

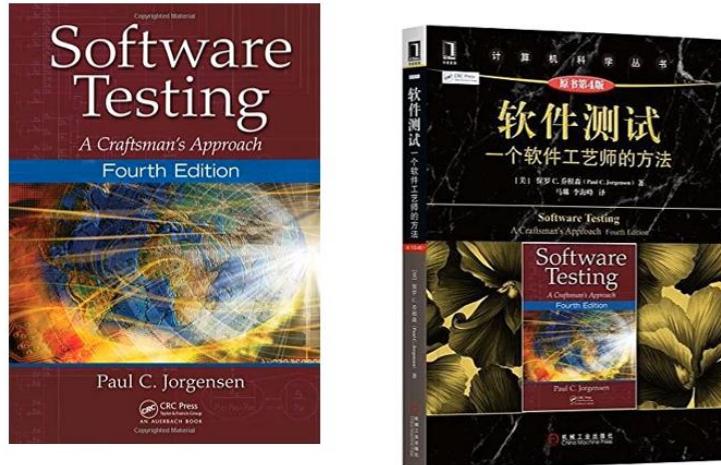
## 目录

2022/2/14.....	2
软件测试介绍.....	3
RIPR 模型.....	11
测试里面的原则.....	13
Myers principle.....	13
独立性准则.....	14
完整性准则.....	14
2022/2/21.....	14
代码走查（Walk through） .....	18
代码检查（Inspection） .....	19
基于执行的测试方法.....	20
黑盒测试.....	21
白盒测试.....	22
2022/2/28.....	24
功能性测试（Function Test） .....	27
边界值测试.....	27
健壮性测试.....	28
最坏情况测试.....	29
随机测试.....	29
等价类测试.....	30
弱一般等价类.....	30
强一般等价类.....	32
弱健壮等价类测试.....	33
强健壮等价类测试.....	34
2022/3/7.....	35
5.3 基于决策表的测试.....	39
结构性测试（Structure Test） .....	46
2022/3/14.....	48
Junit 的使用.....	50
IDEA.....	54
MOCK 工具.....	59
PIT.....	61
变异测试修改的东西.....	64
测试计划怎么写.....	65
路径测试.....	71
DD 路径.....	73
测试覆盖指标.....	75
2022/3/21.....	76
条件覆盖.....	77
分支/条件覆盖.....	79
数据流测试.....	86
定义-使用路径（du-path） .....	87
定义-清除路径（dc-path） .....	87

2022/3/28.....	90
全定义准则.....	90
全使用准则.....	90
基于程序片的测试.....	91
2022/4/2.....	95
用于方法评估的指标.....	96
基于生命周期的测试.....	98
2022/4/11.....	106
基于路径的集成.....	110
系统测试.....	112
2022/4/18.....	118
Selenium.....	118
Load Runner.....	130
系统测试的覆盖率指标.....	148
2022/4/25.....	151
回归测试.....	151
Test-All 方法.....	153
Test selection using execution trace and execution slice.....	157
Test selection using test minimization.....	161
Test selection using test prioritization.....	164
2022/5/7.....	167
WEB 网站测试.....	167
变异测试.....	177
2022/5/9.....	179
变异分数.....	180
变异测试的例子.....	183
变异体的特性.....	186
熟练程序员假设（CPH）.....	191
2022/5/16.....	195
嵌入式软件测试.....	195
计算 Worst-Case Execution Time.....	199

## 2022/2/14

软件测试 2 个学分，3 学分压缩过来，后面会讲的快一些。这门课有一些基本的参考书放在 canvas 上了。



大家注意英文版参考书自己用就行了。中文版的翻译过程中有很多错误。我们课的 60% 内容在书里，其他 40% 在别的参考书里。基本软件测试内容是按照这两本书整理的。

**Projects 1~4: 40%,**

**Usual test : 10%**

**Final exam: 50%**

有四次大作业，尽量联系性比较强一些，难度不会很大。usual test 是会穿插一些课堂测验。去年的开卷考试以应用题为主，有很多例子做软件测试的内容。

- ④ Class Prerequisites : Java/ C++, Software Engineering
- ④ Project requirement: A group, should complete **4 projects** as follows;
- ④ **Project 1: Black-box test-case generation** - during week 3~6; take an application with requirement document; create black-box test cases; execute test; report bugs in a bug tracking tool.
- ④ **Project 2: White-box test-case generation and coverage** - during week 7~10; use tools; evaluate coverage; create additional white-box test cases to satisfy some coverage criteria.
- ④ **Project 3: GUI test** :during week 11~14; Select a software GUI with at least 2 levels; Use selenium, to complete GUI test.
- ④ **Project 4: Performance test:** during week 15~17; select a software of web server, use LoadRunner, to complete performance test.

今年尽量要求大家四次作业，能够找到一个例子把四次作业贯通地做一下会比较好。例子都是自己找，可以找之前的一些开源的例子，尽量不要找一些重复的例子，例子会查重一下。

软件测试为什么要学？我们同学的就业去向 90%往往是做开发工作，现在很多开发人员和测试人员是一体的，自己做完的开发工作往往自己还要做测试。所以大家把软件测试基本内容要系统地学习一下。

现在同学也会做一些基本的测试，但是软件测试的内容是比较多的，分的体系比较多，针对不同的测试内容都要针对性地做测试。所以我们在学习的时候会对每一个专题去做测试，我们就要把学习到的专题技术应用到测试里去，要针对一个项目做完整的测试报告。

## 软件测试介绍

从学科的发展来讲，目前做软件测试研究的人越来越少了。软件测试本身变成了基础的开发工作的一部分。我们学这门课的目标就是对一个软件规范化地完成测试过程(测试方案、计划、用例报告、用例的设计过程)。

很多测试的软件都有一套完整的、体系化的测试过程，针对功能需求和非功能需求都完成了测试。我们软件测试对应到软件的质量，软件质量有一部分对应到大家开发代码的质量。不同人的代码风格差异很大，代码质量也是软件测试里的基本内容。软件质量在静态过程和动态过程中都有体现。

我们还会介绍软件测试里基本的一些概念，我们这节课快速的梳理。计算机软件最后沉淀下来的技术不多，沉淀下来的技术都是在技术发展体系里非常重要和实用的。



各个专业都在做软件，软件现在在很多产品里面的主要部分。我们作为软件工程的学生，要高质量完成软件开发过程。

软件测试有一个误区，软件测试都针对安全关键系统来做的？eg: 航天中的火箭、汽车、飞机。这些应用要求的软件的质量高。

安全关键系统有一个指标，软件可靠性。比如故障率要小于 $10^{-7}$ ，即在整个软件生命周期里发生故障的概率小于1次。

其实软件测试需求不仅仅在安全关键系统中，比如我们的信息系统也要做软件测试，但是它的质量和可靠性的要求不一定要这么高，出了故障它的损失相对小。

我们原先的 12306 系统天天出故障，这类系统虽然不是安全关键系统，但是故障所带来的损失也是很大的；又比如金融界的一些交易系统，那个也是很关键的，故障会带来经济损失，也会有软件可靠性的要求。

所以软件测试本身分为不同级别的对软件质量的要求。比如安全关键系统对于软件测试过程中就要求很高，经济类的也很重要。对于日常软件的重要程度相对低一些。

**Mars Climate Orbiter**

- Purpose: to relay signals from the Mars Polar Lander once it reached the surface of the planet
- Disaster: smashed into the planet instead of reaching a safe orbit
- Why: Software bug - failure to convert English measures to metric values
  - \$165M
  - 1mile = 1609M

**Shooting Down of Airbus 300**

- In 1988, US Vicennes shot down Airbus 300
- Mistook airbus 300 for a F-14
- 290 people dead
- Why: Software bug - cryptic and misleading output displayed by the tracking software

The slide contains two main sections. The left section, titled 'Mars Climate Orbiter', discusses a software bug where the satellite failed to convert English units to metric units, resulting in a crash into Mars. It includes a small image of the satellite in space. The right section, titled 'Shooting Down of Airbus 300', discusses a 1988 incident where a US missile shot down an Airbus 300 due to a software bug in the tracking software, which misinterpreted the aircraft as a F-14 fighter jet. It includes a map of the shootdown location over Iran and an image of an Airbus A300 airplane.

早期的探测器英制和公制单位转换出错导致的问题。这些简单错误的失效完全是可以通过软件测试方法保证的。



## London Ambulance Service

- London Ambulance Service Computer Aided Dispatch (LASCAD)
  - Purpose: automate many of the human-intensive processes of manual dispatch systems associated with ambulance services in the UK
- Functions: Call taking
  - Failure of the London Ambulance Service on 26 and 27 November 1992



救护车分配系统出错。这类例子还有很多。问题在于软件测试体系不完整，会导致一些低级错误，或者一些事件导致整个系统失效。

eg: 千年虫事件。用 2 个数字来表示 20 世纪的年份，导致到 2000 年的时候整个系统都需要升级。

软件测试是通过体系化方法保证软件质量的有效过程，对于平常的系统也要做软件测试。

即使代码写得再好的程序员和工程师，也会写出错误。我们要承认，只要有软件程序在，里面大概率存在错误。根据统计，在程序中 1000 行会有 5 个错误。有些错误是致命的，比如只在某一个场景下才会触发，比如 Win XP 有 45M 代码，那么就会有  $45 \times 5000 = 225,000$  个错误。

但是统计是假设程序员都是熟练的程序员。所以这些错误从开发的角度来说是真实存在的。我们要通过软件测试把这些错误找出来并且修复。

Q: 大家希望软件测试在找错误的时候以什么效果去找？比如两个测试工程师都找到了错误，我们怎么评价这两个工程师的好坏？

A: 做完作业就会发现，软件测试挺累的，要设计很多测试用例。所以要有效率地去找，软件测试就是在测试本身的代价和它达到的效果做平衡。

没有人保证测试完能把所有错误都找到，我们只是以最大努力找到了每个错误。所以软件测试就是在有限时间内尽可能多地找到任务。公司做测试工作会耗费很多人力物力财力。



## How get high quality software

### Two ways:

- Process management
  - CMM,PSP,TSP
  - Engineering process
- Testing

讲到软件质量，我们学过软件工程，软件工程的开发过程大家有所体会。比如迭代开发、成熟度模型、团队软件开发过程。都是通过软件开发过程的管理来提高质量的。我们这门课学的是通过测试手段来提高过程质量。

所以从软件质量保证来讲，先要把软件开发好，不要把软件质量的保证全部转移到测试来做，测试只是最终的防线。前期的开发过程中，大家就要注意代码质量和开发过程的管理。



# What is Software Testing?

- ④ **Testing = process of finding input values to check against a software (*focus of this course*)**

Test case consists of test values and expected results

- ④ **Debugging = process of finding a fault given a failure**

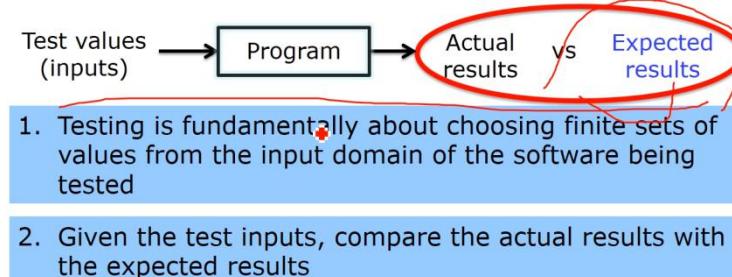
什么是软件测试？测试本身是通过一些输入来测试软件是否满足我们的设计要求，所以软件测试的核心是设计测试用例。

测试用例是软件测试课里要针对性地学习的。测试用例的好坏就是软件测试工作的好坏的评价标准。测试用例一般包含测试的输入和期望的结果。如果我们不知道一个软件的期望的输出结果，那么我们是不能测试的。

Q: 期望的结果去哪找？

A: 其实开发过程中有很多开发的文档，第一步是需求文档。其实需求文档里面已经给出了软件正确执行功能的需求，它是软件开发的整个要求，会包含功能需求和非功能需求。期望的结果应当对应到需求文档里去找。软件测试要对应到软件的需求文档，如果没有需求文档，我们不能做测试。因为我们不能靠主观判断软件的对错。

所以测试用例的期望结果是大家需要尤其关注的，也是经常被人忽略的。后面做作业的时候，大家需要尽可能把需求文档收集一下。我们找到了错误以后，就要去做 debug。



测试的时候，输入有很多，比如测试加法  $a+b$ 。 $a$  和  $b$  有很多可能性，但是我们不能举出所有的可能性，所以软件测试是用有限的输入去完整地测试这个程序。所以测试用例的输入从测试本身来讲，我们希望是越少越好。eg: 用 3 个测试用例就可以完整地测试功能。

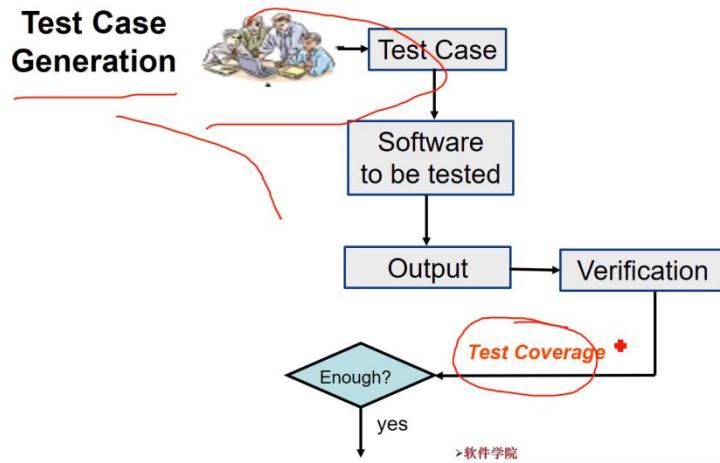
所以测试的规则就是尽可能少地设计测试用例来保证测试的完整性。

Q: 测试什么时候停下来？

A: 一个软件测试可以做一个月，也可以做十年。如何平衡软件测试的代价和效率？我们测试一定是以比较小的代价找到了比较多的软件错误。



## When to Stop?

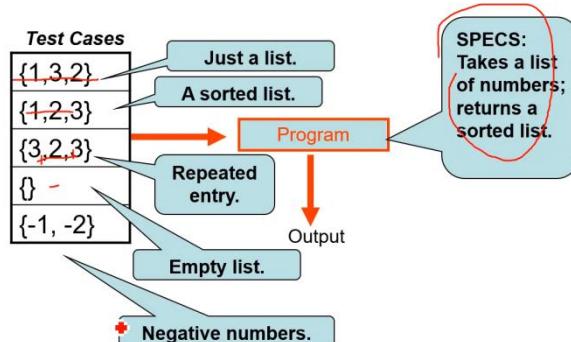


方法的覆盖率分析是指导我们测试停下来的标准。我们认为达到了一定的覆盖率以后，测试就是足够的了。所以说完整性是客观地计算不同种类的覆盖率。

下面我们看一些简单的例子体验一下。



## A Real Testing Example



右边的 spec 就是需求，对数字排序。我们要对输入做一些代表性的分类，针对性地做测试，这样覆盖率就是完整的。

另外就是我们的软件测试还有一些特殊的地方。现在开发的过程不一样。

- **Different Software Types**

- Object-Oriented
- Component-Based
- Concurrent
- Distributed
- Graphical-User Interfaces
- Web

- **Different goals of testing**

- **Function**
- Performance
- Security
- Usability
- Correctness

- ...

>软件学

这些软件类型有一些共用和特殊的方法。测试的目的也不同。开发过程里，需求分析主要是针对于功能需求和非功能需求。

我们简单看一下测试的基本内容。



### 1.2.1 Goals of testing

- Testing is a process of executing a program with the intent of finding an error.
- A good test is one that has a **high probability of finding an as yet undiscovered error.**
- A successful test is one that uncovers an as yet undiscovered error.
- The objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.

测试就是要找到软件里的错误，好的测试一定是以比较大的概率去找到错误，这是评价测试用例设计好坏的标准。

对于一个成功的测试来讲，它一定是可以找到里面还没有发现的错误。好的测试用例不一定真正地能找到错误。所以测试过程一定要有一些最基础的原则。

另外就是测试过程里面要注意的，比如我们要针对需求规约（specification）去做测试。规约分为功能性的和非功能性的，分门别类去做测试。做完测试不能说没有错误了，但是能找到大部分错误。

- Demonstrate that software functions appear to be working according to specification. 7
- That performance requirements appear to have been met.
- Data collected during testing provides a good indication of software reliability and some indication of software quality.

~~Testing cannot show the absence of defects, it can only show that software defects are present.~~

### 1.2.2 Facts About Testing

- Question “does program P obey specification S” is undecidable! +
- Every testing technique embodies some compromise between accuracy and computational cost
- Facts
  - Inaccuracy is not a limitation of the technique. It is theoretically impossible to devise a completely accurate technique
  - Every practical technique must sacrifice accuracy in some way

结论：判断程序 P 是否符合规范 S，是 undecidable 的。

但是我们可以通过测试手段，针对规范设计测试用例来在程序上执行，这样我们通过这个过程间接地说明程序 P 是否满足设计需求 S。软件测试不是一个很严格的技术，它不会去证明一些事情，它只是去做软件测试本身找错误的过程。

- The current development methods of software are:
  - Depend on people's intelligence
  - Specification is informal or even chaos
  - Software development tools are not self-contained
  - Software development process is elective and without standards on detail+

在当前的软件开发方法学指导下开发软件，软件测试仍是保证软件质量和可靠性的重要手段。



## Some examples

### 1 Exception

```
if(state == 0)
{
    ...
}
if(state == 1)
{...} ;
```

### 2 Data overflow

```
Int count( int &a, int month_num)
{
    int month_income = 0, i;
    for( i = 0; i < month_num; i++)
        month_income+=a[i];
    return month_income;
}
//1000<a[i]<10000
```

左边的话是 state 的类型，右边的话就是 int 类型的 month\_income 的溢出的情况。



### 1.2.3 Glossary

- **Error (ISO)(错误)**

—A good synonym is mistake. Errors tend to propagate;

- **Fault(故障)**

—A fault is the result of an error. Defect (see the ISTQB Glossary) is a good synonym for fault.

- **Failure (IEEE)(失效)**

—The inability of a system or component to perform its required functions within specified performance requirements.

- **Incident(事件)**

—When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

我们去看一些企业的规范的时候，会涉及到上面的这些术语。错误就是程序中的 bug，会向后传播导致程序故障，程序故障会导致系统失效。



### Example

```
public static int numZero (int[] arr)
{
    // Effects: If arr is null throw NullPointerException
    // else return the number of occurrences of 0 in arr
    int count = 0;
    for (int i = 1; i < arr.length; i++)
        if (arr[i] == 0)
            count++;
    return count;
}
```

There is a simple fault in numZero

Where is the fault location in the source code?

How would you fix it?

Can the fault location be reached? How does it corrupt program state? Does it always corrupt the program state?

我们可以看一下这个测试的模型是怎么表示的。我们要定位到错误，并且看看错误在程序里是怎么传播的。

Failure: propagation of erroneous state to the program outputs  
Happens as long as `arr.length > 0` and `arr[0] = 0`

Test 1: [4, 6, 0], expected 1  
Error: i is 1, not 0, on the first iteration  
Failure: none

Test 2: [0, 4, 6], expected 1  
Error: i is 1, not 0, error propagates to the variable count  
Failure: count is 0 at the return statement

我们可以看一下这个整个程序的过程中的状态传递模型。状态传递模型会通过变量和程序的指针来表示程序的状态。

Assume that we want to represent program states using the notation `<var1 = v1, ..., varn = vn, PC = program counter>`

Sequence of states in the execution of `numZero([0, 4, 6])`

2: < arr={0, 4, 6}, count=0, PC=[i=1 (L2)] >  
3: < arr={0, 4, 6}, count=0, i=1, PC=[i<arr.length (L2)] >  
...  
< arr={0, 4, 6}, count=0, PC=[return count; (L5)] >

状态的依赖会影响到程序状态的执行。所以通过程序的运行以后，程序的状态是通过一步步的程序执行变成一个个的状态表示，状态表示之间会存在很大的依赖关系。所以过程里面，我们会通过分析，找到里面程序状态的对应关系。它会变成我们错误是如何来传递的因素。

通过这个例子，我们可以看一下状态在程序中的转移情况。



### Example – Error State

#### Error state

The first different state in execution in comparison to an execution to the state sequence of what would be the correct program

If the code had `i=0` (correct program), the execution of `numZero([0, 4, 6])` would be

1: < arr={0, 4, 6}, PC=[int count=0 (L1)] >  
2: < arr={0, 4, 6}, count=0, PC=[i=0 (L2)] >  
3: < arr={0, 4, 6}, count=0, i=0, PC=[i<arr.length (L2)] >  
...

Instead, we have

1: < arr={0, 4, 6}, PC=[int count=0 (L1)] >  
2: < arr={0, 4, 6}, count=0, PC=[i=1 (L2)] >  
3: < arr={0, 4, 6}, count=0, i=1, PC=[i<arr.length (L2)] >

The first error state is immediately after i=1 in line L2

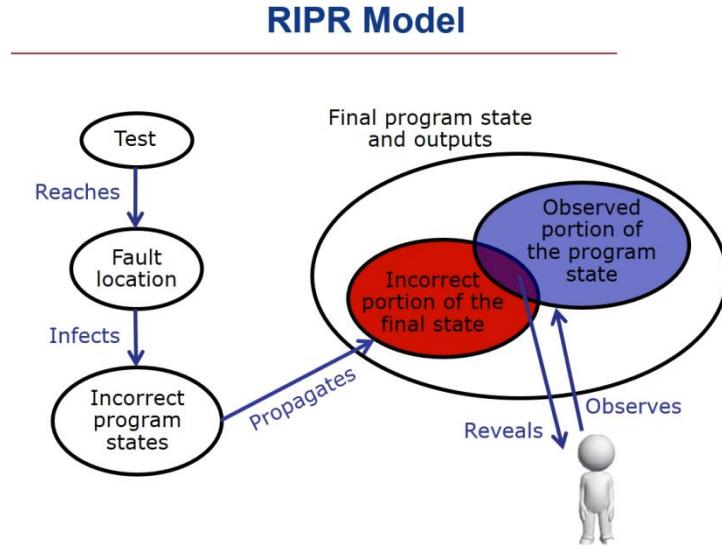
## RIPR 模型

通过这个例子，我们有一个 RIPR 模型。

一个错误被观测到的四个条件：

1. **reachability** (可达性): 错误要能够到达, 测试用例需要保证错误的状态能够到达
2. **infection** (传染): 这个错误会导致程序状态的改变
3. **propagation** (传播): 程序的状态通过一步步的依赖关系传播到最后的输出结果
4. **reveability** (可揭示性): 最终能够看到错误的结果。

一个好的测试用例也要满足上述的特点, 否则就测不出对应的错误。这个模型是后面的  
所有测试用例设计的时候遵循的基本模型。



测试要达到错误的位置, 传播到程序的状态, 最终影响到程序的输出, 最终的输出有时候能观测到, 而有些不能观测到。也有可能虽然错了, 但是我们观测不到最终的错误。



### Example – RIPR (Error, Failure)

```

public static int numZero (int[] arr)
{
    // Effects: If arr is null throw NullPointerException
    // else return the number of occurrences of 0 in arr
L1   int count = 0;
L2   for (int i = 1; i < arr.length; i++)
L3       if (arr[i] == 0)
L4           count++;
L5   return count;
}

```

Revisit the example, apply RIPR to design tests that

Reach a fault (i.e., execute the fault)

Cause the program state to be incorrect (i.e., error)

Propagate (i.e., failure)

One possible test is [0, 4, 6] – now, design some more

How does RIPR model help designing tests?

首先我们看能够执行这个错误的一个好的测试用例, 有了这个执行完了以后, 状态能发生改变, 变成一个错误状态, 导致程序状态不断往下传播, 导致整个测试结果不同。

# 测试里面的原则

## 1.2.4 The basic concepts of software test

- Basic principle of software test (Myers principle)
- Independent principle
- Completing principle
- The types of software test

很多准则是测试工作中的工作原则。

## Myers principle



### Myers principle

- Early test and continual test
- Avoid test software by programmer him/her self
- Test case = input data + expected result
- Test case should include expected (valid) input data and unexpected (invalid) input data
- Test program = what it should do + what it should not do
- The probability of existing errors is direct proportion to the errors have been uncovered.
- Check each test result roundly
- Software test is an most innovative work

有些看起来很无聊，如果我们能把它变成我们做工作的规则的话，就会起效果。

- 早测试：边开发边持续测试。
- 测试和开发尽量避免是同一个人，有时候自己开发的东西找错误很难找。
- 测试用例包含输入和期望结果。
- 测试应该包含两部分输入数据，一部分是有效输入，一部分是无效输入。大部分错误会犯到无效输入上。
  - 被测试的程序行为需要知道应该做什么事情和不应该做什么事情。
  - 我们要发现很多错误，一般发现错误的概率和程序里存在错误的数量成正比。（程序里错误越多越容易发现），比如我们发现测试中错误的曲线不增长了，那么我们就可以停止测试了。
  - 每一个测试过程是一轮轮不断地去做的。
  - 测试过程是按部就班的苦力活。

它把测试过程中的规律做了一些总结，每个人会有不同的体验的角度。还有一些准则。

## 独立性准则

**Software test should be execute by the group which is independent from development group.**

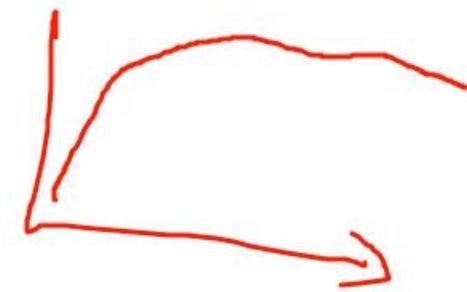
## 第三方测试

工作的时候很多软件要做第三方测试，测试过程不能自己去做，要让别人去做。

## 完整性准则

- Complete **all test cases according to the test plan**
- Use **stated test case generate methods.**
- Reach **expected object (such as coverage criteria)**
- Draw a curve **for analyze, which shows the errors for different time period.**
- Should uncover an **expected number of errors.**

我们要完整地把所有的 test case 做测试，另外就是使用测试方法产生测试用例。还有一些覆盖率要求，还要做一些分析来绘制成曲线。错误曲线一般是这样的。



**2022/2/21**

比如我们课本第三章面向测试的离散数学我们就不讲了，因为我们都学过离散数学。另外，大家要看一下第二章的程序的示例，因为我们课程讲解过程中都是面向例子去讲。比如三角形问题的程序的伪代码和基本结构，因为我们有很多测试过程都是针对于例子来做的。还有一个就是日期（计算下一天）的例子，这是白盒测试、集成测试相对常见的一个例子。其他的话，还有一些单独的小例子，都是课堂里偶尔会用到的。中文课本里面可能会有一些错误。

我们上节课讲到测试里面会遵循一些概念和准则。这些准则都是前人在工作实践中给到我们的一些总结，可以用到我们的项目中。包括 Myers 准则、独立性准则和完整性准则等。这些规则大家也不用去记，主要是融会贯通。

## Steps of software test

- ◆ Design test plan
- ◆ Generate test case
- ◆ Establish test environment
- ◆ Execute test case
- ◆ Evaluate test result
- ◆ Complete test documentation : test plan, test explain and test report

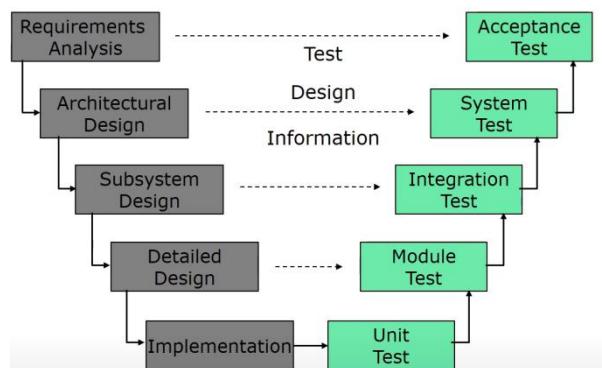
软件测试都会按照这个基本的步骤来做，首先要有测试计划。不是一开始就来设计测试用例。测试计划是指导我们测试的蓝本，我们要做什么工作都要放到软件测试计划里。然后我们就按照计划来做。有了测试计划内容，我们要测试的内容以及对应的软件测试项都定好了。接下来，我们就要生成测试用例，软件测试中测试用例是核心的东西。比如四个人都完成了测试，评价测试的好坏就是看测试用例是否满足“最小的代价满足了完整性和覆盖性”。这样我们就可以评判测试用例的好坏。所以生成测试用例也要遵循准则。

这个过程里面，大家可以体会到软件测试是有方法的指导的。测试环境是执行我们测试用例的基础。然后我们就会去评价和验证测试的结果，我们针对测试结果要去评判测试用例是否能够正常运行。测试结果的反馈就能够告诉我们测试用例的目的是否通过。

完成了测试用例以后，我们就可以撰写完成测试文档。它有比较正规的格式。

每个项目都会有测试方案的评审，特别是验收测试，会根据我们的测试文档来看，这个验收测试会有测试方案评审，它是一个大纲性的，如果条目通过了，那么整个项目都通过了。所以，测试的文档在很多软件项目里面是比较重要的，这也是对于我们软件项目是否能顺利结题的一个标志。

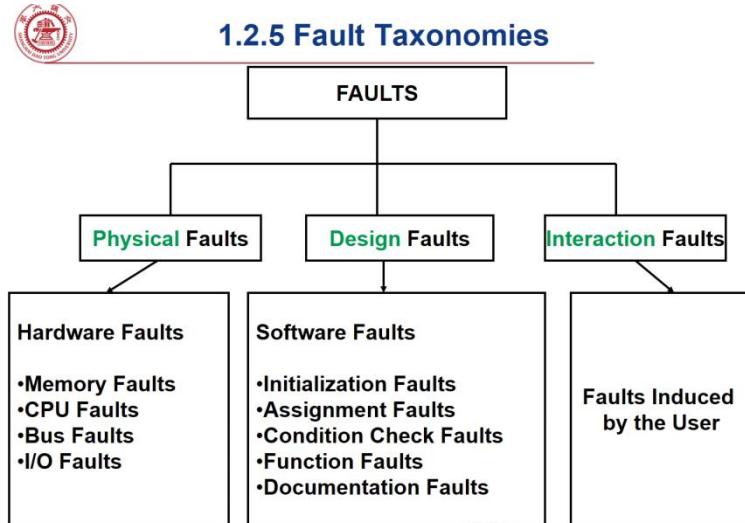
## Testing and SW Development Lifecy



讲到软件测试的时候，在软件的整个周期里面，如果把开发过程和测试过程一起来理解呢？软件开发里通常会用瀑布式的软件开发模型。比如软件都有需求分析、架构性的设计、子系统设计、详细设计、实现，最后才会有代码。所以软件的开发也是遵循一定的准则，大家需要思考一下软件开发为什么要遵循这个过程。其实这也是一些经验，不遵循过程失败的

概率比较大。

这样的话，我们软件开发过程也对应了相应的软件测试过程。实现的代码先会进行单元测试，针对每个实现的 `function`、`module` 去做一个个测试。再往上会有集成在一起的模块、集成测试、系统测试、验收测试。测试往往从单元到模块到集成到系统到验收，这是相对开发过程中相对反向的一个过程。不同软件开发阶段会采用不同的方法。



另外的话，后面大家会做作业测软件里的 BUG。错误有很多种类，这些种类都是我们经常在软件里看到的错误。有物理性的错误（CPU,IO,总线）、有软件设计的错误（初始化、条件、逻辑）、有交互式的错误（接口不对等）。

我们现在每一组都在找一个例子。大家在做软件测试作业的时候可以人为地加一些错误，主要是体验软件测试的过程，并不在意软件本身的质量。有一些软件错误注入的工具，注入错误的好处就是不同的软件测试方法针对不同的错误有不同的效果。

下面我们开始讲软件测试基本的方法。第三章软件测试的离散数学基础自学。

Q：结合自己的编程和软件开发经历，如何提高代码的质量？

架构性的思考、需求文档、etc。写代码的时候，写注释写给将来的自己和同组的人看。写代码的时候可以参考设计模式，这些都是经验的总结，将来返工可能比较少。



这些方法往往是很有效的。都是几十年来沉淀下来的、经过实践证明有效的。我们先看测试如何来划分，对测试的划分可以按软件开发阶段划分、按程序运行划分等。静态、动态的划分就是这个测试是否真正执行了程序，比如代码审查就是一个静态测试。白盒测试需要知道已知所有代码的结构。回归测试是对软件版本更新中迭代版本的测试。软件测试有很多方法，我们还要有总体的大局观。

在讲总体测试之前，我们先讲一些基本问题。比如非执行测试也属于一种测试呢？有人提出没有执行的测试不叫测试。

## 3.1 Non-execution based Verification

- Not execute the program, tester could use some tools to review and analyze the specification and program code.
  - Walk through (走查)
  - Inspection (审查)

我们认为非执行测试也是代码质量的一种保证方法。

## 代码走查 (Walk through)



### 3.1.1 Walk through

- Consist a walk through test group
- Check program code logic
- Generate test case, include input data and expect result.
- Put the data into program code, if the calculate result unequal the expect result, find an error.
- Check the code line by line.

比如我们在脑子里 or 纸上验算输入输出。

Q: 为什么要这样做呢？

A: 在程序执行代码可能不具备执行条件，可以快速做一些逻辑检查。



The walkthrough team should consist of four to six individuals

企业里代码走查也有组织方式，比如 SQA 成员、用户代表、执行成员等。代码走查就是通过程序逻辑推演找到程序中的错误。过程中会有一些列表项，针对里面的一些问题做问题的定位。

- The goal of the walkthrough team is to detect faults, not to correct them.
- The person leading the walkthrough guides the other members of the team through the code.
- The walkthrough can be driven by the lists of issues compiled by team members or by the code itself, with team members raising their concerns at the appropriate time.
- In both cases, each issue will be discussed as it comes up and resolved into either a fault that needs to be addressed or a point of confusion that will be cleared up in the discussion

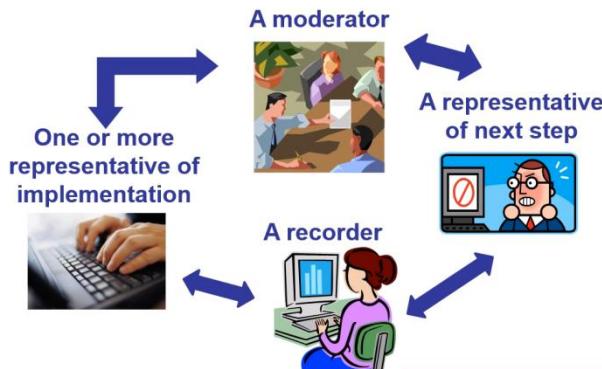
## 代码检查 (Inspection)

代码检查也是一种静态的测试方法，它比代码走查正式一些。有一个主持人和程序实现的代码、客户代表等。最终形成一个代码审查报告。



### 3.1.2 Inspection

- An inspection is a far more formal activity than a code walkthrough.
- It is conducted by a team of four to six people.



The review process consists of five formal steps:

- In the overview step, the author of the module gives a presentation to the team.
- In the preparation step, the participants try to understand the code in detail and compile lists of issues, ranked in order of severity(严重性). They are aided in this process by a checklist of potential faults for which to be on the lookout.
- In the inspection step, a thorough walkthrough of the code is performed, aiming for fault detection through complete coverage of the code. Within a day of the inspection, the moderator produces a meticulous written report.
- In the rework step, the individual responsible for the code resolves all faults and issues noted in the written report. (**Individual Reviewer Issues Spreadsheet**)
- In the follow-up step, the moderator must make sure that each issue has been resolved by either fixing the code or clarifying confusing points. (**Review Report Spreadsheet**)

程序员需要讲解设计过程和实现过程，让大家知道每一个模块是怎么实现的。如果一些

逻辑很明显不对，那么就可以在很早期就发现问题。静态测试会伴随着开发过程，好处就是早发现。

静态测试往往有多个不同部门的人去看，不仅仅是开发人员。当是同一群人的时候，BUG可能就不太容易发现了。

- An important product of an inspection is the **number** and kinds of **faults** found rated by severity.
- If a **module** comes through an inspection exhibiting a significantly **larger number** of faults than other modules in the system, it is a good candidate for **rewriting**.
- If the inspection of **two or three** modules reveals a large number of errors of specific types, this may warrant (re)**checking** other modules for similar errors.
- If more **than 5 percent** of the material inspected must be reworked, the team must reconvene for a full **re-inspection**.

代码审查过程中找到的错误会交给设计人员去改正。如果一个模块通过静态测试方法找到大量的错误，那么可能就会重写这个模块。这就是把很多风险的事情放在早期发现。这个程序很有可能在早期不具备执行条件，但是静态测试就可以找到问题。

## 基于执行的测试方法

还有一部分就是执行的测试。真正在环境里运行，并且有期望的结果，和已有的结果对应看看是否满足要求。对照软件需求规范（黑盒测试）还有对照测试我们的代码（白盒测试）。



### 3.2 Execution-based Verification

- Use **test cases** to execute the program, get all results from the **execution**.
- There are **two basic approaches** to testing modules, each with its own weaknesses.
  - Testing to **Specifications**, also known as **black-box test**.
  - Testing to **Code**, also called **glass-box, white-box test**.

黑盒测试以后已经满足了需求规范了，白盒测试是否多此一举？它对应的不是软件输入输出需求，而是逻辑的正确性。黑盒测试都通过了以后，往往还会有很多潜在的逻辑风险，会在很特定很特定的条件下触发，会使得软件完全失效。这种情况下我们还是需要白盒测试去弥补黑盒测试发现不了的逻辑错误。黑盒测试的优点可以非常明确地对应到我们软件需求的开发文档。

## 黑盒测试

判断一个方法是黑盒还是白盒就是这个测试需不需要看程序代码。如果只看输入输出，就是一个黑盒测试。



### 3.2.1 Black-box test



- Seeks to verify that the module conforms to the **specified** input and output while **ignoring** the actual **code**
- It is **rarely possible** to test all modules for **all** possible input cases, which may be a **huge** number of modules

一般都是有确定的输入和输出。黑盒测试里输入的设计是很关键的，对于一个程序把所有的输入都尝试一遍是不可能的。黑盒测试的核心就是用最小的输入去说明我们的测试是完整的。

我们后面就会讲到等价类方法和边界值方法。

- **Equivalence testing** is a technique based on the idea that the input specifications give ranges of values for which the software product should work the same way.
- **Boundary value analysis** seeks to test the product with input values that lie on and just to the side of boundaries between equivalence classes.

等价类就是选择有代表性的输入类别。

For example the input is (a,b), there are at least 5 equivalence classes:



**Functional testing**, the tester identifies each item of functionality or each function implemented in the module and uses data to test each function.

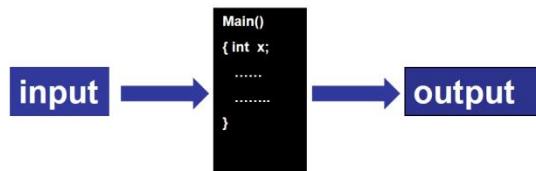
选五个测试用例就是每一类都证明了系统工作是正常的。核心是对输入的分析，要求等价类并在一起是全集。

# 白盒测试

比如数据流测试、路径测试，测试的方法是需要里面的代码的，没有程序的源代码，测试方法是不能使用的。



## 3.2.2 White-box test



- Also called **logic-driven**, or **path-oriented testing**.
- Testing each path through the code is generally not feasible, even for simple flowcharts.
- Moreover, it is **possible** to test every path **without** finding existing faults, because, for example, the fault lies in the decision criterion for **selecting between paths**.

白盒测试也有输入输出，是针对里的代码结构做测试。它通常叫做逻辑驱动、路径导向的测试。逻辑结构的设计是否存在一定的风险。这些风险是黑盒测试所测试不到的，因为黑盒测试选代表性的可能忽略了一些关系。白盒测试会讲覆盖率的表述。

- Statement coverage and amounts to running tests in which every statement in the code is executed at least once.
- Branch coverage, makes sure that each branching point is tested at least once.
- Path coverage, make sure that every different path is tested at least once.

覆盖率是测试证明自己是完整的方法。比如达到了所有语句的覆盖、所有分支条件都覆盖到了、所有路径都覆盖了等。所以完整性都是相对的完整。白盒测试的完整性是通过覆盖率指标说明的。

Q: 黑盒测试怎么说明完整性呢？

A: 比如等价类按照划分来选择。每一个代表性都有，就是完整的。

## Example : black-box testing

- Discussion: MAC/ATM machine
- Specs
  - Cannot withdraw more than \$300
  - Cannot withdraw more than your account balance

ATM 机的例子，这样我们就可以做简单的测试。

### Test case?

X

1. >300
2. 0~300
3. <0

#### Balance:

1. =>300,
2. 0~300

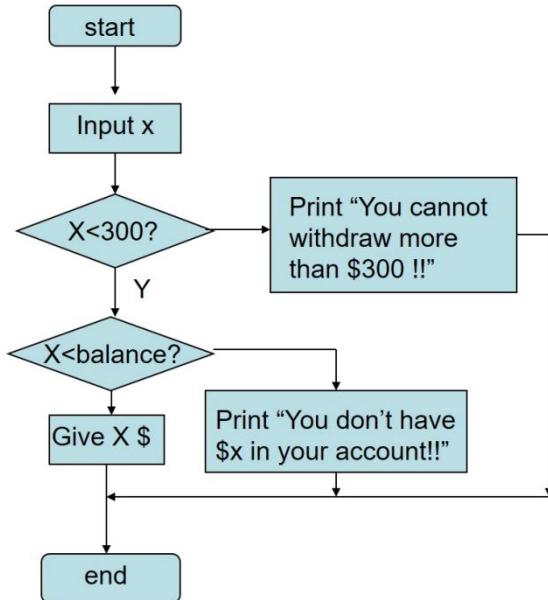
我们要测试系统不应该做的事情是否确实没有做，还有应有的功能。集合划分也有不同的种类，不同的划分的粒度不太一样。



## White-box Testing

### Example

```
// x: 1..1000;
1 INPUT-FROM-USER(x);
  If (x <= 300) {
2   INPUT-FROM-FILE(BALANCE);
    If (x <= BALANCE)
      GiveMoney x;
4     else Print "You don't have $x in your account!!"
      else
5   Print "You cannot withdraw more than $300";
6 Eject Card;
```



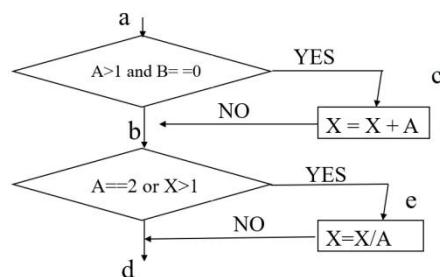
软件对应到结构图，我们就可以知道软件是怎么做逻辑判断的，从而进行白盒测试。

## 2022/2/28

后面我们会讲到各种各样方法在不同的软件测试场景中发挥的必要重要的作用。上节课我们讨论了黑盒测试和白盒测试的关系，包括有了黑盒测试以后为什么需要白盒测试。我们来看一下有白盒测试程序以后，我们究竟要测试什么内容。

我们现在有了这个程序，如果没有学软件测试，我们应该考虑测试哪些内容？

How about example 2 ?



Q: 分支测试能找到代码里对应的什么错误呢？

A: 分支是否可以正确地进入、条件是否一致。

每一种测试方法都有相对明确的任务，有一些错误也是找不到的。

- For example 2, branch coverage is: point A, should have  $A > 1$ ,  $A \leq 1$ ,  $B = 0$ ,  $B \neq 0$ , point B, should have  $A = 2$ ,  $A \neq 2$ ,  $X > 1$ ,  $X \leq 1$ .  
so {2, 0|2, 5} and {0, 1|0, 0} is enough.
- How about path coverage?

4

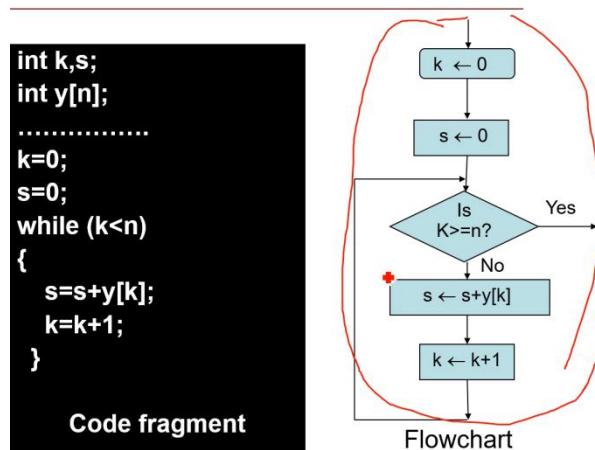
上图中的路径是 4 条。

Q: 分支覆盖和路径覆盖的目标有什么不同？

A: 分支是是否能正确进入一个状态，进入分支以后还要执行一定的程序，所以每条路径还有期望的结果。所以路径不仅仅和分支相关，还有其他的错误的可能性。

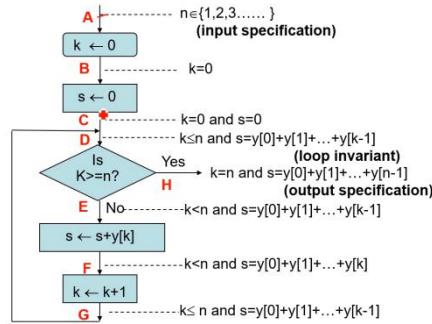
软件测试方法是多维度地找软件里的错误，我们要会去选择相应的方法。如果我们要测试每一个判断都正确，也就是每一个谓词的覆盖，那么我们就需要  $2^4 = 16$  种测试用例。这门课一半以上都在讨论白盒测试方法，从不同的角度分析程序的错误。

软件测试和验证都是保证软件质量的不同方法。软件验证也有很多规范性的方法和证明性的方法。我们提到形式化的软件验证是从数学角度保证程序的正确性。有些程序是有证明过程的，这个程序不断地做循环把数组里的数加起来。



我们证明它正确有很多不同的方法。如果我们把这个程序分片，变成一个个程序的状态，对于一个程序来讲，每个程序状态正确执行最终有一个输出，对应需求规范。到 input 到最终的输出，通过状态的转移过程可以变成一个可计算的过程。我们可以根据程序之间的状态去推导传递过程，去验证最终程序的状态是否符合我们的程序逻辑。

An assertion is placed before and after each statement,  
at the place labeled with the letters A through H;



Input specification,  $n$  is a positive integer,

$$A: \quad n \in \{1, 2, 3, \dots\} \quad (1)$$

Output specification is that if control reaches point H, the value of  $s$  contains the sum of  $n$  values stored in array  $y$ ,

$$H: \quad s = y[0] + y[1] + \dots + y[n-1] \quad (2)$$

A stronger output specification is,

$$H: \quad k = n \text{ and } s = y[0] + y[1] + \dots + y[n-1] \quad (3)$$

How proven from A to H? key is proven the invariant of the loop, that is,

$$D: \quad k \leq n \text{ and } s = y[0] + y[1] + \dots + y[k-1] \quad (4)$$

$$\begin{array}{ll} \text{First} & B: \quad k = 0 \\ \text{This is easy to prove, because of assignment statement} & \end{array} \quad (5)$$

$k \leftarrow 0$  is executed.

