南京大学
NANJING UNIVERSITY

# 《软件工程与计算II》

## ch19 软件测试

# 主要内容

- V&V基础
- 软件测试基础
- 软件测试的层次
- 软件测试技术
- 软件测试活动
- 软件测试的度量


# Software Correctness

- Software Correctness
  - A program P is considered *with respect to a specification* S, if and only if:
    - For each valid input, the output of P is in accordance with the specification S
- Software is never correct no matter which developing technique is used
- Any software must be validated and verified
  - Verification and Validation


# V&V：你生产的产品正确吗？

- 确认（Verification）：产品是否被正确地建立，判断标准是：产品的表现与规格是否一致。
  - 模块表现是否符合模块规格
  - 类的表现是否符合类的规格
  - 方法的表现是否符合方法接口规格
  - ……
- 验证（Validation）：所建立的产品是“正确”的那个吗？判断标准是：产品表现是否符合需求的要求
  - 设计是否符合需求？
  - 代码是否符合需求？
  - 测试是否符合需求？
  - ……

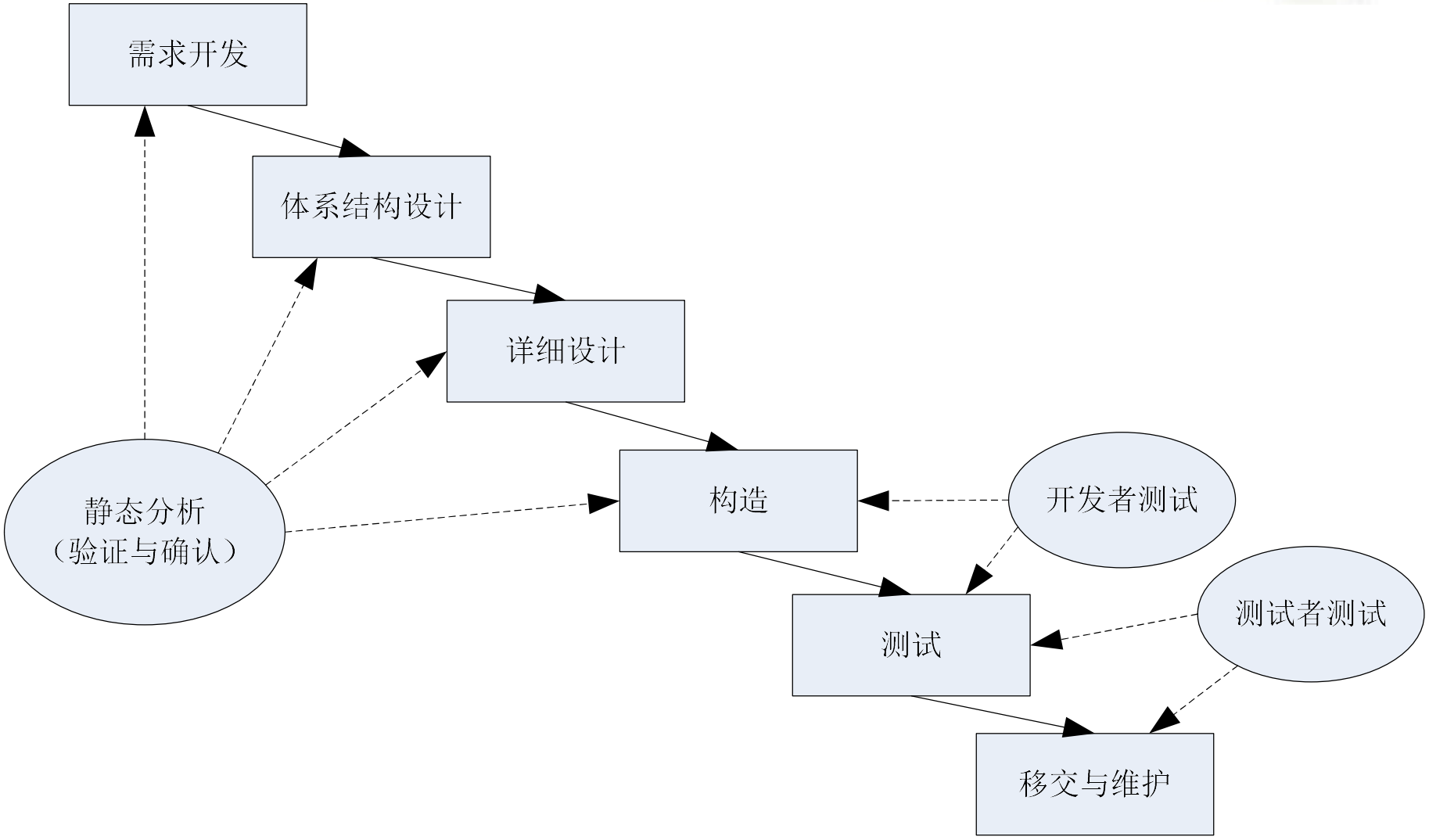
# 如果产品不正确… Bug?

