开发者测试介绍

MOOCTEST.net

开发者测试

软件的持续快速迭代需求，大大压缩了软件开发的发布流程，使得一部分测试任务开始迁移，由软件开发人员担任这部分跟代码相关的软件测试工作，我们统称为开发者测试。开发者测试包括了传统的单元测试、集成测试、接口测试甚至部分系统测试相关的任务。

开发者需要对自己开发的程序代码承担质量责任。在软件质量管理机制下，一 般要求开发者首先自行对自己编写的代码进行审查和测试，保证提交的代码必须达 到一定的质量标准。开发者测试中的单元测试和集成测试，主要采用白盒测试方法，要求做测试的人员对软件代码非常熟悉。这样的测试任务，由软件开发人员开发者来做效率会更高。

开发者测试

开发者测试

为了更全面的评价测试脚本的质量，我们从（1）测试代码覆盖率、（2）Bug 检测率、（3）脚本可维护性、（4）脚本运行效率、（5）脚本编写 效率，五个方面生成关于测试脚本的测试能力评估及其雷达图。

（1） 代码覆盖率：直接采用指定的某种代码覆盖率（语句覆盖、分支覆盖等）， 最小为 0 分，最大为 100 分（代表 100%）。

（2） Bug 检测率：以本次任务最大（以后可以记录历史最大）的变异杀死率最 为 100 分能力值计算。例如最大变异杀死率是 90%，选手的杀死率是 60%，则选手得分为 60/90=66.7 分。

开发者测试

（3）脚本可维护性：按照阿里或其他知名公司的风格要求，采用 checkstyle 计 算相应的满足项。例如，检查项 5 项，每项满分 20，选手某单项出错一 次扣 2 分，单项扣完为止。

（4） 脚本运行效率：采用代码覆盖率除以运行时间，最大值记为 100 分，其他 选手成绩采用线性化归一计算。例如，假设覆盖率 80%/100 秒=0.8 为最 大值，先记为 100 分。选手 A 覆盖率 40%，运行时间 80 秒，则 40/80=0.5，0.5/0.8*100=62.5 分；选手 B 覆盖率 100%，运行时间 200 秒，则 100/200=0.5，也是 62.5 分。

开发者测试

（5） 脚本编写效率：采用最高代码覆盖率除以最快达到这一覆盖率的编写时间， 最高者记为 100 分，其他选手成绩采用线性化归一计算。例如，假设覆盖率 80%/100 分钟=0.8 为最大值，先记为 100 分。选手 A 在第 80 分钟第一 次达到他个人最高覆盖率 40%，则 40/80=0.5，0.5/0.8*100=62.5 分；选手B 在第 200 分钟第一次达到最高覆盖率 100%，则 100/200=0.5，则能力为62.5 分。