At point C, the second assignment statement  $s \leftarrow 0$  is executed, the follow assertion is true:

$$k = 0 \text{ and } s = 0 \quad (6)$$

Now the loop is entered. Before the loop execute, assertion(6) holds  $k=0$  and  $s=0$ . By assertion(1),  $n \geq 1$ . so it follows that  $k \leq n$  is required. Because  $k=0$ , it follows  $k-1=-1$ , so the sum in (4) is empty  $s=0$ . there for loop invariant (4) is true just before the first loop is entered.

Now, assume that, at some stage during the execution of the code fragment, the loop invariant holds. That is  $k$  to some value  $k_0$ ,  $0 \leq k_0 \leq n$ .

$$D: \quad k_0 \leq n \text{ and } s = y[0] + y[1] + \dots + y[k_0-1] \quad (7)$$

We should to prove execute the code fragment, the loop invariant is still true or get the output specification.

27

上述过程不做要求。

当然后面还会有相应的专题化的测试过程，比如 Def-use Test (数据流测试)、变异测试、回归测试、错误统计等。

对白盒测试来讲，我们经常关心里面程序的结构正确性，但是数据的使用没有检测到，所以我们就会有数据流测试。变异测试实际上就是注入一些程序错误，如果程序执行完依旧是对的，那么就会出问题。

## Summary

- Non-execution based Verification
  - Walkthrough
  - Inspection
- Execution based Verification
  - Black-box
  - White-box
- Formal verification +
- Other methods
  - Def-use test
  - Mutation test
  - Regression Test
  - Fault Statistics and Reliability Analysis
  - Clean room

# 功能性测试（Function Test）

测试的基准来源于软件的需求，有一大类就是功能性需求。功能测试有专门的测试方法。



## 提纲

- 边界值测试
- 等价类测试
- 基于决策表的测试
- 测试的效率

### 边界值测试



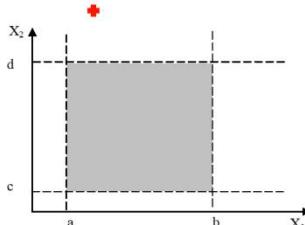
## 5.1 边界测试

### 5.1.1 边界值

对于函数:  $Y = f(x_1, x_2)$

若  $a \leq x_1 \leq b$

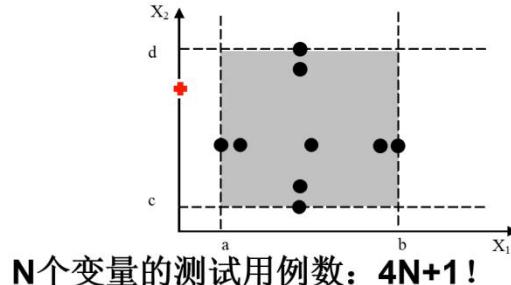
$c \leq x_2 \leq d$



比如我们程序实现了这个函数的功能，程序的输入有对应的范围。比如拿这个例子来讲，我们做测试应该怎么去思考它？我们的依据是什么？

每个测试都有目的和针对性，我们需要知道每个用例的目标是找到什么错误。对于我们的程序输入，分为有效输入（阴影范围内）和无效输入（阴影外）。可能无效输入的测试比有效输入更重要。

- 边界值测试的取值: 最小值、略高于最小值、正常值、略低于最高值、最高值。
- **单缺陷假设:** 失效极少是由2个(或更多)缺陷同时发生引起的。



**4N+1** 的依据就是单缺陷假设，失效极少是由 2 个或多个缺陷同时引起的。多缺陷假设就是缺陷也可以由多个缺陷引起的。因为没有测试四个角落，所以很明显这个测试就是基于单缺陷假设的，设计测试用例取值的时候，我们只考虑一个维度的。没有考虑多个维度之间的互相影响过程，所以单缺陷假设就变成了边界值测试的基础。

**Q:** 单缺陷假设的好处是什么？

**A:** 线性的测试数量比较少；定位方便，如果一个测试用例出现问题了，我们很容易定位到对应维度的变量。但是多缺陷假设就会有耦合效应。比如四个角落就同时考虑了  $X_1$  和  $X_2$  的边界，它就会有多缺陷的耦合效应，很难定位是谁造成的。

**Q:** 这么简单的问题为什么要想这么多呢？

**A:** 同学们软件开发的经验多了以后，就会遇到 Debug 的挑战。很多企业找错误的时候通常是一个团队找很久，软件测试可以通过软件测试方法结合物理性的测试区针对性地定位到里面特定的错误。很多做工程开发的人，知道有错误很简单，但是找到它就会有点难了。所以，大家一定要结合很多工作的准则，通过软件测试手段定位到错误。

所以，边界值区域是针对于输入的有效区域去做的测试，并且是基于单缺陷假设的。这就形成了  $4n+1$  的测试数量。

## 健壮性测试

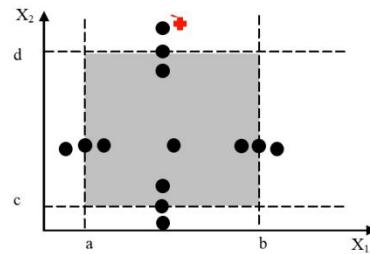
同学们可能会问，边界值测试为什么只针对有效区域呢？它是一个很基础的测试，就是这么定义的。如果我们也要加上无效区域，那么它就会变成健壮性测试。



## 健壮性测试

除了**5**个边界值，再增加一个略小于最小值（**min-**），一个略大于最大值（**max+**）。

即：**要考虑无效值的输入！**



**N**个变量的测试用例数： **$6N+1$ ！**

健壮性测试的时候，也是基于单缺陷假设去做的。只是它考虑了无效区域。

## 最坏情况测试

我们觉得单缺陷假设有点不够用，因为很多复杂的错误都是很多条件相互作用以后导致的结果。如果程序的安全性很高，那么单缺陷假设就不能够满足。就会有了最坏情况测试，我们要考虑输入的组合，所以就要考虑多缺陷的情况，它基于多缺陷假设。



## 最坏情况测试

- 拒绝单缺陷假设，考虑全部边界输入的组合，即各个变量输入的笛卡儿积（**多缺陷**）。
- 最坏情况测试：** $N$ 个变量的测试用例： $5^N$ 。
- 健壮（增加2个无效输入）最坏情况测试：** $N$ 个变量的测试用例： $7^N$

**事例：**

- 三角形问题 边界值测试：a、b、c三个变量
- 一般法（边界值）：测试用例= $4 \times 3 + 1 = 13$ 个
- 最坏情况：** $5^3 = 125$ 个

## 随机测试

随机测试就是取一些随机值作为测试用例来避免人为的偏见。



## 随机测试

即使完成边界值的全部测试，也不能发现程序的全部错误，有些错误会存在于非边界值之中。使用随机函数取出测试值，避免了人为的测试偏见。  
但是多少测试用例才是充分的？（后面再说明）  
何时停止用例生成？  
保证每类输出至少有一个。

## 等价类测试

### 弱一般等价类



## 5.2 等价类测试

划分：互不相交的一组子集，这些子集的并是全集。

1) 弱（单缺陷）一般等价类

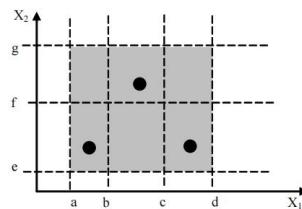
对于函数： $Y = f(x_1, x_2)$

若  $a \leq x_1 \leq d$  等价区间： $[a, b], [b, c], [c, d]; N=3;$

$e \leq x_2 \leq g$  等价区间： $[e, f], [f, g]; M=2;$

覆盖单缺陷

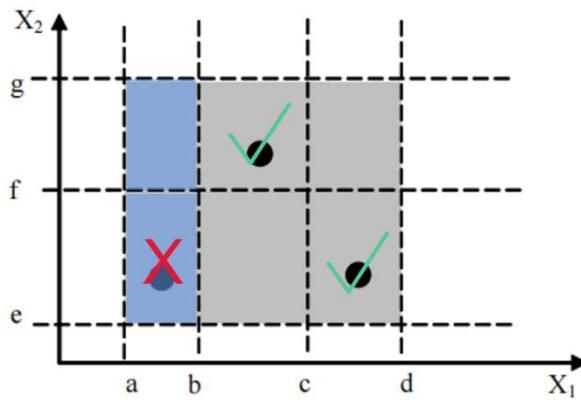
测试数：取N、M中的大者。



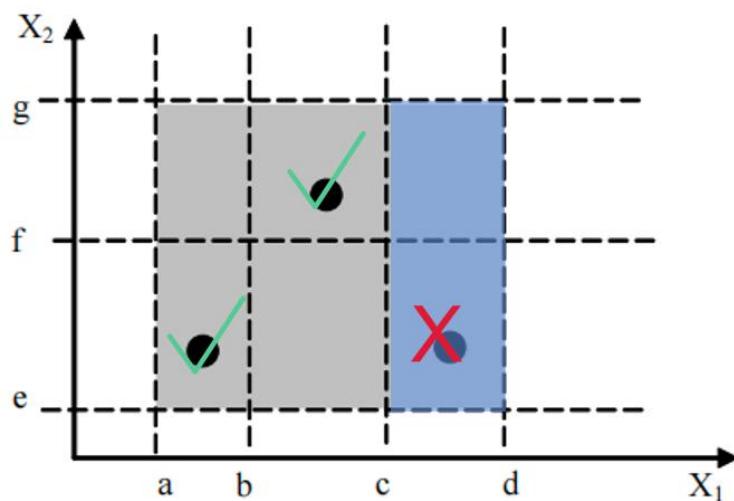
等价类测试是有数学基础的，基于集合的划分。对于之前的  $f(X_1, X_2)$  这个例子来说，我们把输入区域做等价划分的时候，可以划分成一个个互不相交的子集（6个）。弱指的就是单缺陷。要覆盖一个等价类最简单就是选用6个。但是在弱一半等价类的情况下，我们就可以选用  $\max(N, M)$  作为测试用例数。

Q：为什么3个和6个的测试用例的覆盖效果是一样的呢？

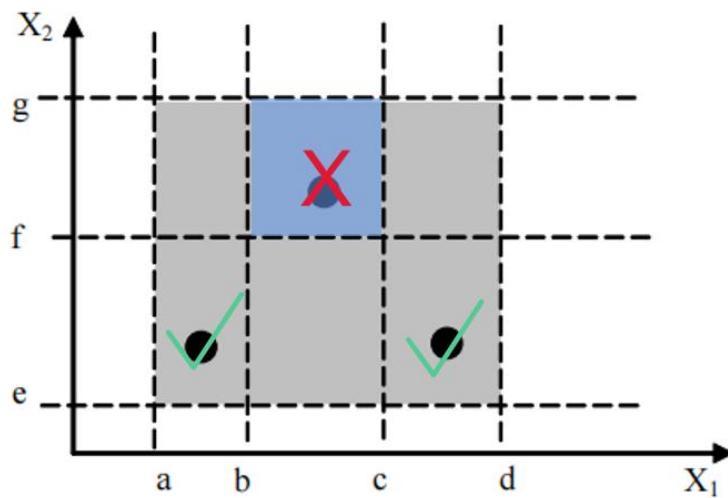
A：对于每个变量来说，我们需要覆盖对应的区间。单缺陷设计的测试用例要把这几段都覆盖掉。对于我们这个例子来说，我们不应该二维的去考虑这个问题，单缺陷假设下，我们只需要分别考虑两个变量即可。



如上图所示，如果测出来有如图的错误，那么我们就可以定位到是  $X_1$  的  $[a, b]$  区间。



如上图所示，如果测出来有如图的错误，那么我们就可以定位到是  $X_1$  的  $[c, d]$  区间。



如上图所示，如果测出来有如图的错误，那么我们就可以定位到是  $X_1$  的  $[b, c]$  区间和  $X_2$  的  $[g, f]$  区间。

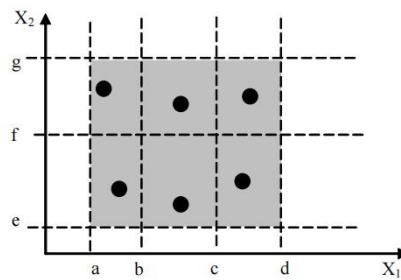
所以我们看到这种方法不能完全测试到对应的位置，需要多缺陷假设补充。

## 强一般等价类

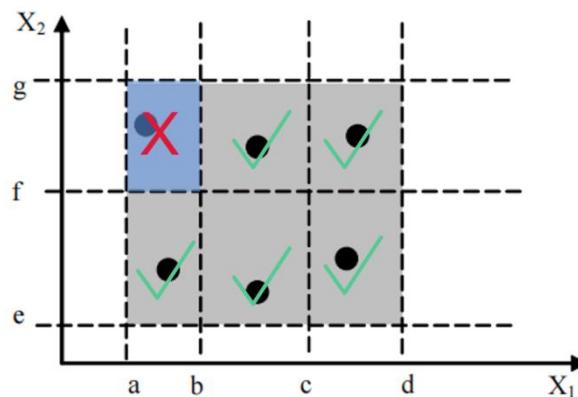
### 2) 强一般等价类

基于多缺陷假设，考虑不同划分的笛卡儿积。

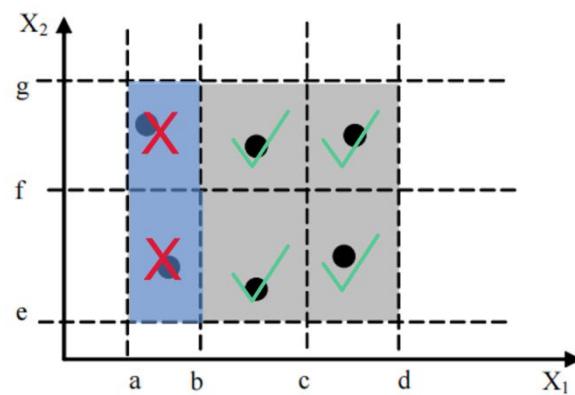
覆盖多缺陷：测试数 $N \times M$ 。



Q: 多缺陷的时候，123456 有不同的测试结果，我们该怎么定位里面的缺陷呢？



比如我们只有一个测试用例是错的，那么我们可以定位到是蓝色区域的 bug。



如上图所示，如果测出来有两个错误，那么我们就可以定位到是  $X_1$  的  $[a,b]$  区间。

## 弱健壮等价类测试

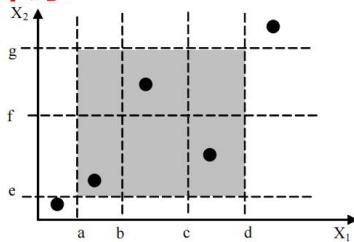
### 3) 弱健壮等价类测试

**弱：单缺陷**

**健壮：考虑无效值**

**提问：测试数取N、M的大者（3）再加无效值（2）。**

**对不对？**



因为要考虑健壮特性，所以我们要考虑无效值，放了两个测试用例。

Q：思考这样选取无效值对不对？

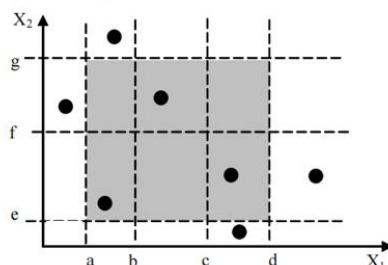
A：如果满足单缺陷的覆盖目的，那么就满足了要求。但是我们发现角落的这两个无效值测试里存在多缺陷耦合效应，如果出错不能定位到对应的变量。所以这样是不够的，我们需要继续增加无效值。

### 3) 弱健壮等价类测试

**弱：单缺陷！**

**健壮：考虑无效值**

**测试数：取N、M的大者（3）再加无效值（ $2 \times$ 变量个数）。**



我们希望做的测试方法最终和我们能和软件测试错误的定位一一对应，一一对应和软件错误是很难的，所以我们要去体验不同方法的时候，一定要把软件测试用例和用例的组合对应到用例的位置上。软件测试方法的产生还是依赖于数学基础的。

单缺陷假设总有点事情说不明白，但是单缺陷假设确实把测试用例数大大减少了，所以这里有一个平衡。我们把测试用例减少了以后就会发现可能漏掉很多事情。

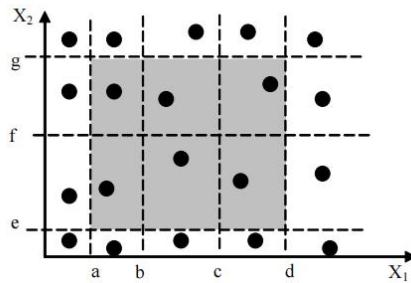
## 强健壮等价类测试

### 4) 强健壮等价类测试

强：多缺陷组合（笛卡儿积）

健壮：考虑无效值

测试数： $(N+2) \times (M+2)$ 。



这个方法在定位错误的时候就很方便。

我们测试的时候要掌握几个准则：

- 用最小的测试用例达到我们测试的目的。
- 测试完了以后要能准确地定位到里面的错误。

所以有的时候我们去评价测试方法的时候，我们就要非常实际地分析它的能力。



## 总结

### ④ 两种软件缺陷特征假设：

- 单缺陷假设（弱）
- 多缺陷假设（强）

### ⑤ 考虑无效输入

- 健壮性

### ⑥ 最坏情况

- 基于多缺陷假设

额外地，我们上传了一个工具使用的 PPT，下节课助教会带大家过一下，内容需要大家理一下。简单演示一下里面主要的功能，我们的作业要基于工具去做。作业主要是把我们学到的知识和工具结合起来。

2022/3/7

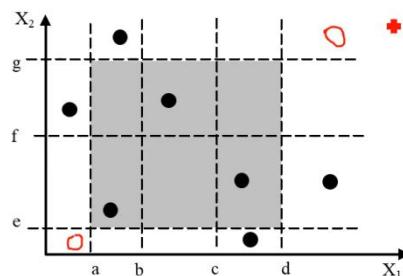
为什么这两个红的不行？

### 3) 弱健壮等价类测试

**弱：单缺陷！**

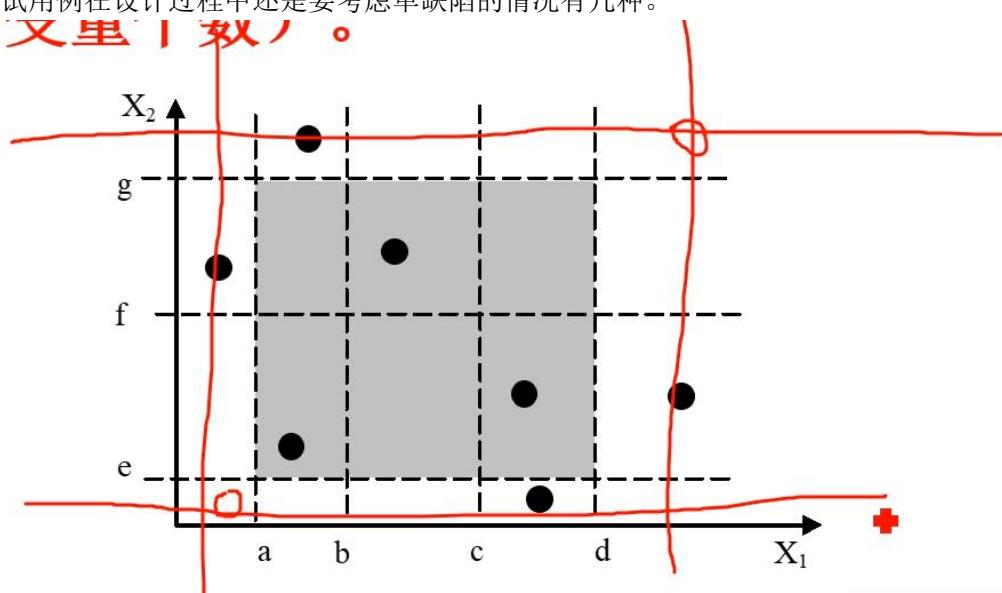
**健壮：考虑无效值**

**测试数：取N、M的大者（3）再加无效值（2×变量个数）。**

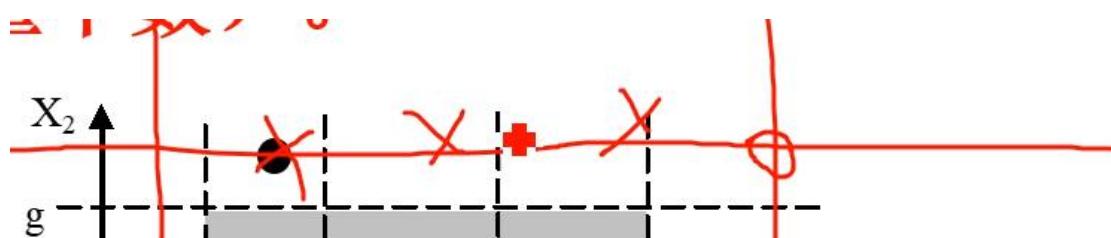


测试用例在设计过程中还是要考虑单缺陷的情况有几种。

**又里 1 双ノ。**



单缺陷要覆盖这4条红线（缺陷类型）。两个角落上的圆点是覆盖了两个缺陷，它是不符合单缺陷假设的。



覆盖这个缺陷的时候，三个位置都可以。大家还是要理解它的原理。

我们开始讲测试的实例，原来我们是针对两个典型例子去做。一个是三角形的问题和日期的问题。



## 测试实例1：三角形

### 三角形问题：

- 输入：a,b,c三个边长
- 输出：三角形类型

### 输出等价类：

- R1={等边三角形}
- R2={等腰三角形}
- R3={不等边三角形}
- R4={非三角形}

对于黑盒测试来讲，伪代码本身也不是特别主要。我们主要看问题本身的输入和输出。我们认为等边三角形和等腰三角形是没有交集的。

Q：等价类是如何选择的？

A：等价类之间是集合的划分，之间没有交集。对于我们系统输出来讲，等价类并在一起就成为了全集。



## 测试实例1：三角形

### 四个弱一般等价类测试用例

测试用例	a	b	c	预期输出
1	5	5	5	等边三角形
2	2	2	3	等腰三角形
3	3	4	5	不等边三角形
4	4	1	2	非三角形



## 测试实例1：三角形

考虑无效值的弱健壮等价类测试所增加的用例

测试用例	a	b	c	预期输出
1	-1	5	5	a取值小于最小值
2	5	-1	5	b取值小于最小值
3	5	5	-1	c取值小于最小值
4	201	5	5	a取值大于最大值
5	5	201	5	b取值大于最大值
6	5	5	201	c取值大于最大值

我们就要考虑增加相应的无效的输入。弱健壮等价类测试的时候，是单缺陷的。这样我们考虑无效的时候，我们只考虑abc单个维度的。我们增加相应的无效测试用例。



## 测试实例1：三角形

强健壮等价类测试,还要考虑无效值的组合,再增加的用例:

测试用例	a	b	c	预期输出
1	-1	-1	5	a、b取值小于最小值
2	5	-1	-1	b、c取值小于最小值
3	-1	5	-1	a、c取值小于最小值
4	-1	-1	-1	a、b、c取值小于最小值

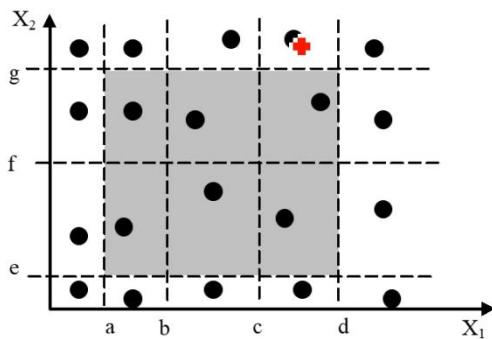
大于最大值，再增加的用例类似，不再说明。

#### 4) 强健壮等价类测试

强：多缺陷组合（笛卡儿积）

健壮：考虑无效值

测试数： $(N+2) \times (M+2)$ 。



#### 测试实例2：NextDate

##### ④ NextDate测试

- 输入：日、月、年
- D1 (日) = 1~31
- M1 (月) = 1~12
- Y1 (年) = 1812~2012

首先我们要做等价类的划分。

#### NextDate：弱一般等价类

用例	月份	日期	年	预期输出
1	6	12	2001	2001年6月13日

1900年是平年，2000年是闰年。所以对于有效输入等价类划分的时候，取决于大家是否了解问题的本身。大家可以试想一下，如果我们不了解日期的算法，我们可能只能如上图所示一个个划分等价类。如果我们将对系统本身运行的规律了解的话，我们就可以细化地去分。

所以测试系统的时候，还是取决于大家对系统的认知程度。每一种分法都对，但是效果是不一样的。这样测试的时候，测试用例的针对性和细化程度会很强。课本上给了这么一个最简单的分法，也就是我们只知道它的输入输出。

等价类只要符合集合划分条件，都是对的。

### ~~弱健壮~~测试用例：

用例	月份	日期	年	预期输出
1	6	12	2001	2001年6月13日
2	-1	12	2001	无效月份（负数）
3	13	12	2001	无效月份(>12)
4	6	-1	2001	无效日期（负数）
5	6	31	2001	无效日期(>31)
6	6	12	1811	无效年份(<1812)
7	6	12	2013	无效年份(>2012)

无效值可以有年份无效值、月份无效值和日期无效值。



### NextDate：弱一般等价类

- 强健壮等价类测试需要考虑无效值的组合,不再进一步说明.
- 但事实上,我们还可以进一步的分析,月份还有大月(31天)、小月(30天)、2月(28天)之分;日期有29、30、31的边界;而年还有平年和闰年之分;
  - 月={6, 7, 2}; 日={14, 29, 30, 31},
  - 年={1996、2000、2002};

关键是如何根据集合划分理论得到等价类。

## 5.3 基于决策表的测试

这也是一种黑盒测试的方法。

桩部分

决策表有四个部分：桩部分；条目部分；  
条件部分；行动部分。

桩	规则1	规则2	规则3、4	规则5	规则6	规则7、8
c1	T	T	T	F	F	F
c2	T*	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
		X		X		
a3		X		X		
a4			X			X

我们看第一个画红圈的 T 怎么解释。

为了使用决策表标识测试用例,可以把条件看作输入,把行动看成输出。仍以三角形问题为例, 进行说明。

c1:a,b,c 构成三角形	*	N	Y	Y	Y	Y	Y	Y	Y	Y
c2: a=b?	—	Y	Y	Y	Y	N	N	N	N	N
c3: a=c?	—	Y	Y	N	N	Y	Y	N	N	N
c4: b=c?	—	Y	N	Y	N	Y	N	Y	N	N
a1: 非三角形	X									
a2: 不等边三角形										X
a3: 等腰三角形					X		X	X		
a4: 等边三角形	X									
a5: 不可能			X	X		X				

条件部分, 我们是用 4 个条件来说明决策的过程。我们关注第一个画圆圈的 N。也就是 a,b,c 不能构成三角形, 那么对应到行动表的输出就是“非三角形”

为了使用决策表标识测试用例,可以把条件看作输入,把行动看成输出。仍以三角形问题为例, 进行说明。

c1:a,b,c 构成三角形	*	N	Y	Y	Y	Y	Y	Y	Y	Y
c2: a=b?	—	Y	Y	Y	Y	N	N	N	N	N
c3: a=c?	—	Y	Y	N	N	Y	Y	N	N	N
c4: b=c?	—	Y	N	Y	N	Y	N	Y	N	N
a1: 非三角形	X									
a2: 不等边三角形										X
a3: 等腰三角形					X		X	X		
a4: 等边三角形	X*									
a5: 不可能			X	X		X				

第二个能构成三角形, 但是不能确定唯一输出, 还要继续看其他条件。只有 c2,c3,c4 都

是 True 的时候，才对应了等边三角形。

如果条件换成：

- c1:a<b+c;
  - c2:b<a+c;
  - c3:c<a+b;
  - c4:a=b;
  - c5:a=c;
  - c6:b=c;
- 构造决策表的方法是一样的

## 例2：Nextdate 函数测试

第一次尝试，等价类如下：

M1 = {月：有30天}；

M2 = {月：有31天}；

M3 = {月：2月}；

D1 = {日：1~28}；

D2 = {日：29}；

D3 = {日：30}；

D4 = {日：31}；

Y1 = {年：闰年}；

Y2 = {年：平年}。

第一次我们把等价类这么划分。月份分成三类，日期分成四类，年分成闰年和平年。这样我们把对应的每一个变量都进行了等价类划分。

条件		
c1: 月在M1中?	*	
c2: 月在M2中?		
c3: 月在M3中?		
c4: 日在D1中?		
c5: 日在D2中?		
c6: 日在D3中?		
c7: 日在D4中?		
c8: 年在Y1中?		
a1: 不可能		
a2: NextDate		

共有 $2^8=256$ 条规则!

八个条件，每一个条件都有一个 True 和 False 两种情况，对应的规则就是 256 条规则。

Q: 我们拿这种情况做软件测试，我们应该怎么做呢？

同学 A 的回答： c1,c2,c3 只有一个 True。 c4,c5,c6,c7 只有一个 True。  $3 * 4 * 2 = 24$ .

老师标答 A: 决策表大家不要考虑太多的逻辑关系，想太多也不是决策表考察的内容。决策表对应的测试用例设计就是表里展示出来的。决策表通过这种列之间的关系，把测试用例一个个对应上去。

决策表有四个部分:桩部分;条目部分;条件部分;  
行动部分。

桩	规则1	规则2	规则3、4	规则5	规则6	规则7、8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

决策表测试很简单，关键是构建这张表。这个图里面有 6 个测试用例。大家想太多反而不好测，我们就按照每一个列的情况去设计测试用例就可以了。决策表关键就设置条件 c 和动作 a，我们如何来进行相应的设计。

条件		
c1: 月在M1中?		
c2: 月在M2中?		
c3: 月在M3中?		
c4: 日在D1中?		
c5: 日在D2中?		
c6: 日在D3中?		
c7: 日在D4中?		
c8: 年在Y1中?		
a1: 不可能	+	
a2: NextDate		

我们每一个都直接按照 T 和 N 来做，那么就有 256 个测试用例。

### 第二次尝试，注意力集中在闰年问题上

- M1 = {月：有30天}；
- M2 = {月：有31天}；
- M3 = {月：2月}；
- D1 = {日：1~28}；
- D2 = {日：29}；
- D3 = {日：30}；
- D4 = {日：31}；
- Y1 = {2000}；
- Y2 = {年：闰年,而且非世纪年}；
- Y3 = {年：平年}。

第二次尝试，对年份分的更多了。



## 第二次测试，主要考虑闰年问题

	1	2	3	4	5	6	7	8
c1:月份在	M1	M1	M1	M1	M2	M2	M2	M2
c2:日期在	D1	D2	D3	D4	D1	D2	D3	D4
c3:年份在	—	—	—	—	—	—	—	—
规则数	3	3	3	3	3	3	3	3
a1:不可能			X					
a2:日期增1	X	X			X	X	X	
a3:日期复位			X					X
a4:月份增1			X					?
a5:月份复位								?
a6:年增1								?

计算下一天的动作有不同的操作，比如日期+1、日期复位、月份+1、月份复位等。

第一列的例子：月份在 M1 (30 天)，日期在 (1~28 天)，年份就不管在哪一年，动作都是日期增加 1。

第八列的例子：月份在 M2 (有 31 天的月)，日期在 D4 (31 号)，这个情况能测吗？

eg: 2000/12/31, 2000/10/31。我们要满足和 action 的唯一的对应关系。

Q: Action 到底起什么作用？

A: 它对应的就是期望输出。一个测试包含测试用例和期望输出，对应期望输出我们就知道这个测试用例是否通过了。



## 第三次尝试，对2月的27进行特别划分：

- M1 = {月: 只有30天};
- M2 = {月: 有31天, 12月除外};
- M3 = {月: 12月};
- M4 = {月: 2月};
- D1 = {日: 1~27};
- D2 = {日: 28};
- D3 = {日: 29};
- D4 = {日: 30};
- D5 = {日: 31}
- Y1 = {年: 闰年};
- Y2 = {年: 平年}。



共22条测试规则

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
c1:月份在	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2	M3	M3	M3	M3	M3	M4						
c2:日期在	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2
c3:年份在	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—
行为			*																			
a1:不可能			X																			
a2:日期增1	X	X	X			X	X	X	X											X	X	X
a3:日期复位				X							X											
a4:月份增1				X							X										X	X
a5:月份复位																				X		
a6:年增1																			X			

\*例子的解读略。

我们发现每条规则都对应了唯一的 action。有的规则不考虑年份，有的则要考虑世纪年的情况。22 条规则把所有情况都对应上了。



进一步，我们考虑等价类的合并

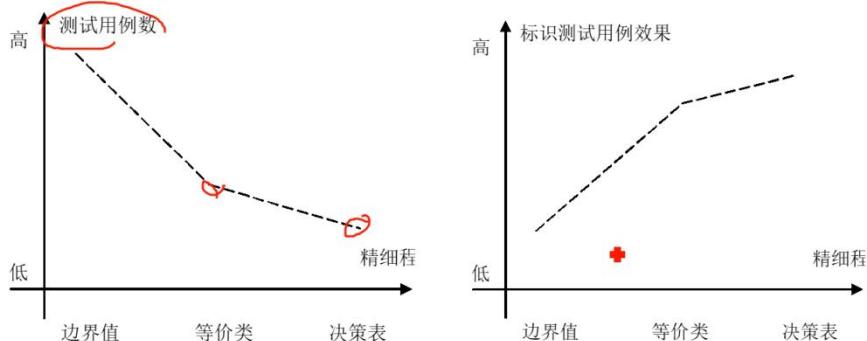
	1~3	4	5	6~9	10	11~14
c1:月份在	M1	M1	M1	M2	M2	M3
c2:日期在	D1,D2,D3	D4	D5	D1,D2,D3,D4	D5	D1,D2,D3,D4
c3:年份在	—	—	—	—	—	—
行为	*					
a1:不可能		X				
a2:日期增1	X			X		X
a3:日期复位		X			X	
a4:月份增1		X			X	
a5:月份复位						
a6:年增1						

- 从第一次的256个测试规则到现在的13个测试规则，测试等价类的完备性是一样的。
- 换言之，从决策表测试方法，我们可以充分地认识到，测试用例越多，并不能增加测试的完备性。而通过分析与合并，使用尽可能少的测试用例来达到测试完备性要求，正是测试技术研究的核心内容和目标。



## 5.4 测试的效率

从前面的介绍,我们显然可以得到以下的结论:



分析的颗粒度不一样,边界值只是从表面上去看,并没有分析逻辑关系。而决策表是分析了其中的逻辑关系。但是决策表的设计过程和代价是相对复杂的。



## 功能测试的指导方针

- 如果变量引用的是物理量, 可采用边界值分析和等价类测试;
- 如果变量是独立的, 可采用边界值分析和等价类测试;
- 如果变量不是独立的, 可采用决策表测试;
- 如果可保证是单缺陷假设, 可采用边界值分析和健壮性测试;
- 如果可保证是多缺陷假设, 可采用最坏情况测试、健壮最坏情况测试和决策表测试;
- 如果程序包含大量例外处理, 可采用健壮性测试和决策表测试;
- 如果变量引用的是逻辑量, 可采用等价类测试和决策表测试。

独立很好理解, 独立的逻辑就很简单, 决策表测试就无所谓。决策表适合变量依赖、逻辑复杂的情况。

黑盒测试就给大家讲完了。黑盒测试相对比较简单, 也要把功能黑盒测试的方法都要掌握。比较难的是灵活应用到不同的场景中。

## 结构性测试 (Structure Test)

白盒测试有一种非常明显的测试方法就是结构化测试。有两类不同的方法, 可以做路径的测试和数据流的测试。路径对应程序的结构。数据流对应数据使用的方法。



## 白盒测试

### ④ 结构性测试的方法：

- 路径测试
- 数据流测试

### ⑤ 结构性测试力求提高测试覆盖率。

Q：黑盒测试有没有覆盖率（边界测试、等价类测试、决策表测试）？

A：看似没有，实际上是有的。比如边界测试的弱一般等价类是  $\max(M, N)$ ，覆盖的是缺陷类型。等价类测试就是要求等价类的划分覆盖全集。黑盒测试有时候只给出了测试用例的设计过程，其实也隐含了覆盖率要求。

但是我们的结构性测试会专门讨论覆盖率的指标。

### ⑥ 结构性测试主要用于软件验证。

- “我们在正确地构造一个系统吗？”
- ⑦ 功能性测试是一种确认技术。
  - “我们在构造一个正确的系统吗？”



## 黑盒测试与白盒测试的比较

- ⑧ 黑盒测试：从用户观点出发，按规格说明书要求的输入数据与输出数据的对应关系设计测试用例。因此它是根据程序外部特征进行测试。
- ⑨ 白盒测试：根据程序内部逻辑结构进行测试。
- ⑩ 这两类测试方法是从完全不同的起点出发，并且是两个完全对立的出发点。这两类方法各有侧重，在测试的实践中都是有效和实用的。在进行单元测试时大都采用白盒测试，而在系统测试中采用黑盒测试。



## 有了“黑盒”测试，为什么还要“白盒”测试？

- ⑤ 黑盒测试是从用户的观点出发，根据程序外部特性进行的测试。如果外部特性本身有问题或规格说明的规定有误，用黑盒测试方法是发现不了的。
- ⑥ 黑盒测试只能观察软件的外部表现，即使软件的输入输出都是正确的，却并不能说明软件就是正确的。因为程序有可能用错误的运算方式得出正确的结果，这种情况只有白盒测试才能发现真正的原因。
- ⑦ 白盒测试能发现程序里的隐患，像内存泄漏、误差累计问题。在这方面，黑盒测试存在严重的不足。

2022/3/14

大家要保持乐观和积极的状态。对于线上课程来讲我们也经历过，问题不是很大。这两天我也在给大家送饭。这节课是实践课，主要讲课程作业的软件和环境怎么去配。

我们今天讲的主要内容是第二次作业白盒测试可能用到的单元测试的软件。我们会从以下几个点来讲：

1. Java
2. JUNIT
3. EMMA
4. IDEA
5. Mock
6. PITest

EMMA：代码覆盖率测试插件

最后结合 IDEA 来讲 JUNIT 和 Mock 的使用。

我们这一讲会以 Java 来讲，我们建议以课程项目作业来做单元测试。Java 拿到源代码和文档容易，且它的测试环境完备。

### Why is Java ?

- Easy to get source code and documentation
- Completed test environment
- More free test tools

### 安装JDK ( Java Development Kit )

- JDK 是整个Java的核心，包括了Java运行环境，Java工具和Java基础的类库。

要使用 Java 要先安装 JDK。我们要下载哪个版本的 JDK 呢？对于我们的课程作业，只需要使用 SE 标准版。我们推荐至少下载 JDK8。

- ④ JDK共有3个版本，分别为：
  - SE(Java SE), standard edition, 标准版
  - EE(Java EE), enterprise edition, 企业版
  - ME(Java ME), micro edition, 主要用于移动设备、嵌入式设备上的java应用程序
- ④ 推荐下载JDK版本：Java SE Development Kit 8u201
- ④ 下载地址：
  - <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- ④ 目前较新的JDK版本为：Java SE Development Kit 15。
- ④ 下载地址：
  - <https://www.oracle.com/java/technologies/javase-downloads.html#JDK15>

Windows 在安装后还需要配置环境变量。

- ④ 安装与配置
  - 下载与安装，假设安装到C:\Program Files\Java\jdk1.8.0\_60
  - 配置PATH：相当于C++中的GCC配置
    - 我的电脑点右键，选择“属性”，选择“高级”标签，进入“环境变量设置，在系统变量里找到PATH变量，选择→编辑，里面已经有很多的变量值，变量值的最前面加上C:\Program Files\Java\jdk1.8.0\_60\bin;
  - 配置CLASSPATH：寻找标准类库位置
    - 我的电脑点右键，选择“属性”，选择“高级”标签，进入环境变量设置，在系统变量那一栏中点新建→CLASSPATH，并且设置
      - classpath=.; C:\Program Files\Java\jdk1.8.0\_60\lib;C:\Program Files\Java\jdk1.8.0\_60\lib\tools.jar;

```
+ ~ java -version
java version "17.0.1" 2021-10-19 LTS
Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)
```

我们怎么知道已经配置成功了，通过 `java -version` 即可。

## Junit 的使用

JUnit 是一个回归测试框架，因为是白盒测试，所以需要代码。

Q: 为什么要使用 JUnit?

A: 除了我们写逻辑代码，写测试代码也是很重要的一部分。



- ④ For application developers, testing forms an integral part of the development life-cycle.
- ④ As old code is modified for reasons ranging from adding new functionality to speed optimizations, the risk of dependent functionality getting broken increases.
- ④ Tests that are **written once**, and can **be run repeatedly** is the only solution to avoid manually QA testing.
- ④ To write these tests, for applications written in java, JUnit provides a solid framework for writing unit tests. Many JUnit integrations also exist to integrate your JUnit tests into build and development tools.

