class is the set that x≠1, 3, 7, 15。

按输入的条件进行划分：输入有个条件那么 1 个有效 1 个无效，多种条件就是 1 个有效和 n 个无效

E.g. The value of input variable must be an odd integer. The valid class is the set of all odd integer, and invalid class is the set of all the others.

E.g.: An input string with length 8 must begin with ‘a’ and consist of characters ‘a’~ ‘z’. The valid class is the set of strings that satisfy all conditions. And there are at least 3 invalid classes.

输入

quivalence Partitioning for Triangle

| | Class 1 | Class 2 | Class 3 |
|---|---|---|---|
| Relationship between 1st edge and 0 | >0 | =0 | <0 |
| Relationship between 2nd edge and 0 | >0 | =0 | <0 |
| Relationship between 3rd edge and 0 | >0 | =0 | <0 |

输入细化

| | Class 1 | Class 2 | Class 3 | Class 4 |
|---|---|---|---|---|
| Relationship between 1st edge and 0 and 1 | >1 | =1 | = 0 | <0 |
| Relationship between 2nd edge and 0 and 1 | >1 | =1 | = 0 | <0 |
| Relationship between 3rd edge and 0 and 1 | >1 | =1 | = 0 | <0 |

功能

| | Class 1 | Class 2 | Class 3 | Class 4 |
|---|---|---|---|---|
| Types of triangle | Normal | Isosceles | Equilateral | Invalid |

# P13：边界值划分

Equivalent classes

Class(i) x<0

Class(ii) x>=0

} Need 2 test cases

Boundary values

Boundary values of (i) are 0 and -MAX

Boundary values of (ii) are 0 and +MAX

Test cases

- -MAX
- -MIN
- 0
- +MIN
- +MAX

| | Boundary | Test cases |
|---|---|---|
| string | Min length-1 / Max length+1 | A input string with 1 ~ 255 character:<br>Valid inputs: 1 character, 255 character<br>Invalid input: 1 character, 256 character |
| int | MIN-1 / MAX+1 | 16 bit unsigned integer:<br>Valid inputs: 0, 65535<br>Invalid input: -1, 65536 |
| loop | loop number | for(int i=0;i<n;++i)<br>for(int i=0;i<=n;++i)<br>for(int i=1;i<n;++i) |

| | range |
|---|---|
| bit | 0 or 1 |
| byte | 0 ~ 255 |
| word | 0~65535(16bit) or 0~4294967295(32bit) |
| K | 1024 |
| M | 1048576 |
| G | 1073741824 |

如何分析?

等价类范围[min, max]

先选一个代表性的值 nom，然后在边界上取 min，min+（略大于 min），max，max-，非法输入 min-，max+

Example: Two input variables x1 (a≤x1≤b) and x2(c≤x2≤d).
Test cases include:

<x1nom,x2min>, <x1nom,x2min+>, <x1nom,x2nom>,
<x1nom,x2max>, <x1nom,x2max->, <x1min,x2nom>,
<x1min+,x2nom>, <x1max,x2nom>, <x1max-,x2nom>

# P14：组合测试

决策表，考虑了输入输出的联系
多输入程序：等价类划分后完全组合数量太多
在此基础上抽样，两两组合（Pair-wise Testing）
T-wise
可变粒度的组合测试

Interaction Relationship:
R={{Input A, Input B, input C},
{input A, input D},
{input C, input D}}

Test cases

| Input A | Input B | Input C | Input D |
|---|---|---|---|
| A1 | B1 | C1 | D1 |
| A1 | B1 | C2 | D2 |
| A1 | B2 | C1 | D3 |
| A1 | B2 | C2 | D3 |
| A2 | B1 | C1 | D2 |
| A2 | B1 | C2 | D1 |
| A2 | B2 | C1 | D2 |
| A2 | B2 | C2 | D1 |