备注：假如一次考试或比赛有多道题，则覆盖率，检测率，时间，检查项等累加 计算后进行相关除法运算。

分支覆盖

分支覆盖要求程序中每个条件判定语句的真值结果和假值结果都至少出现一次

每个判断的真值结果和假值结果都至少出现一次，相当于每个判断的 真分支和假分支至少执行一次

分支覆盖不仅考虑了各个条件判定语句的覆盖需求，还考虑了这些语句分支的覆盖需求，因而较语句覆盖测试强度更高

分支覆盖

- 程序 P1 包含 两条条件判定语句，这就要求相关的真假分支 ④、⑤、⑦、⑧ 至少被 执行一次

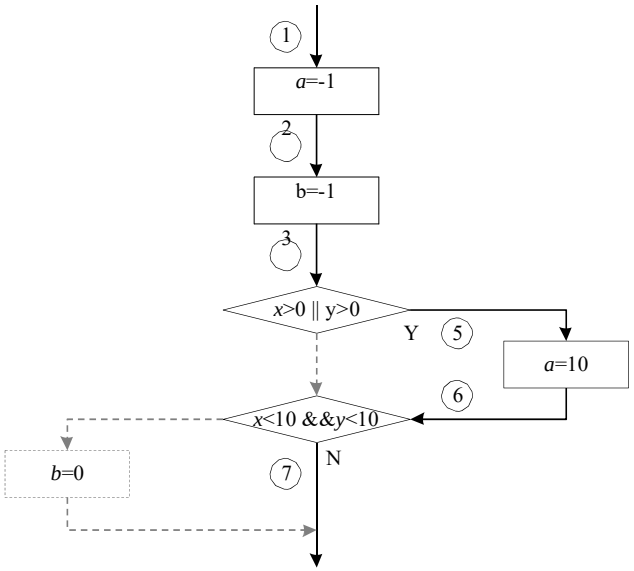


图 1(a) P1在t4=(20, 20)下的程序流程图

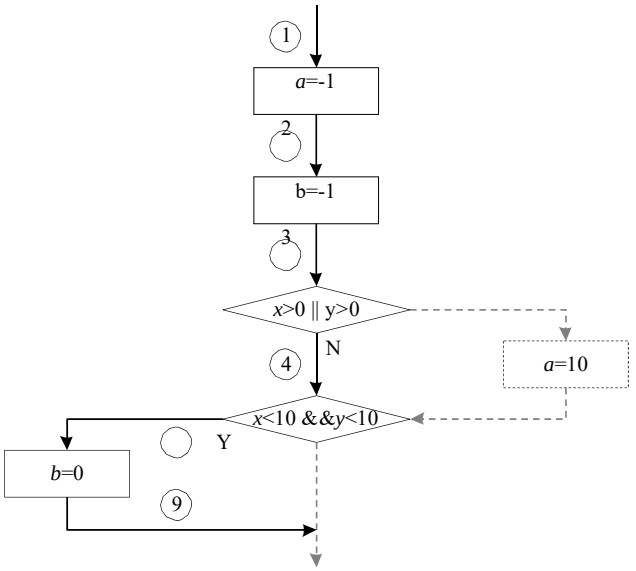


图 1(b) P1 在t5 =(-2, -2)下的程序流程图

分支覆盖

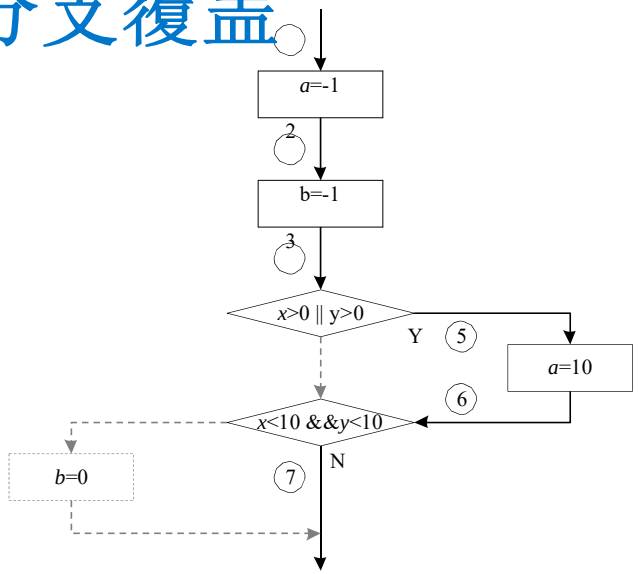


图 2(a) P1在t4=(20, 20)下的程序流程图

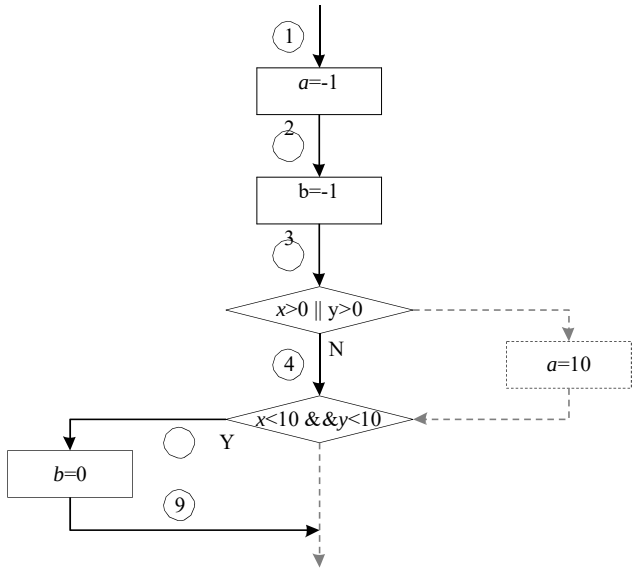


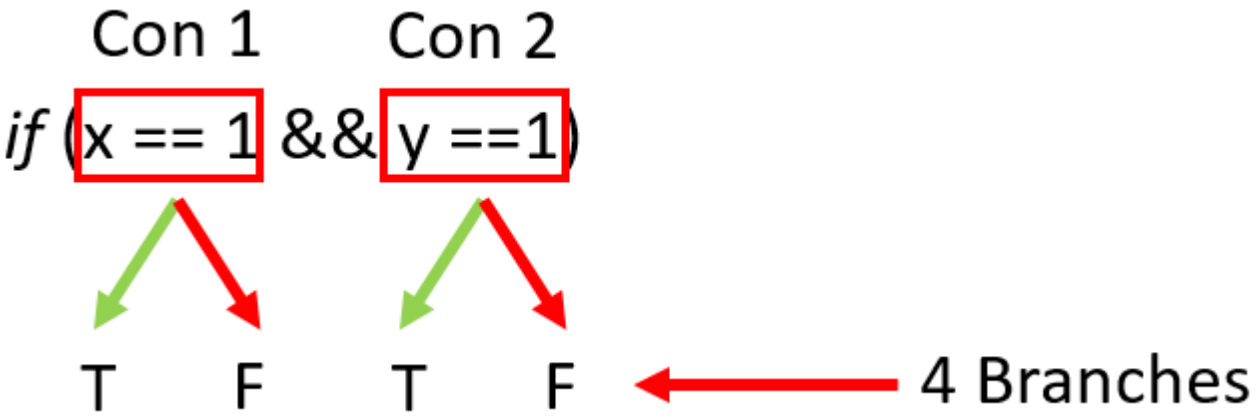
图 2(b) P1 在t5 =(-2, -2)下的程序流程图

表 1 P1在测试用例t_4-t_5下的分支覆盖表

测试用例	x	y	分支覆盖结果	
			x > 0 || y > 0	x < 10 && y < 10
t_4	20	20	⑤	⑦
t_5	-2	-2	④	⑧

分支覆盖

基于字节码的分支覆盖



分支覆盖

- 分支覆盖得分（Branch Score）是一种评价测试用例集充分检测有效性的度量指标
- 分支覆盖得分 $score_{\text{branch}}$ 的值介于 0 与 1 之间，数值越高，表明覆盖的分支数越多，测试用例集覆盖的流程越多，反之则越少
  - 当$score_{\text{branch}}$的值为 0 时，表明测试用例集没有覆盖任何一个分支