我们希望大家在写自己项目的测试用例的时候，尽可能想着把测试用例写完一遍之后，后续程序的迭代中尽可能复用测试用例。



如果我们不使用 IDEA，可以用最传统的方式安装 JUnit。

## 常见JUnit5标注

注解	描述
@Test	表示方法是一种测试方法。
@ParameterizedTest	表示方法是参数化测试
@RepeatedTest	表示方法是重复测试模板
@DisplayName	为测试类或者测试方法自定义一个名称
@BeforeEach	表示方法在每个测试方法运行前都会运行
@AfterEach	表示方法在每个测试方法运行之后都会运行
@BeforeAll	表示方法在所有测试方法之前运行
@AfterAll	表示方法在所有测试方法之后运行
@Disabled	用于禁用测试类或测试方法

JUnit 的工作方式：我们注意到有很多注解，它也分了好几类注解。以 Test 为后缀的注解就是用来注册我们的测试代码的。

DisplayName 主要在测试报告的时候显示什么样子。

BeforeEach 就是测试的前期准备工作，其余同理。

Disabled 就是禁用某个测试类。

## 常见JUnit测试代码结构

```
public class CalculatorTest {  
    private Calculator calculator;  
  
    @BeforeEach  
    void setUp()  
    {  
        calculator=new Calculator();  
    }  
  
    @AfterEach  
    void tearDown()  
    {  
    }  
  
    @Test  
    void testAddMethod()  
    {  
        assertEquals( expected: 2,calculator.add( a: 1, b: 1));  
    }  
}
```

这个代码主要是对 caculator 的类做了测试。assertEquals 就是我们希望 caculator 执行  $1+1$  的值为 2。

JUnit会以以下顺序执行测试：（大致的代码）

```
④ try {  
    HelloTest test = new HelloTest(); // 建立测试类  
    实例  
    test.setUp(); // 初始化测试环境  
    test.testAdd(); // 测试某个方法  
    test.tearDown(); // 清理资源  
}  
catch...
```

- ④ `setUp()`是建立测试环境，这里创建一个Calculator类的实例；
- ④ `tearDown()`用于清理资源，如释放打开的文件等等。

这两个非常像构造函数和析构函数。

- ④ 以test开头的方法被认为是测试方法，JUnit会依次执行`testXxx()`方法。如果方法返回值与期待结果相同，则`assertEquals`不会产生异常。
- ④ 如果有多个`testXxx`方法，JUnit会创建多个XxxTest实例，每次运行一个`testXxx`方法，`setUp()`和`tearDown()`会在`testXxx`前后被调用，因此，不要在一个`testA()`中依赖`testB()`。
- ④ JUnit通过单元测试，能在开发阶段就找出许多Bug，并且，多个Test Case可以组合成Test Suite，让整个测试自动完成，尤其适合于XP方法。
- ④ 每增加一个小的新功能或者对代码进行了小的修改，就立刻运行一遍Test Suite，确保新增和修改的代码不会破坏原有的功能，大大增强软件的可维护性，避免代码逐渐“腐烂”。

我们将逻辑代码和测试代码分离，每次改动都可以跑一次之前的Test，也可以加一些新的Test进去。

提到单元测试，我们肯定会考虑测试覆盖率。测试覆盖率也是评价测试完整的重要指标之一。

### ④ 测试覆盖率 (Code Coverage)

- ④ 测试覆盖率，简单的说，就是评价测试活动覆盖产品代码的指标。测试的目的，是确认产品代码按照预期一样工作，也可以看作是产品代码工作方式的说明文档。进一步考虑，测试覆盖率可以看作是产品代码质量的间接指标——之所以说是间接指标，因为测试覆盖率评价的是测试代码的质量，并不是产品代码的质量。
- ④ 代码覆盖率是一种白盒测试，因为测试覆盖率是评价产品代码类内部的指标，而不是评价系统接口或规约。测试覆盖率尤其用于评价测试代码是否已经覆盖了产品代码所有的路径。



- ④ EMMA 是一种用来衡量java软件覆盖率的软件。它对于不可达代码的检测，以及验证测试集和交互式测试对程序的覆盖情况是很必要的。
- ④ EMMA的设计目标有以下几个，
  - ④ 1. 在不增加创建或执行开销的情况下，提供丰富的覆盖分析数据；
  - ④ 2. 促进团队开发效率，同时，也能加快单元开发—测试周期；
  - ④ 3. 促进小型java软件的开发效率，同时，提高包含大量类的大型企业软件的开发。
- ④ EMMA与其他覆盖测试工具的区别在于它面向极限编程，目的是促使循环的开发—测试软件开发模型。

我们在衡量测试覆盖率指标的时候，有很多覆盖指标。基本块的覆盖就是把没有分支的代码合成基本块。Decision Coverage 就是 if-else 分支是否都走到。还有路径覆盖评价代码开始到结束的所有路径是否都覆盖了。

- ④ 衡量测试覆盖率的指标很多，常用的指标有：
- ④ Statement coverage，也称作Line coverage，用于评价测试的代码语句覆盖率。
- ④ Basic block coverage，是Statement coverage的一个变种，它把没有一个分支的代码区域作为一个计量单位，而不是简单的代码行，用于一个if-else分支代码行数远远大于另一个的情况，在这种情况下，statement coverage 指标并不适用。
- ④ Decision coverage（也称作Branch coverage），用于评价代码分支地测试覆盖率。
- ④ Path coverage，和Decision coverage相似，用于评价代码从开始到结束所有路径的测试覆盖率。
- ④ Function coverage，用于评价代码方法的测试覆盖率。
- ④ EMMA目前支持四种Coverage类型：
  - ④ class、method、line和basic block。

# IDEA

我们来讲一下在 IDEA 下怎么使用 JUnit。具体来说，它内置了 JUnit，也内置了测试覆盖度的工具，但是相对来说覆盖指标比较简单。



- ④ 内置JUnit4&5
- ④ 内置测试覆盖度工具
- ④ 测试类生成插件
- ④ 导出测试报告及测试覆盖率报告



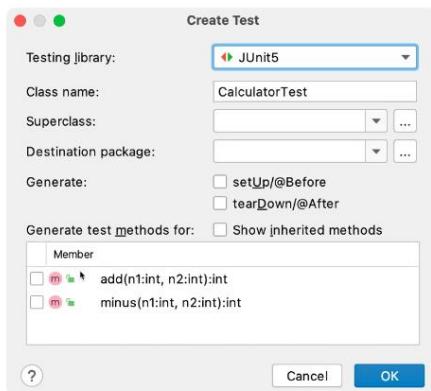
- ④ 新建java-demo项目，采用maven进行包管理
- ④ 待测对象：Calculator类
- ④ 为了保持良好的代码结构，一般将测试类与主程序放置在不同的根目录下

The screenshot shows the IntelliJ IDEA interface. On the left is the Project tool window displaying a Java project named 'java-demo'. It contains a 'src' directory with 'main' and 'test' sub-directories. Under 'main/java', there is a 'Calculator' class. The code editor on the right shows the 'Calculator.java' file with the following code:

```
public class Calculator {  
    public int add(int n1,int n2)  
    {  
        return n1+n2;  
    }  
  
    public int minus(int n1, int n2)  
    {  
        return n1-n2;  
    }  
}
```

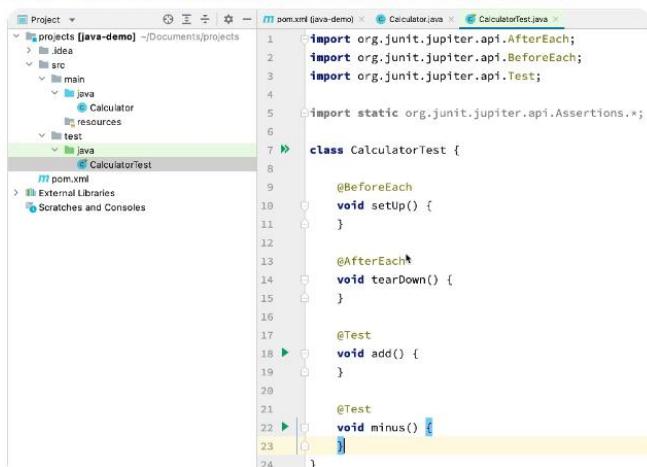
我们下面的例子就是 Calculator 类为测试类。

- ④ 选择合适的JUnit版本，待测类名以及待测类的成员函数



我们可以设置测试类生成的文件里要包含那些函数的测试。

## ④ 自动生成对应类的测试代码文件与代码框架



The screenshot shows an IDE interface with a project structure on the left and two code editors on the right. The left editor shows a Maven project named 'java-demo' with a 'src' folder containing 'main' and 'test' subfolders. The 'main/java' folder contains 'Calculator.java' and 'resources'. The 'test/java' folder contains 'CalculatorTest.java'. The right editor shows the content of 'CalculatorTest.java':

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void add() {
    }

    @Test
    void minus() {
    }
}
```

有些同学会发现 Import 报红。

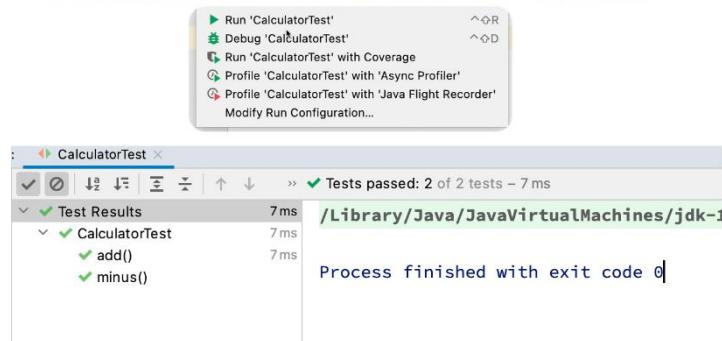
## ⑤ 若提示junit包导入失败，在pom.xml中加入junit依赖



我们写完测试逻辑之后，就可以开始测试。



## ⑥ 右键test类或在test类文件中点击 ➡ 图标，进行测试

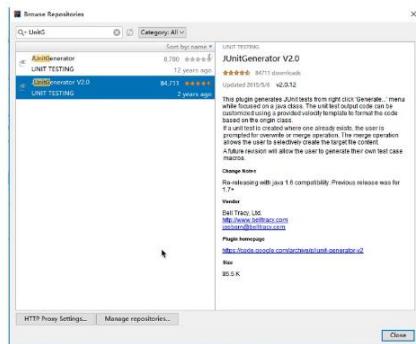


第三个选项就是拿到简单测试报告。



## 使用插件生成测试代码

- ④ 打开File->Settings->Plugins->Browse Repositories，搜索JUnitGenerator并安装（可能要挂代理）。



插件也可以生成测试代码模板，可以在 IDEA 中搜索插件 JUnitGenerator 下载。



## 生成测试代码

在被测代码文件里右键选择Generate->JUnit Test->JUnit 3/4。或者使用快捷键Alt+Insert。



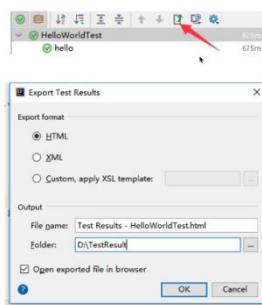
该插件会为测试类下的每个方法生成测试函数，可以使用 IntelliJ自带的方式选择只为部分函数生成测试函数

也能得到和我们之前创建测试模板一样的测试母版。对 IDEA 来说，可以导出测试报告。

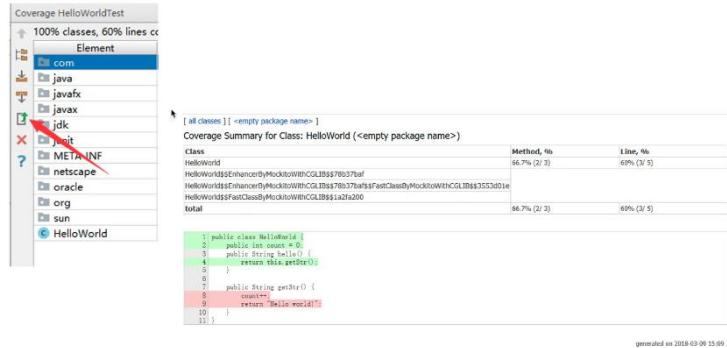


## 导出测试报告

在测试控制窗口界面点击Export Test Results，选择导出格式和路径。



在测试覆盖率窗口点击**Generate Coverage Report**, 选择导出路径。



我们还可以导出测试覆盖率的报告，也就是对每个程序，类、方法、代码行数各自覆盖了多少。当然在这里看到 100%，并不代表代码的路径都覆盖了，需要大家在设计测试用例的时候考虑。

还有一些重要的功能。

- ④ **参数化测试：** 测试用例与测试代码分离，提高测试代码效率，方便测试用例的替换。
  - JUnit4/5
  - TestNG
- ⑤ **Mock（打桩）：** 模拟对其他模块的调用，避免测试代码被外部代码干扰。
  - Mockito
  - PowerMock
- ⑥ **变异测试：** 修改源码使其产生变异体，观察变异体的测试结果，以评判测试用例的充分性。
  - Pitest

JUnit，因为有 `assertEqual`，测试用例和测试代码并没有分离。参数化测试就可以帮我们分离。

第二个就是 Mock 功能，模拟对其他模块的调用，假如 Calculator 依赖于另一个类，但是那个类没有源代码或者没有实现完成，我们可以使用 Mock 来模拟那个类。我们会将 Mockito。

变异测试主要是衡量我们的测试用例是否充分，会讲 Pitest。



- ④ JUnit4中已经包含了对参数化测试的支持，但是代码编写较为繁琐
  - JUnit5对参数化测试的接口进行了简化。

#### JUnit4参数化测试代码示例

```
// annotate this test as parameterized test
@RunWith(Parameterized.class)
public class HelloWorldTest {
    // variables to be tested
    private int absInput;
    private int absResult;

    // parameterized constructor
    public HelloWorldTest(int absInput, int absResult) {
        super();
        this.absInput = absInput;
        this.absResult = absResult;
    }

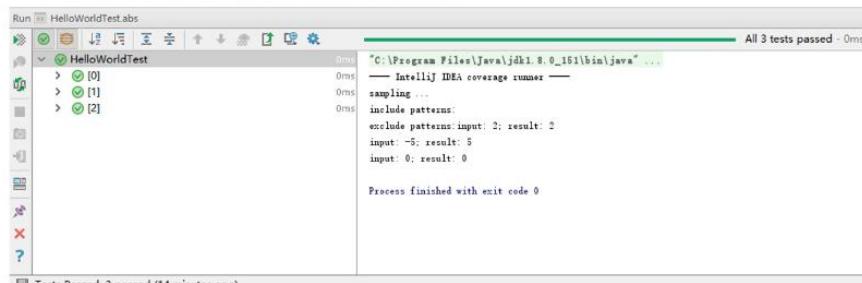
    // prepare test data
    @Parameterized.Parameters
    public static Collection<Object[]> data0 {
        Object[][] data = new Object[][] {
            { 2, 2 },
            { -5, 5 },
            { 0, 0 }
        };
        return Arrays.asList(data);
    }

    @org.junit.Test
    public void abs0() {
        System.out.printf("%dinput: % result: %n", absInput, absResult);
        HelloWorld helloWorld = new HelloWorld();
        int actualResult = helloWorld.abs(absInput);
        assertEquals(absResult, actualResult);
    }
}
```

我们现在有 HelloWorldTest，Object 来存我们的测试数据。我们用注解标记为参数，我们在测试的时候就调用 Object 进行测试，最后我们的测试逻辑还是一样的。



- ④ @RunWith(Parameterized.class)注解告诉Junit这是一个参数化测试类
- ④ 声明变量作为测试参数
- ④ 实现测试类的构造器，以测试变量为参数
- ④ 实现一个public static类型的函数，返回测试数据（固定写法）
- ④ 测试方法中使用测试参数进行测试



一般来说在 JUnit4 里会设计静态的函数来返回测试数据，看起来会比较麻烦一些。

### ④ JUnit5中的参数化测试

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

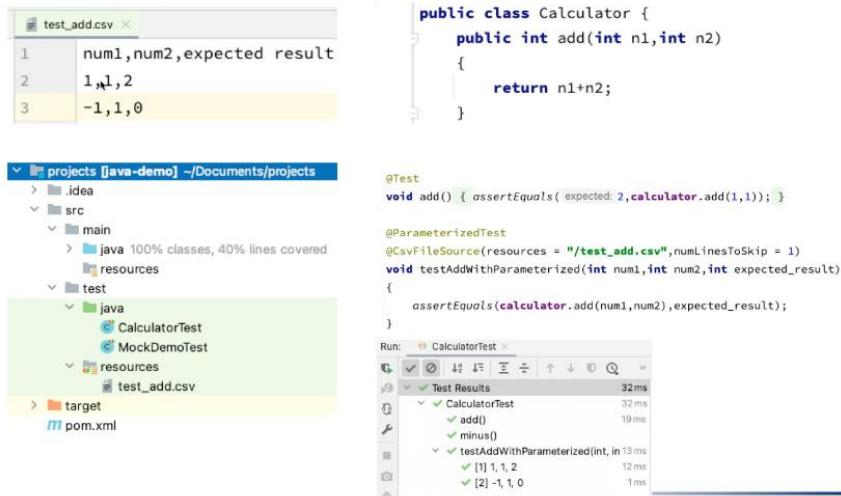
```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String first, int second) {
    assertNotNull(first);
    assertEquals(0, second);
}
```

*two-column.csv*

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

在 JUnit5 中，对参数化测试需要设置注解`@ParameterizedTest` 告诉它数据源是要从其他地方获取的，我们可以设置`@CsvFileSource` 从文件中读取（第一个填入 `first`，第二个填入 `second`），或者`@ValueSource` 直接设置。

### ④ JUnit5中的参数化测试的使用样例



The screenshot shows the following components:

- CSV File:** `test_add.csv` containing three rows of data:

num1	num2	expected result
1	1	2
-1	1	0

- Java Class:** `Calculator` with a method `add`.

```
public class Calculator {
    public int add(int n1,int n2) {
        return n1+n2;
    }
}
```

- Project Structure:** A Java project named `java-demo` with the following structure:
  - `src/main/java`: Contains `Calculator.java` and `MockDemoTest.java`.
  - `src/test/java`: Contains `CalculatorTest.java` and `MockDemoTest.java`.
  - `src/test/resources`: Contains `test_add.csv`.
  - `pom.xml`
- Test Results:** A screenshot of the IDE's "Run" window showing the execution of `CalculatorTest`. It runs the `add()` method with parameterized data from the CSV file, showing two successful test cases.

我们在 CSV 中放入三行数据。我们在 `test` 中新建一个 `resources` 文件夹，把 csv 放到里面去，然后使用参数化测试。如果把测试数据耦合进代码里，后续测试也会很麻烦。这是比较好的做法。

## MOCK 工具

它主要是模仿一些类的行为，屏蔽掉那些类里可能有的 BUG 对我们待测试类的影响。

## ④ Mockito: Tasty mocking framework for unit tests in Java

<https://site.mockito.org/>

- 对数据对象的 mock 的原则：如果不包含业务代码，mock 用处不大；如果包含业务代码，则应当用 mock 将测试用例与其分离

## ④ 主要接口

- mock() / @Mock: create mock
  - optionally specify how it should behave via Answer / ReturnValues / MockSettings
  - when() / given() to specify how a mock should behave
  - If the provided answers don't fit your needs, write one yourself extending the Answer interface
- spy() / @Spy: partial mocking, real methods are invoked but still can be verified and stubbed
- @InjectMocks: automatically inject mocks/spies fields annotated with @Spy or @Mock
- verify(): to check methods were called with given arguments
  - can use flexible argument matching, for example any expression via the any().
  - or capture what arguments were called using @Captor instead

如果 mock 的对象只是一个简单的数据对象，不用 mock 也是影响不大的，但是如果包含了业务代码，最好还是 mock 一下来把待测试代码分离。它的主要接口有 4 个：

- mock，我们指定一个类，它在调用某个函数的时候应该返回哪个值。类原本的方法不会被调用。
- spy，类对应方法会被调用一次，但是会返回我们想要的值。
- @InjectMocks 自动化注释
- verify，和 assert 语句比较像。

## ④ Maven 导入:

Apache Maven  
[maven.apache.org](http://maven.apache.org)

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-core</artifactId>
<version>3.8.0</version>
</dependency>
```

## ④ Gradle 导入:

```
repositories { mavenCentral() }
dependencies { testImplementation "org.mockito:mockito-core:3.+" }
```

然后具体来说，mock 的具体使用如下：



## Mockito代码示例

### ④ Calculator.java

```
public class Calculator {  
    public int add(int n1,int n2) { return n1+n2; }  
    public int minus(int n1, int n2) { return n1-n2; }  
    public String info()  
    {  
        *  
        return this.getInfo();  
    }  
    public String getInfo()  
    {  
        return "Test info";  
    }  
}
```

### ⑤ CalculatorTest.java

```
private Calculator calculator;  
  
@BeforeEach  
void setUp() {  
    calculator=spy(Calculator.class);  
    when(calculator.getInfo()).thenReturn("Mock info");  
}  
@Test  
void getInfo()  
{  
    assertEquals(expected: "Mock info",calculator.getInfo());  
}
```

假如 calculator 的 info 函数就是调用内部的 getInfo 函数。我们用 spy 把 calculator 对象 mock 住，当我们调用 getInfo 的时候，它会返回 “Mock info”，和它本身的实现是不同的。我们 Mock 之后不会采样它原有的输出。

## PIT



## 变异测试工具

PIT is a state of the art **mutation testing** system, providing **gold standard test coverage** for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling



pitest.org

### pox.xml

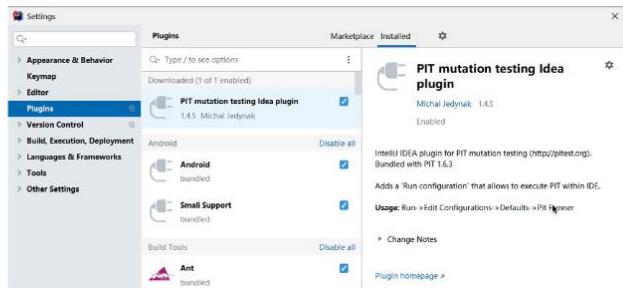
```
<plugin>  
    <groupId>org.pitest</groupId>  
    <artifactId>pitest-maven</artifactId>  
    <version>LATEST</version>  
</plugin>
```

它的使用方法也是引入插件。



## PiTTest

### ④ Install IDEA plugin



### ⑤ Add Maven dependency (for JUnit 5)

```
<dependency>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-junit5-plugin</artifactId>
    <version>0.12</version>
</dependency>
```

第一次作业的黑盒测试可以使用变异测试自动化地注入一些软件错误。可以有效地去验证我们的一些方法。实际上 4 次作业是不包含变异测试的。



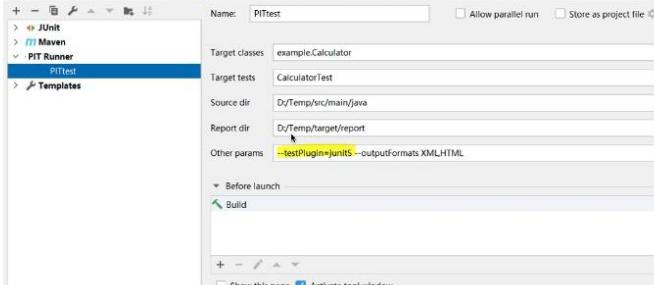
## PiTTest

### ⑥ Run default mutation coverage goal

```
mvn org.pitest:pitest-maven:mutationCoverage
```

### ⑦ Configurations

- Run->Edit Configurations->PIT Runner



### ⑧ Run



这个配置需要配置对。然后我们就可以在 IDEA 里进行变异测试了。它的总体想法就是往代码里注入一些错误。

## ④ Output / Report in target/pit-reports/YYYYMMDDHHMI

```

Statistics
>> Generated 6 mutations Killed 1 (17%)
>> Ran 1 tests (0.17 tests per mutation)

Mutators
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalBoundaryMutator
  Generated 1 killed 0 (0%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

> org.pitest.mutationtest.engine.gregor.mutators.EmptyTableMutator
  Generated 1 killed 0 (0%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

> org.pitest.mutationtest.engine.gregor.mutators.InvertNegateMutator
  Generated 1 killed 0 (0%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

> org.pitest.mutationtest.engine.gregor.mutators.ReturnAllMutator
  Generated 2 killed 1 (50%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

> org.pitest.mutationtest.engine.gregor.mutators.ReturnConditionalMutator
  Generated 1 killed 0 (0%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

> org.pitest.mutationtest.engine.gregor.mutators.ReplaceConditionalMutator
  Generated 1 killed 0 (0%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

> org.pitest.mutationtest.engine.gregor.mutators.ReplaceMethodMutator
  Generated 1 killed 0 (0%)
  KILLED & SURVIVED 0 TIMED_OUT 0 NO_MUTABLE
  NO_COVERAGE 1

```

## Pit Test Coverage Report

### Project Summary

Number of Classes	Line Coverage	Mutation Coverage
1	50% 2/4	17% 1/6

### Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
com.sjtu.yanjasen4.hello	1	50% 2/4	17% 1/6

Report generated by Pit 1.3.2

## Pit Test Coverage Report

### Package Summary

#### com.sjtu.yanjasen4.hello

Number of Classes	Line Coverage	Mutation Coverage
1	50% 2/4	17% 1/6

### Breakdown by Class

Name	Line Coverage	Mutation Coverage
HelloWorld.java	50% 2/4	17% 1/6

Report generated by Pit 1.3.2

我们跑了之后，它会出来一个网页版的报告，有哪一些注入的错误没有被抓到，这就是说明我们的测试用例需要进一步地改进。

## ④ Report in target/pit-reports/YYYYMMDDHHMI

### HelloWorld.java

```

1 package com.sjtu.yanjasen4.hello;
2
3 public class HelloWorld {
4     public String hello() {
5         return this.getStr();
6     }
7
8     public String getStr() {
9         return "Test String";
10    }
11
12    public int abs(int n) {
13        return n >= 0 ? n : (-n);
14    }
15 }

Mutations
5 1. mutated return of Object value for com.sjtu.yanjasen4.hello/HelloWorld::hello to ( if (x != null) null else throw new RuntimeException ) → KILLED
6 1. mutated return of Object value for com.sjtu.yanjasen4.hello/HelloWorld::getStr to ( if (x != null) null else throw new RuntimeException ) → NO_COVERAGE
7 1. changed conditional boundary → NO_COVERAGE
8 3. removed negation → NO_COVERAGE
9 4. negated conditional → NO_COVERAGE
10 4. replaced return of integer sized value with (x == 0 ? 1 : 0) → NO_COVERAGE

```

### Active mutators

- INCREMENTS MUTATOR
- VOID METHOD CALL MUTATOR
- RETURN VALS MUTATOR
- MATH MUTATOR
- NEGATE CONDITIONALS MUTATOR
- EERT MSS MUTATOR
- CONDITIONALS\_BOUNDARY MUTATOR

### Tests examined

- com.sjtu.yanjasen4.hello.HelloWorldTest.hello(com.sjtu.yanjasen4.hello.HelloWorldTest) (140 ms)

Report generated by Pit 1.3.2

绿色就是把一些返回值给变掉了，测试用例抓住了就是完备的（KILLED），否则就是NO\_COVERAGE。

# 变异测试修改的东西



## Mutate Operator

### ④ Conditionals Boundary Mutator

- $</ <=$ ,  $>/ >=$

### ④ Negate Conditionals Mutator

- $== / !=$ ,  $<= / >$ ,  $>/ <=$

### ④ Remove Conditionals Mutator

- $a == b / true$

### ④ Math Mutator

### ④ Increments Mutator

### ④ Return Values Mutator

它会修改一些测试条件语句和数学计算、返回值等。这些工具主要是通过插件修改源代码的方式来看测试指标。还有一个我们会讲测试计划怎么写。

# 测试计划怎么写



### 1、简介

#### 1.1 编写目的

#### 1.2 项目背景

#### 1.3 测试范围

#### 1.4 参考文档

### 2、测试参考文档和测试提交文档

#### 2.1 测试参考文档

#### 2.2 测试提交文档

### 3、测试进度

### 4、测试资源

#### 4.1 人力资源

#### 4.2 测试环境

#### 4.3 测试工具

### 5、测试策略



## 5.1功能测试

### 5.1.1 XXX系统管理软件功能测试

### 5.1.2 XXX系统实时通信软件功能测试

### 5.1.3 目标三维动态显示模型及可视化软件功能测试

### 5.1.4 XXX三维动态显示模型及可视化软件功能测试

### 5.1.5 XXX交会显示软件功能测试

### 5.1.6 XXX系统数据管理软件功能测试

### 5.1.7 XXX文件与AutoCAD、Creator的转换软件功能测试

## 5.2性能测试

### 5.2.1 通讯性能测试

### 5.2.2 实验性能测试

### 5.2.3 数据库性能测试

### 5.2.4 显示性能测试

### 5.2.5 其它性能测试

## 5.3故障测试

### 5.3.1 网络故障

### 5.3.2 服务器故障

### 5.3.3 管理计算机故障

### 5.3.4 计算节点机故障

### 5.3.5 图形工作站故障测试

## 5.4安全性测试

### 5.4.1 管理软件安全性测试

### 5.4.2 数据管理软件安全性测试

▲ CC BY



# 1. 简介

- ④ 1.1 编写目的
  - ④ 为了全面、系统地对“XXX支撑软件与可视化软件”进行评估与测试，从而保证系统长期稳定的运行，组织对该软件进行系统的总体综合测试。
- ④ 1.2 项目背景
  - ④ “XXX验证平台”是中国兵器工业XXX研究所承担的国防基础科研项目。XXX所与XXX大学合作，对该项目的关键技术——XXX支撑技术和可视化技术展开研究。本项目就是在此背景下开展的，通过应用虚拟试验技术、计算机网络技术、可视化技术、数据库技术等，研制“XXX的支撑软件和可视化软件”，并最终集成在“XXX中。
- ④ 本软件系统由以下7个二级软件组成：  
    系统管理、实时通讯、三维显示、.....

- ④ 1.4参考文档
- ④ 《GB 8566-88 计算机软件开发规范》
- ④ 《GJB 438A-97 武器系统软件开发文档》
- ④ XXXXXXXX

## 2、测试参考文档和测试提交文档

- ④ 2.1测试参考文档
- ④ 本测试计划参照GB8567——88标准编制，下表列出了制定测试计划时所使用的文档，并标明了各文档的可用性：

### 2.2测试提交文档

在测试完成后，需要提交给用户相关的测试评估报告与改进报告，并提交详细的测试报告。



上海交通大学  
SHANGHAI JIAO TONG UNIVERSITY

## 3、测试进度

测试活动	计划开始日期	实际开始日期	结束日期
单元功能测试	2006年 3月16日	2006年3月16日	2004年3月26日
集成功能测试	2006年3月27日	2006年3月27日	2006年3月31日
性能测试	2006年4月2日	2006年4月3日	2006年4月5日
故障测试	2006年4月8日	2006年4月8日	2006年4月9日
安全测试	2006年4月10日	2006年4月10日	2006年4月11日
对测试进行评估	2006年4月15日	2006年4月16日	2006年4月18日
用户验收测试	2006年5月11日		

## 4.2 测试环境

下表列出了测试的系统环境：

机器名 环境	数据库服务器	图形工作站	管理计算机
软件环境 (相关软件、 操作系统等)	Windows XP Oracle 9i		Windows XP
硬件环境 (设备、网 络等)	Xeon Mp 2.7×2 CPU/4smp/2GB/DVD/1 000M×2	HP Workstation xw6200	HP Workstation xw6200

## 4.3 测试工具

下表列出此项目测试中使用的工具：

用途	工具名称	生产厂商/自产	版本
压力与性能测试	Jmeter	Jakarta	1.9.1
压力测试网络检测	Webstress	国外软件	6.18
申请一定量的内存	Mem	自产	1.0

测试策略是比较看重的一点。我们考虑功能测试的哪些点选用的测试方法是什么样的。

## 五、测试策略

### 5.1 功能测试

由于对测试对象的功能测试应侧重于所有可直接追踪到用例或业务功能和业务规则的测试需求。这种测试的目标是核实数据的接受、计算、处理和输出是否正确，以及业务规则的实施是否恰当，所以此次的功能测试全部为黑盒测试，测试的主要目的是检测以下错误：

- 1) 是否有不正确或遗漏的功能？
- 2) 在接口上，输入是否能正确的接受？能否输出正确的结果？
- 3) 是否有数据错误或外部信息（例如数据文件）访问错误？
- 4) 是否有初始化或终止性错误？

第一次作业关注的是等价类的划分、怎么设计测试用例，对边界值怎么分析等。

## 作业一：黑盒测试

目的：熟练掌握黑盒测试方法原理、测试过程、测试环境与工具。

要求：

1. 寻找一个具有需求说明的软件，要求被测软件具有至少五个独立功能，如人员管理系统、线上购物网站等，建议采用课程项目进行测试。
2. 对不少于5个独立功能进行黑盒测试，在边界值、等价类、决策表三种测试方法中任选两种方法生成测试用例，并在测试计划中体现。
3. 执行测试用例并记录测试结果，并完成相应的文档：
  1. 《测试计划》
  2. 《测试报告》
  3. 被测软件《需求规格说明书》

备注：

1. 在测试报告中包含小组成员贡献与比例
2. 若被测软件本身无需求说明书，需要按照模板撰写需求规格说明书
3. 提交方式：将项目打包成zip，以小组编号命名后提交到canvas中

软件测试的目的是找到里面的错误，找到了以后软件错误就结束了。作业不需要改正了。可以说黑盒测试为什么要增加白盒测试做补充。

1. 项目最好有源码。
2. 不少于5个独立功能只是一个参考，让每个同学都参与到测试工作里面去。作业要求用现实中的测试工作去做。
3. 这些文档是必要的。需求规格说明书是要对应到我们黑盒测试里去的。其实软件工程对于规范化的要求是有一些的。评分的时候，规范性会成为其中的一部分。
4. 因为同学选的软件的例子都不一样，大家如果对等价类测试有问题可以联系助教和老师。

上节课我们对白盒测试开了一个头，黑盒测试和白盒测试有不同的出发点。从用户的角度来讲，需要从输入和输出来考虑。黑盒测试关注的更多的是外部的状态。所以我们后面做黑盒测试作业的时候，也是针对需求规则说明进行相应的黑盒测试。大家尽量想办法体验一下黑盒测试对一些深层次复杂软件的错误是发现不了的。第一次作业有这种情况的话，大家也可以加进测试报告里面。



## 有了“黑盒”测试，为什么还要“白盒”测试？

- ④ 黑盒测试是从用户的观点出发，根据程序外部特性进行的测试。如果外部特性本身有问题或规格说明的规定有误，用黑盒测试方法是发现不了的。
- ⑤ 黑盒测试只能观察软件的外部表现，即使软件的输入输出都是正确的，却并不能说明软件就是正确的。因为程序有可能用错误的运算方式得出正确的结果，这种情况只有白盒测试才能发现真正的原因。
- ⑥ 白盒测试能发现程序里的隐患，像内存泄漏、误差累计问题。在这方面，黑盒测试存在严重的不足。

白盒测试更倾向于发现程序中的一些隐患。所以我们白盒测试主要讲两部分的内容，一个是路径的测试，另一个是数据流的测试。所以我们几个测试都是进行相应的专题性的介绍。

## 路径测试

对于路径测试，我们要看程序图。我们要回顾一下程序图的基本的画法：



### 程序图

- ④ 程序图是一种有向图，图中的节点表示语句片段，边表示控制流。
- ⑤ 如果  $i$  和  $j$  是程序图中的节点，从节点  $i$  到节点  $j$  存在一条边，当且仅当对应节点  $j$  的语句片段可以在对应节点  $i$  的语句片段之后立即执行。

所以我们的程序图都可以通过有向图的方式把一段程序代码变成相应的图形结构，这就是建模过程，是为了之后的分析。



## 三角形程序的程序流图

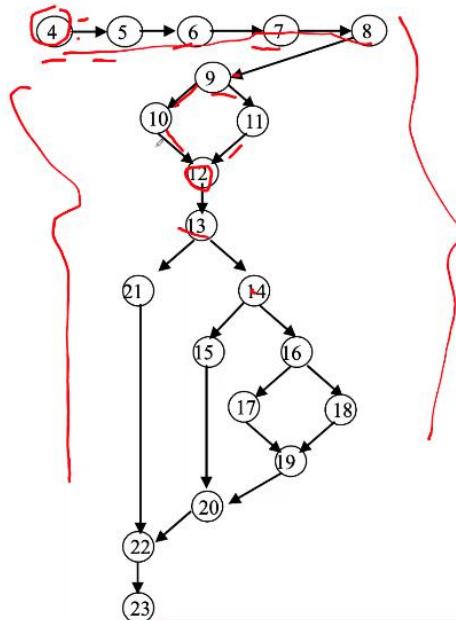
```
1. Program triangle2
2. Dim a,b,c as Integer
3. Dim IsATriangle As Boolean
4. OutPut("Enter 3 integer which are sides of a triangle")
5. Input(a, b, c)
6. OutPut("Side A is ", a)
7. OutPut("Side B is ", b)
8. OutPut("Side C is ", c)
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then IsATriangle = True
11. Else IsATriangle = False
12. EndIf
13. If IsATriangle
14. Then If (a = b) AND (b = c)
15. Then Output("等边三角形" )
16. Else If (a<>b) AND (a <> c) AND (b <> c)
17. Then Output("一般三角形" )
18. Else Output("等腰三角形" )
19. EndIf
20. EndIf
21. Else Output("非三角形" )
22. EndIf
23. End triangle2
```

三角形程序流图都是可以通过一个图的方式把三角形的伪代码变成一个程序流图。



## 三角形程序的程序流图

程序伪代码 (略)  
**(5分钟程序阅读)**  
程序流图：  
**4~8是顺序执行；**  
**9~12是分支语句；**  
**13~22是分支的嵌套语句；**  
**没有循环！**  
**所以是有向无循环图.**



这里面的数字都是程序的行号。这样我们就可以把代码放到可视化的模型里去。通过这种方式就可以清晰地看到程序的分支和循环等结构。有了这个以后，我们就可以发现一些规律，4~8 行是顺序执行，里面没有控制流的流入和流出。9~12 行就是有两个分支语句。到 13 以后就是嵌套的语句。

# DD 路径

DD-路径就是决策到决策的路径。

## 7.1.1 DD-路径图（决策到决策）

- 程序流图是一种有向图，其中的节点表示语句，边表示控制流。
- 对于给定的程序，可以使用多种不同的程序流图，所有这些程序流图，都可以简化为唯一的DD-路径图。
- 还是以三角形程序为例进行说明。

我们是按照程序流图的方式去建模我们的程序，给定的程序里面，如果没有规范性的约束要求，可以有很多种不同的程序图。比如上图中的 4 和 5 可以合并成一个节点。所以对程序图如果没有规范化的约束，不同的人画出来可能是不同的形状。



## DD-路径定义

**DD-路径是程序图中的一条链，使得：**

- 情况1：由一个节点组成，入度=0。
- 情况2：由一个节点组成，出度=0。
- 情况3：由一个节点组成，入度 $\geq 2$ 或出度 $\geq 2$ 。
- 情况4：由一个节点组成，入度=1并且出度=1。
- 情况5：长度 $\geq 1$ 的最大链(单入单出的最大序列)。

五种情况都是一种 DD 路径。

情况 1：路径就是一个节点，入度是 0，也就是没有控制流的流入，它是开始节点。

情况 2：路径就是一个节点，出度是 0，控制流没有流出，所以它是终止节点。

情况 3：对应的就是分支（出度 $\geq 2$ ）或者汇合（）的情况。

情况 4：普通节点，没有额外控制流。

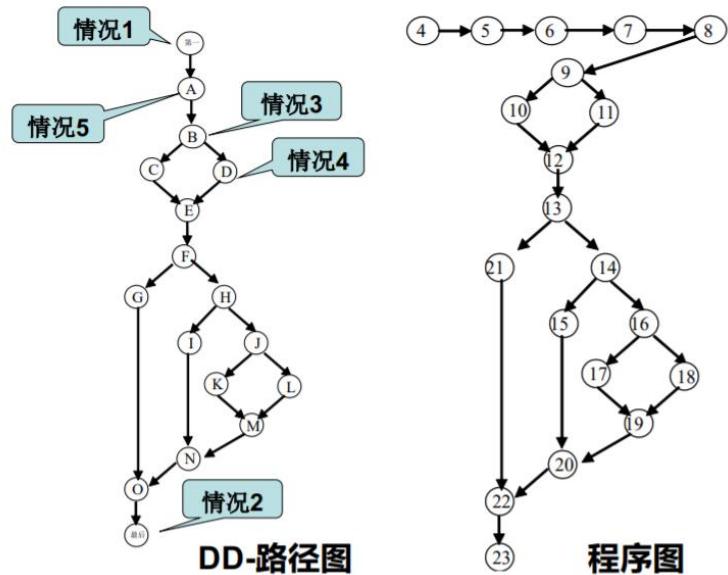
情况 5：前四种情况都是一个节点组成，而第 5 种情况是最大链。



上图中间圈起来的是就是单入单出的最大序列。

**换言之：DD-路径是程序图中的最小独立路径，不能被包括在其它DD-路径之中。**

对于结构来讲，DD 路径是程序图中的最小单位。



我们回到刚才的例子，B 对应的是 9。23 是情况 2。10,11 都是单入单出节点，所以是情况 4。

**DD路径定义:**对给定用命令语言编写的一段程序，其DD路径是有向图，其中节点表示程序图的DD路径，边表示连续DD路径之间的控制流。

换言之，DD路径是一种压缩图，其中的一个节点（DD路径），对应程序中的语句片段。

对100行以内的程序生成DD路径是可行的，如果超过这个规模，一般需要有分析工具的支持。

它的作用就是把程序流图进行压缩和简化，变成一个唯一的图。如果代码不是很长，往往可以手画出 DD 路径。通过唯一的这么一个定义的方式，都可以形成最终归一化的形状，这样我们最后做基于路径的测试的话，模型是有唯一的结构形状的。



## 7.1.2 测试覆盖指标

- ① 测试的主要评测方法包括覆盖和质量。
- ② 测试覆盖是对测试完全程度的评测。测试覆盖是由测试需求和测试用例的覆盖或已执行代码的覆盖表示的。
- ③ 覆盖指标提供了“测试的完全程度如何？”这一问题的答案。最常用的覆盖评测是基于需求的测试覆盖和基于代码的测试覆盖。

测试里有很多覆盖指标，是对我们测试完整度的评测。所以覆盖的指标提供了测试完整度如何的答案。

这些指标都是通用化的叫法。

指标	覆盖描述
$C_0$	所有语句（语句覆盖）
$C_1$	所有DD-路径（分支覆盖）
$C_{1p}$	所有判断的每种分支（条件判断）
$C_2$	$C_1$ 覆盖+循环覆盖
$C_d$	$C_1$ 覆盖+DD-路径的所有依赖对偶
$C_{mcc}$	多条件覆盖
$C_{ik}$	包含最多K次循环的所有程序路径（通常K=2）
$C_{start}$	路径具有“统计重要性”的部分
$C_\infty$	所有可能的执行路径

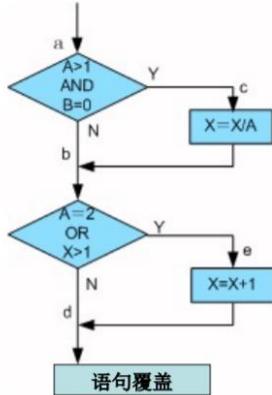
语句覆盖就是所有语句至少执行过一遍，但是在实际的工程里面，可能语句覆盖都没有达到，有很多冗余语句基本都没有做过执行。比如所有 DD 路径的覆盖就是分支的覆盖。如果加上循环就是  $C_2$ 。比如覆盖指标本身就把循环的次数包含进去。

我们的目标是在测试指标的指导下完成最少的测试用例的设计。



## 语句覆盖( $C_0$ )

- ④ 使程序中每一可执行语句至少执行一次；



为使程序中每个语句至少执行一次，只需设计一个能通过路径**ace**的例子就可以了，例如选择输入数据为：  
**A=2, B=0, X=3**  
就可达到“语句覆盖”标准。

这个情况下我们只关心节点都覆盖到了，而不是边的覆盖。如果我们要进行测试用例的设计，那就是回答最少通过几个测试用例来达到语句的覆盖。有些冗余代码通过任何的测试用例可能都执行不了，这样就可以反向地说明程序设计的一些问题。

通过至少执行一边的测试用例也可以说明里面的一些问题，看看语句测试代码是否能够正常运行。

**2022/3/21**

回顾一下上节课。



## DD-路径定义

**DD-路径是程序图中的一条链，使得：**

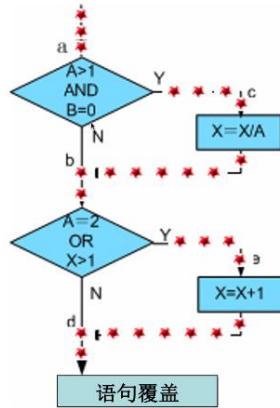
- **情况1：由一个节点组成，入度=0。**
- **情况2：由一个节点组成，出度=0。**
- **情况3：由一个节点组成，入度 $\geq 2$ 或出度 $\geq 2$ 。**
- **情况4：由一个节点组成，入度=1并且出度=1。**
- **情况5：长度 $\geq 1$ 的最大链(单入单出的最大序列)。**

软件测试书里经常把覆盖指标变成一个个符号。



## 语句覆盖( $C_0$ )

④ 使程序中每一可执行语句至少执行一次；



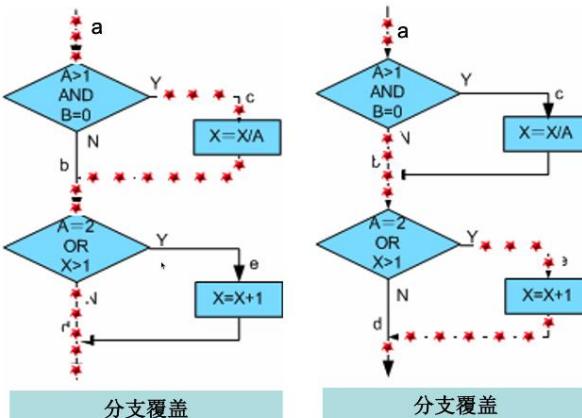
为使程序中每个语句至少执行一次，只需设计一个能通过路径**ace**的例子就可以了，例如选择输入数据为：  
**A=2, B=0, X=3**  
就可达到“语句覆盖”标准。

通过最小的测试用例把语句片段执行一遍，语句片段对应我们程序流图的节点，但是不能覆盖到边。我们通过一个测试用例就能实现相关的覆盖。

DD 路径的测试有两种，分为分支覆盖（每个逻辑判断的真假至少取一次）和条件覆盖（所有判断的每个分支至少取一次）。



## 分支覆盖( $C_1$ )



分支覆盖我们可以通过两个测试用例覆盖两个分支。指的是整个判断的真假各取一次。

## 条件覆盖

而条件覆盖是 if 判断里的每个条件都要覆盖到，即  $A > 1$ 、 $A \leq 1$ 、 $B = 0$ 、 $B \neq 0$ ，但是我们不需要考虑 if 判断中的条件组合。

## 条件覆盖 (C<sub>1</sub>p)

例1的程序有四个条件:

A>1、B=0、A=2、X>1

为了达到“条件覆盖”标准，需要执行足够的测试用例  
使得在a点有：

A>1、A≤1、B=0、B≠0

等各种结果出现，以及在b点有：

A=2、A≠2、X>1、X≤1

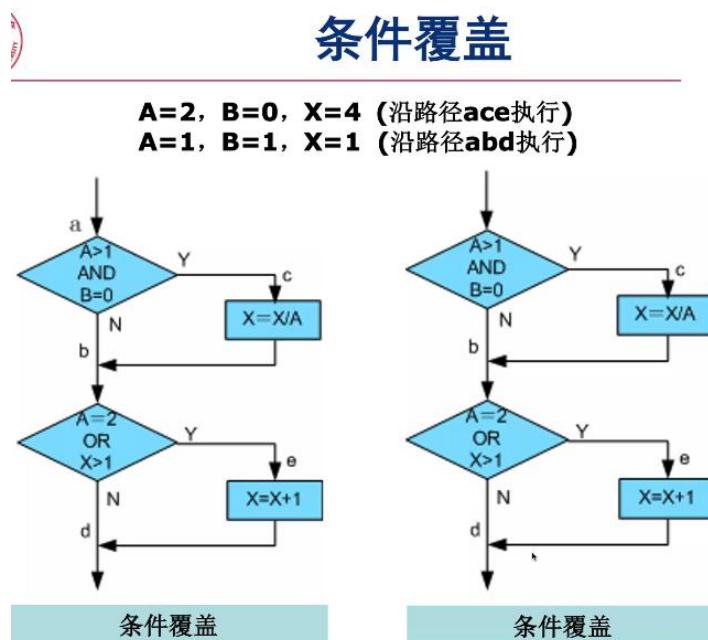
等各种结果出现。

④ 现在只需设计以下两个测试用例就可满足这一标准：

① A=2, B=0, X=4 (沿路径ace执行);

② A=1, B=1, X=1 (沿路径abd执行)。

我们需要设计测试用例去覆盖条件真假的判断。我们只需要2个就可以实现所有条件的覆盖。



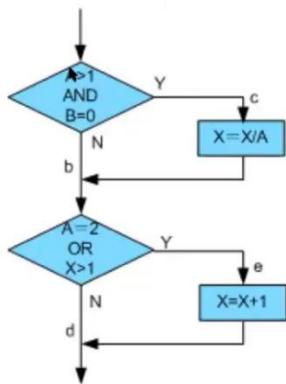
## 多条件覆盖C<sub>mcc</sub>

- ④ 使得每个判断表达式中条件的各种可能组合都至少  
    出现一次；(条件组合覆盖) ↴

我们每个判断的额表达式里有条件的组合，这就面临条件组合的覆盖。



## 多条件覆盖( $C_{mcc}$ )



再看例1的程序，我们需要选择适当的例子，使得下面 8种条件组合都能够出现：

- 1) A>1, B=0      2) A>1, B≠0
- 3) A≤1, B=0      4) A≤1, B≠0
- 5) A=2, X>1      6) A=2, X≤1
- 7) A≠2, X>1      8) A≠2, X≤1