- 缺陷（Defect/Fault故障）：系统代码中存在的不正确的地方，例如计算时存在除0可能。
- 错误（Error）：如果系统执行到缺陷代码，就可能使得执行结果不符合预期且无法预测，表现出来不稳定状态就称为错误。例如对计算时存在除0可能的代码，一旦执行了除0操作，就会发生错误。
- 失败（Failure）：错误的发生会使得软件的功能失效，比如，系统某个功能输出不正确、异常终止、不符合时间或者空间的限制等。
- 缺陷→错误→失败
  - 缺陷会一直存在，直到环境条件使得软件执行到了代码缺陷的地方，软件的运行就会出现错误，这个错误反映到整个系统以及与外界的交互上，就是失败。

# V&V的主要方法

- 静态分析
  - 不依赖于软件的动态运行
  - 依据开发文档、模型等制品
  - 评审是最有效的静态分析方法
- （动态）测试
  - 检查软件的动态运行情况

# 软件开发中的验证与确认活动



# 主要内容

- V&V基础
- 软件测试基础
- 软件测试的层次
- 软件测试技术
- 软件测试活动
- 软件测试的度量

# Software Testing

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
  - – Def: The *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior
  - Test Case: set of inputs and expected output
- Objective: to find errors
- Can detect the presence of defects, but not their absence

# 软件测试的目的

- 向开发者和用户展示软件满足了需求，表明软件产品是一个合格的产品；
  - 有效性测试
  - 1960s出现-（1970s-1980s）主流-1990s later稳定
- 找出软件中的缺陷和不足。
  - 缺陷测试
  - 1990s之后主流

哪一个目的更重要?

# 只有发现了缺陷的测试才是成功的测试!

- 缺陷测试具有更大的重要性
  - 经验表明，短期内生产者的水平比较稳定，如果这些生产者之前生产的产品中每1KLOC中有N个错误，那么新产品中每1KLOC也就应该有N个错误，测试的目的就是尽力找出这N个错误
  - 软件测试不可能找出所有的缺陷，只要经过测试的软件仍存的缺陷数少于之前产品的历史数据，测试就是成功的。
    - 仍存的缺陷数需要在维护期间采集，或者根据预植入缺陷的发现情况进行度量

# 软件测试的目的

## 软件测试的目标或定义

测试是为了发现程序中的错误而执行程序的过程

- 一个好的测试用例在于能发现至今未发现的错误
- 一次成功的测试是发现了至今未发现的错误的测试

# 软件测试 VS 产品质量

- 与历史数据比，软件测试发现缺陷较少未必是因为软件质量较好，也有可能是因为软件测试比较失败（例如测试者没有尽力）
  - 需要回归审视整个测试工作是否存在瑕疵
- 软件测试发现的缺陷较多未必是因为软件质量不好，也有可能是因为软件测试过于成功（例如测试者表现超出历史水平）
  - 需要综合审视发现的缺陷情况和测试工作

# 应该由谁来执行测试?

*开发者*

更倾向于“有效性测试”