  - 当$score_{\text{branch}}$的值为 1 时，表明测试用例集覆盖了所有分支

$$score_{\text{branch}} = \frac{num_{\text{covered}}}{num_{\text{total}}}$$

变异测试

- 变异测试也称为“变异分析”，是一种对测试数据集的有效性、充分性进行评估的技术，能为研发人员开展需求设计、单元测试、集成测试提供有效的帮助
- 变异测试通过对比源程序与变异程序在执行同一测试用例时差异来评价测试用例集的错误检测能力
- 在变异测试过程中，一般利用与源程序差异极小的简单变异体来模拟程序中可能存在的各种缺陷

变异测试

变异测试应用场景

- 在软件测试时，若当前测试用例未能检测到软件缺陷，则存在两种情形：

  ① 软件已满足预设的需求，软件质量较高；

  ② 当前测试用例设计不够充分，不能有效检测到软件中的缺陷。

- 逻辑测试和路径测试方法，分别从程序实体覆盖和路径覆盖的角度来评估软件测试的充分性。然而，这些方法并不能直观地反映测试用例的缺陷检测能力

- 变异测试方法可用于度量测试用例缺陷检测能力

变异测试

变异测试的基本假设

- 变异测试的可行性主要基于两点假设：
  - 熟练程序员假设，关注于熟练程序员的编程行为；
  - 变异耦合效应假设，关注于变异程序的缺陷类型。
- 熟练程序员假设是指熟练程序员由于开发经验丰富、编程水平较高，其编写的代码即使包含缺陷，也与正确代码十分接近。此时，针对缺陷代码仅需做微小的修改即可使代码恢复正确
- 变异耦合效应假设是指若测试用例能够杀死简单变异体，那么该测试用例也易于杀死复杂变异体

变异测试

变异的定义

- 程序变异指基于预先定义的变异操作对程序进行修改，进而得到源程序变异程序（也称为变异体）的过程
  - 当源程序与变异程序存在执行差异时，则认为该测试用例检测到变异程序中的错误，变异程序被杀死
  - 当两个程序不存在执行差异时，则认为该测试用例没有检测到变异程序中的错误，变异程序存活