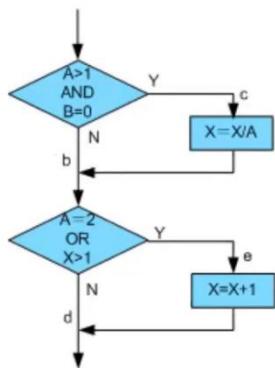
5)、6)、7)、8) 四种情况是第二个 IF语句的条件组合，而X的值在该语句之前是要经过计算的，所以还必须根据程序的逻辑推算出在程序的入口点X的输入值应是什么。

上例中的 4 个条件，就要进行两两组合。但是我们不考虑两个 if 之间的复合，我们只去看每个分支判断的两两复合。每个分支里面两两组合有 4 个情况，可以把不同的判断形式去覆盖，就出现了 8 种组合。



## 多条件覆盖( $C_{mcc}$ )

下面设计的四个例子可以使上述 8种条件组合至少出现一次：



- ① A=2, B=0, X=4  
使 1)、5)两种情况出现；
- ② A=2, B=1, X=1  
使 2)、6)两种情况出现；
- ③ A=1, B=0, X=2  
使 3)、7)两种情况出现；
- ④ A=1, B=1, X=1  
使 4)、8)两种情况出现。

这 8 个条件组合，肯定是可以设计 8 种测试用例来覆盖的，不过我们希望使用的测试用例越少越好。我们发现可以设计 4 个测试用例来覆盖 8 个条件的组合。

我们可以看到上面这个例子虽然满足多条件组合的覆盖，但是它不能覆盖程序中的每一条路径，例如路径 acd 就没有执行，因此，条件组合覆盖标准仍然不彻底。它更能去考察谓词判断的表达式的所有的情况，但是对于路径来讲，它并没有去考虑。

## 分支/条件覆盖

所以这种情况下，就有另外一种分支/条件覆盖。



## 分支/条件覆盖

- ④ 针对上面的问题引出了另一种覆盖标准—“分支 / 条件覆盖”，它的含义是：执行足够的测试用例，使得分支中每个条件取到各种可能的值，并使每个分支取到各种可能的结果。

— 对例1的程序，前面的两个例子

①  $A=2, B=0, X=4$  (沿ace路径)

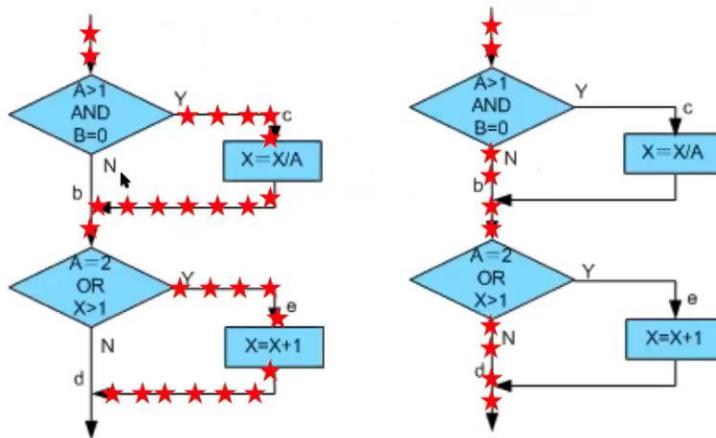
②  $A=1, B=1, X=1$  (沿abd路径)

是满足这一标准的。

它把分支和条件覆盖综合在了一块。

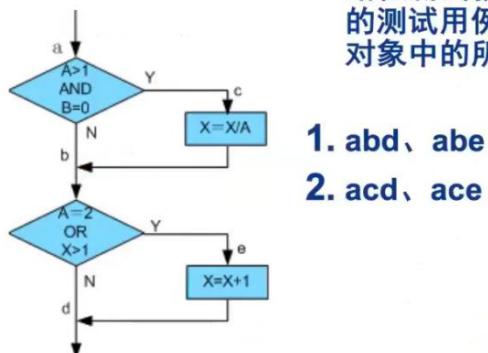
①  $A=2, B=0, X=4$  (沿ace路径)

②  $A=1, B=1, X=1$  (沿abd路径)



## 路径测试( $C_\infty$ )

- ⑤ 路径测试就是设计足够多的测试用例，覆盖被测试对象中的所有可能路径。



1. abd、abe

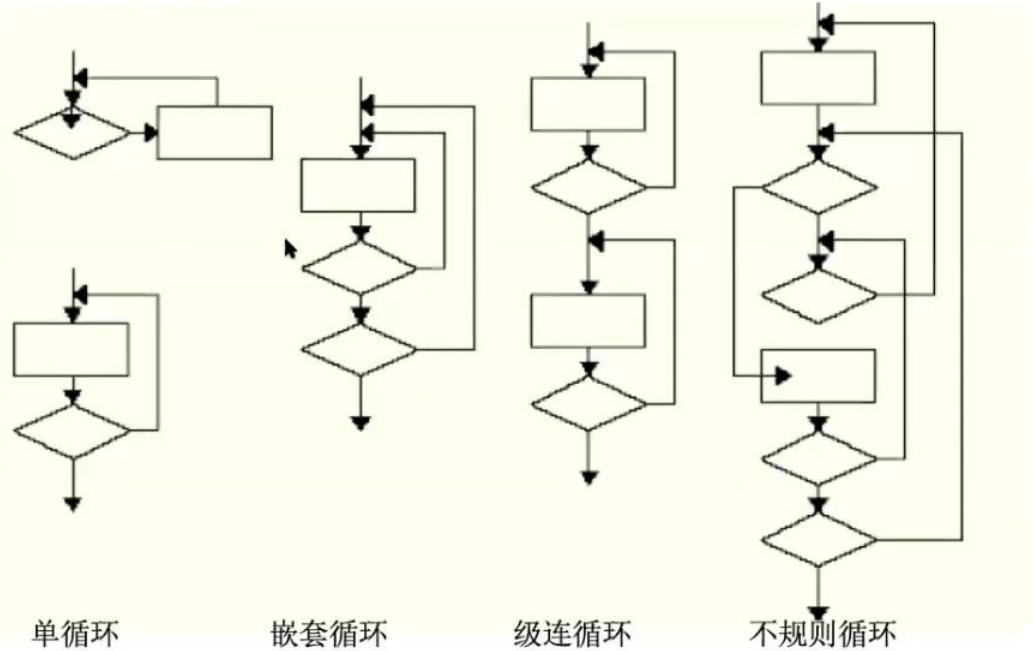
2. acd、ace

上节课主要讲了 DD 路径，实际上我们在程序里经常做路径测试，它的形式就是  $C_\infty$ ，

表示我们测试用例设计过程里可以测试所有可能的程序存在的路径，我们可以把里面所有的路径都测一遍就可能发现里面就可能多的情况。

我们还有 DD 路径的依赖对偶，我们放在数据流测试（第十章）去讲。

## ④ 循环测试



另外，我们很多程序里都会存在循环，因此存在循环的情况下，我们应该怎么来做呢？循环的形式有很多：单循环、嵌套循环、级联循环、不规则循环等。大致的基本的原则就是根据循环次数去测试它的循环。

### ■ 单循环测试

假设循环次数为  $N$

- a. 直接跳过循环
- b. 循环次数为 1
- c. 循环次数为 2
- d. 循环次数为  $M$ ,  $M < N$
- e. 循环次数为  $N-1$ ,  $N$ ,  $N+1$

包括循环的边界的情况，循环次数为  $N-1$ ,  $N$ ,  $N+1$  的情况。对于循环，我们要选择不同次数来测，另外还有嵌套循环。嵌套循环先测试最内部的循环，其他外层循环的次数置成 1，这就变成了一个单循环测试。

### ■ 嵌套循环测试

- a. 先测试最内部循环，其它循环次数为 1
- b. 测试第二层循环，其它循环次数为 1，  
c. 直到最外部循环完成测试

### ■ 级连循环测试

- a. 分别采用单循环测试方法进行测试

### ■ 不规则循环测试 无法测试——重新设计！

级联循环一般采用单循环方式测试。一般来讲不需要进行级联循环之间的组合。不规则循环是不满足代码的指标的，可以直接就不测。

我们现在把基于路径的几个测试方法和流程从原理上分析出来，我们还是要从讲的工具里面去做测试用例的生成和测试，包括不同的方法有一个覆盖率的要求。我们经常希望它是100%，但是有时候在实际系统里面，有些我们的方法达不到100%。所以不同方法测试的效率和完备性是有差别的。大家在做测试方法的时候可以评估一下，我们采用的方法在我们的例子的情况下，它的表现和覆盖率是多少。覆盖率是在特定情况下算出来的一个数字。原来有同学说，是不是每一个方法都有一个大致的覆盖率呢？覆盖率一定是在例子本身里计算出来的百分比，不同例子来讲，不同覆盖率的表现差别还是比较大的。

而且，即使我们的覆盖率是100%，也不能把所有的程序的错误和缺陷揭露出来，不同方法找到的错误是不一样的，这种情况下就希望大家采用不同方法去测试，这样我们整个测试的完备性和充分性会做的比较好。

## 小结

- 无论哪种测试覆盖，即使其覆盖率达到百分之百，都不能保证把所有隐藏的程序欠缺都揭露出来。
- 提高结构的测试覆盖率只能增强我们对被测软件的信心，但它绝不是万无一失的。

第二次作业，我们希望大家还是选择一个例子，200行代码以上。大家在测试的时候，尽量人工去注入一些错误，如果错误不注入，对于不同方法的评估好像都差不多，我们建议大家一定在源代码中注入不同类型的错误，然后再用相应方法做测试，把错误给找到。这样我们就可以去评估方法的表现应该怎么样。

之前的覆盖要求都是针对路径本身去做覆盖的，是最基本的路径覆盖的要求。因为我们第二次作业会布置一些基于路径覆盖的白盒测试覆盖方法的体验。

接下来我们介绍一下关于基路径的测试方法。



### 7.1.3 McCabe圈数

#### 基路径的概念与方法

- ④ 基：向量空间的概念，向量空间的基是相互独立的一组向量，基“覆盖”整个向量空间。使得该空间中的任何其它向量都可以用基向量表示。

其实“圈数”本身是在基路径测试里比较重要的概念。二维向量空间里有2个基向量。通过2个基向量，我们可以表示整个空间里的任意向量。

- ④ 基路径：程序图中相互独立的一组路径，使得该程序中的所有路径都可以用基路径表示。

基路径实际上是把基向量的一些特性采用到了我们路径里来。我们所有路径都可以通过

基路径的线性表示。基向量是正交的，它是相对独立的，而在基路径里面也采用了类似的“相互独立的路径”。

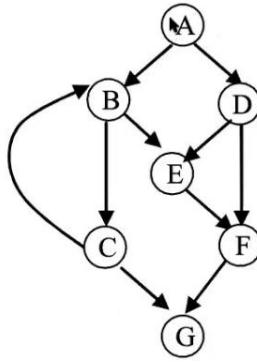
④ 圈复杂度：是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于计算程序的基本的基本路径数目。

一段程序代码可以算出圈复杂度，它是一个软件复杂度的定量表示。复杂度其实就是我们基本路径的数目，比如我们图里有 1000 个路径，有 10 个基本路径，路径都可以被基本路径使用线性组合的方式来表示。

⑤ 基本路径必须从起始点到终止点的路径。

—包含一条其他基本路径不曾用到的边。或至少引入一个新处理语句或者新判断的程序通路。

基本路径必须是完整的路径。



这是我们变成图的形式的程序，首先我们有一个圈复杂度的计算公式。

**McCabe圈数V ( G ) 计算方法：**

$$V(G) = e - n + 2p$$

e : 边数；

n : 节点数

p : 连接区域数

$$\text{对于右图 : } V(G) = 10 - 7 + 2 \cdot 1 = 5$$

10 条边，7 个节点，区域数是 1。所以这段程序变成一个图以后，圈复杂度是 5。对于 5 这个数字来讲，表示程序里基本路径的数量就是 5。

**参考课本说明强连接有向图的计算公式(不满足单入口、单出口条件)。**

有些书上公式也会写成  $e - n + p$ ，但是这两种公式运用的情况是不一样的。如果  $e - n + p$  就会约束图是强连通图。

圈数计算出来以后有什么用呢？

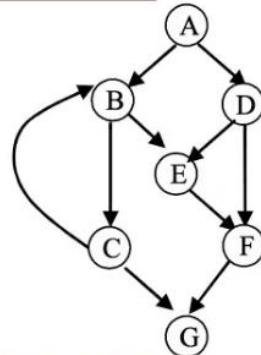
程序的圈数越大，程序的复杂度越高。如果圈数越小，那么我们程序的逻辑越简单，我们可以很容易地度量。接下来，我们需要用到基本路径的个数。圈数 5 就代表有 5 条基本路径，这 5 条基本路径的找法就是搜索的方法。我们找的时候，一定是找从起点到终点的路径。

比如我们先找到了 P1，第二次找的时候，我们需要在新路径中至少包含一条 P1 不包含

的边，如 C-B 边；P3,P4,P5 同理。

### ⑤5条基本路径是：

- P1: A, B, C, G;
- P2: A, B, C, B, C, G;
- P3: A, B, E, F, G;
- P4: A, D, E, F, G;
- P5: A, D, F, G.

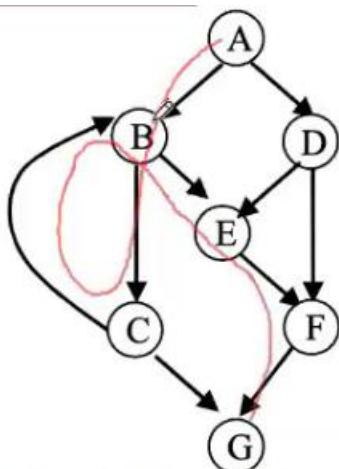


任何路径都可以看作是上述5条路径的组合形成的。

我们实际上可以通过搜索的方法找到 5 条基本路径。我们有个原则：有了一条基本路径以后，要增加新的基本路径，必须增加至少一个控制流或者新路径。这样通过搜索的方式一定能把基本路径的全集给找到。

我们需要验证“任何路径都可以看作是上述 5 条路径组合形成的”。

比如路径 A,B,C,B,E,F,G



这个路径和我们每个基本路径都不一样，但是它可以通过 P2 + P3 - P1 来组合而成。

例：A, B, C, B, E, F, G 是 P2 + P3 - P1

其中，加法是一条路径后接一条路径；乘法是路径的重复，而减法则只有数学上去除边的含义，而缺少实际意义。

$$\begin{aligned} &= \cancel{A, B, C, B, C, G} + \cancel{A, B, E, F, G} - \cancel{A, B, C, G} \\ &AB + BC + CB + CG + AB + BE + EF + FG - AB - BC - CG \\ &= A, B, C, B, E, F, G \end{aligned}$$

这个过程就类似于基向量的计算方法，从原理和思想上是一致的。

找基本路径的算法就是通过搜索的方法。

## 如何寻找McCabe路径?

图的搜索与遍历算法!

- 深度优先算法
- 广度优先算法



## 基本复杂度

- McCabe在圈复杂度上的工作,在一定意义上,对程序设计的改进要大于对测试的改进。
- 对于结构化程序设计的程序,可以将if-then-else构造压缩为一个节点。
- 则三角形程序最终可以压缩为只有一条路径的图!
- 而对于非结构化程序,必然增加McCabe圈的数量。

一般的,一个单元模块的最大复杂度V(G) < 10。

这是我们程序设计过程中的一个经验值,超过这个复杂度的话,程序可能就太复杂了。  
对于路径的测试,我们之前讲到了一些不同的覆盖指标。



### 7.1.4 基于路径的测试讨论

基于路径的测试,从另一个角度去考虑程序测试的完备性,但是没有考虑数据功能,即使满足DD-路径覆盖,和考虑圈复杂度指标,也仅是测试的下限指标。

如左图: S: 已说明行为的路径集合;

P: 已编程实现的路径集合;

T: 拓扑可达的路径集合;

区域: 1: 测试的重点, 可测;

2、6: 已实现的,

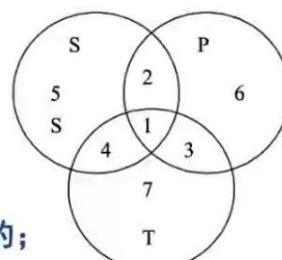
3: 已实现和可达的,

4: 已描述的, 拓扑可达, 却没有实现的;

要增加, 可测;

5: 没有实现的已说明路径, 不可测,

7: 未说明, 未实现, 拓扑可达, 测试无意义!



可行的和在拓扑  
结构上可能的路  
径

有时候大家也可以画一些图的形式,S是我们需求里面已经说明的路径集合,而P是编程实现的集合,T是拓扑可达的路径集合,通常情况下这三个集合不会重合。

Q: 拓扑可达路径指的是什么?

A: 拓扑可达和实际可达是不一样的,拓扑可达指的是看似这条路径是有的,但是实际情况是达不到的。比如两个条件是互斥的,有一些路径就是达不到。

## 数据流测试

这些总结只是说大家在实际的工程测试里要有一些测试的原则。后面我们对基于数据流的测试开一个头。数据流测试和我们路径测试不太一样，对白盒测试来讲一个要测试程序的结构（逻辑是否正确），接下来我们还要关注数据是否正常使用。这对我们程序的正确性也非常重要。其实数据流测试和我们之前讲的结构化测试是一个并行的概念，但是两个关注点是不一样的，因为数据流测试是从数据的角度来关注，而结构化测试是从程序逻辑的角度来进行测试的。

对数据流来讲，它也是一种逻辑的覆盖。



## 7.2 数据流测试

- **数据流测试是从关注程序中数据及其使用的角度，来设计测试用例的。类似一种路径测试覆盖，但关心的是数据变量而不是程序结构。**

### 定义-使用 (def-use) 测试

- 对于程序 $P$ ，有程序图 $G(P)$ ， $V$ 是程序中变量的集合，节点表示语句片，边表示节点序列。
- 节点 $n \in G(P)$  是变量 $v \in V$ 的**定义节点**，记做 $\text{DEF}(v, n)$ ，当且仅当变量 $v$ 的值有对应节点 $n$ 的语句片处**定义**。

**输入语句、赋值语句、循环控制语句和过程调用语句，即：  
向内存地址写入值的语句。**

定义-使用是什么概念呢？我们有变量的定义节点。定义有很多的形式，就是向内存地址写入一个值，能够写入值的都是定义节点体现的范围。另外还有变量的使用：

- 节点 $n \in G(p)$  是变量 $v \in V$ 的**使用节点**，记做 $\text{USE}(v, n)$ ，当且仅当变量 $v$ 的值在对应节点 $n$ 的语句片处**使用**。

**输出语句、赋值语句、条件语句、循环控制语句和  
过程调用语句，都是可能的使用语句。但它们不改  
变变量的值。即：从内存地址读取值的语句。**

很多情况都会在不改变变量值的情况下读取变量，这些都会对内存进行读取某些值。

## 定义-使用路径 (du-path)

- 使用节点 **USE** ( $v, n$ ) 是一个谓词使用 (记做 P-use)，当且仅当语句  $n$  是谓词语句；否则， **USE** ( $v, n$ ) 是计算使用 (记做 C-use)。
- 关于变量  $v$  的 **定义-使用** 路径 (记做 du-path) 是 **PATHS** ( $P$ ) 中的路径，使得对某个  $v \in V$ ，存在 **定义** 节点 **DEF** ( $v, m$ ) 和 **使用** 节点 **USE** ( $v, n$ )，使得  $m$  和  $n$  是该路径的 **起始** 和 **终止** 节点。

数据流测试里我们定义了新的路径，我们通过对变量数据的定义，定义出了新的 du 路径。

## 定义-清除路径 (dc-path)

- 定义：关于变量  $v$  的 **定义清除** 路径 (记做 dc-path)，是具有 **起始** 和 **终止** 节点 **DEF** ( $v, m$ ) 和 **USE** ( $v, n$ ) 的 **PATHS** ( $P$ ) 中的路径，使得该路径中 **没有其它节点是  $v$  的定义节点**。

除了 du 路径以外，我们还有 dc 路径。它在之前的 du 路径基础上，再进行一个约束。所以说，定义清除路径也很简单，在我们定义使用路径中间没有对这个变量重新进行定义，这就是我们的定义清除路径。

我们还是通过例子：

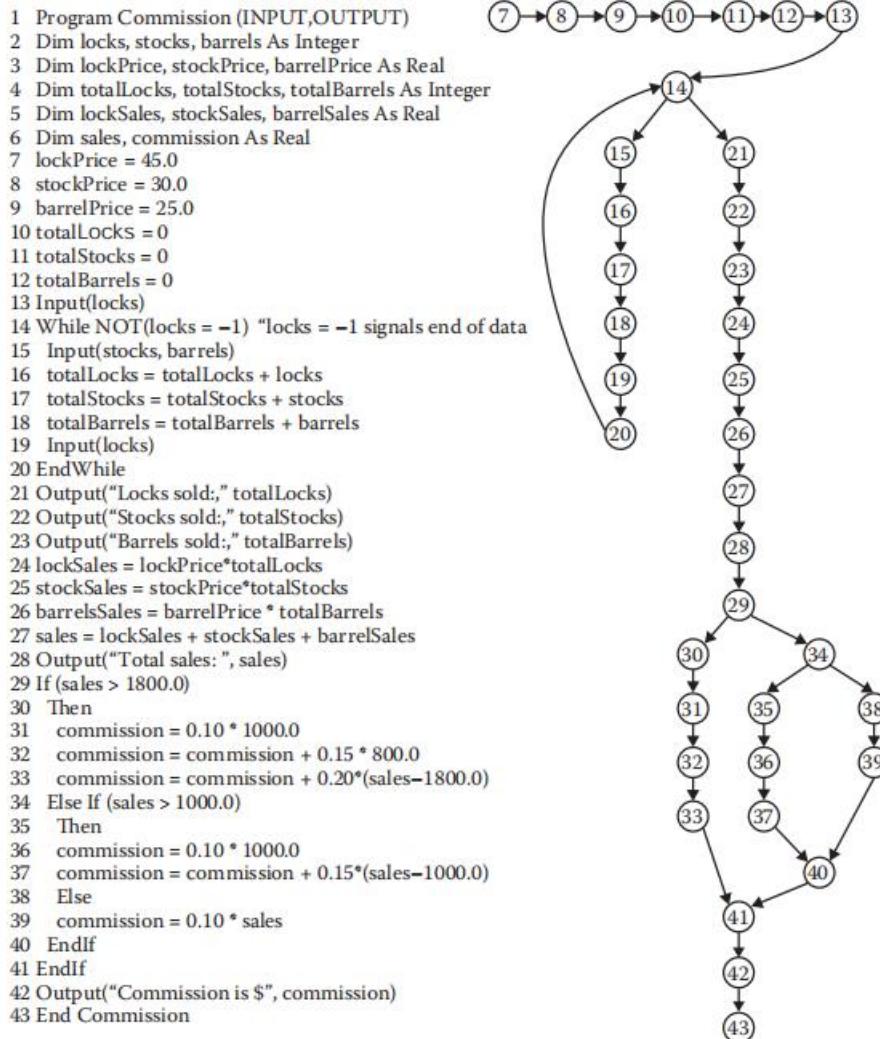


Figure 9.1 Commission problem and its program graph.

图 1 英语书 P162

基于数据流测试会讲这个佣金问题(Commission Problem)，在书上的 P109 页。它里面有很多对变量的计算。其实数据流测试的话，它的测试方法特别适合对数据计算的一些场景。

接下来，我们就需要去填这一张表：

变量名	定义节点	使用节点
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10,16	16,21,24
totalStock	11,17	17,22,25
totalBarrels	12,18	18,23,26
locks	13,19	14,16
stocks	15	17
barrels	15	18
locksales	24	27
stocksales	25	27
barrelsales	26	27
sales	27	28,29,33,34,37,38
commission	31,32,33,36,37,39	32,33,37,42

我们可以把所有的定义节点和使用节点对应我们的变量找出来。这样我们就可以继续分析我们的定义-使用路径 (DU 路径)。

例子：

对于变量 stocks:

1. 变量stocks: 有DEF(stocks,15)和USE(stocks,17),  
路径<15,17>是一个stocks的定义-使用路径,是定义清除  
路径.

因为中间没有对 stocks 重新定义, 所以它也是一个定义清除路径。

对于变量 locks:

2. 变量locks: 有DEF(locks,13)、DEF (locks, 19)  
USE (locks, 14) 、 USE (locks, 16) 。

产生的路径: p1=<13,14>

p2=<13,14,15,16>  
p3=<19,20,14>  
p4=<19,20,14,15,16>

对于变量 totalLocks:

3. 变量totalLocks: 问题比较复杂,有2个定义节  
点,DEF(totalLocks,10)和DEF(totalLocks,16);3个使用节  
点,USE(totalLocks,16)、USE(totalLocks, 21)和USE  
(totalLocks, 24) 。

有路径:

p1=<10,11,12,13,14,15,16> 是定义清除的。  
p2=<10,11,12,13,14,15,16,17,18,19,20,14,21>因为节  
点16是可循环的,存在totalLocks再定义现象,不是定义清除  
的。

因为中间节点 16, 重新把变量定义了, 所以 p2 属于 du-path, 但是不属于 dc-path。

**p3=<10,11,12,13,14,15,16,17,18,19,20,14,21,22,23,24>**

**=<p2,22,23,24>**,当然不是定义清除的.

**p4=<16,16>** 不作为定义-使用路径

自己定义自己的情况，不能作为 du-path 概念。

**p5=<16,17,18,19,20,14,21>**

**p6=<16,17,18,19,20,14,21,22,23,24>**

对于一个变量，我们只需要把定义和使用的节点号码找出来以后，这样就直接可以得到 du 路径了。

所以，我们的方法就是找到变量，构建一个表格。然后去找定义使用路径，然后再把同一个节点同时定义和清除的情况去除掉即可。

## 2022/3/28

这门课的 60% 内容在课本上，而 40% 内容课本里没有。上节课我们讲了定义使用路径和定义清除路径怎么找。接下来我们讲，我们找到了很多定义使用路径（du 路径）以后，我们应该怎么做测试呢？做测试的时候，我们也有对应的覆盖指标。

## 全定义准则

**全定义准则：**每个**定义节点**到一个**使用节点**的定义清除路径。就是说要覆盖每一个定义节点到一个使用的定义清除路径。

**Q：**针对这个测试覆盖指标做测试的时候应该怎么做呢？

**A：**原来我们做结构化测试的时候，比如要覆盖分支，那么就是要找到覆盖分支的几个用例。做数据流测试的时候，按照全定义准则去做数据流测试。第一步就是先要找到这些定义清除路径，每一个定义清除路径，我们做测试的时候就应该针对它去做测试用例。

例子如下：

变量名	定义节点	使用节点
totalLocks	10, 16	16, 21, 24

针对 totalLocks 这个变量，我们要覆盖全定义准则的话，只需要设计 2 个测试用例，因为是**每个**定义节点到**一个**使用节点的定义清除路径。

大家设计测试用例的时候，有针对不同的覆盖指标设计测试用例的思路。

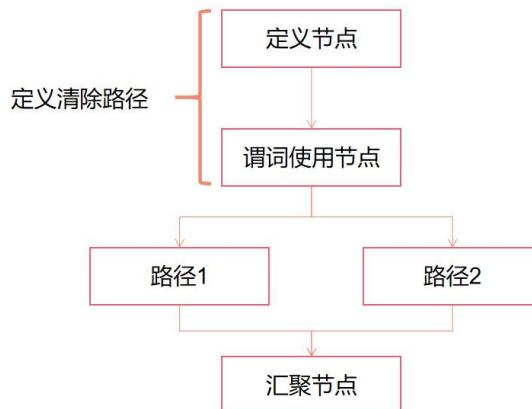
## 全使用准则

**全使用准则：****每个定义节点**到**所有使用节点**以及**后续节点**的定义清除路径。首先，我们理解一下每个定义节点到所有使用节点的定义清除路径，比起之前的全定义准则范围更大。

**Q：**为什么在定义中，还有“**后续节点**”？

**A：**后续节点实际上对整个程序是有影响的，所有的使用节点后面的节点也要考虑进去。因为使用节点对后续节点有不同的东西。比如有一个特殊的使用叫做“**谓词使用**”，如下图所

示：



如果我们只考虑定义节点到谓词使用节点这一段的定义清除路径的话，我们并不知道是要走路径 1 还是路径 2，这在设计测试用例的时候是不合适的。所以，我们要考虑后续节点。

Q：为什么全定义准则里并没有考虑使用节点的后续节点？

A：书上是这样定义的。但是对于使用节点来说，我们还是尽量要考虑谓词使用的特殊性。

- 全谓词使用/部分计算使用准则：每个定义节点到所有谓词使用的定义清除路径，若无谓词使用，至少有一个计算使用的定义清除路径。
- 全计算使用/部分谓词使用准则：每个定义节点到所有计算使用的定义清除路径，若无计算使用，至少有一个谓词使用的定义清除路径。
- 全定义-使用路径准则：每个定义节点到所有使用节点以及后续节点的定义清除路径。包括有一次环路和或无环路的路径。

数据流覆盖的层次结构关系图9-5 P116。

如果含有所有的谓词使用的情况的话，我们认为还是需要包含后续节点的影响。从严格意义来讲，在使用谓词的情况下，尽量要考虑后续节点的影响。这些准则最终都是要使用测试用例去覆盖的，我们的每一条路径都对应了每一个测试用例的设计。这些覆盖指标都能对应到路径有多少条，也就意味着测试的时候要完成多少测试用例的设计。我们要理解设计的初衷，这样就可以更好地理解测试的过程。

## 基于程序片的测试

对于数据流测试还有一块很重要的就是基于程序片的测试。首先，我们要知道程序片的定义。

- 定义：给定一个程序P和P中的一个变量集合V，变量集合V在语句n上的一个片，记做S(V, n)，是P中对V中的变量值作出贡献的所有语句（编号）的集合。

所以一个程序片是语句的集合，它是对变量 V 做出贡献的语句（能够影响到变量值的

语句)。

- **USE (使用) 的形式有:**

谓词使用、计算使用、输出使用、定位使用、  
迭代使用

- **DEF (定义) 的形式有:**

输入定义、赋值定义

我们看一下程序片怎么去找。

```
13 Input(locks)
14 While NOT(locks = -1) "locks = -1 signals end of data
15 Input(stocks, barrels)
16 totalLocks = totalLocks + locks
17 totalStocks = totalStocks + stocks
18 totalBarrels = totalBarrels + barrels
19 Input(locks)
20 EndWhile
```

首先，我们考察变量 locks，

变量名	定义节点	使用节点
locks	13, 19	14, 16

对于  $S(\text{locks}, 13)$  来说，影响变量 locks 的只有 13。故  $S(\text{locks}, 13)=\{13\}$ 。

对于  $S(\text{locks}, 14)$  来说，影响变量 locks 的有 13 的 input，并且循环控制流 14 和 20 会影响 19 的 input 的次数（决定了是否能执行到 19 这一行），故  $S(\text{locks}, 14)=\{13, 14, 19, 20\}$ 。

对于  $S(\text{locks}, 19)$  来说，有  $S(\text{locks}, 19)=\{13, 14, 19, 20\}$ 。

做出贡献最简单的就是赋值，但是能不能进到循环里面这是通过间接的方式对 Locks 进行影响。

Q: 对于  $S(\text{locks}, 16)$  来说，有  $S(\text{locks}, 16)=\{13, 14, 19, 20\}$ 。为什么 16 不属于  $S(\text{locks}, 16)$  呢？

A: 16 只是对 locks 进行使用，没有对 locks 的值进行改变。

我们考察变量 stocks，

变量名	定义节点	使用节点
stocks	15	17

$S(\text{locks}, 15)=\{13, 14, 15, 19, 20\}$ 。

$S(\text{locks}, 17)=\{13, 14, 15, 19, 20\}$ 。

我们考察变量 barrels，

变量名	定义节点	使用节点
barrels	15	18

$S(\text{barrels}, 15)=\{13, 14, 15, 19, 20\}$ 。

```

10 totalLocks = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(lock)
14 While NOT(lock = -1) "locks = -1 signals end of data
15 Input(stocks, barrels)
16 totalLocks = totalLocks + locks
17 totalStocks = totalStocks + stocks
18 totalBarrels = totalBarrels + barrels
19 Input(lock)
20 EndWhile
21 Output("Locks sold: " totalLocks)
22 Output("Stocks sold: " totalStocks)
23 Output("Barrels sold: " totalBarrels)
24 lockSales = lockPrice*totalLocks

```

我们考察变量 `totalLocks`,

变量名	定义节点	使用节点
<code>totalLocks</code>	10, 16	16, 21, 24

对于  $S(\text{totalLocks}, 10)$  来说, 影响变量 `totalLocks` 的只有 10。故  $S(\text{totalLocks}, 10) = \{10\}$ 。

对于  $S(\text{totalLocks}, 16)$  来说, 16 又使用了又定义了 `totalLocks`, 注意到 13、14、19、20 虽然是对 `locks` 的使用, 但是影响了控制流, 所以间接地对 `totalLocks` 产生了影响, 也要计算在内。故  $S(\text{totalLocks}, 16) = \{10, 13, 14, 16, 19, 20\}$ 。

$S(\text{totalLocks}, 24) = S(\text{totalLocks}, 21) = S(\text{totalLocks}, 16) = \{10, 13, 14, 16, 19, 20\}$ 。

- ④ 变量 `lockPrice`, `stockPrice`、`barrelPrice` 都是 1 个定义, 1 个使用, 不涉及循环, 不多说明。
- ④ `sales` 变量只有 1 个定义节点 27, 有 28、29、33、34、37、38 共 6 个使用节点, 但影响其值的变量是除 `commission` 以外的所有变量。所以,  
`sales` 变量在所有相关语句上的程序片都一样:

$S_{24-30}:$

$= \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27\}$

我们考察变量 `commission`,

变量名	定义节点	使用节点
<code>commission</code>	31, 32, 33, 36, 37, 39	32, 33, 37, 42

④ 变量commission有6个定义节点，4个使用节点。所以

$S_{31}: S(\text{commission}, 31) = \{7-20, 24-31\}$

$S_{32}: S(\text{commission}, 32) = \{7-20, 24-32\}$

$S_{33}: \text{包含所有变量的影响, } S(\text{commission}, 33) = \{7-20, 24-33\}$

$S_{34}: S(\text{commission}, 36) = \{7-20, 24-36\}$

$S_{35}: S(\text{commission}, 37) = \{7-20, 24-37\}$

$S_{36}: S(\text{commission}, 39) = \{7-20, 24-39\}$

$S_{37}: S(\text{commission}, 42) = \{7-2-, 24-39\}$

Q: 程序片找出来以后，怎么用到程序片测试里呢？

A: 我们其实就是把佣金计算的公式变成我们的程序。在计算的时候有很多变量的使用和定义，最后我们整个程序就是把一个比较长的公式变成我们的程序，我们在测试数据流的时候，其实就是在看变量之间的影响关系。变量之间的依赖关系通过程序片的定义就很容易了。

程序片的方法，将程序进行了分段划分。就如佣金问题的commission是最终结果，它的计算依赖于sales的计算；而sales的计算，又依赖于lockSales、stockSales、barrelSales的计算；而lockSales、stockSales、barrelSales的计算又分别依赖于totalLocks、totalStocks、totalBarrels的计算等等。

如果所有变量的定义和使用是正确的，当然程序就是正确的！

我们可以把复杂的数据流依赖关系通过程序切片的方法一段一段地进行验证。这样我们就可以相应比较容易地做数据流测试了。

当然里面还有一些说明：

④ 不同变量程序片的补集，为变量的分段划分提供了方法。

如： $S(\text{commission}, 42) \ominus S(\text{sales}, 28)$

$= \{7-39\} - \{7-27\}$

$= \{28-39\};$

将commission的计算分为sales计算和之后的计算，为错误的定位提供了依据。

程序切片就是比较精细地做了程序数据逻辑的验证。



## 7.3 测试的效率

~~什么时候测试可以停止？多少是足够的测试？~~

1. ~~当时间用完时——缺少标准~~
2. 当继续测试没有产生新失效时 } 基于经验的
3. 当继续测试没有发现新缺陷时 } 有效方法
4. 当无法考虑新测试用例时—原因？
5. 当回报很小时—基于分析的方法
6. 当达到所要求的覆盖时—结构化测试的指标
7. 当所有缺陷都已经清除时—难以实现

对于结构化测试，就有一个很清晰的覆盖标准。那么多覆盖指标怎么选呢？实际上我们就是在测试方案里选择几种覆盖，这样我们测试的时候就有依据了。选择了确定的覆盖指标之后，我们在测试的时候就大致可以知道用多少测试用例足够达到我们的覆盖标准。

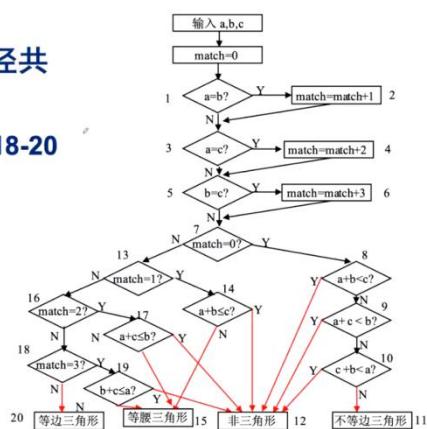


### 7.3.1 漏洞与冗余

以三角形程序为例

我们可以看到可行路径共  
有11条！

P1:1-2-3-4-5-6-7-13-16-18-20



上述这个例子通过程序流图的方式已经画出来了，那么路径是多少条呢？大家找路径的时候要分成拓扑可达路径和实际的路径。

2022/4/2

我们看一下不同的测试方法在测试例子里面的表现如何。它的可行的路径是多少条？从拓扑来讲，我们要考虑到每个分支，最终是乘起来变成很多很多拓扑路径。

Q：拓扑可达的路径和实际的路径可达有什么关系？

A：拓扑路径是相对比较全的路径集合，但是程序可能会包含一些互斥的条件，就会导致拓扑路径实际不可达的情况。

可行路径的话，我们可以从最底下去找。也就是任何一个路径，从 1 个开始节点最终会到达 4 个终止节点之一。等边三角形就是三个边相等，只有一条路径才能达到最终的等边三角形节点。我们从等腰三角形这个终止节点出发，三种情况都可以到达我们等腰三角形的最

终节点。非三角形的情况比较多了，6种情况都对应到了反三角形。

Q：如果我们使用边界值测试方法，我们需要多少个测试用例呢？

表10-1（P129）

测试用例	a	b	c	预期输出	路径
1	100	100	1	等腰三角形	P6
2	100	100	2	等腰三角形	P6
3	100	100	100	等边三角形	P1
4	100	100	199	等腰三角形	P6
5	100	100	200	非三角形	P7
...	...	...	...	...	...

A：普通的情况就是  $4N + 1$ ，用边界值测试可以产生 13 个测试用例。我们发现 13 个测试用例会最终对应到 11 个路径里去，很多是重复的，还有一些没有经过原先的 11 个路径的情况。

④ 测试覆盖的路径有：7

p1、p2、p3、p4、p5、p6、p7)

⑤ 如果采用最坏情况测试，测试用例为  $5^3=125$  个，能够覆盖全部11条路径，但是冗余很多！

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11
边界值	1	3	1	3	1	3	1	0	0	0	0
最坏情况	5	12	6	11	6	16	7	17	18	19	12

最坏情况测试的时候，居然有 19 个测试用例对应到了 p10 路径上。

## 用于方法评估的指标

评价测试用例的好坏的时候，其实是有比较定量的评价指标的。



## 7.3.2 用于方法评估的指标

假设功能性测试技术M生成m个测试用例,并且根据标识被测单元中的s个元素的结构性测试指标S来跟踪这些测试用例.当执行m个测试用例时,会经过n个结构性测试单元。

定义:方法M关于指标S的覆盖是:n与s的比值,记做C(M,S)。

定义: 方法M关于指标S的冗余是m与s的比值, 记做R(M,S)

定义:方法M关于指标S的净冗余是m与n的比值, 记做NR(M,S)。

在我们之前的场景中, m个测试会经过n条路径, 而总路径数量为s(11)。我们期望覆盖是大于等于1的。

冗余带来的问题就是测试方法有巨大的开销, 我们希望冗余是小的, 冗余越小越好。

净冗余, 我们希望m个测试用例经过更多的结构化单元, 越小越好。

### 三角形程序的指标 (P133)

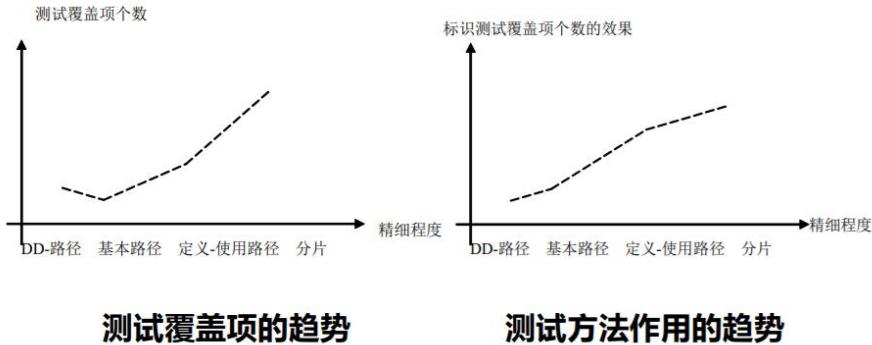
方法	m	n	s	C(M,S)=n/s	R(M,S)=m/s	NR(M,S)=m/n
一般测试	13	7	11	0.64	1.18	1.86
最坏情况	125	11	11	1.0	11.36	11.36
目标	s	s	s	1.0	1.0	1.0

在最坏情况下,  $R(M,S)=11.36$  的意思就是平均 11.36 个测试用例经过了同一条路径。

### 佣金问题的指标

方法	m	n	s	C(M,S)=n/s	R(M,S)=m/s
边界值	25	11	11	1	2.27
决策表	3	11	11	1	0.27
DD-路径	25	11	11	1	2.27
定义-使用路径	25	33	33	1	0.76
程序片	25	40	40	1	0.63

在结构化测试的时候, 路径确实都是 11 条, 但是在定义-使用路径测试和程序片测试中, 我们关心的路径不一样, 关心的是其他的一些特征 (P132)。

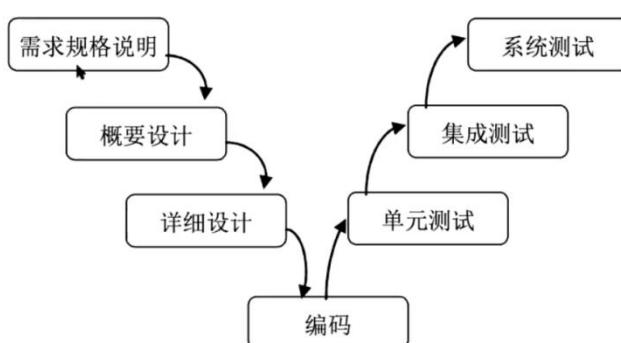


每个测试方法，在测试效率和测试表现情况下是不一样的。  
 自此，我们讲完了白盒测试方法。

## 基于生命周期的测试

软件测试开发都是一个整体性的工作，我们讨论一下整个软件生命周期里，测试起着什么样的作用。

### 8.1.1 传统的观点



### 瀑布式生命周期

比如瀑布模型清晰地表现了软件开发周期。它是基础的软件层次开发过程。有了编码以后，我们就会单元测试。然后单元和单元的集成就是集成测试，单元和单元之间的接口、联系、放在一起会不会有问题。集成测试经常会发现很多错误，在大型项目中，单元和单元是不同项目组开发的，更容易出现单元和单元之间的问题。所以集成测试非常有必要。集成到最后，就可以形成整个系统，我们对照需求设计，看看交付给用户的系统是否满足需求规格的需要。

**Q:** 集成测试和系统测试应该是白盒测试还是黑盒测试？

**A:** 集成测试的时候，有一些模块里面，我们还是需要白盒测试对里面的逻辑进一步的剖析的。但是最后的系统测试往往就是黑盒测试，因为它对应的是需求规格说明书。

从传统瀑布模型的观点：

- 单元测试：面向详细设计，完成对软件独立模块的测试。
- 集成测试：面向概要设计，完成软件模块之间的组合测试。
- 系统测试：面向需求分析，完成系统的功能测试。

系统测试的时候，我们关系系统是不是能交付给用户正确的使用。



## 瀑布模型的优缺点

### ① 优点：

- 瀑布模型框架非常适合层次管理结构
- 每个阶段都有明确的产物，便于进行项目管理
- 详细设计中分工明确，可以并行工作，缩短周期

### ② 缺点？

- 需求规格说明与系统测试之间周期很长，无法加入客户反馈
- 模型几乎排除了综合可能，最先发生在集成测试
- 单元级别大规模并行开发可能会受到人员数量限制
- 开发人员需要对系统“完美”理解

无法加入客户反馈，灵活性差了以后会导致很多问题。

### ④ 增量开发模型：

- 是将一个软件产品分成若干次产品进行提交，每一次新的软件产品的提交，都是在上次软件产品的基础上，增加新的软件功能，直到全部满足客户的需求为止。

### ⑤ 演化开发模型：

- 演化开发模型是在需求分析之前，首先提供给客户或用户一个最终产品的原型（部分主要功能的软件）

### ⑥ 螺旋模型：

- 它与瀑布模型和快速原型模型十分相似。但重要的是，它在每个阶段都增加了风险分析和验证这两个重要的步骤。
- 这三种派生模型的好处，是增加了迭代开发的方法，而不是将成功的风险全部放在了最后阶段！
- 在这样的开发模型中，**回归测试**就成为一种重要的测试方法。
- **回归测试（Regression test）**：对已经完成测试的软件进行修改和增加之后，重新测试软件。
- 回归测试我们在后续章节进行专门的介绍，这里不做特别说明。