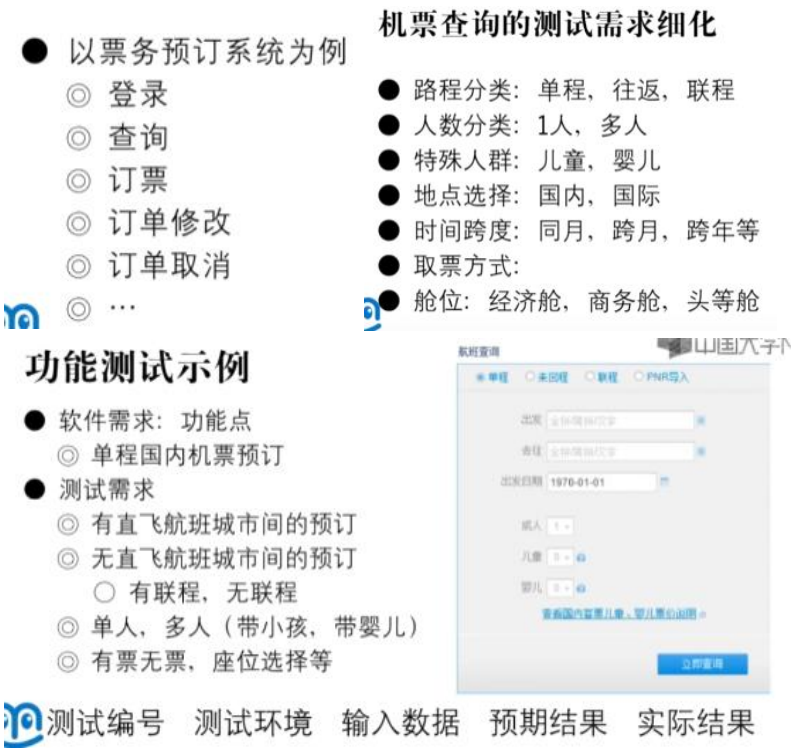
# P15：功能测试

根据产品特性设计需求，验证产品是否满足
常用步骤：
1. 根据需求细分功能点
2. 根据功能点派生测试需求
3. 根据测试需求设计测试用例
4. 逐项执行用例

关于用户界面：

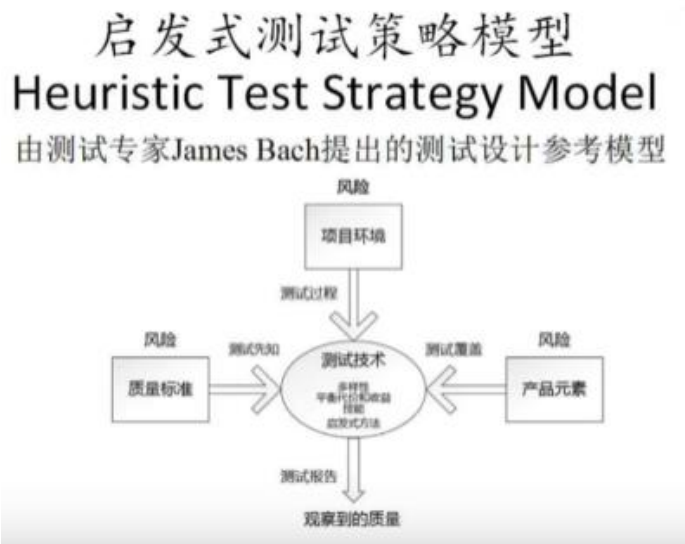
- 非图形化用户界面
  - ◎ 命令行
- 图形化用户界面（GUI）
  - ◎ 桌面：Windows风格，单文档、多文档、资源管理器等
  - ◎ Web：Html元素等，静态页面，动态页面
  - ◎ 移动设备：多触点交互，传感器等

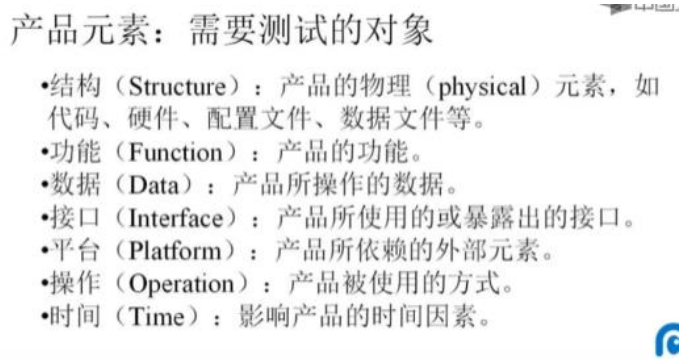
正确性：功能是否与需求、设计文档一致
可靠性：交互是否会引发异常
易用性：完成特定任务的容易成都



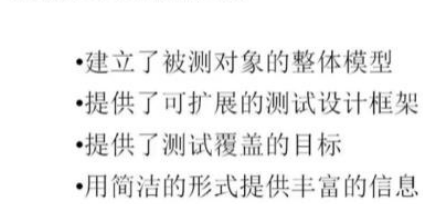

# P16：探索式测试

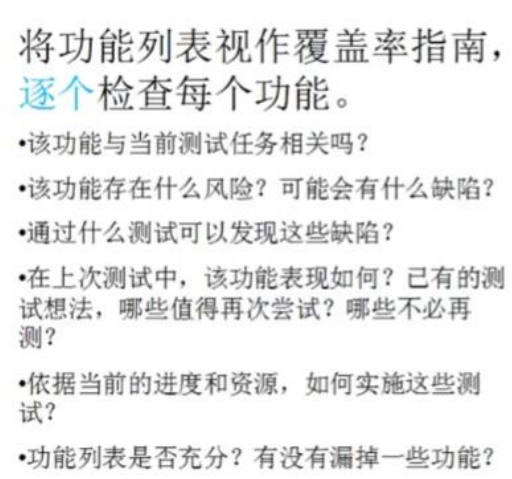
软件测试风格
强调独立测试人员的个人自由和职责
将**测试学习、设计、执行、结果分析**作为相互支持的活动
项目过程中并行执行






关注价值（产品为用户提供更多价值）、风险驱动（损害价值，降低用户体验）
测程（Session）：任务、潜在目标





# P17：性能测试

验证性能在特定负载和环境下能否满足指标
进一步发现瓶颈，优化系统
度量方法不同：
服务端：CPU、内存
客户端：用户响应时间
响应时间：系统对请求作出响应所需时间
服务端响应时间：请求发出到客户端接收最后一个字节所耗时间
客户端响应时间：客户端收到响应数据后响应用户所耗时间
并发用户数取决于测试对象的目标业务场景，先确定业务场景再用相应方法计算并发用户数
与同时在线数有区别，在线不一定施加压力
吞吐量指单位时间内处理的用户请求数量，人数/天，页面数/秒，请求数/秒……

性能计数器：描述性能的一些指标，与系统配置、架构、开发方式有密切联系
负载测试是性能测试一种，验证系统在正常负载下的行为
性能行为通过一些指标体现
方式：直接到达负载数、逐步增加负载数
压力测试评估系统超过预期负载的行为
关注的不一定是性能行为，还有 bug 例如同步、内存泄漏
压力增加时性能应该下降但不应该崩溃

## P18：Jmeter1

脚本、事务、参数化

## P19：Jmeter2

集合点、断言（Bean Shell）、测试结果展示

## P20：移动应用测试

与传统软件测试区别：关注移动
主要是安卓手机，但碎片化，只能覆盖市场占有率高的
传感器和屏幕碎片化
一般先从模拟器上开始测试
众包测试：分配给非专业测试人员