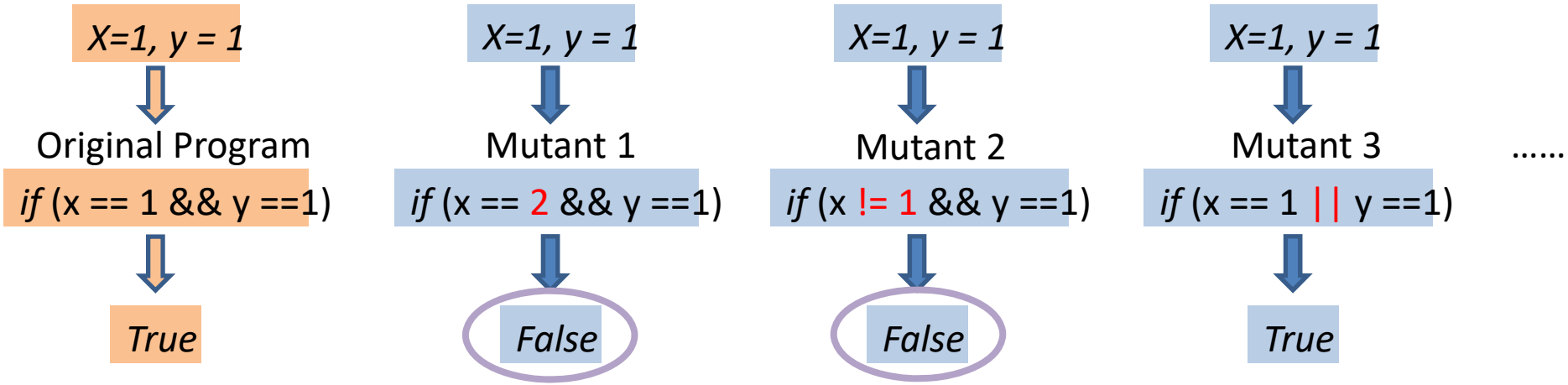
变异测试

变异算子	描述
运算符变异	对关系运算符 “<”、“<=”、“>”、“>=” 进行替换，如将 “<” 替换为 “<=”
	对自增运算符 “++” 或自减运算符 “--” 进行替换，如将 “++” 替换为 “--”
	对与数值运算的二元算术运算符进行替换，如将 “+” 替换为 “-”
	将程序中的条件运算符替换为相反运算符，如将 “==” 替换为 “!=”
数值变异	对程序中整数类型、浮点数类型的变量取相反数，如将 “i” 替换为 “-i”
方法返回值变异	删除程序中返回值类型为void的方法
	对程序中方法的返回值进行修改，如将 “true” 修改为 “false”

变异测试

变异算子	描述
继承变异	增加或删除子类中的重写变量
	增加、修改或重命名子类中的重写方法
	删除子类中的关键字super，如将 “return a*super.b” 修改为 “return a*b”
多态变异	将变量实例化为子类型
	将变量声明、形参类型改为父类型，如将 “Integer i” 修改为 “Object i”
	赋值时将使用变量替换为其它可用类型
重载变异	修改重载方法的内容，或删除重载方法
	修改方法参数的顺序或数量