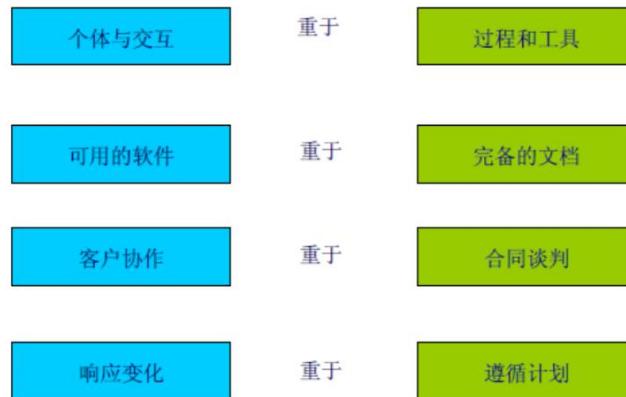
- 事实上，并不仅仅是软件维护阶段的工作。即使在软件开发阶段，软件的实现过程就是一个不断迭代和升级的过程！

### 敏捷开发

- 是软件开发方法的一个很大的范畴，包括Scrum、极限编程（XP）、测试驱动（TDD）、适应性软件开发、结对编程等。

### 推荐书目：

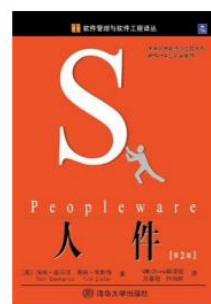
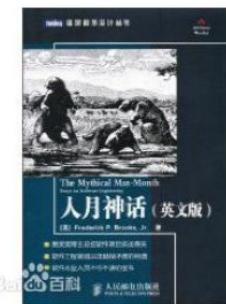
- 《敏捷软件开发：原则、模式与实践》，[美]马丁著，邓辉译，清华大学出版社。
- 《Java敏捷开发——使用Spring、Hibernate和Eclipse》，（美）Anil Hemrajani;韩坤,徐琦，人民邮电出版社。



**敏捷开发的核心思想是：以人为本，适应变化。**

### ◎ 个体和交互 胜过 过程和工具

- 人是软件项目获得成功最为重要的因素
- 合作、沟通能力以及交互能力比单纯的软件编程能力和工具更为重要
- 方法和工具是“死”的，人是“活”的，人要是协作不好，再强大的方法和工具都是“白扯”；



### 可用的软件 胜过 完备的文档

- 过多的面面俱到的文档往往比过少的文档更糟
- 软件开发的主要和中心活动是创建可以工作的软件
- 直到迫切需要并且意义重大时，才进行文档编制
- 编制的内部文档应尽量短小并且主题突出

### ◎ 客户协作 胜过 合同谈判

- 客户不可能做到一次性地将他们的需求完整清晰地表述在合同中
- 为开发团队和客户的协同工作方式提供指导的合同才是最好的合同

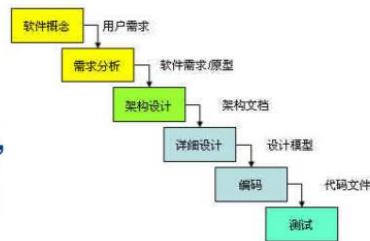
### 响应变化 胜过 遵循计划

- 变化是软件开发中存在的现实
- 计划必须有足够的灵活性与可塑性
- 短期迭代的计划比中长期计划更有效

## 敏捷方法 VS. 瀑布模型

### ④ 瀑布模型

- 固定的、没有弹性的。
- 很困难去达到互动。
- 假如说需求没有完全的被了解，或是可能需要完全地改变项目的需求，瀑布式的模型是比较不适合的。



### ⑤ 敏捷方法

- 完整地开发，每少数几周或是少数几个月里可以测试功能。
- 强调在获得最简短的可执行功能的部分，能够及早给予企业价值。
- 在整个项目的生命周期里，可以持续的改善、增加未来的功能。

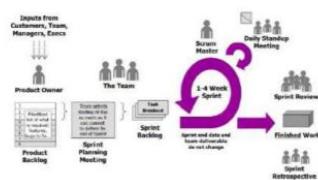
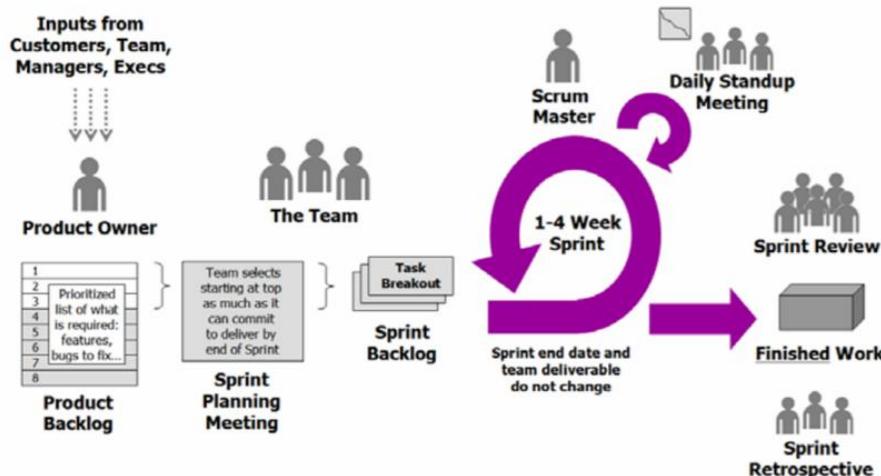


Figure 1. Scrum  
图示 1 Scrum

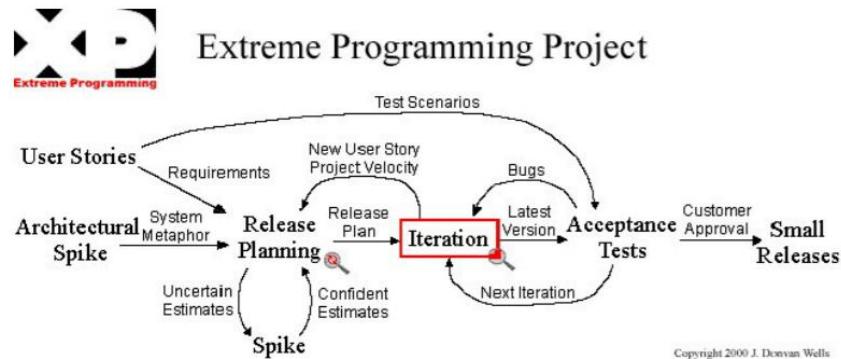
我们希望及早介入到软件测试里。

## Scrum敏捷开发流程



相关网站：[agilemodeling.com](http://agilemodeling.com)  
[extremeprogramming.org](http://extremeprogramming.org)

④ XP活动：倾听→测试→编码→设计（重构）



Copyright 2000 J. Donvan Wells

如果到不了 acceptance test 要求，那么就需要额外的迭代 debug。

## XP的一些典型实践活动：共12项

- (1) .现场客户 (On-site Customer)
- (2) .计划游戏 (Planning Game)
- (3) .系统隐喻 (System Metaphor)
- (4) .简单设计 (Simple Design)
- (5) .代码集体所有 (Collective Code Ownership)
- (6) .结对编程 (Pair Programming)
- (7) .测试驱动 (Test-driven)
- (8) .小型发布 (Small Releases)
- (9) .重构 (Refactoring)
- (10) .持续集成 (Continuous integration)
- (11) .每周40小时工作制 (40-hour Weeks)
- (12) .代码规范 (Coding Standards)

### (1) 现场客户

始终在开发团队中有一位客户。

现场客户的工作：

- 回答问题
- 编写验收测试
- 运行验收测试
- 指导迭代
- 接受版本

### (2) 计划游戏

- 计划是持续的、循序渐进的。每2周，开发人员就为下2周估算候选特性的成本。
- 以业务优先级和技术估计为基础，决定下一版本发布的范围。

### (3) 系统隐喻

在XP中，隐喻是一种概念框架并提供名称的描述系统，类似于其他方法中的体系结构（或体系结构基准）。

- 共识
- 共享的术语空间

例子：Windows风格的界面、网上购物站点的购物车

#### (4) 简单设计

系统应设计得尽可能简单。

#### (5) 代码集体所有

整个团队拥有所有代码。任何人都可以更改他们需要更改的部分。没有唯一对代码有所有权的人。

#### (6) 结对编程

- 结对编程是让两个人共同设计和开发代码的实践。结对者是全职合作者，轮流执行键入和监视；这提供了持续的设计和代码评审。
- 不是两个人做一个人的事情。



#### 学到的经验

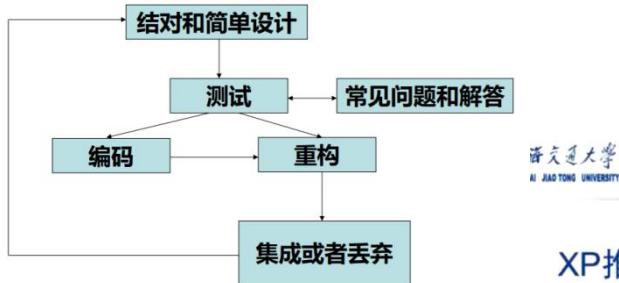
- 程序员和设计人员\协调人结对编程更有效。
- 键盘输入效率！
- 自愿结对编程。

我们行业的主要问题实质上更侧重于社会学而不是科学技术。

——《人件》



#### (7) 测试驱动



#### (8) 小型发布

XP推荐小而频繁的有意义发布。

测试驱动是软件开发里面比较特殊的一种方法，就是很多设计现有测试用例，再进行编码，编码有问题还可以去重构。这样我们一开始就知道软件的功能要求。



#### (9) 重构



#### (10) 持续集成

- 重构是XP的一个重要组成部分。所谓重构是指在不改变代码外在行为的前提下对代码做出的修改，以改进代码的内部结构。
- 重构是一种有纪律的、经过训练的、有条不紊的代码整理方法，可以将整理过程中不小心引入错误的可能性降到最低。从本质上说，重构就是在代码写好之后改进它的设计。
- 重构的节奏：重新推理、小的更改、重新推理、小的更改、重新推理…
- 持续集成的思想是任何时候只有一项任务完成，就集成新代码，构造系统并测试。持续集成是每日构建\每晚构建的一种极限形式，是XP的重要基础。
- 每日构建\每晚构建是将一个软件项目的所有最新代码取出，从头开始编译、链接，用安装软件包将链接好的程序安装好，运行安装后的软件，使用测试工具对主要功能进行测试，发现错误并报告错误的完整过程。

我们需要判断重构有没有必要，必须知道我们为什么要去重构，而不仅仅是因为看不懂而重构代码。

持续集成相对来说还是比较频繁的，每日/每晚构建，这样可以有效发现我们新的代码出现了些什么问题。

- 让开发人员在第一时间了解到软件的错误，并迅速排除错误，是每日构建\每晚构建最重要的目标之一。
- 每日构建\每晚构建必须出日志和报告，并发布构建结果的有关信息，最好能够使用自动化工具发出电子邮件通知。

**每日构建是项目的心跳。如果一个项目的心跳停止了，这个项目就死亡了。**

**Treat the daily build as the heartbeat of the project.**

**If there is no heartbeat, the project is dead.**

一些好的 github 开源项目是每天都在更新，项目组成员还是对项目有信心的。



## 作用

- 降低集成风险
- 加强错误诊断
- 降低不确定性
- 加快开发速度
- 增强团队合作
- 对项目参与者是重要激励



### (11) 每周四十小时工作制

在这里40是一个概数，不是确数。

- 如果能够努力地工作8小时，超过这个时间后就不适于有效地工作了——8小时燃烧
- 再学习
- 你无法改变时间，但是可以改变你的任务。



### (12) 代码规范

- 系统中所有的代码看起来就好像是被单独一人编写的。

# 2022/4/11

我们这节课还是从集成测试和系统测试这两个类别做一个详细的讲解。上节课我们开了一个头，集成测试和系统测试有一些明显的区别：

## 将集成测试与系统测试分开

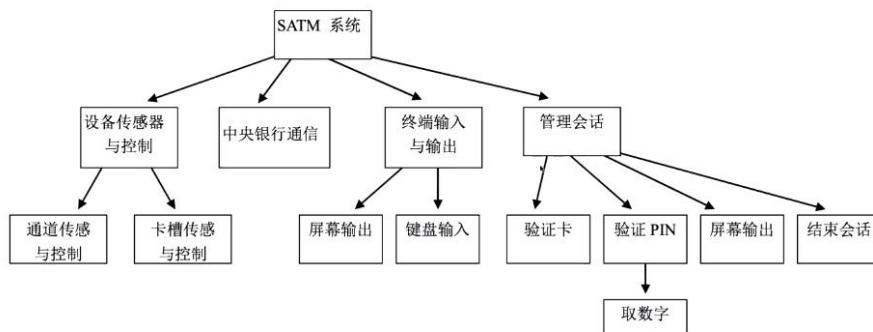
- 从前面的介绍，我们知道集成测试针对的是模块之间的关系；而系统测试针对的是整个系统的功能。
- 集成测试需要了解程序的结构，是一种结构化的测试方法，有路径覆盖的含义。
- 系统测试不需要了解程序的结构，是一种黑盒的测试方法，是功能覆盖的意义。
- 集成测试是由软件开发人员完成的；而系统测试往往是需要用户的参与的。

集成测试是一个典型的白盒测试方法，但我们系统测试是不需要了解程序结构，是通过黑盒测试来功能覆盖的。

我们 PPT 是基于 ATM 给大家讲了一下。我们的课本第十三章集成测试是针对日历来做的。课下大家再针对日历的例子来看一下集成测试和系统测试的过程。我们先进入到集成测试的环节里来。

一般来说我们开发了一个 ATM 机软件系统，它有内在存在的软件模块的调用关系。比如这个系统会调用“设备传感器与控制”，形成一系列的调用关系。

## 事例 SATM



设计完做测试的时候，如果针对这个系统做集成测试，我们应该怎么去做呢？通过这个图，我们知道集成测试是解决模块与模块之间的调用过程是否存在软件的错误。集成测试倾向测试软件调用之间的问题。

Q：如果做典型的集成测试，有哪些方法可以给我们选择呢？

A：从典型的集成策略来讲，可以分为以下集中：自顶向下集成（主程序去调用底下的模块形成调用关系，下面的程序以桩程序出现。桩程序是替代真实模块的代码，是以简单的形式模拟真实的代码，这个过程可以很清晰地定位到调用关系是不是有错误。），自底向上集成（这种情况下我们从叶子结点开始集成，我们要开发驱动器的程序，驱动器就是模拟调用过程）。

### 8.2.1 集成测试的方法

1) **自顶向下集成** 从主程序(顶层)开始,所有下层程序都以“**桩程序**”出现。完成顶层测试后,以真实程序代替“**桩程序**”,向下进行下一层测试。



“**桩程序**” (stub): ?

模拟被调用程序的代码。一般以表格形式存在。

2) **自底向上集成** 从程序的最下层节点(叶子)开始,通过编写“**驱动器**”完成测试,然后以真实程序代替“**驱动器**”,向上进行上一层测试。



“**驱动器**”: 模拟对测试节点的调用驱动。

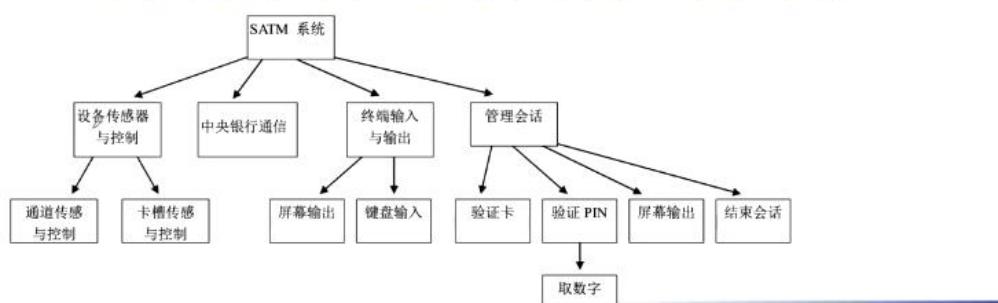
所以,无论是从上往下还是从下往上,一定有一方被模拟。

另外还有结合的方式:三明治集成。它结合自顶向下和自底向上两种策略,往中间做集成。可以充分利用两者的优点,减少桩程序和驱动的开发,特别是程序层次多了以后,这个效果非常明显。另外还有一些不规则的测试方法,大爆炸集成就是不太好的测试方法。

3) **三明治集成** 是自顶向下和自底向上测试的组合,即可以同时从顶和底向中间层集成,可以减少**桩程序**和**驱动的数量**。

4) **大爆炸测试** 不分层次,将所有单元放在一起编译,并进行一次性测试。

对于**SATM系统**,我们知道需要开发**(节点-1)**个**桩程序**:?个;需要开发**(非叶子节点)**个**驱动器**:?个。



如果我们从上往下做集成,每一个边都是一个调用关系,我们要开发**13**个**桩程序**;如果我们从下往上做集成,对每个非叶子结点我们都要开发一个**驱动器**,我们要开发**6**个**驱动器**。所以**桩程序**和**驱动器**的数量是有区别的。

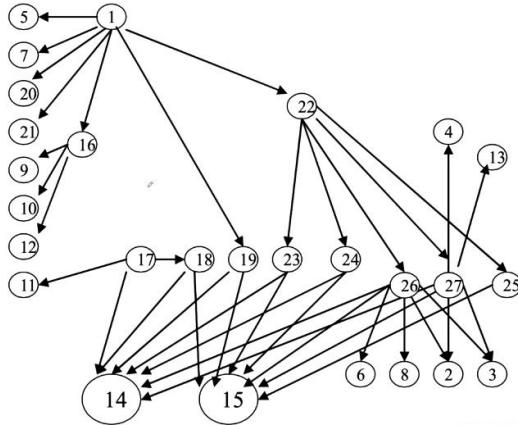
另外还有一种基于调用图的集成。我们从模块之间调用和被调用的关系入手。比如有些软件的层次化不是很清楚,可能只有一个调用图。这种情况下我们只能通过调用图来进行集

成。每个边都是一个调用和被调用的关系。



## 8.2.2 基于调用图的集成

换一个角度，从模块之间调用关系的角度，我们可以得到SATM的调用图。



1. 成对集成，每一对都是调用和被调用的关系。

为减少桩程序和驱动器的开发，可采用调用对的测试方法。SATM的成对测试集，就是调用图中边的数量，共？个。

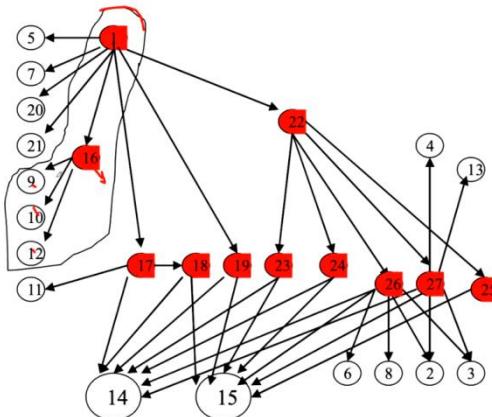
2. 相邻集成，它有如下概念：

为减少测试的数量，以相邻节点为集合，进行测试。

相邻节点：包括所有直接前驱和所有直接后继节点。

对于SATM，显然就是除去叶子节点的所有节点的相邻集合，共？个。

相邻节点包括所有的直接前驱和直接后继的节点。



我们就可以把这个区域的节点做相邻的集成。这样就可以大大减少测试的数量。每个相邻节点纳入直接前驱和直接后继，我们把这个区域做一个测试。

节点1没有直接前驱，但是有很多直接后继。所以它也是一个相邻节点，也会形成一个相邻节点的测试关系。

# 基于路径的集成

前面我们都是针对程序的调用图，看看集成测试过程中考虑节点模块之间的调用关系的时候，通过前面的方法都可以做测试。另外一种方法是基于路径的测试。路径的概念是白盒测试里每个程序模块中，通过画数据流图可以做对应的测试。在程序调用的这里应该怎么有路径的概念呢？

## 8.2.3 基于路径的集成

在面向结构的测试中，我们经常采用路径覆盖的方法，在集成测试中，我们也有类似的概念。

### ◆ 概念

**定义：**程序中的**源节点**是程序开始或重新开始处的语句片段。

**汇节点**是程序执行结束处的语句片断。

**模块执行路径**是以**源节点**开始，以**汇节点**结束的一系列语句，中间没有插入**汇节点**。

**消息**是一种程序设计语言机制，通过它，一个单元将控制转移给另一个单元。

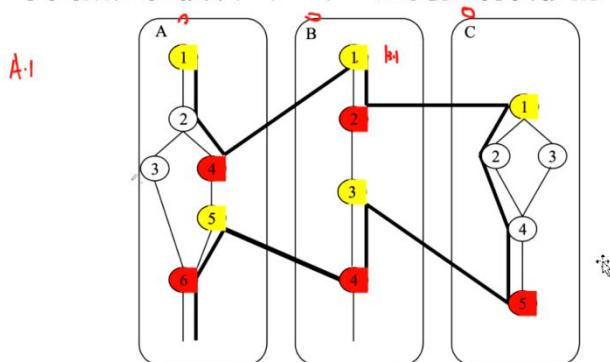
我们现在考虑的不是程序里的路径，而是模块和模块之间的调用关系。我们现在为了做集成测试，我们引入了“模块执行路径”的概念，这是粒度更粗的一种路径的概念。模块执行路径也是以源节点（模块）开始、汇节点（模块）结束。它中间一定没有插入汇节点。

## MM-路径是穿插出现模块执行路径和消息的序列。

MM 路径就是模块到模块的路径。其实这个有点像 DD 路径的概念。

我们通过下面这个例子来看一下，ABC 是三个模块，源节点和汇节点一定是和模块相关的。下图就是 ABC 这三个模块的调用图。每个模块里面有一个程序流图。比如 A.4 和 B.1 之间的边就是通过调用来完成模块之间信息的传递。

下图就是在模块A、B、C之间控制转移的MM-路径



源节点：● 汇节点：●

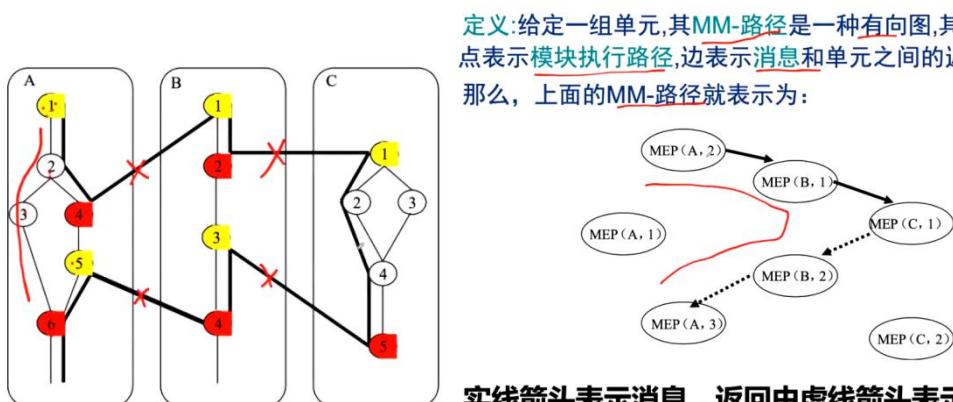
首先我们要找模块执行路径。比如  $MEP(A, 1) = <1, 2, 3, 6>$ , 确实是一条从源节点和汇节点的执行路径。

$MEP(A, 2) = <1, 2, 4>$ , 也是源节点到汇节点的。因为模块执行路径是模块内的, 所以我们不会管跳转的情况。

模块 A 中第三条模块执行路径就是  $MEP(A, 3) = <5, 6>$ 。以此类推, 我们还有其他 4 条模块执行路径。

$$\begin{aligned} MEP(B, 1) &=<1, 2> \\ MEP(B, 2) &=<3, 4> \\ MEP(C, 1) &=<1, 2, 4, 5> \\ MEP(C, 2) &=<1, 3, 4, 5> \end{aligned}$$

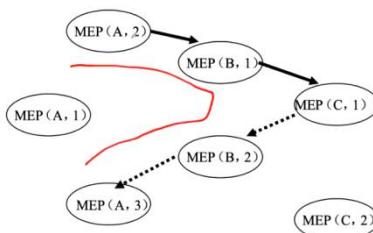
下一步就是找模块到模块之间的执行路径。标记为叉的四条边其实就是完成了模块之间的消息的传递。



定义:给定一组单元,其MM-路径是一种有向图,其中的节

点表示模块执行路径,边表示消息和单元之间的返回。

那么, 上面的MM-路径就表示为:



其实右图的 MM 路径对应的就是左图中粗黑线描述的最长的模块调用路径。

我们来区分一下 DD 路径和 MM 路径的对应关系:

- ④ DD-路径: 模块内的程序执行路径;
- ④ MM-路径: 模块间的模块执行路径序列。

请同学自己阅读 NextDate 的例子 (P169~P174)

其中, UML序列图是MM-路径图的表示方式, 因此, UML序列图可以作为集成测试的路径覆盖标准。



## 集成测试策略比较

策略	对接口的测试能力	对交互功能的测试能力	故障分离辨别力
功能分解	满意但有可能不可靠	有限单元对	好, 尤其是有故障单元
调用图	满意	有限单元对	好, 尤其是有故障单元
MM路径	优秀	全部单元	优秀, 尤其对有故障单元执行路径

三个策略有不一样的特点, MM 路径相应做了程序之间的调用逻辑的分析, 所以它的故障的辨别力也比较好。而功能分解和调用图还是在有限个单元对中去做。

# 系统测试

我们首先看一下测试的层次：



## 8.3 系统测试

从前面的介绍我们知道有：

- 单元测试：白盒(侧重)、详细设计文档；
- 集成测试：白盒、概要设计文档；
- 系统测试：从外部特性对软件进行测试，功能测试。

### 标准：需求规格说明！

系统测试有别于前两个，只管外部表现。我们要基于需求规格说明书来做相应的测试。功能测试的黑盒方法也是一些具体的方法，如何针对一个系统做相应的测试方法。从系统集成的角度去梳理：

- 如何描述系统的功能，其实是通过系统的输入输出来分析。
- 换言之，我们可以通过系统从输入到输出的行为线索来描述系统。

我们关注怎么描述系统的功能（输入和输出的分析）。我们引入线索的概念，线索在不同的关键测试层次中有不同的理解，可以理解为测试的依据，可以对应到测试用例里去。

#### 8.3.1 线索(thread)

线索有不同的层次，单元级线索被理解为指令执行路径，或DD-路径；集成测试线索是MM-路径，即模块执行和消息交替序列。那么系统级线索，就是原子系统功能序列。

另外我们从系统集测试去找线索的，其实我们对应了我们很多功能串起来。比如我们的原子系统功能串起来形成一个序列。

这样的定义并不令人满意，换个角度，单元测试的线索是模块内的路径执行序列（有意义的最小单元）；集成测试的线索是模块间的路径执行序列（是小于系统级的意义单元）；系统级的线索，是系统输入到输出的路径（是功能的最小单元）。

本章，我们讨论系统级线索。

我们以 ATM 机为例：

## SATM中的可能线索：？

- 数字输入-屏幕输出
- 密码（PIN）验证
- 单一事务：存款、取款、查询余额
- 多个事务：包括两个或多个业务活动
- 当然，仅对于密码验证而言，还可以细化为插卡和密码输入2个线索。
- 而密码可以有三次尝试。

Q：实际上来讲，对密码输入来讲，可以细分为插卡和输入密码。输入密码分为三次尝试，那么到底一次尝试是线索还是输入密码总的是线索呢？

**定义：**原子系统功能（ASF）：是一种在系统层可以观察得到的端口输入和输出事件的行动。

- ASF具有事件静止特性，ASF开始于一个端口输入事件，遍历一个或多个MM路径，以一个端口输出事件结束。
- ASF具有事件序列原子化（不愿再细分）特性。
- 卡输入
- PIN输入？是ASF吗？

原子的意思就是说，功能不能再分了。细粒度的不能再分的功能就是原子系统功能。它一定能够观察到端口输入和输出事件（可观测性），我们不能定义一个外部没有表象的功能。原子系统功能的粒度的把握就是“不愿细分”，此时主观认识比较重要，也就是在这个过程中，我们不愿意再细分它了。

我们从端口输入开始遍历 MM 路径，然后从端口输出结束。比如我们可以认为“卡输入”、“密码输入”都是一个原子系统功能。“密码输入”也可以根据我们是否愿意再细分，分成 3 次“密码具体输入操作”。

我们希望把原子系统功能通过有向图的方式组织起来。

**我们总是希望有图形化的方法，来描述系统需求。因此，我们就需要有能够描述系统功能的图。**

**定义：**给定通过原子系统功能描述的系统，系统的ASF图是一种有向图，其中的节点表示ASF，边表示串行流。

**定义：**源ASF是一种原子系统功能，在系统ASF图中作为源节点出现。汇ASF也是一种原子系统功能，在系统ASF图中作为汇节点出现。

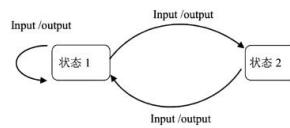
这个就是原子系统功能图，作为分析的依据。原子系统功能图经常会使用有限状态图（FSM）的图的形式。我们通过它可以建立原子系统功能图的表示。

比如我们可以看下左图，状态 1 可以转移到状态 2，伴随着 input 和 output。当然也允许状态的自转移。



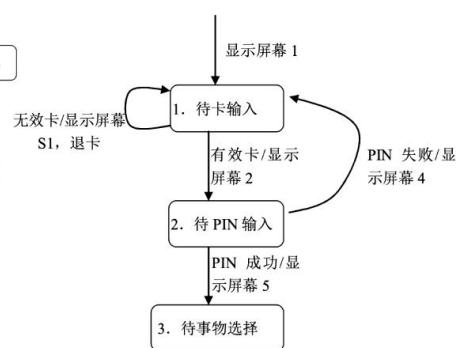
### 8.3.3 建立系统ASF图

#### FSM的一般形式



**注: Input, output  
可以是多值组合**

#### 顶层的SATM的FSM图:



**定义:** 系统线索在系统的ASF图中，是从源ASF到汇ASF的路径。

#### 8.3.2 需求规格说明的基本概念

**数据:** 整数、浮点数、字符串、数组、结构体等；

**操作:** 输入、输出、转换、处理、活动、任务、方法、服务；

**设备:** 端口设备、系统I/O接口；

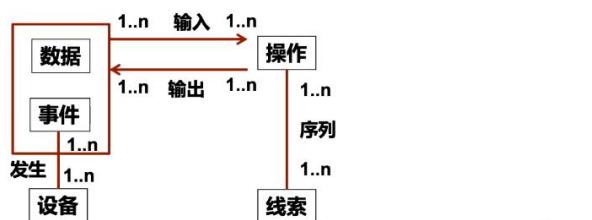
**事件:** 发生在端口设备上的系统级输入（或输出），是操作+数据。

源节点到汇节点就可以形成系统级的路径。我们找的时候是在“原子系统功能图”里去找。上图中还有一些需求规格中的基本概念。

线索的概念是很难把握的，我们先要画出原子系统功能图（系统 ASF 图），然后我们去找源节点到汇节点的路径。

**线索:** 最难把握的概念，根据定义，是从源ASF到汇ASF的路径。因此，要测试线索，就必须画出系统ASF图。

#### 基本概念之间的E/R图（图14-3）



首先我们要画出这个比较抽象的图，我们首先要切分功能，再去花功能之间的关系。功能之间的关系通常会在需求规格中存在。

大家都学过用例图和用例分析（画用例图和写出用例描述）。用例描述通常对应了我们取钱的用例场景，它会把几个功能串起来。这个用例描述其实是有原子功能集的线索的概念。

我们首先找 ASF 图的节点，再去找边的关系（对应边和行动）。



## 基于模型的线索

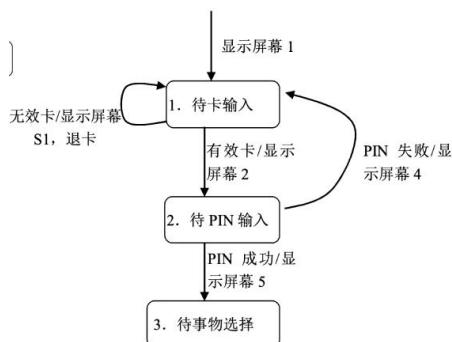
- 建立系统的ASF图，是软件建模的方式之一。换言之，是建立需求规格的形式化方法之一。
- 有限状态机（FSM）是研究系统功能级线索的有效手段。换言之，有限状态机恰是我们前面定义的ASF图。

FSM：节点=ASF

边=事件和行动

我们以下图为例子：

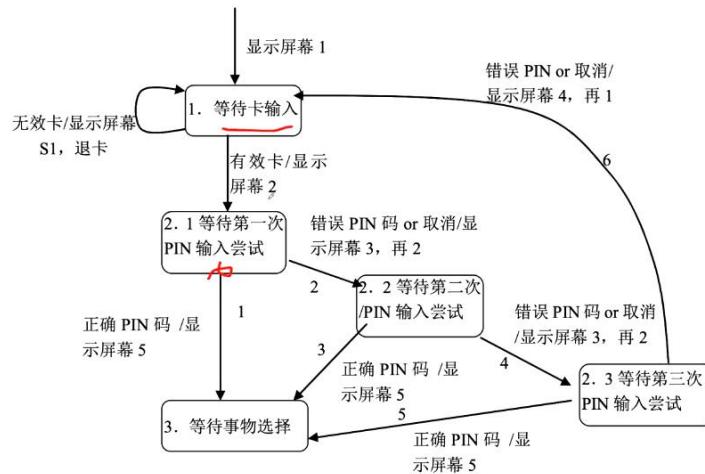
顶层的SATM的FSM图：



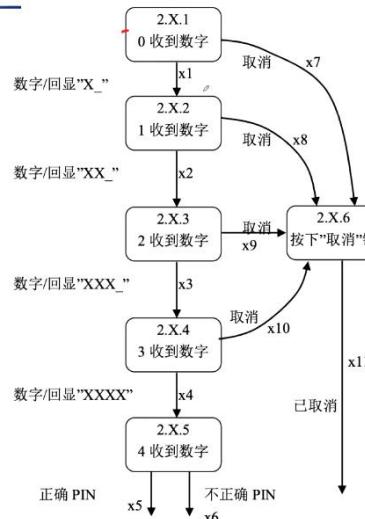
待卡输入就是一个静止状态，还有待输入密码、待事务选择的状态。从系统级来讲，我们有这么几个状态，每个状态都是静止的，可以观测到输入和输出的时间。插入卡就是一个输入，并且会自动显示给用户相应状态。

密码的不同尝试，我们对应到不同的状态。细化以后就变成一个新的状态图。

## 对PIN输入的细化状态机



- 进一步的细化,我们知道每次PIN需要有4位数字输入,而且在<4个数字输入后,都可以按“取消”键返回。则PIN的过程可以细化为右图。
- 为了使测试用例明确,我们不妨设定PIN密码为:1234



这个里面有取消键的,有四位的输入,每次输入后都可以按取消键。我们看一下密码验证有几次路径:

我们来算算有多少不同的线索 (从源到汇的路径)

仅计算一下正确PIN的路径:

第1次PIN正确: 1条;  
第2次PIN正确: 5条;  
第3次PIN正确: 25条;  
共31条;

不正确的路径: 125条,为什么?

第1次PIN正确的事件序列: P183 表14-1

第1次密码正确肯定要输入1234,就是一条路径。第二次密码正确的情况取决于第一次错误的路径有多少个。密码不正确的情况分为没有输入数字、输入1个按取消、输入2个按取消、输入3个按取消、输入4个密码错误,显然有5种情况。

所以第三次密码正确的情况就是25条。

我们找线索的时候，也是通过路径去找系统级别的线索，系统级别的路径是能够对应到我们实现的功能的一些解释。状态图的画法也是对系统功能的抽象，对应不到代码的实现。所以这是黑盒测试。

这样，我们就可以根据不同的情况来设计测试用例了：

情况	输入
第1次 PIN 正确	1234
第2次 PIN 正确	取消-1234 1-取消-1234 13-取消-1234 789-取消-1234 4567-1234

用例场景都是对应我们系统级的线索，也就是原子系统功能图的路径的概念，也对应了我们功能的解释。这样我们就可以通过路径的概念完成相应的系统级测试。



### 8.3.4 线索测试的结构策略

在获得系统状态图之后，如何确定测试用例呢？

既然已经有系统的有限状态图，当然测试用例的选择就是考虑对路径的覆盖了。

对于PIN的细化状态图，共有6条路径。

输入事件序列	路 径
1234	x1、x2、x3、x4、x5
1235	x1、x2、x3、x4、x6
C	x7、x11
1C	x1、x8、x11
12C	x1、x2、x8、x11
123C	x1、x2、x3、x10、x11

用例描述就是用文字去描述一个用例，它对应了一个个的场景。



### 基于用例的线索

#### ④ 用例的层次

- 高级用例（非常类似于一个敏捷开发故事）
- 基本用例
- 基本扩展用例

#### ④ 问题：

- 如何利用需求分析中的用例进行系统测试？
- 用例=用例图+用例描述 (?)

系统测试和集成测试我们没有布置相应的作业。真正去做基于线索的测试的时候，我们要通过用例描述画出原子系统功能图，然后再通过路径去做。

**2022/4/18**

我们把后面大家要用到的工具和流程给大家梳理一下。后面涉及的工具也会提前在这里讲完。下面两次作业，一次是 Web 的 GUI 测试，用到的工具是 Selenium，还有一个是性能测试的工具 Load Runner。

## Selenium



### 第一部分

- 1. Selenium 简介**
- 2. 什么情况下选用WebDriver**
- 3. Webdriver对浏览器的简单操作**
- 4. Webdirver对浏览器的支持**
- 5. 基本操作**
- 6. 如何对页面元素进行操作**

第一部分是基础的介绍，在做大作业的时候第一部分是最重要的。



### Selenium 简介

- ④ **Selenium 是ThroughtWorks 公司一个强大的开源 Web 功能测试工具系列，**
- ④ **Selenium 可以使用录制工具录制脚本，测试页面。**
- ④ **Selenium 可以生成类html 代码，java 代码，ruby 代码等。**
- ④ **Selenium 录制工具根据id 属性定位html 元素**
- ④ **Selenium IDE 仅支持Selenium 语言。**
- ④ **Selenium RC 支持很多语言，如：C# , Java , Python , Ruby 等。**

我们需要通过 Id 定位元素来模拟用户操作。我们这次作业使用的是 Selenium 2.0。主要就是增加了 webdriver 这样一个测试框架。



## Selenium 2.0

- ④ **selenium2.0 = selenium1.0+ webdriver**
- ④ WebDriver项目是由Simon Stewart 提出的，它是一个轻便简洁的自动化测试框架。WebDriver通过尝试不同的方法去解决Selenium1.0所面临的问题。不单单是使用javascript，WebDriver会使用任何一种更合适的机制来操作浏览器。
- ④ 通过更灵活的机制去操控浏览器，就能很好的绕过浏览器javascript的安全限制。当这些技术还不够用时，可以调用系统设备操作，尤其是当你需要一些键盘和鼠标操作时，通过这些技术，可以更好的模拟用户的真实浏览器操作
- 

这次作业我们就会选用 WebDriver，因为 Selenium1.0 不支持一部分的浏览器功能。



## 什么情况下选用WebDriver？

- ④ **Selenium-1.0不支持的浏览器功能。**
- ④ **multiple frames, multiple browser windows, popups, and alerts.**
- ④ **页面导航。**
- ④ **下拉。**
- ④ **基于AJAX的UI元素。**

Selenium 不需要特定的安装环境。对于测试环境只需要在工程文件中添加 selenium 对应的库文件。



## 安装selenium说明

- ④ **Selenium无需安装环境；**
- ④ **只需在工程文件中，增加库文件：selenium-server-standalone-X.X.X，即可使用；**
- ④ **或者使用maven导入：**

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.11.0</version>
    </dependency>
</dependencies>
```

- ④ **需要对应browser和browser driver支持**

Selenium 还需要特定的浏览器和 webdriver 的支持。对于不同的浏览器，需要创建不同的 WebDriver 的实例对象。这边我们比较推荐 Firefox 和 Chrome 的 driver。



## Webdriver对浏览器的简单操作

### A ) 打开一个测试浏览器

```
public class OpenBrowsers {  
    public static void main(String[] args) {  
        //打开默认路径的firefox  
        WebDriver diver = new FirefoxDriver();  
  
        //打开指定路径的firefox,方法1  
        System.setProperty("webdriver.firefox.bin","C:\\Program Files\\Mozilla Firefox\\firefox.exe");  
        WebDriver driver1 = new FirefoxDriver();  
  
        //打开指定路径的firefox,方法2  
        File pathToFirefoxBinary = new File("C:\\Program Files\\Mozilla Firefox\\firefox.exe");  
        FirefoxBinary firefoxbin = new FirefoxBinary(pathToFirefoxBinary);  
        WebDriver driver2 = new FirefoxDriver(firefoxbin,null);  
  
        //打开ie  
        System.setProperty("webdriver.ie.driver","E:\\2013上半年\\Web应用项目测试\\software\\IEDriverServer_Win32\\IEDriverServer.exe");  
        WebDriver ie_driver = new InternetExplorerDriver();  
  
        //打开chrome  
        System.setProperty("webdriver.chrome.driver","E:\\2013上半年\\Web应用项目测试\\software\\chromedriver_win\\chromedriver.exe");  
        System.setProperty("webdriver.chrome.bin","C:\\Program Files\\Google\\Chrome\\Application\\chrome.exe");  
        WebDriver chrome_driver = new ChromeDriver();  
  
        String url = "http://www.51.com";  
        driver1.get(url);  
        driver2.get(url);  
        ie_driver.get(url);  
        chrome_driver.get(url);  
    }  
}
```



## Webdriver对浏览器的简单操作

### B ) 打开1个具体的url

```
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class OpenUrl {  
    public static void main(String[] args) {  
        String url = "http://www.51.com";  
        WebDriver driver = new FirefoxDriver();  
  
        //用get方法  
        driver.get(url);  
  
        //用navigate方法, 然后再调用to方法  
        driver.navigate().to(url);  
    }  
}
```

测试完了我们可以关闭浏览器。

### D) 返回当前页面的url和title

```
public class GetUrlAndTitle {  
    public static void main(String []args){  
  
        String url = "http://www.51.com";  
        WebDriver driver = new FirefoxDriver();  
  
        driver.get(url);  
  
        //得到title  
        String title = driver.getTitle();  
        //得到当前页面url  
        String currentUrl = driver.getCurrentUrl();  
  
        //输出title和currenturl  
        System.out.println(title+"\n"+currentUrl);  
    }  
}
```

### C) 关闭浏览器

```
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class CloseBrowser {  
    public static void main(String []args){  
        String url = "http://www.51.com";  
        WebDriver driver = new FirefoxDriver();  
        driver.get(url);  
  
        //用quit方法  
        driver.quit();  
        //用close方法  
        driver.close();  
    }  
}
```

## E) 其他方法

- ④ **getWindowHandle()**    返回当前的浏览器的窗口句柄
- ④ **getWindowHandles()**    返回当前的浏览器的所有窗口句柄
- ④ **getPageSource()**    返回当前页面的源码

我们有时候可能在同一个浏览器里打开不同的窗口进行切换。

HtmlUnit Driver 是轻量级的，运行速度很快，但是对 js 支持不是很好。所以在作业里不建议使用 Html Unit Driver 和 InternetExplorer Driver。



## Webdriver对浏览器的支持

### HtmlUnit Driver

- ④ **优点：**HtmlUnit Driver不会实际打开浏览器，运行速度很快。对于用FireFox 等浏览器来做测试的自动化测试用例，运行速度通常很慢，HtmlUnit Driver 无疑是可以很好地解决这个问题。
- ④ **缺点：**它对JavaScript的支持不够好，当页面上有复杂JavaScript时，经常会捕获不到页面元素。
- ④ **使用：**  
**WebDriver driver = new HtmlUnitDriver();**
- ④ 在selenium3中取消了对HtmlUnit Driver的支持

### InternetExplorer Driver

- ④ **优点：**直观地模拟用户的实际操作，对JavaScript提供完善的支持。
- ④ **缺点：**是所有浏览器中运行速度最慢的，并且只能在Windows下运行，对CSS以及XPATH的支持也不够好。
- ④ **使用：**  
**WebDriver driver = new InternetExplorerDriver();**

推荐的是 FireFox Driver，但是它启动比较慢。

### FireFox Driver

- ④ **优点：**FireFox Dirver对页面的自动化测试支持得比较好，很直观地模拟页面的操作，对JavaScript的支持也非常完善，基本上页面上做的所有操作FireFox Driver都可以模拟。
- ④ **缺点：**启动很慢，运行也比较慢，不过，启动之后 Webdriver的操作速度虽然不快但还是可以接受的，建议不要频繁启停FireFox Driver。
- ④ **使用：**  
**WebDriver driver = new FirefoxDriver();**

在 WebDriver 中都是通过 findElement 来找到页面的元素，一般情况下我们通过 id 和

name 来进行查找。



## 基本操作

### 如何找到页面元素

- ④ Webdriver的findElement方法可以用来找到页面的某个元素，最常用的方法是用id和name查找。下面介绍几种比较常用的方法。

#### By ID

假设页面写成这样：

```
<input type="text" name="passwd" id="passwd-id"/>
```

那么可以照样找到页面的元素：通过id查找

```
WebElement element =  
driver.findElement(By.id("passwd-id"));
```

例：

```
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver; import org.openqa.selenium.WebElement;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class ByUserId {  
  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub WebDriver dr = new FirefoxDriver();  
        dr.get("http://www.51.com");  
  
        WebElement element = dr.findElement(By.id("passport_51_user"));  
        System.out.println(element.getAttribute("title"));  
    }  
}
```

#### By Name

通过name查找：

```
WebElement element =  
driver.findElement(By.name("passwd"));
```

例：

```
public class ByClassName {  
public static void main(String[] args) { WebDriver driver = new  
FirefoxDriver();  
driver.get("http://www.51.com");  
  