开发者的成就感来源于开发工作——产品符合需求且没有缺陷

*独立的测试者*

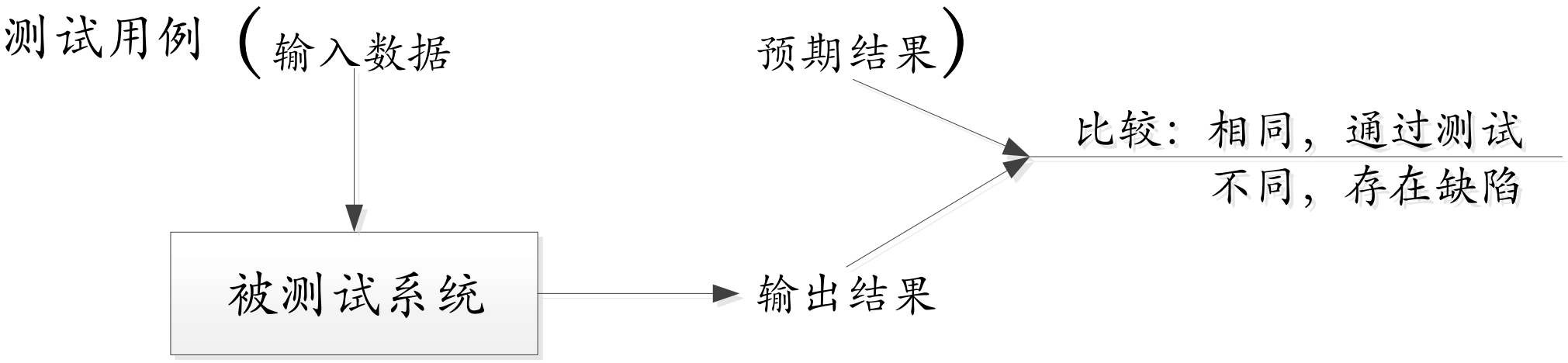
更倾向于“缺陷测试”

测试者的成就感来源于测试工作——成功地将软件缺陷找了出来!

软件测试阶段需要独立的测试者!

# 测试用例

- 软件测试的中心工作就是设计有效的输入数据，观察软件运行结果，并与预期结果进行比较



# 测试工作需要受控的环境

## 桩与驱动

驱动程序

- 设置环境
- 输入测试用例的输入数据
- 得到被测试部件的输出结果
- 将输出结果与预期结果比较
- 给出测试用例执行结论

被测试部件

桩程序

- 接收被测试部件请求
- 按照设定规则向被测部件返回结果

# 主要内容

- V&V基础
- 软件测试基础
- 软件测试的层次
- 软件测试技术
- 软件测试活动
- 软件测试的度量

# 依据测试对象划分层次 [SWEBOK2004]

| 类别 | 描述 |
| --- | --- |
| 单元测试 | 验证独立软件片段的功能，软件片段可以是单个的子程序、或者是由紧密联系的单元组成的较大的组件。 |
| 集成测试 | 验证软件组件之间的交互。 |
| 系统测试 | 关注整个系统的行为，评价系统功能需求和非功能性需求，也评价系统与外界环境（例如其它应用、硬件设备等）的交互。 |

# 依据测试目标划分层次[SWEBOK2004]

| 类别 | 描述 |
| --- | --- |
| 验收测试 | 按照客户的需求检查系统。这个测试活动可能需要开发人员的参与，也可能不需要他们的参与。 |
| 安装测试 | 在目标环境中通过安装来验证软件。 |
| α与β测试 | 在软件发布前，让小规模、有代表性的潜在用户试用，可以在开发机构中进行（α测试），也可在用户处进行（β测试）。通常，α与β测试不需要控制。 |
| 性能测试 | 特别针对性能需求验证软件。 |
| 易用性测试 | 评价终端用户学习和使用软件（包括用户文档）的难易程度、软件功能在支持用户任务的有效程度、从用户的错误中恢复的能力。 |
| 可靠性测试 | 验证和评价系统可靠性的测试。 |

# 依据测试目标划分层次[SWEBOK2004]

| 类别 | 描述 |
| --- | --- |
| 安全测试 | 验证系统内的安全机制保护系统不受非法入侵的能力。 |
| 恢复测试 | 验证软件在“灾难”后的重启动能力。 |
| 压力测试 | 以设计的最大负载运行软件，并以超过最大负载运行软件，验证软件的负载能力。 |
| 配置测试 | 分析软件在规格说明的不同配置下的行为。 |
| 回归测试 | 在变更系统后进行，重新执行已经测试过的测试用例子集，以确保变更没有造成未预期的副作用。每个测试级别的测试（不论何种测试对象、功能还是非功能需求）都可以进行回归测试。 |


# 单元测试

## 单元测试的几点说明

- 单元测试集中检验软件设计中最小单元--模块。
- 根据详细设计的说明，应测试重要的控制路径，力求在模块范围内发现错误。
- 由于测试范围有限，测试不会太复杂，所能发现的错误也是有限的。
- 单元测试总是采用白箱测试方法，而且可以多个模块并行进行。