变异测试

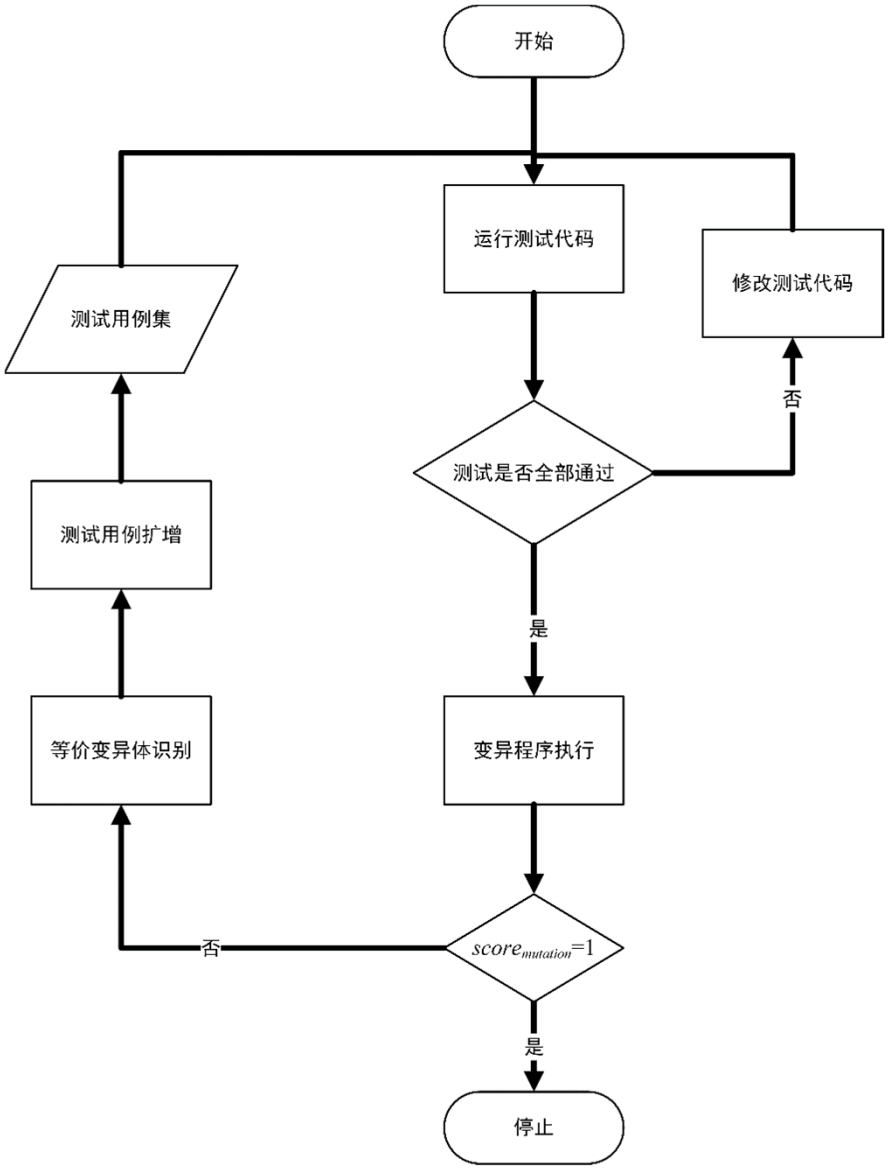

X=1, y = 1
Original Program
if (x == 1 && y ==1)
True
X=1, y = 1
Mutant 1
if (x == 2 && y ==1)
False
X=1, y = 1
Mutant 2
if (x != 1 && y ==1)
False
X=1, y = 1
Mutant 3
if (x == 1 || y ==1)
True
……

变异测试

变异测试过程

- 在变异测试过程中，程序与变异程序的执行差异主要表现为以下两个情形：
  - (1) 执行同一测试用例时，源程序和变异程序产生了不同的运行时状态
  - (2) 执行同一测试用例时，源程序和变异程序产生了不同的执行结果
- 根据满足执行差异要求的不同，可将变异测试分为弱变异测试 (Weak Mutation Testing) 和强变异测试 (Strong Mutation Testing)
- 在弱变异测试过程中，当情形 (1) 出现时就可认为变异程序被杀死，而在强变异测试过程中，只有情形 (1) 和 (2) 同时满足才可认为变异程序被杀死

变异测试



变异测试基本流程图

变异测试

等价变异体

若变异体 p′ 与原有程序 p 在语法上存在差异， 但在语义上与 p 保持一致， 则称p′ 是 p 的等价变异体。

```
for(int i=0; i<10; i++) {
    if(i*2 <= currentSize && array[i]>array[2*i]) {
        return false;
    }
}
```

```
for(int i=0; i!=10; i++) {
    if(i*2 <= currentSize && array[i]>array[2*i]) {
        return false;
    }
}
```

变异测试

- 变异得分（Mutation Score）是一种评价测试用例集错误检测有效性的度量指标
- 变异得分 $score_{mutation}$ 的值介于 0 与 1 之间，数值越高，表明被杀死的变异程序越多，测试用例集的错误检测能力越强，反之则越低
  - 当$score_{mutation}$的值为 0 时，表明测试用例集没有杀死任何一个变异程序
  - 当$score_{mutation}$的值为 1 时，表明测试用例集杀死了所有非等价的变异程序

$$score_{\text{mutation}} = \frac{num_{\text{killed}}}{num_{\text{total}} - num_{\text{equivalent}}}$$

算分核心

```
{
  "categories": [       //题目中包含的覆盖点的全部类别，可自行定义
    {
      "name": "SC"
    },
    {
      "name": "MUTATION"
    }
  ],
  "nodes": [            //覆盖点数据
    {
      "name": "statement1",     //必选 覆盖点的名称，用于展示；作为覆盖点的唯一标识，不允许重复
      "description": "",        //可选 覆盖点的描述，仅用于展示 黄勇
      "location": "Triangle/src/Triangle.java/28",  // 必选 覆盖点的定位，用于在评分时判断该点是否被覆盖
      "category": "SC",         //必选 覆盖点的类别，与categories数组中的类别名称对应
      "weight": 1,              //可选 表示该节点的在覆盖中的权重，缺省为1
      "img": "",                //可选 覆盖点的图片说明，仅用于展示
      "extra": ""               //可选 附加信息
    }
  ],
  "edges": [            //覆盖点之间的关系
    {
      "source": "statement1",   //有向边，以覆盖点的name标识
      "target": "statement2"
    },
    {
      "source": "statement2",
      "target": "statement3"
    }
  ]
}
```

软件定义世界

质量保障未来

MOOCTEST.net