WebElement e=dr.findElement(By.name("passport_51_user"));  
System.out.println(element.getTagName());  
}  
}
```

Class 可能一个表格下面有很多元素都属于同一个 Class Name，所以我们要 findElements

找到所有符合条件的。

### By Class Name

假设页面写成这样：

```
<div class="cheese"><span>Cheddar</span></div>
<div class="cheese"><span>Gouda</span></div>
```

可以通过这样查找页面元素：

```
List<WebElement>cheeses =
driver.findElements(By.className("cheese"));
```

### By XPATH

通过XPATH查找：

```
WebElement element
=driver.findElement(By.xpath("//input[@id='pass wd-
id']"));
```

### By CSS

假设CSS写成这样：

```
<div id="food" >
<span class="dairy" >milk</span>
<span class="dairy aged" >cheese</span>
</div>
```

通过CSS selectors查找：

```
WebElement element =driver.findElement
(By.cssSelector( "#food span.dairy.aged"));
```

我们还可以通过超链接的文字进行查找。

### By Link Text

假设页面元素写成这样：

```
<a href="http://www.google.com/search?q=cheese">cheese</a>>
```

那么可以通过这样查找：

```
WebElement cheese=driver.findElement(By.linkText("cheese"));
```

我们在游览器中比较多的情况就是选择了某个输入框输入内容、点击某个按钮，我们需要对页面元素进行操作。



## 如何对页面元素进行操作

### 输入框 ( text field or textarea )

找到输入框元素 :

```
WebElement element = driver.findElement(By.id("passwd-id"));
```

在输入框中输入内容 :

```
element.sendKeys("test");
```

将输入框清空 :

```
element.clear();
```

获取输入框的文本内容 :

```
element.getAttribute("value")
```

我们还可以通过 `getAttribute` 得到输入框的输入内容。

### 多选项(checkbox)

#### 单选项(Radio Button)

找到单选框元素 :

```
WebElement radio=driver.findElement(By.id("BookMode"));
```

选择某个单选项 :

```
radio.click();
```

清空某个单选项 :

```
radio.clear();
```

判断某个单选项是否已经被选择 :

```
radio.isSelected();
```

多选项的操作和单选的差不多 :

```
WebElement checkbox =
```

```
driver.findElement(By.id("myCheckbox."));
```

```
checkbox.click();
```

```
checkbox.clear();
```

```
checkbox.isSelected();
```

```
checkbox.isEnabled();
```

### 左右选择框 (穿梭框)

#### 按钮(button)

找到按钮元素 :

```
WebElement saveButton = driver.findElement(By.id("save"));
```

点击按钮 :

```
saveButton.click();
```

判断按钮是否enable:

```
saveButton.isEnabled();
```

也就是左边是可供选择项，选择后移动到右边的框中，反之亦然。例如：

```
Select lang = new Select(driver.findElement(By.id("languages")));
```

```
lang.selectByVisibleText( "English" );
```

```
WebElement
```

```
addLanguage=driver.findElement(By.id("addButton"));
```

```
addLanguage.click();
```

### 弹出对话框(Popup dialogs)

```
Alert alert = driver.switchTo().alert();
```

```
alert.accept();
```

```
alert.dismiss();
```

```
alert.getText();
```

我们需要传入文件的路径。

#### 上传文件 (Upload File)

上传文件的元素操作 :

```
WebElement adFileUpload = driver.findElement(By.id("WAP-upload"));
```

```
String filePath = "C:\\test\\uploadfile\\media_ads\\test.jpg";
```

```
adFileUpload.sendKeys(filePath);
```

或

```
approve.submit(); //只适合于表单的提交
```

在作业中，第二部分的内容我们可以选择性地用到一些。第三次作业的要求是使用不少于数量的元素操作，可能就会遇到多个对象的定位问题、层级定位等问题。



## 第二部分

### 1. 多个对象的定位方法

### 2. 层级定位

### 3. frame的处理

### 4. 弹出窗口

### 5. 处理select下拉框

### 6. 操作Cookies

### 7. 拖放元素



### 多个对象的定位方法

- ④ `findElements()`方法可以返回一个符合条件的元素 List 组

```
import java.util.List;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class FindElementsStudy {
    public static void main(String[] args) {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.51.com");

        List<WebElement> element = driver.findElements(By.tagName("input"));
        for (WebElement e : element) {
            System.out.println(e.getAttribute("id"));
        }
    }
    driver.quit();
}
```

层级定位就是我们可以先定位到父元素。我们可以根据父元素再调用 `findElements` 方法。



### 层级定位



### 层级定位

- ④ 层级定位的思想是先定位父元素，然后再从父元素中精确定位出其我们需要选取的子元素。
- ④ 层级定位一般的应用场景是无法直接定位到需要选取的元素，但是其父元素比较容易定位，通过定位父元素再遍历其子元素选择需要的目标元素，或者需要定位某个元素下所有的子元素。

```
import java.io.File;
import java.util.List;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxBinary;
import org.openqa.selenium.firefox.FirefoxDriver;

public class LayerLocator {

    public static void main(String[] args) {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.51.com");

        WebElement element = driver.findElement(By.className("login"));
        List<WebElements> el = element.findElements(By.tagName("label"));
        for(WebElement e : el) {
            System.out.println(e.getText());
        }
    }
}
```

软件学院

对于网页中嵌套网页的情况，我们就可以通过 `iframe` 方法进入子网页。



## iframe的处理

selenium webdriver中提供了进入一个iframe的方法：

```
WebDriver org.openqa.selenium.WebDriver.  
TargetLocator.frame(String nameOrId);
```

提供了一个返回default content的方法：

```
WebDriver  
org.openqa.selenium.WebDriver.TargetLocator.default  
Content()
```

比如在 main.html 里就嵌入了 iframe 子网页。



实例：main.html

```
<html>  
<head>  
<title>FrameTest</title>  
</head>  
<body style="background-color: #990000;">  
<div id = "id1">this is a div!</div>  
<iframe id = "frame"    frameborder="0"  
scrolling="no" style="left:0;position:absolute;" src  
= "frame.html"></iframe>  
</body>  
</html>
```



实例：frame.html

```
<html>  
<head>  
<title>this is a frame!</title>  
</head>  
<body style="background-color: #009900;">  
    <div id = "div1">this is a div, too!</div>  
    <label>input:</label>  
    <input id = "input1"></input>  
</body>  
</html>
```

我们在外层网页里是拿不到 iframe 里的元素的，我们要先拿到子网页并且切换过去。这样才可以拿到子网页里的元素，在子网页里我们也拿不到父网页里的元素。



## 测试

```
public class FrameTest {  
    public static void main(String[] args) {  
        WebDriver dr = new FirefoxDriver();  
        String url = "file:///C:/Users/LRQ/workspace/2013-selenium-webdriver-2/main.html"  
        dr.get(url);  
        //在default content定位id="id1"的div  
        dr.findElement(By.id("id1"));  
        //此时，没有进入到id="frame"的frame中时，以下两句会报错  
        dr.findElement(By.id("div1")); //报错  
        dr.findElement(By.id("input1")); //报错  
        //进入id="frame"的frame中，定位id="div1"的div和id="input1"的输入框。  
        dr.switchTo().frame("frame");  
        dr.findElement(By.id("div1"));  
        dr.findElement(By.id("input1"));  
        //此时，没有跳出frame，如果定位default content中的元素也  
        //会报错  
        dr.findElement(By.id("id1")); //报错  
        //跳出frame,进入default content;重新定位id="id1"的div  
        dr.switchTo().defaultContent();  
        dr.findElement(By.id("id1"));  
    }  
}
```

软件学院

弹出窗口指的是打开一个新的浏览器窗口。



## 弹出窗口

在selenium 1.X里面得到弹出窗口是一件比较麻烦的事，特别是新开窗口没有id、name的时候。在 selenium webdriver中得到新开窗口相对简单的多，它无关新开窗口的id、name等属性。以下面的 popup.html为例：

```
<span style="white-space: normal; background-color: rgb(255, 255, 255);">test.html</span>

<html>
<head><title>Test Popup Window</title></head>

<body>
<a id = "51" href = "http://www.51.com/" target = "_blank">Let's go!</a>
</body>

</html>
```

这种情况下，我们要先拿到窗口再进行迭代。



## 测试

```
public class PopupWindowTest {
public static void main(String[] args) { WebDriver dr = new FirefoxDriver();
String url ="file:///C:/Users/LRQ/workspace/2013-selenium-webdriver-2/popup.html
dr.get(url);
dr.findElement(By.id("51")).click();
//得到当前窗口的句柄
String currentWindow = dr.getWindowHandle();
//得到所有窗口的句柄
Set<String> handles = dr.getWindowHandles();
Iterator<String> it = handles.iterator(); while(it.hasNext()){
String handle = it.next(); if(currentWindow.equals(handle)) continue; WebDriver
window = dr.switchTo().window(handle); System.out.println("title=
"+window.getTitle());
System.out.println ("url="+window.getCurrentUrl());

}
}
```



## 弹出窗口

- ④ 捕获或者说定位弹出窗口的关键在于获得弹出窗口的句柄。
- ④ 在上面的代码里，使用windowhandle方法来获取当前浏览器窗口的句柄，使用了windowhandles方法获取所有弹出的浏览器窗口的句柄，然后通过排除当前句柄的方法来得到新开窗口的句柄。
- ④ 在获取新弹出窗口的句柄后，使用switchto.window(newwindow\_handle)方法，将新窗口的句柄作为参数传入既可捕获到新窗口了。
- ④ 如果想回到以前的窗口定位元素，那么再调用1次 switch\_to.window 方法，传入之前窗口的句柄既可达到目的。

我们拿到点击按钮之后，就可以使用 switchTo().alert()方法拿到警示弹窗。最后如果 accept 方法模拟点击确认按钮。

## 处理alert、confirm、prompt对话框

- ④ alert、confirm、prompt这样的js对话框在 selenium1.X时代也是难啃的骨头，常常要用 autoit来帮助处理。selenium webdriver中处理这些对话框十分方便简洁。以下面Dialogs.html代码为例：

```
<html>
<head>
<title>Alert</title>
</head>

<body>
<input id = "alert" value = "alert" type = "button" onclick = "alert("欢迎！请按确认继续！");">
<input id = "confirm" value = "confirm" type = "button" onclick = "confirm("确定吗？");">
<input id = "prompt" value = "prompt" type = "button" onclick = "var name = prompt('请输入你的名字：';请输入你的名字);document.write(name)"/>
</body>
</html>
```

## 测试代码

```
public class DialogsTest {
public static void main(String[] args) { WebDriver dr = new
FirefoxDriver();
String url = "file:///C:/Users/LRQ/workspace/2013-selenium- webdriver-
2/Dialogs.html";
dr.get(url);

//点击第一个按钮，输出对话框上面的文字，然后擦掉
dr.findElement(By.id("alert")).click();
Alert alert = dr.switchTo().alert();
String text = alert.getText();
System.out.println(text);
alert.dismiss();
```

```
//点击第二个按钮，输出对话框上面的文字，然后点击确认
dr.findElement(By.id("confirm")).click();
Alert confirm = dr.switchTo().alert();
String text1 = confirm.getText();
System.out.println(text1);
confirm.accept();

//点击第三个按钮，输入你的名字，然后点击确认，最后
dr.findElement(By.id("prompt")).click();
Alert prompt = dr.switchTo().alert(); String text2 =
prompt.getText(); System.out.println(text2);
prompt.sendKeys("jarvi");
prompt.accept();
}
```

下拉框就要用到父子元素的查找。

## 测试代码

## 处理alert、confirm、prompt对话框

- ④ 从以上代码可以看出dr.switchTo().alert();这句可以得到alert|confirm|prompt对话框的对象，然后运用其方法对其进行操作。对话框操作的主要方法有：

getText() 得到它的文本值  
accept() 相当于点击它的“确认”  
dismiss() 相当于点击“取消”或者叉掉对话  
sendKeys() 输入值

## 处理select下拉框

### 处理select下拉框

- ④ 下面来看一下selenium webdriver是如何来处理select下拉框的，以 select1.html为例。

```
<html>
<head>
<style type="text/css"> div#container{width:500px;
div#header {background-color:#99bbbb;}
div#car-menu {background-color:#ffff99; height:200px; width:200px;
float:left;} div#content {background-color:#EEEEEE; height:200px;
width:300px; float:left;} div#footer {background-color:#99bbbb; clear:both;
text-align:center;}
h1 {margin-bottom:0;}
h2 {margin-bottom:0; font-size:14px;} ul {margin:0;
list-style:none;}
</style>
</head>
```

```
<body>
<div id="container">
<div id="header">
<h1>Main Title of Car Select</h1>
</div>

<div id="car-menu">
<h2>品牌选择</h2>
<select name="cars",id="select">
<option value="volvo">Volvo</option>
<option value="saab">Saab</option>
<option value="fiat" selected="selected">Fiat</option>
<option value="audi">Audi</option>
<option value="bmw">BMW</option>
<option value="Mercedes Benz ">Mercedes Benz </option>
</select>
</div>
```

```

public class SelectsTest {
    public static void main(String[] args) { WebDriver dr = new FirefoxDriver();
        dr.get("file:///E:/软件测试课程/selenium/select1.html");

        // 通过下拉列表中选项的索引选中第二项。
        Select selectCar = new Select(dr.findElement(By.name("cars")));
        selectCar.selectByIndex(4);

        // 通过可见文字“2.5”选中相应项。
        Select selectEngin = new Select(dr.findElement(By.name("engine")));
        selectEngin.selectByVisibleText("2.5");

    }
}

```

浏览器会通过 Cookies 存储一些登录状态，主要就是通过 getCookies 获取当前页面的 cookie。

```

public class CookiesStudy {
    public static void main(String[] args) {
        WebDriver dr = new FirefoxDriver();
        dr.get("http://www.51.com");
        // 增加一个 name = "name", value = "value" 的 cookie
        Cookie cookie = new Cookie("name", "value");
        dr.manage().addCookie(cookie);
        // 得到当前页面下所有的 cookies，并且输出它们的所在域、name、value、有效日期和路径
        Set<Cookie> cookies = dr.manage().getCookies();
        System.out.println(String
            .format("Domain > name > value > expiry > path"));
        for (Cookie c : cookies)
            System.out.println(String.format("%s > %s > %s > %s >%s",
                c.getDomain(), c.getName(), c.getValue(), c.getExpiry(), c.getPath()));
        // 删除 cookie 有三种方法
        // 第一种：通过 cookie 的 name
        dr.manage().deleteCookieNamed("CookieName");
        // 第二种：通过 Cookie 对象
        dr.manage().deleteCookie(cookie);
        // 第三种：全部删除
        dr.manage().deleteAllCookies();
    }
}

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class DragAndDrop {
    public static void main(String[] args) {
        WebDriver dr = new FirefoxDriver();
        dr.get("http://koyoz.com/demo/html/drag-drop/drag-drop.html");
        // 首先拿出要拖入的页面元素对象和目标对象，然后进行插入。
        WebElement element = dr.findElement(By.id("item1"));
        WebElement target = dr.findElement(By.id("drop"));
        (new Actions(dr)).dragAndDrop(element, target).perform();

        // 利用循环把其它 item 也插入
        String id = "item";
        for (int i = 2; i < 6; i++) {
            String item = id + i;
            dragAndDrop(element, target);
        }
    }
}

```

## 拖放元素

- ④ selenium webdriver 可以实现把一个元素拖放到另一个元素里面

主要是使用了 dragAndDrop 方法。

## 第三部分

1. 如何等待页面元素加载完成
2. 验证码
3. 利用 selenium-webdriver 截图
4. 如何处理 table
5. 利用 Actions 类模拟鼠标和键盘的操作
6. 启用 firefox 代理
7. 启用默认情况下被 firefox 禁用的功能
8. 临时指定插件

# Load Runner

接下来我们介绍一下第四次作业，Load Runner 的使用。相比于第三次作业来说，第四次作业难度大一些，因为 Load Runner 不是很稳定。



## 目录

- 1. 理解软件性能**
- 2. LoadRunner 简介**
- 3. LoadRunner功能演示**
- 4. Virtual User Generator**
  - 创建脚本
  - 准备脚本
- 5. Controller**
  - 创建负载测试场景
  - 运行负载测试
- 6. Analysis (分析)**
- 7. Q & A**

不同的用户关注点是不一样的。



### 1 理解软件性能

#### 什么是“软件性能”？

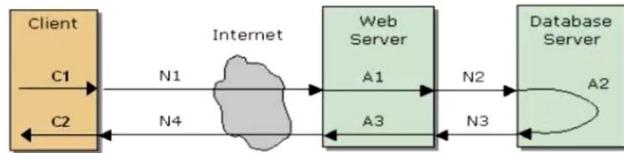
- 响应时间
- 并发用户数
- 吞吐量
- 资源可用度
- 其它？

响应时间其实就是用户在 APP 做了一个操作之后，前端计算->网络传输给后端->请求数据库->数据返回到后端->后端计算->客户端得到需要的数据->展现给用户。响应时间也包含了上述的几个阶段。



## 软件性能指标

### • 响应时间



从图中我们可以清楚的看到一个请求的响应时间是由几部分时间组成的，包括

C1: 用户请求发出前在客户端需要完成的预处理所需要的时间；

C2: 客户端收到服务器返回的响应后，对数据进行处理并呈现所需要的时间；

A1: Web/App Server 对请求进行处理所需要的时间；

A2: DB Server 对请求进行处理所需的时间；

A3: Web/App Server 对 DB Server 返回的结果进行处理所需的时间；

N1: 请求由客户端发出并达到Web/App Server 所需要的时间；

N2: 如果需要进行数据库相关操作，由Web/App Server 将请求发送至DB Server 所需要的时间；

N3: DB Server 完成处理并将结果返回Web/App Server 所需的时间；

N4: Web/App Server 完成处理并将结果返回给客户端所需的时间；

从用户的角度来看，响应时间=(C1+C2)+(A1+A2+A3)+(N1+N2+N3+N4)；但是从系统的角度来看，响应时间只包括(A1+A2+A3)+(N1+N2+N3+N4)。

在理解了响应时间的组成之后，可以帮助我们通过对响应时间的分析来更好的识别和定位系统的性能瓶颈。

我们要分清楚几个概念：



## 软件性能指标

并发用户数 v.s. 系统用户数 v.s. 同时在线用户数

**系统用户数：**系统额定的用户数量，如一个OA系统，可能使用该系统的用户总数是2000个，那么这个数量，就是系统用户数。

**同时在线用户数：**在一定的时间范围内，最大的同时在线用户数量

**平均并发用户数**的计算：

$$C = nL / T$$

其中C是平均的并发用户数，n是平均每天访问用户数，L是一天内用户从登录到退出的平均时间（操作平均时间），T是考察时间长度（一天内多长时间有用户使用系统）



## 软件性能指标

### 吞吐量

在不同的测试工具中，对于吞吐量(Throughput)会有不同的解释。例如，在LoadRunner中，这个指标是以字节数为单位来衡量网络吞吐量的，而在JMeter中则是以事务数/秒为单位来衡量系统的响应能力的。

不过在大多数英文的性能测试方面的书籍或资料中，吞吐量的定义使用的是后者。

0

### 资源可用度

指系统各种资源的使用情况，如cpu占用率、内存占用率等，一般使用“资源实际使用/总的资源可用量”形成资源利用率。



## 1理解软件性能

### 用户视角的软件性能

- 从用户的角度来说，软件性能就是软件对用户操作的响应时间。说得更明确一点，对用户来说，当用户单击一个按钮、发出一条指令或是在Web页面上单击一个链接，从用户单击开始到应用系统把本次操作的结果以用户能察觉的方式展示出来，这个过程所消耗的时间就是用户对软件性能的直观印象。
- 必须要说明的是，用户所体会到的“响应时间”既有客观的成分，也有主观的成分。例如，用户执行了某个操作，该操作返回大量数据，从客观的角度来说，事务的结束应该是系统返回所有的数据，响应时间应该是从用户操作开始到所有数据返回完成的整个耗时；但从用户的主观感知来说，如果采用一种优化的数据呈现策略，当少部分数据返回之后就立刻将数据呈现在用户面前，则用户感受到的响应时间就会远远小于实际的事务响应时间(顺便说一下，这种技巧是在C/S结构的管理系统中开发人员常用的一种技巧)。

对于用户来说，感受到的响应时间还包含了主观的反应时间。

### 管理员视角的软件性能

- 从管理员的角度来看，软件系统的性能首先表现在系统的响应时间上，这一点和用户视角是一样的。但管理员是一种特殊的用户，和一般用户相比，除了会关注一般用户的体验之外，他还会关心和系统状态相关的信息。
- 另一方面，管理员还会想要知道系统具有多大的可扩展性，处理并发的能力如何；而且，管理员还会希望知道系统可能的最大容量是什么，系统可能的性能瓶颈在哪里，通过更换哪些设备或是进行哪些扩展能够提高系统性能，了解这些情况，管理员才能根据系统的用户状况制定管理措施，在系统出现计划之外的用户增长等紧急情况的时候能够立即制定相应措施，进行迅速的处理；此外，管理员可能还会关心系统在长时间的运行中是否足够稳定，是否能够不间断地提供业务服务等。
- 因此，从管理员的视角来看，软件性能绝对不仅仅是应用的响应时间这么一个简单的问题。

开发者实现要保证软件的可用性，然后再考虑并发提升了以后软件能否正常使用。接下来我们再考虑怎么样减少系统的响应时间。

### 开发者视角的软件性能

- 从开发人员的角度来说，对软件性能的关注就更加深入了。对开发人员来说，其最想知道的是“如何通过调整设计和代码实现，或是如何通过调整系统设置等方法提高软件的性能表现”，和如何发现并解决软件设计和开发过程中产生的由于多用户访问引起的缺陷”。
- 因此，其最关注的是使性能表现不佳的因素和由于大量用户访问引发的软件故障，也就是我们通常所说的“性能瓶颈”和系统中存在的在大量用户访问时表现出来的缺陷。



## 2 LoadRunner 简介

- **HP (Mercury) LoadRunner®** 是一种预测系统行为和性能的工业级标准性能测试负载测试工具。通过以模拟上千万用户实施并发负载及实时性能监测的方式来确认和查找问题，LoadRunner 能够对整个企业架构进行测试。通过使用 LoadRunner，企业能最大限度地缩短测试时间，优化性能和加速应用系统的发布周期。
- 目前企业的网络应用环境都必须支持大量用户，网络体系架构中含各类应用环境且由不同供应商提供软件和硬件产品。难以预知的用户负载和愈来愈复杂的应用环境使公司时时担心会发生用户响应速度过慢，系统崩溃等问题。这些都不可避免地导致公司收益的损失。
- **Mercury Interactive 的 LoadRunner** 能让企业保护自己的收入来源，无需购置额外硬件而最大限度地利用现有的IT 资源，并确保终端用户在应用系统的各个环节中对其测试应用的质量，可靠性和可扩展性都有良好的评价。
- **LoadRunner** 是一种适用于各种体系架构的负载测试工具，它能预测系统行为并优化系统性能。LoadRunner 的测试对象是整个企业的系统，它通过模拟实际用户的操作行为和实行实时性能监测，来帮助您更快的查找和发现问题。此外，LoadRunner 能支持广泛的协议和技术，为您的特殊环境提供特殊的解决方案。

软件学院

主要就是模拟上千万个用户并发负载来监控软件的性能。接下来我们介绍 Load Runner

的一些组件：



## 2.1 LoadRunner 的组件

- ④ **Virtual User Generator:** 录制最终用户业务流程并创建自动化性能测试脚本，即 **Vuser** 脚本。
- ④ **Controller:** 组织、驱动、管理并监控负载测试。
- ④ **Analysis:** 用于查看、剖析和比较性能结果。

Virtual User Generator 就是通过录制的方式把用户流程录制下来创建虚拟用户脚本。



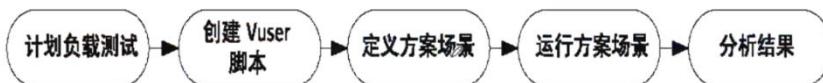
## 2.2 LoadRunner 术语

- ④ **场景 (Scenario)**：根据性能要求定义每次测试期间发生的事件。
- ④ **虚拟用户 (Vuser)**：在场景中，LoadRunner 用虚拟用户 (Vuser) 代替真实用户。Vuser 模仿真实用户的操作来使用应用系统。一个场景可以包含数十、数百乃至数千个 Vuser。
- ④ **Vuser 脚本 (Scripts)**：描述 Vuser 在场景中执行的操作。
- ④ **事务 (Transaction)**：要评测服务器性能，需要定义事务。事务代表要评测的终端用户业务流程。

我们要定义要测的是什么，定义它的性能测试要求。接下来确定了测试目标之后，我们就创建 Vuser 脚本。接下来是定义一定的场景来设置负载测试场景。接下来就是运行场景，并且分析结果。



## 2.3 LoadRunner 测试流程



- **计划负载测试：** 定义性能测试要求，例如并发用户的数量、典型业务流程和所需响应时间。
- **创建 Vuser 脚本：** 将最终用户活动捕获到自动脚本中。
- **定义场景：** 使用 *LoadRunner Controller* 设置负载测试环境。
- **运行场景：** 通过 *LoadRunner Controller* 驱动、管理和监控负载测试。
- **分析结果：** 使用 *LoadRunner Analysis* 创建图和报告并评估性能。

接下来我们看一个具体的例子：



## 实例：HP Web Tours

1 确保示例 Web 服务器正在运行

2 打开 HP Web Tours 应用程序

<http://localhost:1080/WebTours/>

应用程序性能要求：

1. 必须能够成功处理 10 家旅行社的并发操作。
2. 必须能够处理 10 个并发的机票预订操作，且响应时间不能超过 90 秒。
3. 必须能够处理 10 家旅行社的并发航班路线查看操作，且响应时间不能超过 120 秒。
4. 必须能够处理 10 家旅行社的并发登录和注销操作，且响应时间不能超过 10 秒。



## 4 Virtual User Generator

- ④ LoadRunner Virtual User Generator (VuGen) 以“录制-回放”的方式工作。当您在应用程序中执行业务流程步骤时，VuGen 会将您的操作录制到自动化脚本中，并将其作为负载测试的基础。



### 4.1 创建脚本

1 启动 LoadRunner

2 打开 VuGen

3 创建一个空白 Web 脚本

- “新建单协议脚本”选项
- 请确保“类别”是所有协议
- 选择 Web (HTTP/HTML) 并单击创建

4 使用任务向导

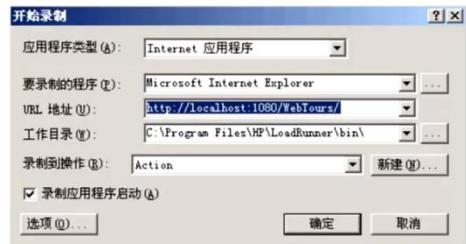
- 录制脚本
- 回放脚本



大家也可以测一些支持并发数量比较高的网页，比如百度、淘宝的页面。Load Runner 只能在 Windows 上操作。

## ④ 录制脚本

- a 单击步骤 1 的“任务”窗格中的录制应用程序。  
b 在说明窗格底部，单击开始录制。



- c 在 URL 地址框中，输入 <http://localhost:1080/WebTours>。在录制到操作框中，选择 Action。单击确定。

这时将打开浮动的“正在录制”工具栏



## 完成应用的操作

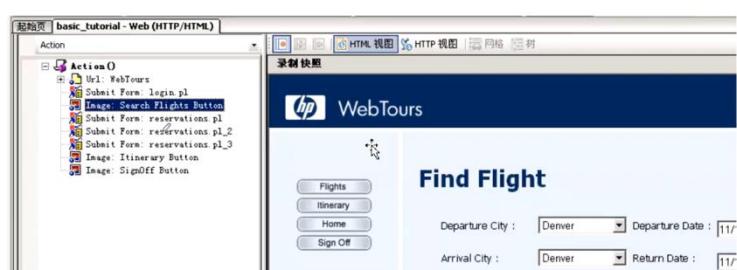
- 1 登录到 HP Web Tours 网站。
- 2 输入航班详细信息
- 3 选择航班
- 4 输入支付信息并预订机票
- 5 查看航班路线
- 6 注销 (Sign Off)

## d 停止录制

## e 保存脚本

### ⑤ 查看脚本

- 现在，您可以查看在 VuGen 中录制的脚本。可以在树视图或脚本视图中查看脚本。树视图是基于图标的视图，其中将 Vuser 的操作作为步骤列出；而脚本视图是基于文本的视图，其中将 Vuser 的操作作为函数列出。
- 树视图
- 要在树视图中查看脚本，请选择“查看”>“树视图”或单击“树视图”按钮。对于录制期间执行的每个步骤，VuGen 都在测试树中生成了一个图标和一个标题。



## - 脚本视图

- 脚本视图是基于文本的视图，其中将 Vuser 的操作作为 API 函数列出。要在脚本视图中查看脚本，请选择“查看”>“脚本视图”或单击“脚本视图”按钮。