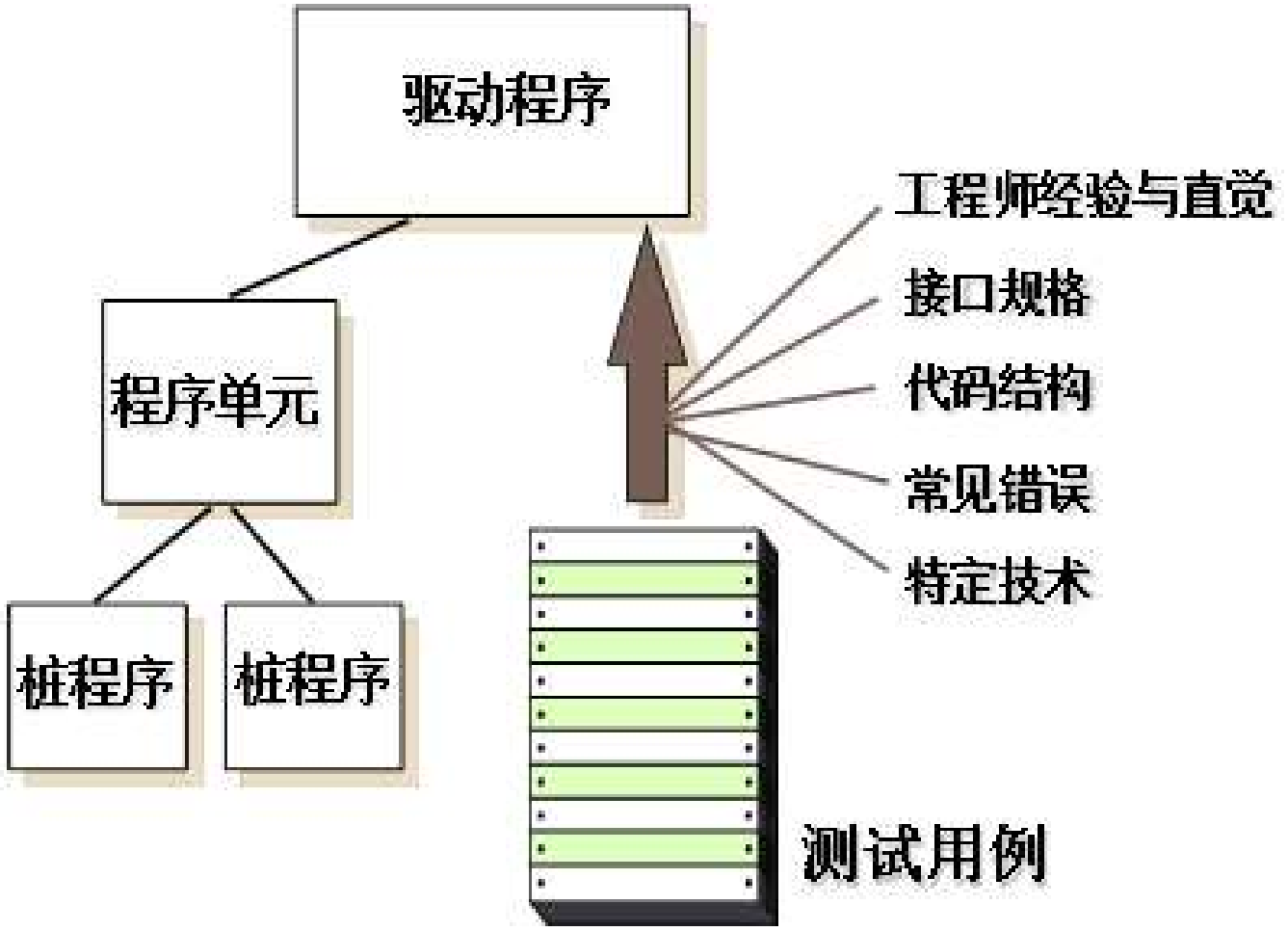
## 单元测试的内容

- 模块接口
- 局部数据结构
- “重要的”运行路径
- 出错处理路径
- 影响以上各点的边界条件

## 单元测试的过程

- 单元测试和编码属于软件开发的同一阶段。
- 模块并不是独立的程序，必须为每个单元测试开发驱动模块和承接模块。
- 驱动模块：是一个“主程序”。接收测试数据，把这些数据传送给被测模块，并且打印结果。
- 承接模块：用以代替被测试模块所调用的模块，采用被代替模块的接口，可以做很少的数据处理，打印数据，并把控制返回给被测模块。
- 驱动模块和承接模块是一种额外的开销，也就是说，它们都是必须编写的软件，但却不作为最终的软件产品提供给用户。