```
起始页 basic_tutorial - Web (HTTP/HTML)
vuser_init
Action()
web_url("WebTours",
    "URL=http://localhost:1080/WebTours",
    "Resource=0",
    "RecContentType=text/html",
    "Referer=",
    "Snapshot=t3.inf",
    "Mode=HTML",
    LAST);
lr_think_time(12);
web_submit_form("login.pl",
    "Snapshot=t4.inf",
    ITEMDATA,
    "Name=username", "Value=johoj", ENDITEM,
    "Name=password", "Value=bear", ENDITEM,
```

### ④ 回放脚本

#### 1. 打开运行时设置对话框

- 运行逻辑
- 步
- 思考时间
- 日志

#### 2. 运行时查看设置

- 工具 > 常规选项，  
然后选择显示选项卡。

选择回放期间显示运行时  
查看器和自动排列窗口选项。

#### 3. 验证回放



## 4.2 准备脚本

### ④ 创建事物

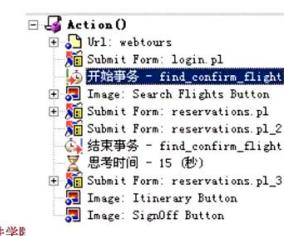
- 在“任务”窗格的增强功能下单击**事务**。单击**新建事务**。将打开事务创建向导。

#### 2. 插入事务开始标记和事务结束标记

(Search flights button~reservations.pl\_2)

#### 3. 指定事务名称: **find\_confirm\_flight**

#### 4. 在树视图中观察事务





## 如何模拟多个用户

- ④ **参数化：**不同用户可以提交不同的数据 (e.g., seatPref)

### 1. 找到要更改数据的部分

在测试树中双击 **Submit Data: reservations.pl** 步骤。将打开“提交数据步骤属性”对话框。

### 2. 将常量值更改为变量值、创建参数

选择第七行中的 **seatPref**。单击 **Aisle** 旁边的 **ABC** 图标。打开“选择或创建参数”对话框，修改参数名。

### 3. 指定示例值来更改数据

单击添加行。VuGen 将向表中添加行

### 4. 定义测试更改数据的方式



## 如何验证网页内容

- ④ **文本检查：**检查特殊文本字符串是否出现在 Web 页面上

### 1. 在“任务”的增强功能下单击**内容检查**

选择工具栏中的 **HTML** 视图以显示缩略图的快照

### 2. 选择包含待检查文本的页面、要检查的文本

3. 选择要检查的文字 (**Find Flight**)，然后右键单击并选择添加文本检查 (**web-reg-find**)

### 4. 查看新步骤

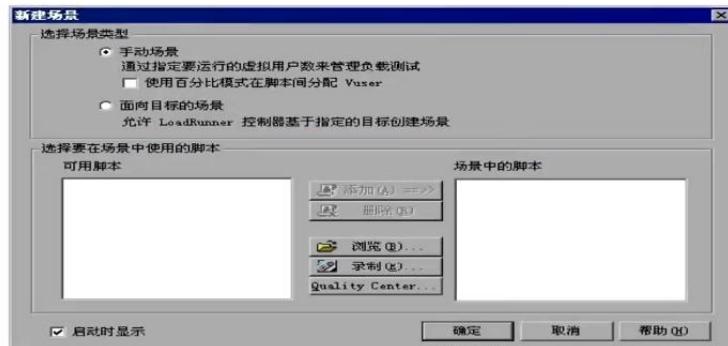
在脚本中插入了一个新步骤：**Service: Reg Find**



## 5 Controller

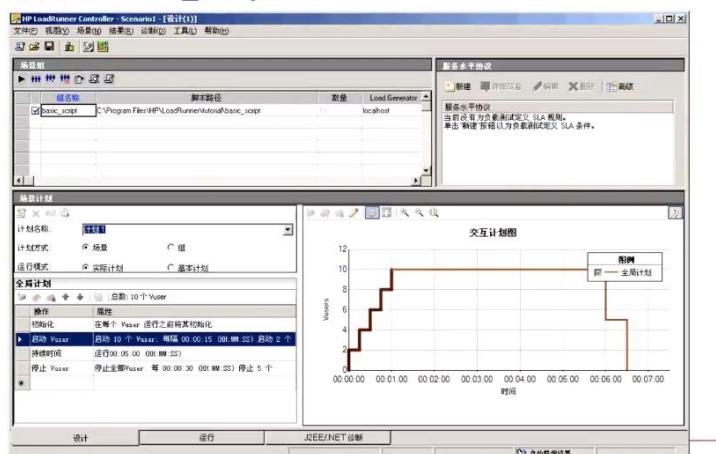
负载测试指在典型的工作条件下测试应用程序，在我们的案例中，典型的工作条件即许多旅行代理同时在相同的航班预订系统中预订航班。在前面，您已完成了第一步—创建脚本，接下来需要设置负载测试环境。

使用 **Controller**，可以将应用程序性能测试需求划分为多个场景。场景定义每个测试会话中发生的事件。例如，一个场景可以定义和控制模拟的用户数、用户执行的操作以及用户运行其模拟时所用的计算机。



### 5.1 创建负载测试场景

① 单击**浏览**按钮找到<LoadRunner 安装位置>\Tutorial 目录中的 basic\_script



② 在系统上生成负载：Load Generator

a)添加 Load Generator

在设计选项卡中，单击 **Load Generator** 按钮

b) 测试 Load Generator 连接

Name: localhost

单击**连接**，状态会从关闭变为就绪。



## ② 在系统上生成负载：Load Generator

单击 Controller 工具栏上的 **Load Generator** 按钮 。Load Generator 对话框打开。



Controller 会尝试连接到 Load Generator 计算机。建立连接后，Load Generator 的状态会从关闭变为就绪。



## ③ 模拟负载行为：配置加压方式

### a) 选择计划类型和运行模式

在**计划定义**区域，确保选中**计划方式：场景**  
和**运行模式：实际计划**



加压方式还要考虑到真实情况，比如用户不会同时登录和退出。包括场景的持续时间和终止方式可能都有所区别。



## ③ 模拟负载行为：配置加压方式

### b) 设置计划操作定义

- 设置 Vuser 初始化：**同时或间隔一定时间**
- 间隔启动 Vuser：每 **XX:XX:XX** 启动 **？** 个 Vuser
- 安排持续时间：
- 安排逐渐关闭：每 **XX:XX:XX** 停止 **？** 个 Vuser





## ④ 模拟不同用户：运行时设置

- ④ **运行逻辑：**用户重复一系列操作的次数。
  - ④ **步：**重复操作之前等待的时间。
  - ④ **日志：**希望在测试期间收集的信息的级别。
- 如果是首次运行场景，建议生成日志消息，确保万一首次运行失败时有调试信息。
- ④ **思考时间：**用户在各步骤之间停下来思考的时间。
  - ④ **速度模拟：**使用不同网络连接（例如调制解调器、DSL 和电缆）的用户。
  - ④ **浏览器模拟：**使用不同浏览器查看应用程序性能的用户
  - ④ **内容检查：**用于自动检测用户定义的错误。

不同的用户可能有不同的参数。

### a) 启用思考时间

选择**使用录制思考时间的随机百分比**选项。指定最小值为 50%，最大值为 150%。

### b) 启用日志记录

- ④ 选择**始终发送消息**。
- ④ 选择**扩展日志**，然后选择  
**服务器返回的数据**



## ⑤ 监控负载下的系统：设置监控器

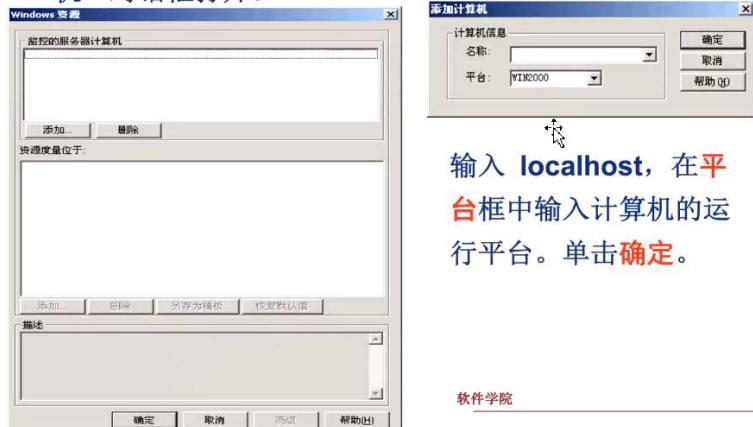
①~④已经定义了 Vuser 在测试期间的行为方式，接下来就可以设置监控器了。

选择 Windows 资源监控器

单击 Controller 窗口中的**运行**选项卡打开“运行”视图



- ④ 右键单击“Windows 资源”图并选择添加度量。  
“Windows 资源”对话框打开，单击添加。“添加计算机”对话框打开。



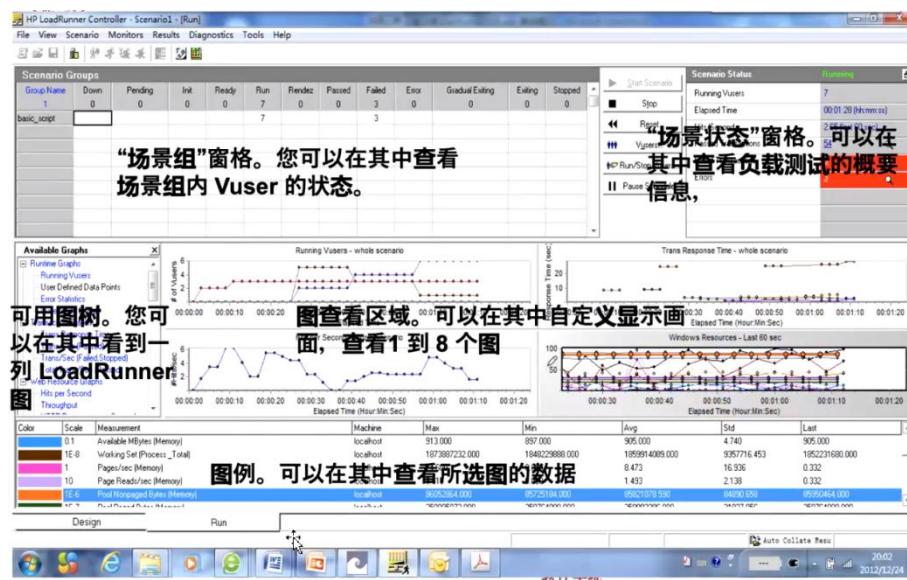
输入 localhost，在平台框中输入计算机的运行平台。单击确定。



## 5.2 运行负载测试

- ④ 打开 Controller 的“运行”视图；  
⑤ 单击开始场景按钮，或者选择场景 > 开始以开始运行测试；

可以看到如下界面





## 监控负载下的应用程序

### 1. 检查性能图

- ④ “正在运行 Vuser – 整个场景” 图。显示在指定时间运行的 Vuser 数。
- ④ “事务响应时间 – 整个场景” 图。显示完成每个事务所用的时间。
- ④ “每秒点击次数 – 整个场景” 图。显示场景运行期间 Vuser 每秒向 Web 服务器提交的点击次数（HTTP 请求数）。
- ④ “Windows 资源” 图。显示场景运行期间评测的 Windows 资源。

### 2. 突出显示单个测量值



## 实时观察 Vuser 的运行情况

### 3. 单击 Vuser 按钮。这时将打开 Vuser 窗口

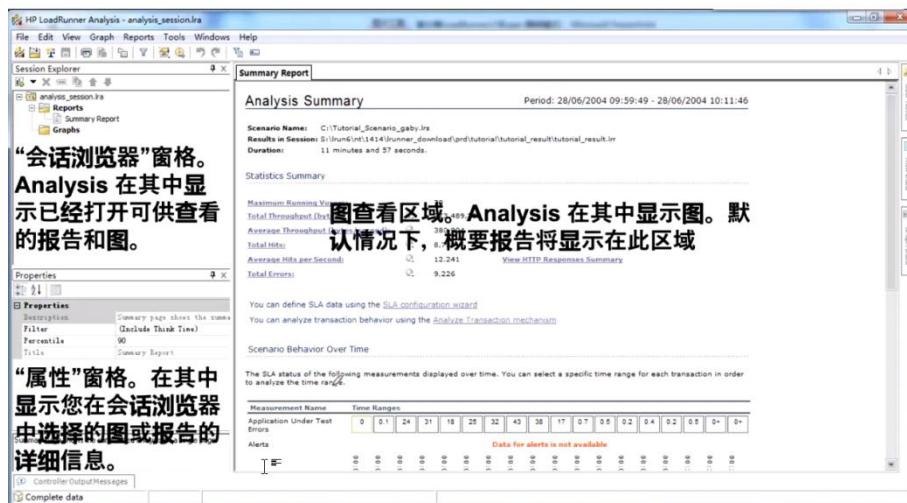
ID	状态	脚本	Load Generator	已用时间
1	正在运行	demo_script	localhost	00:00:02
2	正在运行	demo_script	localhost	00:00:02
3	正在运行	demo_script	localhost	
4	正在运行	demo_script	localhost	
5+	正在初始化	demo_script	localhost	
6+	正在初始化	demo_script	localhost	
7	关闭	demo_script	localhost	
8	关闭	demo_script	localhost	
9	关闭	demo_script	localhost	
10	关闭	demo_script	localhost	



## 测试期间增加负载

- ④ 在“运行”视图中单击运行/停止 Vuser 按钮。“运行/停止”对话框打开，显示当前分配到场景中运行的 Vuser 数。
- ④ 在 # 列中，输入要添加到组中额外的 Vuser 的数目。要运行 2 个额外的 Vuser，请将 # 列中的替换为 2。

组名称	#	Load Generator
travel_agent	2	localhost



**“会话浏览器”窗格。**  
**Analysis 在其中显示已经打开可供查看的报告和图。**

**“属性”窗格。**在其中显示您在会话浏览器中选择的图或报告的详细信息。

④ Complete data



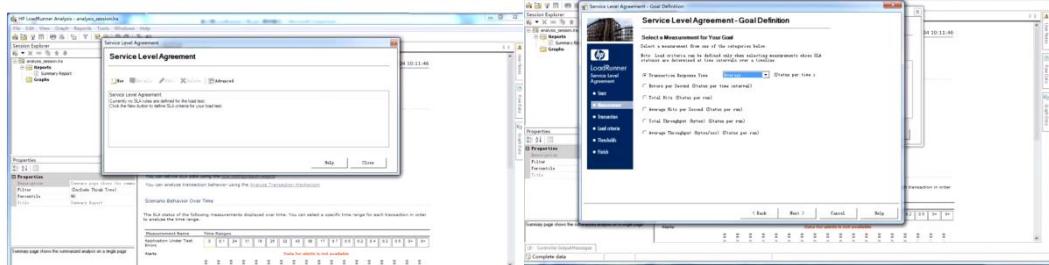
## 服务水平协议

- ⑤ **SLA (service level agreement 服务水平协议)**是您为负载测试场景定义的具体目标。**Analysis** 将这些目标与**LoadRunner**在运行过程中收集和存储的性能相关数据进行比较，然后确定目标的 **SLA 状态**（通过或失败）。
- ⑥ 作为目标定义的一部分，您可以指示 **SLA** 将负载条件考虑在内。这意味着可接受的阈值将根据负载级别（例如，运行的 **Vuser** 数、吞吐量等）而有所更改。随着负载的增加，您可以允许更大的阈值。
- ⑦ **SLA状态确定方式：**通过时间线中的时间间隔确定 SLA 状态；通过整个运行确定 SLA 状态



### 6.1 定义SLA

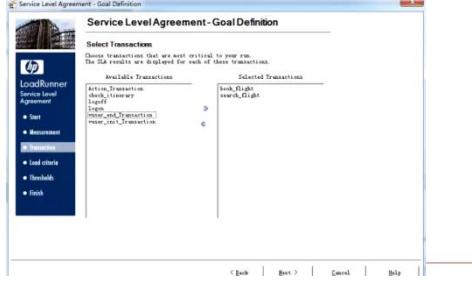
- ① 打开 **SLA 配置向导**。
- ② 选择工具 > 配置 **SLA 规则**。“服务水平协议”对话框打开。单击新建打开向导。



- ③ 在“选择目标度量”页面中选择**事务响应时间：平均值**。单击 **Next**

### ③ 选择事务进行监控

- ④ 从可用事务列表（脚本中的所有事务列表）中选择要监控的事务。双击 **book\_flight** 和 **search\_flight** 事务将它们选中。单击 **Next >**



### ④ 设置加载条件

在“设置加载条件”页面，可以指示 SLA 将不同的加载条件考虑在内。

#### ⑤ 轻负载

**0 ~ 19 个 Vuser**

#### ⑥ 平均负载

**20~49 个 Vuser**

#### ⑦ 重负载

**超过 50 个 Vuser**



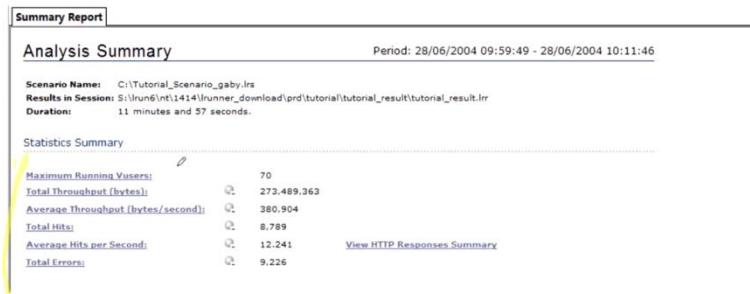
### ⑤ 设置阈值

- ⑥ 保存  
依次单击  
**Next >**、  
**Finish**、  
**Close**。



## 6.2 查看性能报告

### • 场景的总体统计信息



### • 5 个最差事务表

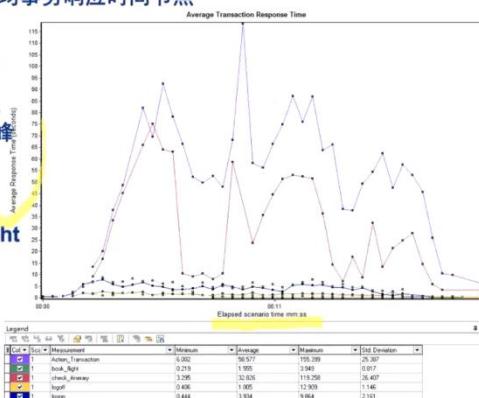
- 超出 SLA 阈值的时间间隔
- 事务的整体性能

## ② 以图形方式查看性能

- ④ 打开“平均事务响应时间”图,请右键单击图节点并在“打开新图”对话框中选择事务: 平均事务响应时间节点

注意 **check\_itinerary** 事务的平均响应时间波动很大,甚至在场景运行 2:56 分后峰值达到 75.067 秒。

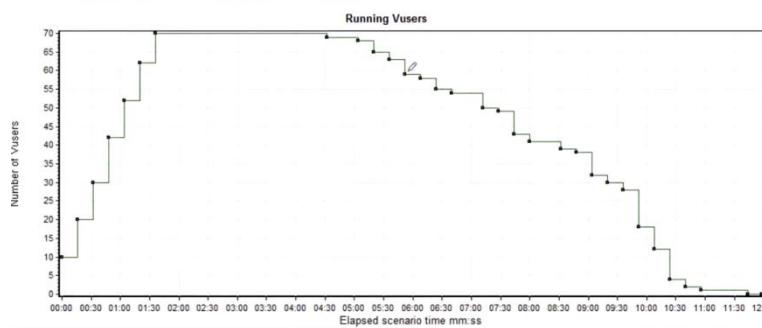
**logon**、**logoff**、**book\_flight** 和 **search\_flight** 事务的平均响应时间相对稳定



## 服务器的性能是否稳定

### ③ 研究 Vuser 的行为

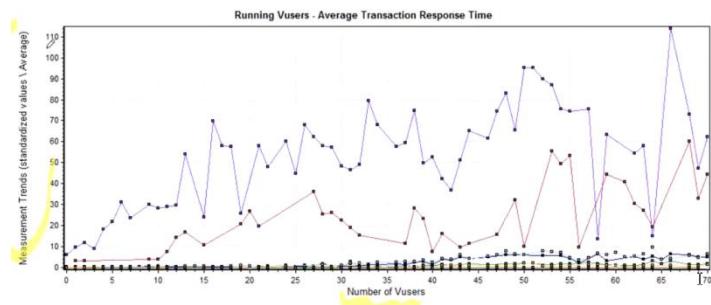
- 在图树中单击运行 Vuser



### ④ 筛选条件

- ④ 在“运行 Vuser”图, 并选择清除筛选器/分组方式
- ④ elapsed time 项: 选 values 设置从 000:01:30 到 000:05:45 的时间范围)
- ④ 将“运行 Vuser”图和“平均事务响应时间”图关联在一起比较数据
- ④ 右键单击该图并选择合并图。
- ④ 在选择要合并的图列表中, 选择平均事务响应时间。
- ④ 在选择合并类型区域中, 选择关联, 然后单击确定。

## ④ 分析关联后的图



- 可以看到随着 Vuser 数目的增加, `check_itinerary` 事务的平均响应时间也在逐渐延长。
- 运行 64 个 Vuser 时, 平均响应时间会突然急剧拉长。我们称之为测试弄崩了服务器。

软件学院



## 6.3 发布结果

- 从报告菜单中选择 HTML 报告....。
- 为报告选择文件名和保存路径。单击保存。



## 可能遇到的问题

### 1. 无法执行应用实例

原因：网络服务器未启动

### 2. 无法录制脚本

原因：防火墙、360浏览器、第三方插件，其它未知网络软件屏蔽

### 3. 无快照图

原因：软件授权过期等原因



1. [https://admhelp.microfocus.com/lr/en/2020/help/WebHelp/Content/Resources/\\_TopNav/\\_TopNav\\_Home.htm](https://admhelp.microfocus.com/lr/en/2020/help/WebHelp/Content/Resources/_TopNav/_TopNav_Home.htm)
2. [https://softwaresupport.softwaregrp.com/web/softwaresupport/document/-/facetsearch/attachment/KM02004789?fileName=LR\\_12.50\\_Tutorial\\_zh.pdf](https://softwaresupport.softwaregrp.com/web/softwaresupport/document/-/facetsearch/attachment/KM02004789?fileName=LR_12.50_Tutorial_zh.pdf)

后面两次作业主要是测试工具这一部分占比比较大。

## 系统测试的覆盖率指标

我们继续讲课程内容。系统测试最终大部分测试的依据都来源于需求规格要求。系统测试也有一些覆盖率的概念，去说明这个测试是否足够。

我们上节课画了一些状态图，系统功能这块包含了事件、端口和数据。我们可以通过这些的定义去找一下测试的线索。

从端口的输入事件考虑，也可以有覆盖率的指标。比如我们系统里有很多端口，这些端口的输入事件都发生也是一个指标。典型序列可以对应到很多用例上。

上海交通大学  
HANGZHOU JIAO TONG UNIVERSITY

### 8.3.5 基于规格说明系统测试的覆盖率

基于状态图的测试方法很有效，但并不是所有的软件系统都能够方便的得到状态图。而系统功能确是各个系统都十分明确的线索，下面我们就从功能定义的三个基本构件：事件、端口和数据，讨论测试的线索。

#### 1) 基于事件的线索测试

从端口的输入事件考虑，有5个覆盖标准：

- PI1：每个端口输入事件发生。
  - PI2：端口输入事件的常见序列发生。
  - PI3：每个端口输入事件在所有“相关”数据语境中发生。
  - PI4：对于给定语境，所有“不合适”的输入事件发生。
  - PI5：对于给定语境，所有可能的输入事件发生。
- PI1是易于达到的，PI2是基本可行的。而PI3就有可能形成测试爆炸，而PI4和PI5往往是供参考的选项。

测试爆炸就是测试用例过多，形成测试用例维度的爆炸。

从端口的输出事件考虑，可以有两种覆盖指标：

- PO1：每个端口输出事件发生。
- PO2：每个端口输出事件在每种情况下发生。
- PO1是基本的要求，PO2不仅要求所有的输出事件，而且要考虑导致这种输出的所有可能原因，这个要求往往也是难以完全满足的。

我们还可以基于端口来进行测试。

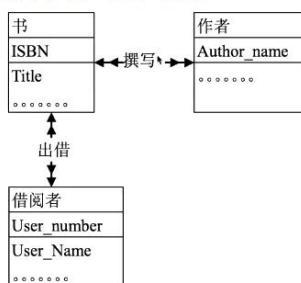
### 2) 基于端口的线索测试

- 基于端口的线索测试是基于事件的测试的有用补充。
- 基于事件的测试以事件为中心，考虑的是事件到端口的一对多测试；
- 基于端口的测试以端口设备为中心，考虑的是端口到事件的一对多测试。

### 3) 基于数据的线索测试

- 基于端口和事件的测试适合主要以事件驱动的系统，但并不是所有的系统都是事件驱动系统，例如以数据库为基础的系统，主要是数据的操作，此时就可以采用基于数据的测试。

以E-R模型作为测试的线索。



覆盖指标：

- DM1：检查每个关系的基数。
  - DM2：检查每个关系的参与。
  - DM3：检查关系之间的函数依赖关系。
- 基数指：关系间的一对一、一对多、多对一、多对多关系。

## SATM 实例（由学生阅读）

系统测试的指导方针，一般常用的方式是使用一些模型来描述系统的行为。表示的

形态有很多种，比如决策表、状态图、Petri网等。代表了系统运行的时候的逻辑关系。

## 8.3.6 系统测试指导方针

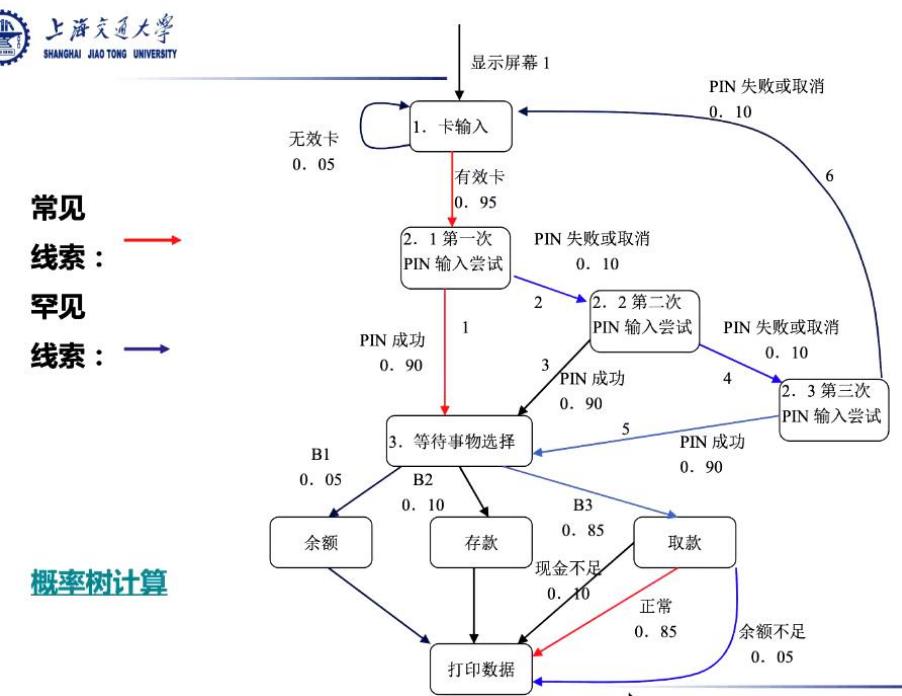
### ④ 伪结构系统测试

- 伪结构是系统的“行为模型”，是系统实际情况的近似模型。
- 决策表、状态图、Petri网是系统功能性测试的常见选择。

### ⑤ 运行剖面

- 齐夫定理（Zipf's Law）在大多数情况下都成立。即80%的活动发生在20%的空间里（二八定律）。
- 因此，确定各种线索的执行频率，对事件发生频率高的线索执行测试，可以大大提高测试的效率。

系统里有很多执行路径，这些路径发生的概率是不一样的。比如大部分用户到ATM机基本上就是取钱和查询余额。取钱大部分的情况下都是“有钱、取钱成功”的概率比较大。



- 运行剖面提供了系统的使用情况，对优化系统测试有意义。
- 更进一步的，对客户使用情况的了解，是改进系统设计的重要依据。对于发生概率极小甚至为零的线索，也许恰恰占用了系统的极大资源！

$$\text{风险} = \text{代价} * \text{发生概率}$$

- 失效代价通过四种风险类别给定：如1代表失效代价低、3为中间值、10为代价高
- 表14-14给出了SATM系统的风险评估结果

**2022/4/25**

这节课我们开始讲回归测试，回归测试的很多内容是我们新加的，有些 PPT 上的内容在我们课本上是没有的。因为课本上讲回归测试比较简略，所以我们课程上会做一些扩展。回归测试在整个软件生命周期里比较重要的。我们根据回归测试的概念找了一些基本的测试方法给大家介绍，但是没有对应到我们课本上的内容。

## 回归测试

回归测试是一个专题内容，它在整个软件生命周期里是很特别的一种方法，我们将从以下几个要点来讲：



### 要点

1. 回归测试的概念
2. 回归测试策略
3. 回归测试的步骤
4. 回归测试用例选择方法

每个技术的产生都是为了解决一些问题，回归测试也是这样的。这个其实是对应到我们整个软件周期的开发过程的。软件工程的开发是遵循一定的工程和开发模式来做的。大家比较熟悉的是迭代开发，这是绝大多数软件的开发模式。迭代过程中会有软件版本的更新，会给测试带来新的问题。因为开发和测试是有对应关系的，如果软件本身发生了修改，那么我们软件测试就需要适应软件开发周期里解决迭代开发所带来的问题。



## 1 回归测试的概念

在软件生命周期中的任何一个阶段，只要软件发生了修改和改变，就可能给该软件带来问题。

- 对问题的修改被遗漏；
- 所做的修改只修正了错误的外在表现；
- 导致软件未被修改的部分产生出新的问题。

我们经常讲如何以最小的代价完成测试的目的。我们会针对软件开发里面有一些问题的修改是否遗漏等。所以回归测试就是在软件整个测试迭代开发过程中，必须要面临的一个问题。回归测试的重点就是考虑到软件生命周期里会迭代开发软件，不同的时期和阶段有不同的版本，所以我们需要有特定的策略。

因此，每当软件发生变化时，我们就必须重新测试现有的功能，以便确定软件修改是否达到了预期的目的，检查修改是否损害了原有的正常功能。同时，还需要补充新的测试用例来测试新的或被修改了的功能。为了验证修改的正确性及其影响，就需要进行回归测试。

我们通过一个简单的模型实例来讲一下软件回归测试里的关键点。比如一个软件有两个版本 v1 和 v2。v1 版本通过了我们所有的测试，发布出去了。发布出去以后，我们后续会有新的版本更新，对应了我们一些功能的修改和增加，形成了我们新的软件。所以 v1 版本是 v2 版本的基础，那么我们测试怎么去做呢？

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 $P'$

图 2 回归测试策略

另外的话，最终我们会形成 2 个测试。1 是增加测试用例去完成新的功能，我们软件版本增加新的功能，自然要增加一些新的测试用例。如果说我们大部分功能是从原来的版本来的，我们原来的版本实际上已经进行了一些完整的测试，这些完整的测试是否还是要做呢？我们原先的功能是否要重新进行测试呢？重新测试带来的问题是，我们需要把原来的测试重新跑一遍，代价是很高的。所以，回归测试里也会面临“如果使用最小的测试代价（用例），去完成测试完整性的工作”。所以这就会有回归测试的测试策略的选择。

## Test-All 方法

这样的话，我们就会面临选择策略的问题。我们总是想用最简单的方法去解决问题，一般来讲，大家会通过一些直接的思考模式去解决问题：

我们原来已经做了一些测试，并且对于原先版本已经有了一些测试用例的产生，是不是我们可以把原来的版本的测试用例做新的版本的测试呢？



### What tests to use?

#### › Idea 1:

› **All valid tests from the previous version and new tests created to test any added functionality. [This is the TEST-ALL approach.]**

› **What are the strengths and shortcomings of this approach?**

这是一个最简单的思路，有一些优缺点：

1. 优点：避免我们分析要做哪些测试要做、哪些不要做
2. 缺点：这样做的代价很高，对于大型软件整个测试用例有很多，测试的人力成本和时间成本也比较高。如果我们采用 TEST-ALL 方法，开销就比较大。对于一些测试用例不多的小型软件，还是可以使用这种方法的。



### The test-all approach

› **The test-all approach is best when you want to be certain that the new version works on all tests developed for the previous version and any new tests.**

› **But what if you have limited resources to run tests and have to meet a deadline? What if running all tests as well as meeting the deadline is simply not possible?**

TEST-ALL 方法的最大问题就是测试开销的问题。

有了这种直接的思考方法之后，我们会有另外一种选择策略。如果把所有的测试用例拿过来代价很高，那么我们是不是能选择一部分测试用例进行相关的测试。原先版本中我们的

测试用例集  $T$ ，我们选择里面的一个子集  $T_r$  进行小部分的测试，从而说明我们新的修改在原来版本的测试上是足够的达到回归测试的目标的。



## Test selection

### ➤ Idea 2:

➤ **Select a subset  $T_r$  of the original test set  $T$  such that successful execution of the modified code  $P'$  against  $T_r$  implies that all the functionality carried over from the original code  $P$  to  $P'$  is intact.**

➤ **Finding  $T_r$  can be done using several methods. We will discuss three of these.**

这样，我们就会面临这个子集怎么选取的问题。选一部分测试子集，并且我们还要说明这部分的测试子集是能够说明我们对软件修改的测试是足够的。我们先来看一下测试的用例集会有不同的特性，我们对原来的测试集  $T$  分成三部分： $T_o$ （废弃的）、 $T_u$ （冗余的）和  $T_r$ （有效的）。

废弃的指的是，软件修改了以后，有一些测试用例集对于我们新的版本已经不产生任何作用了。

冗余的指的是，软件测试用例有冗余的特性。

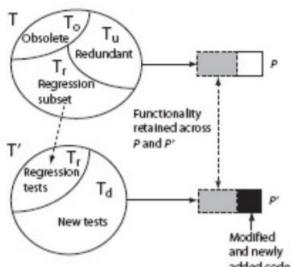
有效的指的是，这一部分是回归测试所关注的测试用例。我们可以把它放到我们新的版本的测试用例中。在新版本的测试集  $T'$  中，除了我们从  $T$  中拿来的  $T_r$ ，还有新的策略用例

集合  $T_d$ 。这是针对新的功能修改所增加的测试用例。

回归测试的核心就是在  $T_r$  这里，我们需要通过一些方法在原来的测试集  $T$  中把  $T_r$  找到。找到了以后，我们还需要去说明这样选取子集能够达到测试的完备性要求，并且我们还希望这个子集是尽可能小的。



## Regression Test Selection problem



Given test set  $T$ , our goal is to determine  $T_r$  such that successful execution of  $P'$  against  $T_r$  implies that modified or newly added code in  $P'$  has not broken the code carried over from  $P$ .

Note that some tests might become obsolete when  $P$  is modified to  $P'$ . Such tests are not included in the regression subset  $T_r$ .

在这个之前, 我们提一些概念。在软件开发整个生命周期里都回去维护一些测试用例库。测试用例库分为很多种, 比如有基线测试用例库。软件基线版本 v1.0 可能会对应一个基线测试用例库。



## 概念

- ④ 测试用例库：对于一个软件开发项目来说，项目的测试组在实施测试的过程中，会将所开发的测试用例保存到“测试用例库”中，并对其进行维护和管理。
- ④ 基线测试用例库：当得到一个软件的基线版本时，用于基线版本测试的所有测试用例就形成了基线测试用例库

有了测试用例库之后，我们也会对它进行相应的维护操作。比如我们会删除过时的、改进不受控制的、删除冗余的、添加新的。

### ④ 测试用例库的维护

— 随着软件的升级，软件的功能和应用接口以及软件的实现都可能发生了演变，测试用例库中的一些测试用例就可能会失去针对性和有效性，而另一些测试用例可能会变得过时，甚至完全不能运行。为了保证测试用例库中测试用例的有效性，必须对测试用例库进行维护。

- 删除过时的测试用例
- 改进不受控制的测试用例
- 删除冗余的测试用例
- 增添新的测试用例

测试方法考虑主要要考虑效率（用最小测试用例去做完备测试）和有效性。我们就会有一些测试用例的选择策略。基于风险的选择，我们讲过软件路径有一些风险比较高。操作剖面就比如 ATM 中取钱功能是比较频繁使用到的用例，我们也要频繁地做测试。



## 回归测试包的选择

选择回归测试策略应该兼顾效率和有效性两个方面。常用的选择回归测试的方式包括：

- **再测试全部用例**

再测试全部用例具有最低的遗漏回归错误的风险，但是测试成本最高。

- **基于风险选择测试**

首先运行最重要的、关键的和可疑的测试，逐步降低风险值，直至满足回归测试要求。

- **基于操作剖面选择测试**

测试用例是基于软件操作剖面开发的，优先选择那些针对最重要或最频繁使用功能的测试用例。

- **再测试修改的部分**

当测试者对修改的局部有足够的信心时，可以通过等价性分析，识别软件的修改情况并分析修改的影响，将回归测试局限于被改变的模块和它的接口上。

如果我们要做真正去回归测试，我们应该做哪些方法的选择？回归测试里我们要识别出哪些软件的部分是修改了。接下来我们要从基线测试用例库中选取出一部分做回归测试。还要生成一部分新的测试用例放到测试用例库去。

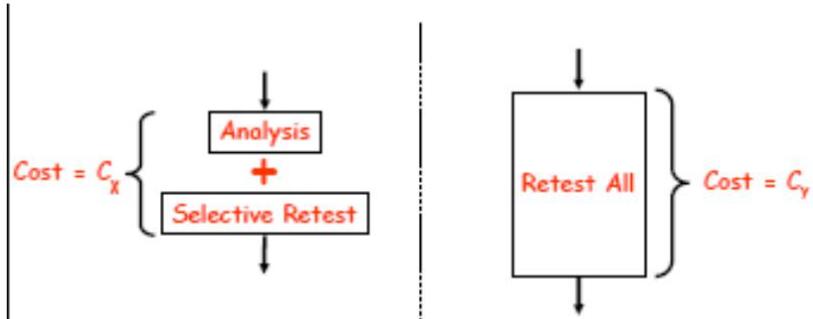


## 3 回归测试的步骤

回归测试可遵循下述基本过程进行：

1. 识别出软件中被修改的部分。
2. 从原基线测试用例库 $T$ 中，排除所有不再适用的测试用例，确定那些对新的软件版本依然有效的测试用例，其结果是建立一个新的基线测试用例库 $T_0$ 。
3. 依据一定的策略从 $T_0$ 中选择测试用例测试被修改的软件。
4. 如果必要，生成新的测试用例集 $T_1$ ，用于测试 $T_0$ 无法充分测试的软件部分。
5. 用 $T_1$ 执行修改后的软件。
6. 第(2)和第(3)步测试，验证修改是否破坏了现有的功能，第(4)和第(5)步测试验证修改工作本身。

讲到回归测试的话，我们希望总的代价比较小（小于 Test-All）。



We want  $C_x < C_y$   
Key is the test selection algorithm/technique  
We want to maintain the same “quality of testing”

我们接下来会讲几个回归测试用例选择的方法。



## 4 回归测试用例选择方法

- **Test selection using execution trace and execution slice**
- **Test selection using test minimization**
- **Test selection using test prioritization**

### Test selection using execution trace and execution slice

我们先看第一个方法：通过分析执行轨迹（执行的程序片）去进行测试的选择。我们来看它的步骤：

首先，回归测试是一个白盒测试方法，我们需要对程序源代码进行分析。程序  $P$  有一个测试用例集  $T$ 。我们首先要构建程序的模型，我们要找到程序  $P$  的执行轨迹。一般来说，对于每个测试都有一个执行轨迹，我们要把这些执行轨迹找出来。

所以我们第一步是要把原来的程序和原来测试用例集进行执行轨迹的分析，我们是针对每一个测试用例进行执行轨迹的分析。

第二步是从执行轨迹中找到测试向量（test vector）。测试向量其实就是把原来的执行轨迹变成一个向量化的表示，执行轨迹对我们程序流图的每一个节点有什么样的关系。

有了这么一个测试向量以后，我们要分析新的程序  $P'$  在句法树上是哪里进行了修改。我们对修改的部分进行相应的测试。

# Overview of a test selection method

➤ **Step 1:** Given  $P$  and test set  $T$ , find the **execution trace** of  $P$  for each test in  $T$ .

➤ **Step 2:** Extract **test vectors** from the execution traces for each node in the CFG of  $P$

➤ **Step 3:** Construct **syntax trees** for each node in the CFGs of  $P$  and  $P'$ . This step can be executed while constructing the CFGs of  $P$  and  $P'$ .

➤ **Step 4:** Traverse the CFGs and determine the a subset of  $T$  appropriate for regression testing of  $P'$ .

讲了以后，大家应该对这个方法的原理有了一个比较抽象的概念。它第一点识别软件的哪些部分进行了修改。然后去识别原来的测试用例集和修改的部分之间的关系。

我们通过一些示例来理解一下，我们要理解执行的轨迹。我们先要对程序  $P$  进行程序流图的建模。



## Execution Trace [1]

➤ Let  $G=(N, E)$  denote the CFG of program  $P$ .  $N$  is a finite set of nodes and  $E$  a finite set of edges connecting the nodes. Suppose that nodes in  $N$  are numbered 1, 2, and so on and that Start and End are two special.

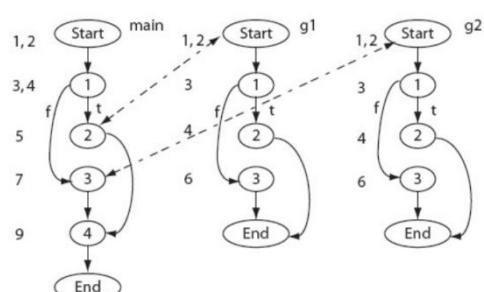
➤ Let  $T_{no}$  be the set of all valid tests for  $P'$ . Thus  $T_{no}$  contains only tests valid for  $P'$ . It is obtained by discarding all tests that have become **obsolete** for some reason.

我们看如下的一个例子：

➤ An execution trace of program  $P$  for some test  $t$  in  $T_{no}$  is the sequence of nodes in  $G$  traversed when  $P$  is executed against  $t$ . As an example, consider the following program.

```
1 main(){           1 int g1(int a, b){ 1 int g2(int a, b){  
2 int x,y,p;      2 int a,b;  
3 input (x,y);    3 if(a+ 1==b)    2 int a,b;  
4 if (x<y)        4 return(a*a);  4 if(a==(b+1))  
5   p=g1(x,y);    5 else          5 return(b*b);  
6   else          6 return(b*b);  6 return(a*a);  
7   p=g2(x,y);    7 }  
8 endif  
9 output (p);  
10 end  
11 }
```

➤ Here is a CFG for our example program.



上右图就是左侧程序的程序流图。那么我们怎么得到测试用例的对应执行轨迹呢？我们考虑如下的 3 个测试用例的集合  $T$ ，我们要对每个测试用例  $t_i$  去分析它的执行路径。

➤ Now consider the following set of three tests and the corresponding trace.

$$T = \left\{ \begin{array}{l} t_1 : \langle x = 1, y = 3 \rangle \\ t_2 : \langle x = 2, y = 1 \rangle \\ t_3 : \langle x = 3, y = 4 \rangle \end{array} \right\}$$

拿  $t_1$  为例, 它会有执行路径  $trace(t_1)$ 。其实就是通过程序流图中的节点执行序列来表示。

最后我们得到了这么一个执行轨迹的表格。测试用例和执行轨迹一一对应。

Test ( $t$ )	Execution trace ( $trace(t)$ )
$t_1$	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
$t_2$	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
$t_3$	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

接下来我们就要得到一个测试向量, 它就是测试用例的一个集合。测试向量表格的行是一个个函数。每一列就是每个函数的每一个基本块。以 `main` 函数为例, 它就有 4 个基本块。每个格子中的测试向量就是经过了对应的函数基本块的测试用例。

## Test vector

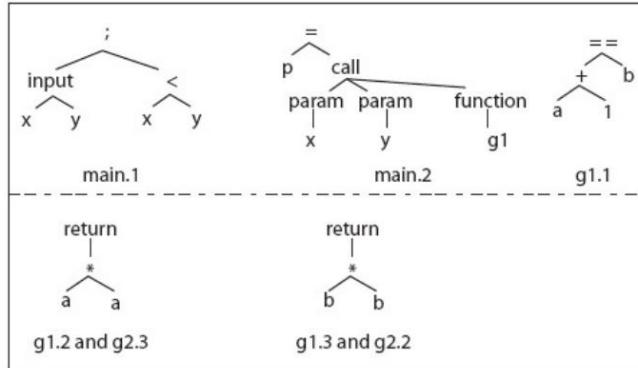
➤ A test vector for node  $n$ , denoted by  $test(n)$ , is the set of tests that traverse node  $n$  in the CFG. For program  $P$  we obtain the following test vectors.

Function	Test vector ( $test(n)$ ) for node $n$			
	1	2	3	4
main	$t_1, t_2, t_3$	$t_1, t_3$	$t_2$	$t_1, t_2, t_3$
g1	$t_1, t_3$	$t_3$	$t_1$	—
g2	$t_2$	$t_2$	None	—

Test ( $t$ )	Execution trace ( $trace(t)$ )
$t_1$	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
$t_2$	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
$t_3$	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

后面我们就针对测试向量表进行回归测试的用例的选择。接下来我们要做句法树的分析, 要看看原来的程序和现有的程序哪些发生了变化。我们通过句法树的形式去看里面语义的变化。

>A syntax tree is constructed for each node of  $\text{CFG}(P)$  and  $\text{CFG}(P')$ . Recall that each node represents a basic block.  
Here sample syntax trees for the example program.



假如我们对  $g1$  的第三行的谓词进行了修改，对应的基本块是  $g1.1$  进行了修改，我们会看过有  $t_1, t_3$  经过了它。所以我们就会选择  $t_1, t_3$  作为我们回归测试的用例集  $T'$ 。所以我们测试用例集的选择是针对测试向量表来做的。



## Test selection example

>Suppose that function  $g1$  in  $P$  is modified as follows.

```

1 int g1(int a, b){ ← Modified g1.
2 int a, b;
3 if(a-1==b) ← Predicate modified.
4   return(a*a),
5 else
6   return(b*b),
7 }
```

Function	Test vector ( $test(n)$ ) for node $n$			
	1	2	3	4
main	$t_1, t_2, t_3$	$t_1, t_3$	$t_2$	$t_1, t_2, t_3$
g1	$t_1, t_3$	$t_3$	$t_1$	—
g2	$t_2$	$t_2$	None	—

>Try the SelectTests algorithm and check if you get  $T'=\{t1, t3\}$ .

选择策略也很简单，就是遍历所有节点。最终我们通过程序流图的执行轨迹进行相应的分析，去找回归测试集。

## **Test selection [1]**

---

➤ Given the execution traces and the CFGs for P and P', the following three steps are executed to obtain a subset T' of T for regression testing of P'.

- Step 1 Set  $T' = \emptyset$ . Unmark all nodes in G and in its child CFGs.
- Step 2 Call procedure `SelectTests` (G. Start, G'.Start'), where G.Start and G'.Start' are, respectively, the start nodes in G and G'.
- Step 3  $T'$  is the desired test set for regression testing  $P'$ .

## **Test selection [2]**

---

- The basic idea underlying the `SelectTests` procedure is to traverse the two CFGs from their respective START nodes using a recursive descent procedure.
- The descent proceeds in parallel and the corresponding nodes are compared. If two nodes N in  $\text{CFG}(P)$  and  $N'$  in  $\text{CFG}(P')$  are found to be syntactically different, all tests in test (N) are added to  $T'$ .

我们可以通过自动地比对把测试用例找出来。

### **Test selection using test minimization**

这种方法叫做最小化测试的方法。它也是回归测试的用例选择方法。我们假定原来的程序 P 有两个函数，分别是 `main` 和 `f`，并且我们使用测试用例  $t_1, t_2$  来测试。观测到测试用例  $t_1$  能导致 `main` 函数执行，而没有执行 `f`。而观测到测试用例  $t_2$  能导致 `main` 和 `f` 函数均执行。

## Test minimization [1]

**Test minimization is yet another method for selecting tests for regression testing.**

**To illustrate test minimization, suppose that P contains two functions, main and f. Now suppose that P is tested using test cases t1 and t2. During testing it was observed that t1 causes the execution of main but not of f and t2 does cause the execution of both main and f.**

Q1: 如果我们的函数 main 在新版本中发生了改变，是不是  $t_1, t_2$  都有必要测一下呢？

Q2: 如果我们的函数 f 在新版本中发生了改变，因为  $t_1$  不能导致 f 执行，是不是我们只需要测试  $t_2$  呢？

如果有一些代码的修改以后，不经过修改代码执行的测试用例是不是可以不用进行测试。这样我们就会有相应的分析。前面的方法我们考虑的是基本块，而在里我们粒度是否经过了某个函数。

## Test minimization [2]

**Now suppose that P' is obtained from P by making some modification to f.**

**Which of the two test cases should be included in the regression test suite?**

**Obviously there is no need to execute P' against t1 as it does not cause the execution of f. Thus the regression test suite consists of only t2.**

如果我们提出 function coverage，我们就可以考虑这两个测试用例能够覆盖函数。如果一个测试用例可以把所有 function 全部覆盖，就可以形成一种最小化的测试。按照函数覆盖指标测试来说，这样的测试就是足够的。

**In this example we have used function coverage to minimize a test suite {t1, t2} to obtain the regression test suite {t2}.**

从本质来讲，它有一些错误不能发现，因为它的粒度很大。我们粒度可以选为 function、program statements、decisions、def-use chains 等。所以这种方法可以针对测试的实体（不同粒度）去最小化测试用例集。

**Test minimization is based on the coverage of testable entities in P.**

**Testable entities include, for example, program statements, decisions, def-use chains, etc.**

**One uses the following procedure to minimize a test set based on a selected testable entity.**

## A procedure for test minimization

**Step 1: Identify the type of testable entity to be used for test minimization. Let  $e_1, e_2, \dots, e_k$  be the  $k$  testable entities of type TE present in P. In our previous example TE is function.**

**Step 2: Execute P against all elements of test set T and for each test  $t$  in T determine which of the  $k$  testable entities is covered.**

**Step 3: Find a minimal subset  $T'$  of T such that each testable entity is covered by at least one test in  $T'$ .**

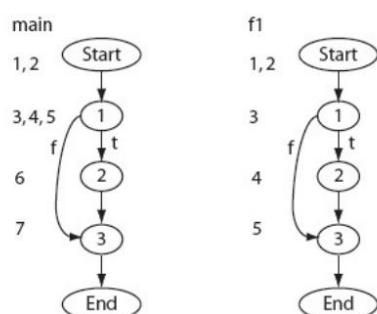
我们可以看一个例子：

在这个测试中，我们选择基本块作为可测试的实体。那么我们的目的就是选择最小的测试用例，把所有的基本块至少都执行一遍。所以在这个例子中，我们就选择了  $t_1, t_3$  作为测试的最小集合。从选择实体的覆盖率的角度来说，这种方法是完备的。



### Test minimization: Example

**Step 1: Let the basic block be the testable entity of interest. The basic blocks for a sample program are shown here for both main and function f1.**



**Step 2: Suppose the coverage of the basic blocks when executed against three tests is as follows:**

- $t_1$ : main: 1, 2, 3. f1: 1, 3
- $t_2$ : main: 1, 3. f1: 1, 3
- $t_3$ : main: 1, 3. f1: 1, 2, 3

**Step 3: A minimal test set for regression testing is  $\{t_1, t_3\}$ .**

## Test selection using test prioritization

我们还有通过测试优先级来进行测试，我们知道不同的路径的优先级不同。我们可以根据测试实体的优先级大小来选择我们的测试用例。



### Test prioritization

- > Note that test minimization will likely discard test cases. There is a small chance that if  $P'$  were executed against a discarded test case it would reveal an error in the modification made.
- > When very **high quality software** is desired, it might not be wise to discard test cases as in test minimization. In such cases one uses **test prioritization**.
- > Tests are prioritized based on some criteria. For example, **tests that cover the maximum number of a selected testable entity could be given the highest priority, the one with the next highest coverage m the next higher priority and so on.**

定义优先级的方法有很多，比如我们可以根据测试的实体去判别它的优先级。比如我们定义一个测试用例如果经过了更多的测试实体，我们认为它更重要。核心就是通过某个基准对测试用例进行排序。排序方法有很多种。



### A procedure for test prioritization

**Step 1: Identify the type of testable entity to be used for test minimization. Let  $e_1, e_2, \dots, e_k$  be the  $k$  testable entities of type TE present in  $P$ . In our previous example TE is function.**

**Step 2: Execute  $P$  against all elements of test set  $T$  and for each test  $t$  in  $T$ . For each  $t$  in  $T$  compute the number of distinct testable entities covered.**

**Step 3: Arrange the tests in  $T$  in the order of their respective coverage. Test with the maximum coverage gets the highest priority and so on.**



## Using test prioritization

Once the tests are prioritized one has the option of using all tests for regression testing or a subset.

The choice is guided by several factors such as the resources available for regression testing and the desired product quality.

In any case test are discarded only after careful consideration that does not depend only on the coverage criteria used.

前面我们讲的是具体方法，很多书上不会具体讲这些东西。为了让大家更容易理解，我们还找了一些原始的方法进行说明。实际上现在的回归测试方法本身已经集成到回归测试的工具里去了，我们在使用工具的时候可以理解一下里面的策略的选择。这样可以比较好地支撑自动化的测试。



## Tools for regression testing

Methods for test selection described here require the use of an automated tool for all but trivial programs.

xSuds from Telcordia Technologies can be used for C programs to minimize and prioritize tests.

Many commercial tools for regression testing simply run the tests automatically; they do not use any of the algorithms described here for test selection. Instead they rely on the tester for test selection. Such tool are especially useful when all tests are to be rerun.

自动化回归测试要把方法本身变成流程化的测试过程。



## Tools for regression testing

TestingWhiz是一款无需编码即可使用的回归测试自动化工具，专门面向Web、移动及云应用，且提供超过290种预定义测试命令以实现测试用例的编写与编辑。



Sahi是一款开源工具，但Sahi Pro则属于面向Web应用的商用测试自动化工具。Sahi Pro能够管理大型测试套件的回归测试自动化事务。



TestComplete是一套来自Smartbear公司的平台，适用于桌面、Web以及移动测试工具。它能够实现功能与回归测试自动化，并支持由JavaScript、C++ Script、C# Script、VB Script、Python、Jscript以及DelphiScript等编写而成的测试。

**SMARTBEAR**

Silk Test是一款由Borland推出的自动化测试工具，旨在执行功能与回归测试。它基于类似于C++的面向对象编程(简称OOP)语言，其中包含对象、类与继承等概念。



## Summary

- **Regression testing is an essential phase of software product development.**
- **In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.**
- **In such situations one can make use of sophisticated technique for selecting a subset of all tests and hence reduce the time for regression testing.**

回归测试的核心就是我们不希望重新把所有测试用例测试一遍，我们希望用最小的代价来进行新的测试。

2022/5/7

## WEB 网站测试



## 2 WEB网站的特性

### 2.1. WEB网站的概念

- ④ 大多数WEB网站都是采用B-S结构。
- ④ 能够交付一组复杂的内容和功能给大量的终端用户。
- ④ WEB网站测试是用于测试高质量Web应用系统的过程，它借用了许多传统软件测试和系统测试的概念和原理。



我们要知道网站的特点才能做测试，比如门户网站测起来的层次会很多。工作量会大一些



## 2.2 WEB网站的特点

### ④ 网络集约性

就本质而言，一个Web网站是网络集约的。它可以驻留在网络上，并且服务于变化多样的客户群的需要。

如时下流行的门户网站或者网络游戏。它们都可以看成一个完善的大型Web应用系统，服务于各种客户群，但其本身只需要一个服务器端，用各式各样的客户端满足不同要求的客户。



- ④ 网络集约性
- ④ 内容驱动性
- ④ 持续演化性

一般来说，Web网站不是为了某个或某些特定用户量身定做的，它们一般都拥有一个广大的服务群体，其服务的内容，往往由这些群体的要求所决定。在大多数情况下，一个Web网站的主要功能是使用HTML（超文本标记语言）javascript等语言来表示文本、图形、音频、视频内容给终端用户。

WEB网站可能每天都在更新。

不同于传统的、按一系列规律发布进行演化  
的应用软件（如微软每隔1-2年发布新的Office办  
公软件），Web网站一般是采取持续演化的模式。  
对于某些Web应用而言，按小时为单位进行更新  
都是司空见惯的。

- ④ 网络集约性
- ④ 内容驱动性
- ④ 持续演化性
- ④ 即时性
- ④ 网络集约性
- ④ 内容驱动性
- ④ 持续演化性
- ④ 即时性
- ④ 安全性



### 3 网页测试

---

网页测试包括以下内容：

#### 3.1. 功能测试

#### 3.2. 可用性测试

功能测试包括以下内容：

- ④ 链接测试

什么是链接？

链接是Web网站的一个主要特征，它是在页面之间切换和引导用户去一些未知地址页面的主要手段。

功能测试包括以下内容：

- ④ 链接测试
- ④ 表单测试
- ④ 数据校验
- ④ Cookies 测试

#### 什么是Cookies？

Cookie是一个由网页服务器放在您硬盘上的非常小的文本文件。它本质上就像您的身份证明一样，并且不能像代码那样被执行或被用来散布病毒。它只能被您使用并且只能由提供的服务器读取。



### Cookies 测试

测试内容：

- ④ Cookies是否能正常工作；
- ④ Cookies是否按预定的时间进行保存；
- ④ 刷新对Cookies有什么影响等。

举例：

- ④ 如果在cookies 中保存了注册信息，应确认该cookie 能够正常工作而且已对这些信息进行加密。
- ④ 如果使用cookie 来统计次数，需要验证次数累计正确。

问题： Cookies 造成的负面影响是什么？

Cookies 容易暴露访问者的一些隐私内容。



### 3.2. 可用性测试

可用性测试包括：

- ① 导航测试：导航描述了用户在一个页面内操作的方式
- ② 图形测试：一个Web 网站的图形可以包括图片、动画、边框、颜色、字体、背景、按钮等。
- ③ 内容测试：内容测试用来检验Web 网站提供信息的正确性、准确性和相关性。
- ④ 整体界面测试：对整个Web 系统的页面结构设计的测试，是用户对系统的一个整体感受。



## 整体界面测试

例如，当用户浏览Web网站时，应考虑

- ④ 是否感到舒适？
- ④ 是否凭直觉就知道要找的信息在什么地方？
- ④ 整个Web应用系统的设计风格是否一致？



### ① 导航测试的内容

- ④ 导航是否直观？
- ④ Web系统的主要部分是否可以通过主页访问？
- ④ Web系统是否需要站点地图、搜索引擎或其他的导航器帮助？
- ④ 测试Web系统的页面结构；
- ④ 导航条、菜单、连接的风格是否一致？
- ④ 各种提示是否准确，确保用户凭直觉就知道是否还有内容，内容在什么地方。
- ④ 最好让最终用户参与导航测试，效果将更加明显。



### ② 图形测试

- 要确保图形有明确的用途，图片或动画不要胡乱地堆在一起，以免浪费传输时间。图片尺寸要尽量地小，并且要能清楚地说明某件事情。
- 验证所有页面字体的风格是否一致。
- 背景色应该与字体颜色和前景颜色相搭配。
- 图片的大小和质量也是一个很重要的因素，一般采用JPG或GIF压缩。



## 4 网站测试

网站测试在网页测试的基础之上还包括：

- 4.1 功能测试；
- 4.2 性能测试；
- 4.3 安全性测试；
- 4.4 兼容性测试



### 4.1 功能测试

功能测试包括以下内容：

#### 1) 数据库测试

在Web应用中，最常用的数据库类型是关系型数据库，  
可以使用SQL对信息进行处理。[常见问题](#)



### 数据库测试

两种主要数据库错误：

#### ④ 数据一致性错误

主要是由于用户提交的表单信息不正确而造成的。

#### ④ 输出错误

主要是由于网络传输速度或程序设计问题等引起的。

数据库测试就要针对这两种情况，分别进行测试。

---

数据库测试是一个专题，一般包括

- 数据一致性
  - 备份与恢复
  - 数据库响应测试（查询时间）
  - 数据库压力测试（数据量）
  - 并发测试
  - 权限管理
- ④ ?



## 4.1 功能测试

---

功能测试包括以下内容：

1) **数据库测试**

在Web应用中，最常用的数据库类型是关系型数据库，  
可以使用SQL对信息进行处理。[常见问题](#)

2) **WEB网站特定的功能需求测试**

依据需求规格说明书对WEB网站特定的功能需求进行验证。

3) **设计语言测试**

不同的Web设计语言版本的差异可以引起客户端或服务  
器端严重的问题；

例如使用Java、JavaScript、ActiveX、VBScript或Perl等  
开发的应用程序也要在不同的版本上进行验证。



## 4.2 性能测试

---

包括以下内容：

① **连接速度测试**

② **负载测试**

③ **压力测试**



## ① 连接速度测试

用户连接方式的不同：都有哪些？

- ④ 电话拨号上网
- ④ 宽带上网
- ④ 局域网（光纤）
- ④ GSM、CDMA、4G、5G
- ④ 无线（WiFi）
- ④ 有线电视网
- ④ 电力网
- ④ 其它？



## ② 负载测试

负载测试的目的：

- ④ 负载测试是为了测量Web系统在某一负载级别的性能，以保证Web系统在需求范围内能正常工作。

负载测试内容：

- ④ 某个时刻同时访问Web系统的用户数量；
- ④ 在线数据处理的数量。

例如：

- ④ 系统最多能允许多少个用户同时在线？
- ④ 如果超过了这个数量，会出现什么现象？
- ④ 系统能否处理大量用户同时对同一个页面的请求？



### ③ 压力测试

压力测试的目的：

压力测试目的是要弄清楚被测试的 Web 服务是不是不仅能够做我们认为它能做的事，而且在被施加了某些高强度压力的情况下仍然继续正常运行。

压力测试的内容：

压力测试必须对 Web 服务应用以下四个基本条件进行有效压力测试。

- ④ 重复 (Repetition)；
- ④ 并发 (Concurrency)；
- ④ 量级 (Magnitude) - (每个操作中的负载量)；
- ④ 随机变化。



#### 压力测试的内容

重复 (Repetition)：

测试的重复就是一遍又一遍地执行某个操作或功能。比如重复调用一个 Web 服务，确定一个操作能否正常执行，并且能否继续在每次执行时都正常。



#### 压力测试的内容

并发 (Concurrency)：

并发是同时执行多个操作的行为。换句话说，就是在同一时间执行多个测试，例如在同一个服务器上同时调用许多 Web 服务。



#### 压力测试的内容

量级 (Magnitude)：

压力测试系统应该应用于产品的另一个条件，需要考虑每个操作中的负载量，即也要尽量给产品增加负担。

例如，改变数据的大小、改变时间延迟的长度、资金数量的转移、输入速度以及输入的变化等。



## 压力测试的内容

随机变化：

任何压力系统都多多少少具有一些随机性。随机使用前面的压力原则中介绍的无数变化形式，就能够在每次测试运行时应用许多不同的代码路径。



## 压力测试用例参考模板

1. 被测试对象的介绍

2. 测试范围与目的

3. 测试环境与测试辅助工具的描述

4. 测试驱动程序的设计

5. 压力测试用例

极限名称 A	如“最大并发用户数量”	
前提条件		
输入/动作	输出/响应	是否能正常运行
如 10 个用户并发操作		
如 20 个用户并发操作		

.....



## 4.3 安全性测试

- ④ 登录验证
- ④ 数据加密
- ④ 超时限制
- ④ SSL 套接字测试
- ④ 目录测试
- ④ 脚本语言
- ④ 日志文件



## Web安全性测试

- ④ 登录验证：一般的网站都会使用登录或者注册后使用的方式，因此，必须对用户名和匹配的密码进行校验，以阻止非法用户登录。在进行登陆测试的时候，需要考虑输入的密码是否对大小写敏感、是否有长度和条件限制，最多可以尝试多少次登录，哪些页面或者文件需要登录后才能访问/下载等。
- ④ 数据加密：某些数据需要进行信息加密和过滤后才能进行数据传输，例如用户信用卡信息、用户登陆密码信息等。（包括加密算法、解密算法）
- ④ 超时限制：WEB应用系统需要有是否超时的限制，当用户长时间不作任何操作的时候，需要重新登录才能使用其功能。
- ④ SSL：越来越多的站点使用SSL安全协议进行传送。SSL是Security Socket Layer(安全套接字协议层)的缩写，是由Netscape首先发表的网络数据安全传输协议。SSL是利用公开密钥/私有密钥的加密技术。（RSA），在位于HTTP层和TCP层之间，建立用户与服务器之间的加密通信，确保所传递信息的安全性。SSL是工作在公共密钥和私人密钥基础上的，任何用户都可以获得公共密钥来加密数据，但解密数据必须要通过相应的私人密钥。
- ④ 目录测试：WEB的目录安全不容忽视的一个因素。如果WEB程序或WEB服务器的处理不适当，通过简单的URL替换和推测，会将整个WEB目录完全暴露给用户，这样会造成很大的风险和安全隐患。
- ④ 服务器脚本语言：脚本语言是最常见的安全隐患，如有些脚本语言允许访问根目录，经验丰富的黑客可以通过这些缺陷来攻击和使用服务器系统，因此，脚本语言安全性在测试过程中也必须被考虑到。
- ④ 日志文件：在服务器上，要验证服务器的日志是否正常工作，例如CPU的占用率是否很高，是否有例外的进程占用，所以的事务处理是否被记录等。



## 4.4 兼容性测试

- ④ **平台测试**: 操作系统类型，最常见的有Windows、Unix、Macintosh、Linux、symbian、ios、windows phone等；
- ④ **浏览器测试**: 不同厂商的浏览器对Java、JavaScript、ActiveX、plug-ins 或不同的HTML 有不同的支持；
- ④ **分辨率测试**: 页面版式在不同(640x400、600x800 或 1024x768) 分辨率模式下是否显示正常？

### 常见的显示器分辨率

- ④ **连接速率测试**: 不同接入介质的速率测试；
- ④ **组合测试**: 前四项的组合。

## 变异测试

### **LEARNING OBJECTIVES**

- What is test adequacy? What is test enhancement? When to measure test adequacy and how to use it to enhance tests?
- What is program mutation?
- Competent programmer hypothesis and the coupling effect.
- Strengths and limitations of test adequacy based on program mutation.
- Mutation operators
- Tools for mutation testing

## What is adequacy?

- Consider a program P written to meet a set R of functional requirements. We denote such a P and R as  $(P, R)$ . Let R contain n requirements labeled  $R_1, R_2, \dots, R_n$ .
- Suppose now that a set T containing k tests has been constructed to test P to determine whether or not it meets all the requirements in R. Also, P has been executed against each test in T and has produced correct behavior.
- We now ask: Is T good enough? This question can be stated differently as: Has P been tested thoroughly?, or as: Is T adequate?

## What is program mutation?

- Suppose that program P has been tested against a test set T and P has not failed on any test case in T. Now suppose we do the following:

Changed to



What behavior do you expect from P' against tests in T?

## What is program mutation? [2]

- P' is known as a mutant of P.
- There might be a test t in T such that  $P(t) \neq P'(t)$ . In this case we say that t distinguishes P' from P. Or, that t has killed P'.
- There might be not be any test t in T such that  $P(t) \neq P'(t)$ . In this case we say that T is unable to distinguish P and P'. Hence P' is considered live in the test process.

## What is program mutation? [3]

- If there does not exist any test case  $t$  in the input domain of  $P$  that distinguishes  $P$  from  $P'$  then  $P'$  is said to be **equivalent** to  $P$ .
- If  $P'$  is not equivalent to  $P$  but no test in  $T$  is able to distinguish it from  $P$  then  $T$  is considered **inadequate**.

*We will refer to program mutation as mutation.*

**2022/5/9**

### 大作业评分细则

- 每次大作业按以下方法进行打分（共4次）
- 小组得分：
  - 满分100分，从测试内容，文档质量，代码质量(第一次作业无)，答辩表现以及加分项（作业亮点）来进行评分。
- 个人得分：
  - 满分100分，根据小组得分和组内贡献度来计算个人得分，计算公式为：  
$$\text{个人分} = 50 + (\text{小组分} - 50) * \text{贡献度} / 25\% \quad (4\text{人})$$
- 每一次大作业的个人贡献度需要组内协商（组员贡献度之和=100%）
- 若个人贡献度低于  $(60/N) \%$ ,  $N$ 为组人数。该生小组分=小组分\*  
$$\text{个人贡献度}/(100/N)$$

我们有一次答辩，基本上是xx周之内，助教在批改大作业的时候可能会漏掉一些信息，或者助教跑不通代码的情况。

答辩前几分钟，PPT简单介绍一下情况和内容。另外老师/助教会针对课程作业里的问题做一些确认，有些小组会超出测试内容做一些亮点，答辩的时候也可以强调。

我们继续上课，我们的课现在第13周，还有一些内容没有讲完。上节课我们对变异测试开了一个头。

它方法里有很多的考虑。变异测试是从软件测试的充分性这个角度去切入的。首先，我们要评估软件的测试是否充分，如果不充分，我们就需要增加额外的测试用例。

## LEARNING OBJECTIVES

- What is test adequacy? What is test enhancement? When to measure test adequacy and how to use it to enhance tests?
- What is program mutation?
- Competent programmer hypothesis and the coupling effect.
- Strengths and limitations of test adequacy based on program mutation.
- Mutation operators
- Tools for mutation testing

一个程序首先是要做了大部分测试，我们不太清楚这些测试够不够。所以我们要通过变异测试的方法评估它。对于黑盒和白盒是有覆盖率的情况，但是脱离覆盖率的时候，我们需要变异测试作为额外的方法对测试覆盖率进行说明。

变异体是对程序的编译，比如我们现在有  $k$  个变异体，这些变异体是形成变异测试的基础。我们针对每一个变异体，看看在原来的测试用例集合  $P$  中，是否存在测试用例能够区分本体和变异体。如果能区分，我们就去掉这个变异体。

### Test adequacy using mutation [1]

- Given a test set  $T$  for program  $P$  that must meet requirements  $R$ , a test adequacy assessment procedure proceeds as follows.
  - Step 1: Create a set  $M$  of mutants of  $P$ . Let  $M = \{M_1, \dots, M_k\}$ . Note that we have  $k$  mutants.
  - Step 2: For each mutant  $M_i$ , find if there exists a  $t$  in  $T$  such that  $M_i(t) \neq P(t)$ . If such a  $t$  exists then  $M_i$  is considered killed and removed from further consideration.

## 变异分数

接下来我们计算变异分数， $e$  是等价变异体的数量。原来有  $k$  个变异体，也就是  $k-e$  个非等价变异体，其中有  $k_1$  个可以被区分。还有  $k-k_1$  个是存活的。

## Test adequacy using mutation [2]

- Step 3: At the end of Step 2 suppose that  $k_1 \leq k$  mutants have been killed and  $(k-k_1)$  mutants are live.

Case 1:  $(k-k_1)=0$ : T is adequate with respect to mutation.

Case 2:  $(k-k_1)>0$  then we compute the mutation score (MS) as follows:

$$MS = k_1 / (k-e)$$

Where e is the number of equivalent mutants. Note:  $e \leq (k-k_1)$ .

- MS=1 的情况，也就是  $k_1=k-e$ ，也就是我们的测试用例可以把原来的所有非等价变异体区分开，这就说明原来的测试用例针对于  $k$  个变异体是足够的，我们认为测试对于这  $k$  个变异体来说是充分的，但是如果换成其他  $k$  个变异体，充分性可能发生变化。

变异测试主要用来针对测试集合进行优化的，可以评估测试集的充分性，并且还可以进行测试集的增强

### 用来测试集充分性adequacy

- 创建变异体集合
- 对于每一个变异体均进行测试，如果有一个测试集中的测试用例使得此变异体无法通过测试，那么杀掉他（因为你既然这么轻易被测试集区分开了，那么你就没啥意义了）
- 假若最终由  $k_1$  个变异体活了下来，而原先有  $k$  个变异体
  - 如果  $k-k_1=0$ ，那么意味着测试集太强了，是充分的
  - 如果  $k-k_1>0$ ，使用  $MS=k_1/(k-e)$  来查看变异的指标， $e$  是等价变异体，因为等价变异体就相当于冗余的，所以去掉

### 用来测试集增强enhancement

- 如果  $MS=1$ ，说明测试集太菜了，并且菜的离谱，所以需要放弃本次实验，选择其他技术
- 如果  $MS<1$ ，说明有突变体存活，因此需要设计一个新的测试用例，至少区分一个本体与变异体，如果区分不了重新设计
- 设计好了之后扩展测试集，这样测试集便更有能力杀死变异体了，下面重新计算  $MS$
- 以此类推
- (所谓的等价变异体可以看做和源程序一样的变异体)

## 用来测试集增强enhancement

1. 如果 $MS=1$ , 说明测试集太菜了, 并且菜的离谱, 所以需要放弃本次实验, 选择其他技术
2. 如果 $MS<1$ , 说明有突变体存活, 因此需要设计一个新的测试用例, 至少区分一个本体与变异体, 如果区分不了重新设计
3. 设计好了之后扩展测试集, 这样测试集便更有能力杀死变异体了, 下面重新计算MS
4. 以此类推
5. (所谓的等价变异体可以看做和源程序一样的变异体)

## 用来错误预测error detection

使用变异来进行错误预测:

1. 设计好了变异体, 并进行了测试
2. 发现存活的变异体之后, 我们便首先进行等价变异体的排除
3. 设计新的测试用例, 进行变异体杀害
4. 在设计新的测试用例之后, 发现原程序中的bug

## 关于等价变异体Equivalent mutants

1. 等价变异体是无法决断的, 也无法使用自动化工具
2. 经验表明5%的变异体会是等价变异体
3. 识别手工变异体通常得人工! 非常沮丧

我们首先要考虑生成的  $k$  个变异体, 再考虑等价变异体的数量。等价变异体是不存在任何测试用例去区分原代码和等价变异体的, 所以我们计算变异分数的时候要减掉。

如果  $MS < 1$ , 说明测试并不充分, 我们可以增强测试的充分性, 增加新的测试用例去区分这些还没有被区分的测试用例

## Test enhancement using mutation

- One has the opportunity to enhance a test set T after having assessed its adequacy.
- Step 1: If the mutation score (MS) is 1, then some other technique, or a different set of mutants, needs to be used to help enhance T.
- Step 2: If the mutation score (MS) is less than 1, then there exist live mutants that are not equivalent to P. Each live mutant needs to be distinguished from P.

## Test enhancement using mutation [2]

- **Step 3:** Hence a new test  $t$  is designed with the objective of distinguishing at least one of the live mutants; let us say this is mutant  $m$ .
- **Step 4:** If  $t$  does not distinguish  $m$  then another test  $t$  needs to be designed to distinguish  $m$ . Suppose that  $t$  does distinguish  $m$ .

## Test enhancement using mutation [3]

- **Step 6:** One now adds  $t$  to  $T$  and re-computes the mutation score (MS). 
- Repeat the enhancement process from Step 1.

这个过程就是不断增加新的测试用例使 MS 等于 1 的过程。

Q: 变异测试是否能够找到程序中存在的一些错误呢?

A: 实验证明变异测试是一个比较有效的方法。因为变异数体生成的时候，会考虑程序员经常犯的一些错误，尽量去模拟，这样就可以发现错误了。

## Error detection using mutation

- As with any test enhancement technique, there is no guarantee that tests derived to distinguish live mutants will reveal a yet undiscovered error in P. Nevertheless, empirical studies have found to be the most powerful of all formal test enhancement techniques.

## 变异测试的例子

我们来看如下的一个例子，`foo` 把里面的加号写成了减号。

## Error detection using mutation [2]

- Consider the following function `foo` that is required to return the sum of two integers `x` and `y`. Clearly `foo` is incorrect.

```
int foo(int x, y){  
    return (x-y); ← This should be return (x+y)  
}
```

比如这个程序原来做了两个测试  $T = \{t_1, t_2\}$ , 对于原程序是可以通过的。

## Error detection using mutation [3]

- Now suppose that `foo` has been tested using a test set  $T$  that contains two tests:

$$\textcircled{T} = \{ t1: \langle x=1, y=0 \rangle, t2: \langle x=-1, y=0 \rangle \}$$

- First note that `foo` behaves perfectly fine on each test in, i.e. `foo` returns the expected value for each test case in  $T$ . Also,  $T$  is adequate with respect to all control and data flow based test adequacy criteria.

接下来我们生成变异体：

## Error detection using mutation [4]

Let us evaluate the adequacy of  $T$  using mutation. Suppose that the following three mutants are generated from `foo`.

M1: int foo(int x, y){ return (x+y); }	M2: int foo(int x, y){ return (x-0); }	M3: int foo(int x, y){ return (0+y); }
--	--	--

接下来我们在原程序的测试集上去测试这几个变异体，发现  $M_1$  和  $M_2$  都是存活的，并且我们人工发现  $M_1$  和  $M_2$  都是非等价变异体。所以非等价变异体没有被这两个测试用例区分开，说明测试用例是不充分的。

## Error detection using mutation [4]

Next we execute each mutant against tests in T until the mutant is distinguished or we have exhausted all tests. Here is what we get.

$$T = \{ t1: \langle x=1, y=0 \rangle, t2: \langle x=-1, y=0 \rangle \}$$

Test (t)	foo(t)	M1(t)	M2(t)	M3(t)
t1 <sub>1</sub>	1 <sub>1</sub> <sup>0</sup>	1	1	0
t2 <sub>1</sub>	-1 <sub>1</sub>	-1	-1	0
		Live	Live	Killed

## Error detection using mutation [5]

After executing all three mutants we find that two are live and one is distinguished. Computation of mutation score requires us to determine if any of the live mutants is equivalent.

*In class exercise: Determine whether or not the two live mutants are equivalent to foo and compute the mutation score of T.*

接下来我们设计一个新的测试用例  $t_3$ 。

## Error detection using mutation [6]

Let us examine the following two live mutants.

M1: int foo(int x, y){	M2: int foo(int x, y){
return (x+y);	return (x-0);
}	}

Let us focus on M1. A test that distinguishes M1 from **foo** must satisfy the following condition:  
 $x-y \neq x+y$  implies  $y \neq 0$ .

Hence we get t3:  $\langle x=1, y=1 \rangle$

## Guaranteed error detection

Sometimes there exists a mutant  $P'$  of program  $P$  such that any test  $t$  that distinguishes  $P'$  from  $P$  also causes  $P$  to fail.  
More formally:

Let  $P'$  be a mutant of  $P$  and  $t$  a test in the input domain of  $P$ . We say that  $P'$  is an **error revealing** mutant if the following condition holds for any  $t$ .

$P'(t) \neq P(t)$  and  $P(t) \neq R(t)$ , where  $R(t)$  is the expected response of  $P$  based on its requirements.

*Is M1 in the previous example an error revealing mutant? What about M2?*

## 变异体的特性

变异体是需要有一些特性的，比如可达性、传染性和传播性。首先测试用例  $t$  一定要覆盖到变异语句  $s$ ，否则就很难查错。其次变异语句要导致程序状态的不一致。

## Distinguishing a mutant

A test case  $t$  that distinguishes a mutant  $m$  from its parent program  $P$  must satisfy the following three conditions:

- **Condition 1:** 可达性: 测试用例  $t$  执行时需要覆盖到变异语句  $s$ 。因为变异体  $m$  内其他语句与  $P$  相同, 如果测试用例  $t$  不能执行  $m$  中的  $s$  语句, 则  $t$  在  $P$  和  $m$  上的运行结果必然一致, 即  $t$  不能检测到变异体  $m$ 。
- **Condition 2:** 传染性: 测试用例  $t$  在执行完  $P$  和  $m$  的语句  $s$  后程序状态不一致。

## Distinguishing a mutant [2]

- **Condition 3:** 传播性: 执行完语句  $s$  后的不一致的程序状态将导致程序  $P$  和  $m$  的输出不一致。

*Exercise: Show that in the previous example both  $t_1$  and  $t_2$  satisfy the above three conditions for  $M_3$ .*

Q: 对于变异体数量巨大的变异测试来说, 我们怎么辨别里面的等价变异体呢?

A: 等价变异体还是比较少见的, 等价变异体通常需要手动识别。

## Equivalent mutants

- The problem of deciding whether or not a mutant is equivalent to its parent program is undecidable. Hence there is no way to fully automate the detection of equivalent mutants.
- The number of equivalent mutants can vary from one program to another. However, empirical studies have shown that one can expect about 5% of the generated mutants to be equivalent to the parent program.
- Identifying equivalent mutants is generally a manual and often time consuming--as well as frustrating--process.

Table 3 A typical equivalent mutant  
表3 一个典型的等价变异体

程序 $p$	变异体 $p'$
for (int $i = 0; i < 10; i++$ ) {	for (int $i = 0; i != 10; i++$ ) {
$sum += a[i];$	$sum += a[i];$
}	}

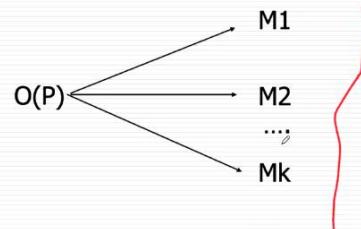


Foundations of Software Testing

现在往往有工具来生成变异体。

## Mutant operators

- A mutant operator  $O$  is a function that maps the program under test to a set of  $k$  (zero or more) mutants of  $P$ .



## Mutant operators [2]

- A mutant operator creates mutants by making simple changes in the program under test.
- For example, the “variable replacement” mutant operator replaces a variable name by another variable declared in the program. An “relational operator replacement” mutant operator replaces relational operator with another relational operator.

## Mutant operators: Examples

Mutant operator	In P	In mutant
Variable replacement	$z=x*y+1;$	$x=x*y+1;$ $z=x*x+1;$
Relational operator replacement	$\text{if } (x < y)$	$\text{if}(x>y)$ $\text{if}(x \leq y)$
Off-by-1	$z=x*y+1;$	$z=x*(y+1)+1;$ $z=(x+1)*y+1;$
Replacement by 0	$z=x*y+1;$	$z=0*y+1;$ $z=0;$
Arithmetic operator replacement	$z=x*y+1;$	$z=x*y-1;$ $z=x+y-1;$

一次变异可能发现一些低阶错误，还可以有多次变异。

## Mutants: First order and higher order

- A mutant obtained by making exactly “one change” is considered first order.
- A mutant obtained by making two changes is a second order mutant. Similarly higher order mutants can be defined. For example, a second order mutant of  $z=x+y$ ; is  $x=z+y$ ; where the variable replacement operator has been applied twice.
- In practice only first order mutants are generated for two reasons: (a) to lower the cost of testing and (b) most higher order mutants are killed by tests adequate with respect to first order mutants. [See coupling effect later.]

## Mutant operators: basis

- A mutant operator models a simple mistake that could be made by a programmer
- Several error studies have revealed that programmers--novice and experts--make simple mistakes. For example, instead of using  $x < y + 1$  one might use  $x < y$ .
- While programmers make "complex mistakes" too, mutant operators model simple mistakes. As we shall see later, the "coupling effect" explains why only simple mistakes are modeled.

变异体好坏也会形成一个评价，一个好的变异体就是可以发现更多的错误。

## Mutant operators: Goodness

- The design of mutation operators is based on guidelines and experience. It is thus evident that two groups might arrive at a different set of mutation operators for the same programming language. How should we judge whether or not that a set of mutation operators is "good enough?"
- Informal definition:
  - Let  $S_1$  and  $S_2$  denote two sets of mutation operators for language  $L$ . Based on the effectiveness criteria, we say that  $S_1$  is superior to  $S_2$  if mutants generated using  $S_1$  guarantee a larger number of errors detected over a set of erroneous programs.

## Mutant operators: Goodness [2]

- Generally one uses a small set of highly effective mutation operators rather than the complete set of operators.
- Experiments have revealed relatively small sets of mutation operators for C and Fortran. We say that one is using “constrained” or “selective” mutation when one uses this small set of mutation operators.

## Mutant operators: Language dependence

- For each programming language one develops a set of mutant operators.
- Languages differ in their syntax thereby offering opportunities for making mistakes that differ between two languages. This leads to differences in the set of mutant operators for two languages.
- Mutant operators have been developed for languages such as Fortran, C, Ada, Lisp, and Java. [See the text for a comparison of mutant operators across several languages.]

## 熟练程序员假设 (CPH)

熟练程序员假设：一般我们认为软件是由熟练程序员来编写的，基本逻辑要求是正确的。

## **Competent programmer hypothesis (CPH)**

- CPH states that given a problem statement, a programmer writes a program P that is in the general neighborhood of the set of correct programs.
- An extreme interpretation of CPH is that when asked to write a program to find the account balance, given an account number, a programmer is unlikely to write a program that deposits money into an account. Of course, while such a situation is unlikely to arise, a devious programmer might certainly write such a program.

熟练程序员只会犯一些简单的错误：

## **Competent programmer hypothesis (CPH) [2]**

- A more reasonable interpretation of the CPH is that the program written to satisfy a set of requirements will be a few mutants away from a correct program.
- The CPH assumes that the programmer knows of an algorithm to solve the problem at hand, and if not, will find one prior to writing the program.
- It is thus safe to assume that when asked to write a program to sort a list of numbers, a competent programs knows of, and makes use of, at least one sorting algorithm. Mistakes will lead to a program that can be corrected by applying one or more first order mutations.

简单的变异也能发现里面的一些问题。

## **Coupling effect**

- The coupling effect has been paraphrased by DeMillo, Lipton, and Sayward as follows: “Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors”
- Stated alternately, again in the words of DeMillo, Lipton and Sayward “..seemingly simple tests can be quite sensitive via the coupling effect.”

## Coupling effect [2]

- For some input, a non-equivalent mutant forces a slight perturbation in the state space of the program under test. This perturbation takes place at the point of mutation and has the potential of infecting the entire state of the program.
- It is during an analysis of the behavior of the mutant in relation to that of its parent that one discovers complex faults.

变异测试是可以工具化的。

## Tools for mutation testing

- As with any other type of test adequacy assessment, mutation based assessment must be done with the help of a tool.
- There are few mutation testing tools available freely. Two such tools are [Proteum](#) for C from Professor Maldonado and [muJava](#) for Java from Professor Jeff Offutt. We are not aware of any commercially available tool for mutation testing. See the textbook for a more complete listing of mutation tools.

我们要针对原程序使用变异操作生成变异操作符，然后我们针对原来的测试用例集进行测试，并且使用变异分数评价测试是否完备。

## Tools for mutation testing: Features

- A typical tool for mutation testing offers the following features
  - A selectable palette of mutation operators.
  - Management of test set T.
  - Execution of the program under test against T and saving the output for comparison against that of mutants.
  - Generation of mutants.

变异测试通常是在重要的小单元里。

## Mutation and system testing

- Adequacy assessment using mutation is often recommended only for relatively small units, e.g. a class in Java or a small collection of functions in C.
- However, given a good tool, one can use mutation to assess adequacy of system tests.
- The following procedure is recommended to assess the adequacy of system tests.

放在系统的框架下，我们的步骤如下

## Mutation and system testing [2]

- **Step 1:** Identify a set U of application units that are critical to the safe and secure functioning of the application. Repeat the following steps for each unit in U.
- **Step 2:** Select a small set of mutation operators. This selection is best guided by the operators defined by Eric Wong or Jeff Offutt. [See book for details.]
- **Step 3:** Apply the operators to the selected unit.

## Mutation and system testing [3]

- Step 4: Assess the adequacy of T using the mutants so generated. If necessary, enhance T.
- Step 5: Repeat Steps 3 and 4 for the next unit until all units have been considered.
- We have now assessed T, and perhaps enhanced it. Note the use of incremental testing and constrained mutation (i.e. use of a limited set of highly effective mutation operators).

## Mutation and system testing [4]

- Application of mutation, and other advanced test assessment and enhancement techniques, is recommended for applications that must meet stringent availability, security, safety requirements.

## Summary

- Mutation testing is the most powerful technique for the assessment and enhancement of tests.
- Mutation, as with any other test assessment technique, must be applied incrementally and with assistance from good tools.
- Identification of equivalent mutants is an undecidable problem--similar the identification of infeasible paths in control or data flow based test assessment.

2022/5/16

## 嵌入式软件测试

这节课讲嵌入式软件测试，其实嵌入式系统测试是整个软件测试的很重要的一个部分。

因为从现有的软件分类来说，有一些基本的共识，比如基础软件、平台软件、嵌入式软件等。嵌入式软件就是嵌入式系统里使用的软件，对这些软件有一些单独的测试方法。这些嵌入式终端一般来讲是嵌入式芯片上运行了嵌入式软件。

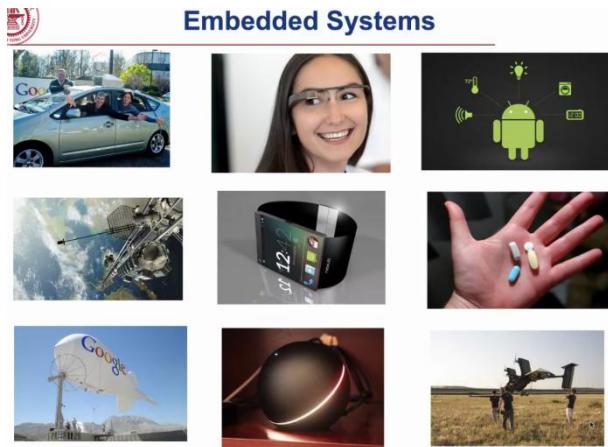
## Computing Systems

- Two types of computing systems
  - Servers, PCs, notebooks
  - Embedded systems
- The next frontier
  - Mainframe computing (60's-70's)
    - Large computers to execute big data processing applications
  - Desktop computing & Internet (80's-90's)
    - One computer at every desk to do business/personal activities
  - Embedded computing (21st Century)
    - “Invisible” part of the environment
    - Transformation of industry



- Number of microprocessor units per year
  - Millions in desktops
  - Billions in embedded processors
- Applications:
  - Automotive Systems
    - Light and heavy automobiles, trucks, buses
  - Aerospace Systems
    - Airplanes, space systems
  - Consumer electronics
    - Mobile phones, office electronics, digital appliances
  - Health/Medical Equipment
    - Patient monitoring, MRI, infusion pumps, artificial organs
  - Industrial Automation
    - Supervisory Control and Data Acquisition (SCADA) systems for chemical and power plants
    - Manufacturing systems
  - Defense
    - Source of superiority in all weapon systems

比如监测健康的一些手环等。



这是谷歌当时做的一些嵌入式系统。

硬件的发展和软件的发展是螺旋式的促进作用。软件的发展背后都有硬件的支持，很多应用只有在软硬件一体化之后才能完全发挥作用。这种情况下，整个系统的运行里如果要做测试，就要考虑实际的要求，比如嵌入式系统是通过电池供电的。



## Hardware Components

- ④ Powerful mobile processors
  - Trend: multi-processors within a single SOC
- ④ Dedicated chips for display driver, touchscreen control, GPS, bluetooth, WiFi more...
- ④ Storage/Memory
- ④ Communications/Connectivity
- ④ Various sensors
  - Accelerometer, proximity, gyro, temperature, etc.
- ④ Camera(s)
- ④ Battery

嵌入式系统也有很多的操作系统，比如 iPhone OS, Android 等。RTOS 本身是实时性的载体，需要 OS 支持实时特性，商用更多的是 VxWorks。



## Mobile OS

- ④ Apple iPhone OS
- ④ Android for Googlephones
- ④ Maemo OS for Nokia phones
- ④ Blackberry RIM OS
- ④ Windows CE (market share dropped from 23% in 2004 to 9% in 2009)
- ④ Windows Phone
- ④ RTOS, such as RT-Linux, VxWorks

.....



实时系统分为两大类：

硬实时系统：一个任务必须要知道应该在什么时间内（如：从到达完成在 2ms 内才能满足系统的要求）完成才能完成目标。

软实时系统：比如 100 个任务要求在 2ms 内完成，但是我们允许 90%的任务在 2ms 中完成，允许一部分超时的任务。比如说我们的多媒体。



## Types of Real-Time Systems

### ④ Hard real-time system

—A system where “something very bad” happens if the deadline is not met

- Examples: control systems for aircraft, nuclear reactors, chemical power plants, jet engines, etc.

### ⑤ Soft real-time system

—A system where the performance is degraded below what is generally considered acceptable if the deadline is missed

- Example: multimedia system

嵌入式系统中，系统是按照时间特性来运行的。里面有一些指标。WCET（最差执行时间）这个指标是很重要的。我们要测试这个WCET的时候，每一次测都不一样，因为影响时间变化的有很多，所以我们要多次测试以后进行估计。



## Time is Central to Embedded Systems

- ④ Worst-case execution time (WCET) estimation
- ④ Threshold property: can you produce a test case that causes a program to violate its deadline?
- ④ Software-in-the-loop simulation: predict execution time of particular program path

ALL involve predicting an execution time property!

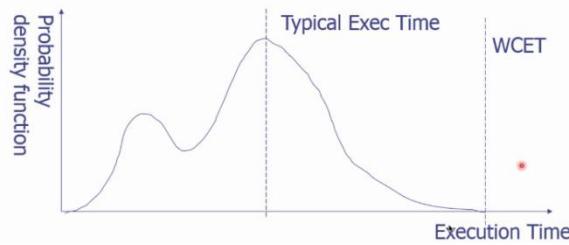
从系统任务的执行时间来看，有系统时间的执行影响。不同OS下的表现情况也不一样。

WCET就是概率密度函数为0的时候的位置。Typical Exec Time就是平均时间。我们需要比较准确地分析最坏情况下的时间。但是分析这个时间是很麻烦的，会遇到很多问题。



## Worst-Case Execution Time (WCET) of a Task

- ④ The longest time taken by a software task to execute->Function of input data and environment conditions
- ④ BCET = Best-Case Execution Time (shortest time taken by the task to execute)



平台由硬件和操作系统组成。嵌入式系统里最重要的就是实时性。

# 计算 Worst-Case Execution Time



## The WCET Problem

### Given

- ④ the code for a software task
- ④ the platform (OS + hardware) that it will run on

Determine the WCET of the task.

- ④ Why is this problem important?

The WCET is central in the design of RT Systems:  
Needed for Correctness (does the task finish in time?) and Performance (find optimal schedule for tasks)

我们根据如下的这个例子看看怎么进行测试。整个程序就是模幂算法。我们通过实时系统去做测试的时候，要测它最长的执行时间，要怎么去做呢？我们首先要把程序建模，否定分析起来很难。直接测的方法是很难遇到最坏情况下的时间的（概率很低）。在这个过程中，我们以程序流图的方式去分析。



## Programs as Graphs

```
1 #define EXP_BITS 32
2
3 typedef unsigned int UI;
4
5 UI modexp(UI base, UI exponent, UI mod) {
6     int i;
7     UI result = 1;
8
9     i = EXP_BITS;
10    while(i > 0) {
11        if ((exponent & 1) == 1) {
12            result = (result * base) % mod;
13        }
14        exponent >>= 1;
15        base = (base * base) % mod;
16        i--;
17    }
18    return result;
19 }
```

基本块很固定，没有控制流的流入和流出，所以直接测基本块的时间是很准的。

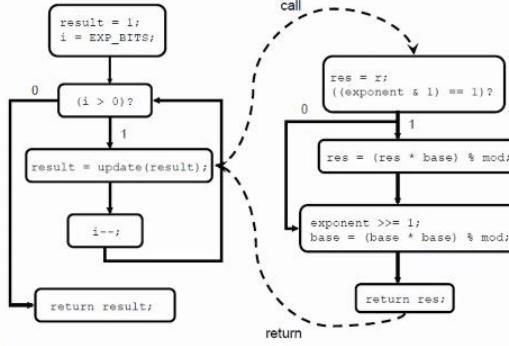


## Function Calls

```

1 #define EXP_BITS 32
2 typedef unsigned int UI;
3 UI exponent, base, mod;
4
5 UI update(UI r) {
6     UI res = r;
7     if ((exponent & 1) == 1) {
8         res = (res * base) % mod;
9     }
10    exponent >>= 1;
11    base = (base * base) % mod;
12    return res;
13 }
14
15 UI modexp_call() {
16     UI result = 1; int i;
17     i = EXP_BITS;
18     while(i > 0) {
19         result = update(result);
20         i--;
21     }
22     return result;
23 }

```



Q: 哪些因素会影响执行时间的变化呢？

A: 比如循环的次数、路径的空间等。



## Factors Determining Execution Time

### ④ Exponential Path Space

**Example 15.6:** Consider the function count listed below, which runs over a two-dimensional array, counting and accumulating non-negative and negative elements of the array separately.

```

1 #define MAXSIZE 100
2
3 int Array[MAXSIZE][MAXSIZE];
4 int Ptotal, Pcnt, Ntotal, Ncnt;
5 ...
6 void count() {
7     int Outer, Inner;
8     for (Outer = 0; Outer < MAXSIZE; Outer++) {
9         for (Inner = 0; Inner < MAXSIZE; Inner++) {
10             if (Array[Outer][Inner] >= 0) {
11                 Ptotal += Array[Outer][Inner];
12                 Pcnt++;
13             } else {
14                 Ntotal += Array[Outer][Inner];
15                 Ncnt++;
16             }
17         }
18     }
19 }

```

还有一些拓扑不可达的路径，也会影响执行时间。



## Factors Determining Execution Time

### Infeasible Paths

**Example 15.7:** Recall Example 12.3 of a software task from the open source Paparazzi unmanned aerial vehicle (UAV) project (Nemer et al., 2006):

```

1 #define PPRZ_MODE_AUTO2 2
2 #define PPRZ_MODE_HOME 3
3 #define VERTICAL_MODE_AUTO_ALT 3
4 #define CLIMB_MAX 1.0
5 ...
6 void altitude_control_task(void) {
7     if (pprz_mode == PPRZ_MODE_AUTO2
8         || pprz_mode == PPRZ_MODE_HOME) {
9         if (vertical_mode == VERTICAL_MODE_AUTO_ALT) {
10            float err = estimator_z - desired_altitude;
11            desired_climb
12                = pre_climb + altitude_pgain * err;
13            if (desired_climb < -CLIMB_MAX) {
14                desired_climb = -CLIMB_MAX;
15            }
16            if (desired_climb > CLIMB_MAX) {
17                desired_climb = CLIMB_MAX;
18            }
19        }
20    }
21 }
```

Cache 也会带来运行时间的影响。命中与否对执行时间的变化非常大。

### Processor Behavior Analysis: Cache Effects

```

1 float dot_product(float *x, float *y, int n) {
2     float result = 0.0;
3     int i;
4     for(i=0; i < n; i++) {
5         result += x[i] * y[i];
6     }
7     return result;
8 }
```

Suppose:

1. 32-bit processor
2. Direct-mapped cache holds two sets
  - 4 floats per set
  - x and y stored contiguously starting at address 0x0

What happens when **n=2?**

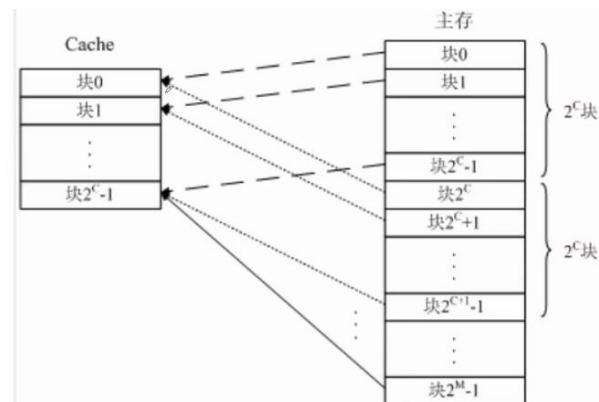


图 3 Cache 概念的回顾

# Some WCET Estimation Tools

Commercial Tools: aiT, RapiTime, ...

University/Research Tools: **GameTime**, Chronos, ...



有了各个基本块的时间之后，我们估计整个图结构里的时间，这样可以考虑进来循环次数和不同的路径空间。



## 软件测试技术——黑盒测试（1）

### ④ 黑盒测试的概念

- 若被测程序与特定的功能相联系，我们可以针对功能设计测试，以证实各功能完全可执行，同时在功能中寻找错误
- 把测试对象看做一个黑盒子，完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否正确。
- 黑盒测试又叫做功能测试或数据驱动测试。
- 黑盒测试类似于中医：使用望、闻、问、切

### ⑤ 黑盒测试的目的

- 是否有错误的或遗漏了的功能？
- 在接口上，输入能否正确地接受？能否输出正确的结果？
- 是否有数据结构错误或外部信息访问错误？
- 是否考虑了软件的出错情况？

黑盒测试完全是在嵌入式系统里用的。



## 软件测试技术——黑盒测试（2）

### ⑥ 黑盒测试的一个常用技巧——打桩（Stub）

- 用在隔离测试中，用以消除其他程序对被测程序的影响
- 当被测程序调用其子模块时，可以使用模拟法，此时被测程序并没有真正调用其他模块，而是从桩模块处得到一个所需的值。这种技术叫做打桩。

### ⑦ 黑盒测试常用方法

- 等价类测试
- 边界测试
- 等等

一个复杂的系统的模块调用是很麻烦的，比如一个系统调用另一个系统的时候，就可以用到打桩。用一个表和值去代替调用块的值，我们只是关系被测试的模块的输入输出的关系。



## 软件测试技术——黑盒测试（3）

- ④ 常用黑盒测试方法——等价分割（测试）
  - 等价分割是一种典型的黑盒测试方法
  - 等价分割将输入输出范围分割成多个等价的区域，然后从每个等价区中选取少数有代表性的数据做为测试用例。所谓等价，是指：
    - 在同一分割区内程序的行为是等价的
    - 等价区之间的依赖性可以忽略
  - 输入和输出并非只是程序参数：
    - 外部数据
    - 时间
    - 执行顺序/记录
    - 状态

等价类测试需要我们自己根据等价类去划分。



## 软件测试技术——黑盒测试（4）

- ④ 常用黑盒测试方法——边界值分析（测试）
  - 边界值分析也是一种黑盒测试方法，是对等价分割方法的补充
  - 经验告诉我们，大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。因此针对各种边界情况设计测试用例，可以查出更多的错误
  - 使用边界值分析方法设计测试用例，首先应确定边界情况。应当选取正好等于，刚刚大于，或刚刚小于边界的值做为测试数据，而不是选取等价类中的典型值或任意值做为测试数据

错误聚集在边界上，缺陷隐藏在角落里！

边界值测试也是 ok 的。



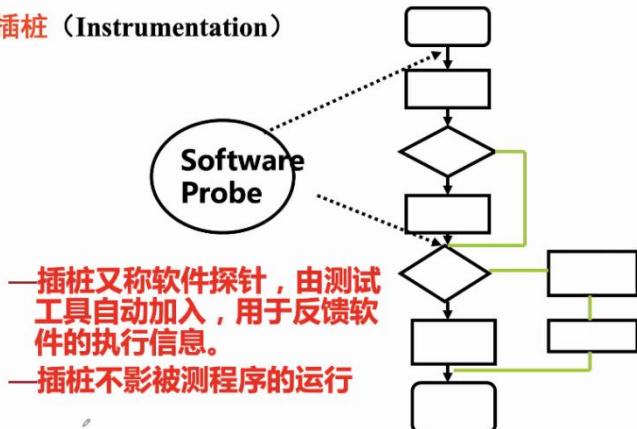
## 软件测试技术——白盒测试（1）

- ④ 什么是白盒测试
  - 若被测程序与特定的结构相联系，我们可以针对结构设计测试，以确保内部的“所有齿轮相吻合”，即软件的内部过程是合理的，是遵照规定执行的
  - 白盒测试又称玻璃盒测试。把测试对象看作一个透明的盒子，充分利用其逻辑结构和有关信息，设计和选择测试用例。
  - 白盒测试又称为结构测试或逻辑驱动测试
  - 白盒测试类似于西医看病，通过X光、CT扫描等手段
- ④ 白盒测试主要用于
  - 结构测试
  - 覆盖测试
  - 静态分析也可以算作白盒测试的一种手段



## 软件测试技术——白盒测试（2）

### ④ 插桩（Instrumentation）



- 插桩又称软件探针，由测试工具自动加入，用于反馈软件的执行信息。
- 插桩不影被测程序的运行

白盒测试里也会加一些探针来实现插桩。



## 软件测试技术——白盒测试（4）

- ④ 代码覆盖 |
- ④ 分支覆盖 (判定覆盖)
- ④ 条件覆盖
- ④ 路径覆盖

还有一些覆盖率的方法。工作以后，因为代码覆盖最简单效率最高，所以我们经常会去做。



## 软件测试技术——回归测试

### ④ 回归测试

- 软件更动后重新进行的测试
  - + 既要测试变更部分，也要测试受影响部分
- 关键在决定哪些测试必须被重复
- 保证测试工作的可重现性
- 尽量利用工具，提供回归测试的自动化水平





## 软件测试技术——静态分析（2）

### ④ 静态分析的主要方法和内容：

—**代码审查，这是软件质量保证（SQA）的重要内容**

—**代码统计分析**

—**软件工程度量**

- McCabe 复杂度
- Halstead 复杂度
- ...



## 嵌入式软件测试方法（1）

### ⑤ 嵌入式软件测式的基本方法

—**拿来主义——充分借用PC软件的测试方法**

• 静态/动态

• 白盒/黑盒

• 单元测试/集成测试/系统测试

• .....

—**全数字模拟测试**

—**交叉测试（Host/Target测试）**

—**真实环境验证**

这些都可以用到嵌入式系统设计里面。



## 嵌入式软件测试方法（2）

### ⑥ 全数字模拟测试

—采用数学平台的方法，将嵌入式软件从系统中剥离出来，通过开发CPU指令、常用芯片、I/O、中断、时钟等模拟器在HOST上实现嵌入式软件的测试

#### —主要特点

- 与嵌入式硬件平台脱钩
- 操作简单，可以借鉴常规的软件测试方法
- 适用于功能测试
- 有局限性



硬件平台没有的情况下，我们就模拟硬件平台的指令集、IO、中断等。



## 嵌入式软件测试方法 (4)

### ④ 交叉测试 (Host/Target测试)

—与目标环境无关的部分在PC机上完成

- 充分利用高级语言的可移植性
- 借鉴常规的软件测试方法
- 与模拟测试不同

—与硬件密切相关的部分在Target上完成

- 需要调试环境支持
- 测试工具需要支持目标环境

—最后在目标环境中确认



## 嵌入式软件测试方法 (6)

### ⑤ 如何开展交叉测试 (Host/Target测试)

- 选用带有**目标支持包(Target Package)**的软件测试工具
- 确定哪些模块与硬件无关，哪些与硬件相关
- 配置相应的调试环境和目标环境
- 分别进行Host和Target测试
  - **Host**: 源代码+测试用例->编译连接->执行->测试结果
  - **Target**: 源代码+测试用例+**目标包**->编译连接->**下载**->执行->反馈测试结果

### ⑥ 交叉测试的嵌入式调试环境

—Simulator

—Emulator

- BDM/JTAG Emulator

12



## 嵌入式软件测试方法 (7)

### ⑦ “拿来主义”

—静态分析很重要

Watts S. Humphrey的说法

- 很多软件工程师认为动态测试比静态测试更重要——并非如此
- 有经验的软件工程师平均每写1000行代码将会出现100个错误
- 80%的软件错误归咎于对于编写语言的错误使用，而这些错误往往不是功能测试能解决的
- 因此，软件工程师应该消除错误，找出根源，预防再次发生同样的问题

—静态分析的重要内容——代码规则检查

- 实施简单、方便
  - 无需执行程序，与嵌入式环境无关
- 早期介入，代价小，见效快
- 有利于降低动态测试的难度
- 有利于养成良好的编程习惯
- 可以执行自定的规范



## 嵌入式软件测试策略（2）

### ④ 质量分析，事半功倍

通过改善代码的结构来分析、改进软件的质量——充分应用结构化  
测试技术

—**软件模块的圈复杂度和逻辑结构能客观地反映软件的质量**

- 逻辑越“复杂”，就越容易出错。
- 结构越“良好”，代码就越可靠

—**代码质量分析的好方法——结构化测试**

□从结构入手分析代码的复杂程度

□逻辑复杂度量化

□客观

□有理论基础

• 复杂度与代码出错的关联性非常强

• 指导测试的执行

• 指出代码质量改进的方向



## 嵌入式常用软件测试工具介绍（8）

### ④ Klocwork——软件缺陷检测工具

加拿大Klocwork公司出品

- 能够分析C、C++、C#和.NET等语言的源代码
- 能够发现的软件缺陷种类：
  - 软件质量缺陷
  - 安全漏洞方面的缺陷
  - 对软件架构、编程缺陷的违反情况
- 对软件进行可视化的架构分析、优化
- 对软件进行各种度量分析
- 能够提供与多种主流IDE开发环境的集成
- 能够分析上千万代码行的超大型软件



**Klocwork**