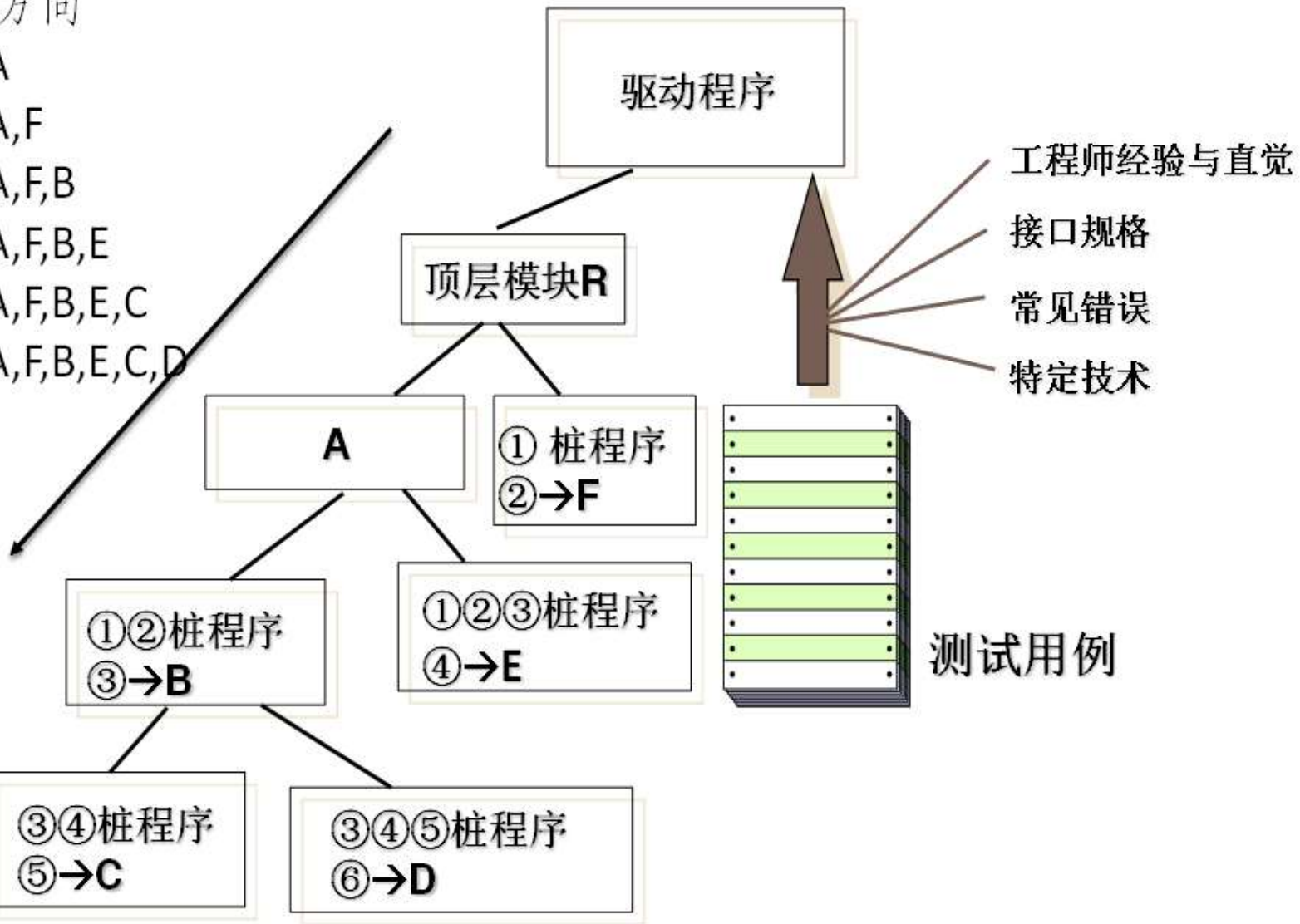
# 单元测试



# 集成测试

## 自顶向下

集成方向
①R,A
②R,A,F
③R,A,F,B
④R,A,F,B,E
⑤R,A,F,B,E,C
⑥R,A,F,B,E,C,D

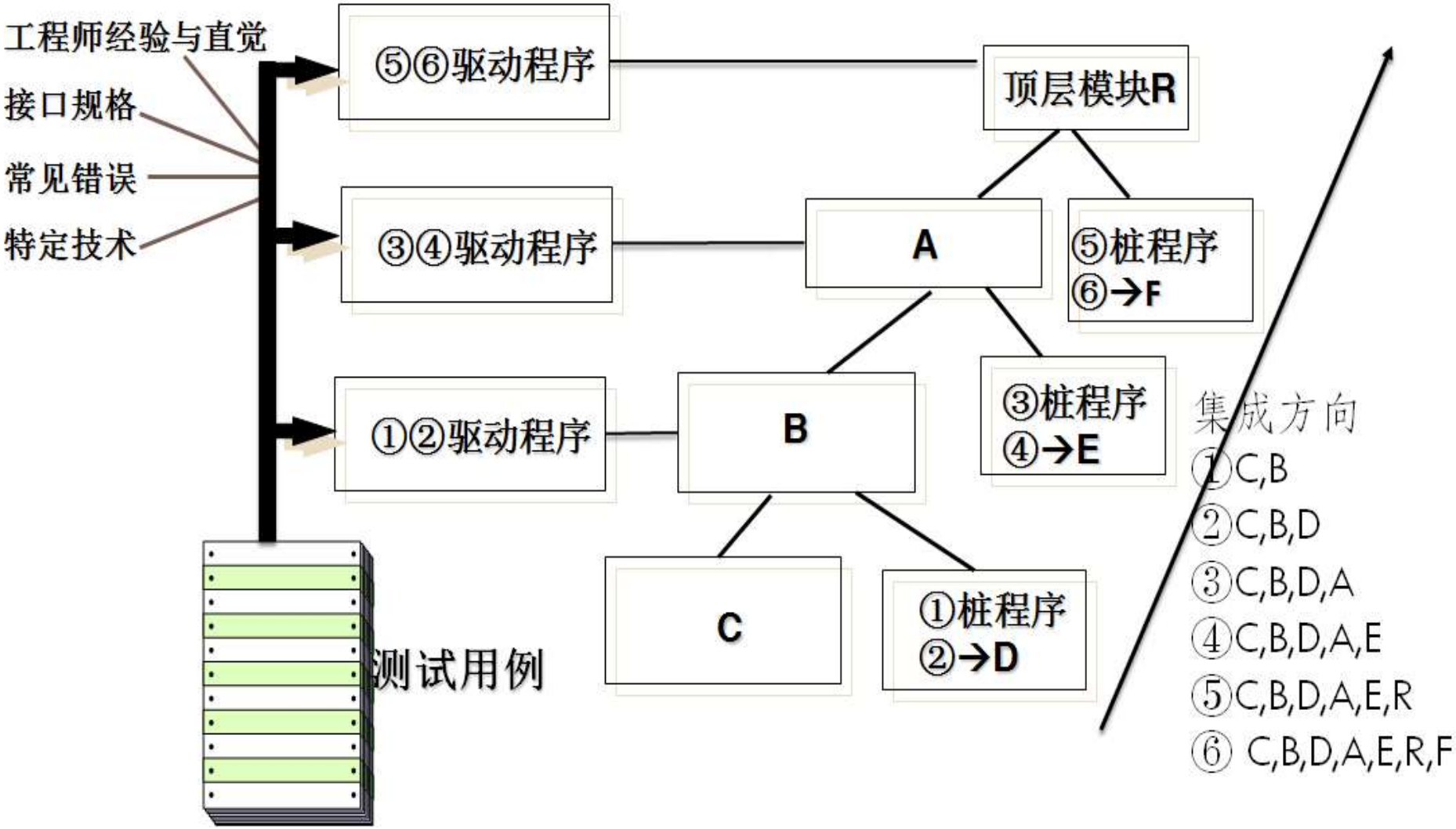


# 集成测试

- 在把经过单元测试的模块按照设计要求组装起来的过程中进行测试，重点查找与接口有关的错误。
- 与接口有关的错误包括：
  - 数据穿过接口时可能丢失；
  - 某个模块对另一个模块可能产生不利的影响；
  - 把子功能组合起来可能不产生预期的主功能；
  - 个别原可以接受的误差可能累计到不能接受的程度；
  - 全程数据结构可能有问题。

# 集成测试

## 自底向上


工程师经验与直觉
接口规格
常见错误
特定技术
⑤⑥驱动程序
③④驱动程序
①②驱动程序
测试用例
顶层模块R
A
B
C
⑤桩程序
⑥→F
③桩程序
④→E
①桩程序
②→D
集成方向
①C,B
②C,B,D
③C,B,D,A
④C,B,D,A,E
⑤C,B,D,A,E,R
⑥ C,B,D,A,E,R,F

## 集成测试方法比较

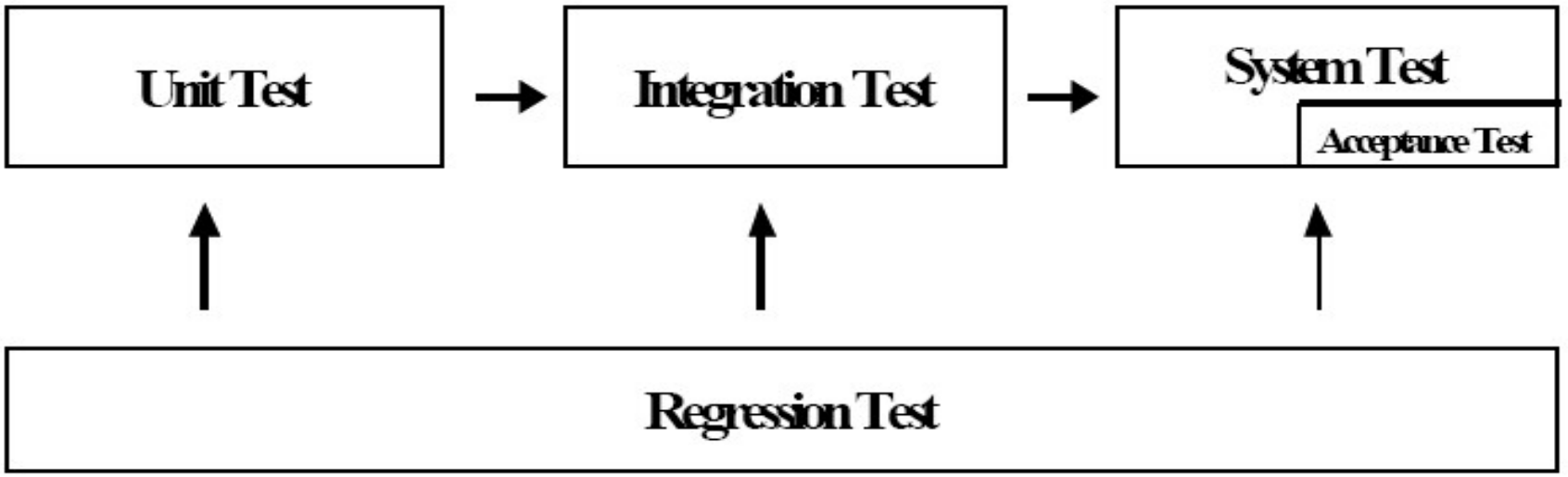
- 自顶向下组装法：不需要驱动模块；能够在测试阶段早期验证系统的主要功能，早期发现接口错误；需要较多的承接模块，可能遇到与之相联系的测试困难；低层关键模块中的错误发现较晚。
- 自底向上组装法：不需要承接模块；需要较多的驱动模块；在最后一个模块装入之前，软件实体并不存在。
- 一般使用两种方法的结合，测试阶段早期采用自顶向下组装法，后期采用自底向上组装法，很多情况下二者可以同时进行。

# 系统测试

- 单元测试、集成测试更加关注技术上的正确性，重在发现设计缺陷和代码缺陷。系统测试则不同，它更关注不符合需求的缺陷和需求自身的内在缺陷。
- 根据测试目标的不同，有很多不同类型的系统测试：功能测试、非功能性测试、验收测试、安装测试等等。但是发生在软件测试阶段，完全由软件测试人员控制和执行的主要是功能测试和非功能性测试。
- 系统测试关注整个系统的行为，所以不依赖于桩程序和驱动程序。但是，使用一些测试工具可以让系统测试过程更加自动化。
- 系统测试的功能测试计划以需求规格说明文档或用例文档为基础，主要使用随机测试和基于规格的测试技术设计功能测试用例。在测试非功能性需求时需要使用针对非功能需求的特定测试技术进行测试计划和测试用例设计。

# Testing Level ---Regression Testing
## 回归测试

*regression test* is not a separate level of but may refer to the retesting of a unit, integration and system after modification, in order to ascertain that the change has not introduced new faults




# An Example of Test Oracle

| RE | Software Architecture | Detail Design | Unit Coding | Integration & Delivery | Maintenance |
|---|---|---|---|---|---|
| • SRS Inspections<br>• Acceptance Test planning | • Architecture Inspections<br>• Generate Oracle<br>• Integration Test planning | Design inspections<br>Generate oracles<br>Unit test planning<br>Automated design analyses | Code inspections<br>Create scaffolding<br>Unit test execution<br>Automated code analyses<br>Coverage analysis | Integration test execution<br>System test execution<br>Acceptance test execution<br>Deliver regression test suite | Regression test execution<br>Revise regression test suite |

This method—a function that determines if the application has behaved correctly in response to a test action—is called a test oracle.

# 主要内容

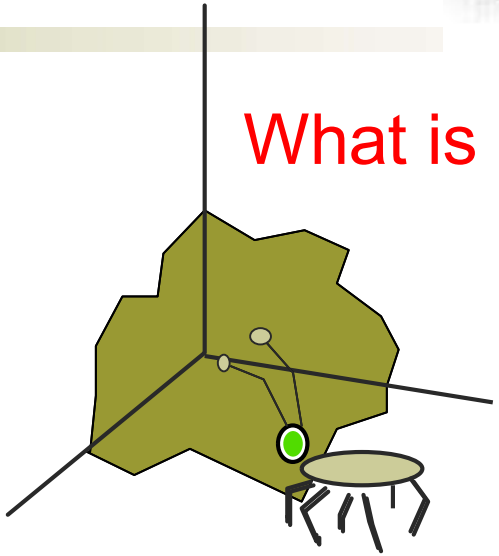
- V&V基础
- 软件测试基础
- 软件测试的层次
- 软件测试技术
- 软件测试活动
- 软件测试的度量

# Test Case Design

What is a “Good” Test?

**"Bugs lurk in corners and congregate at boundaries ..."**

***Boris Beizer***

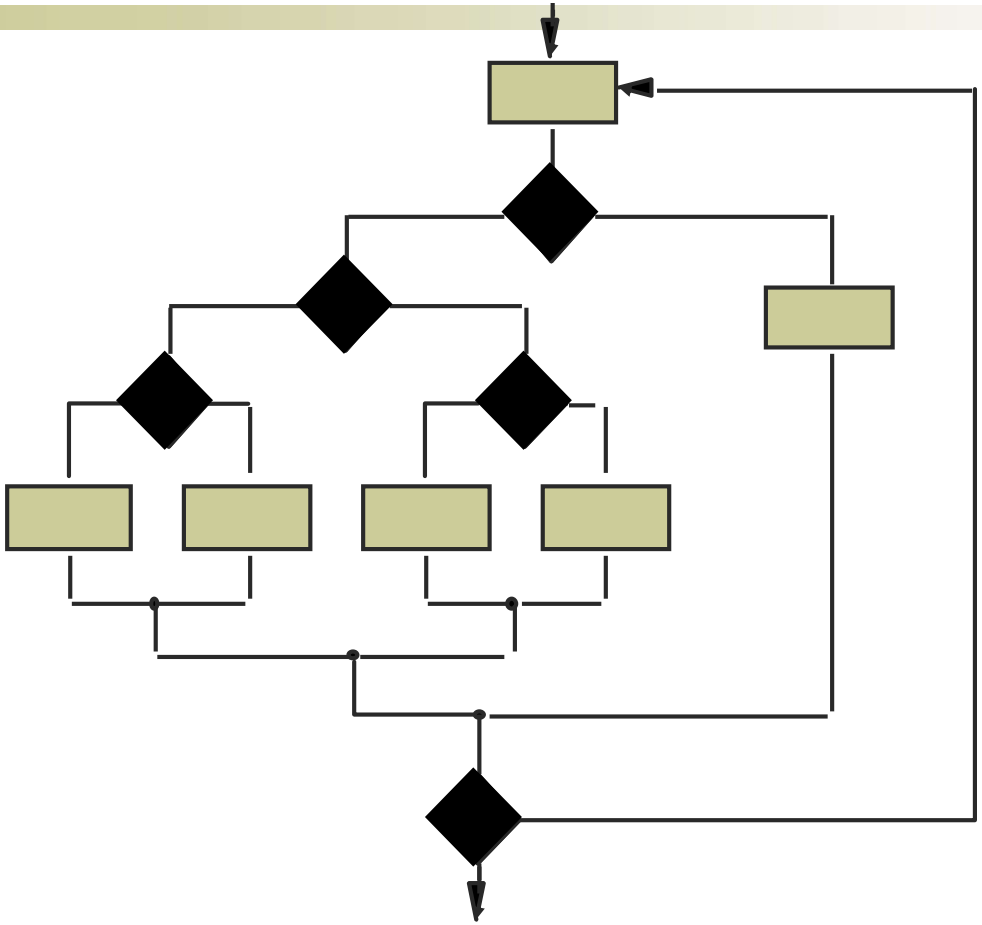


***OBJECTIVE*** **to uncover errors**

***CRITERIA*** **in a complete manner**

***CONSTRAINT*** **with a minimum of effort and time**

# Exhaustive Testing

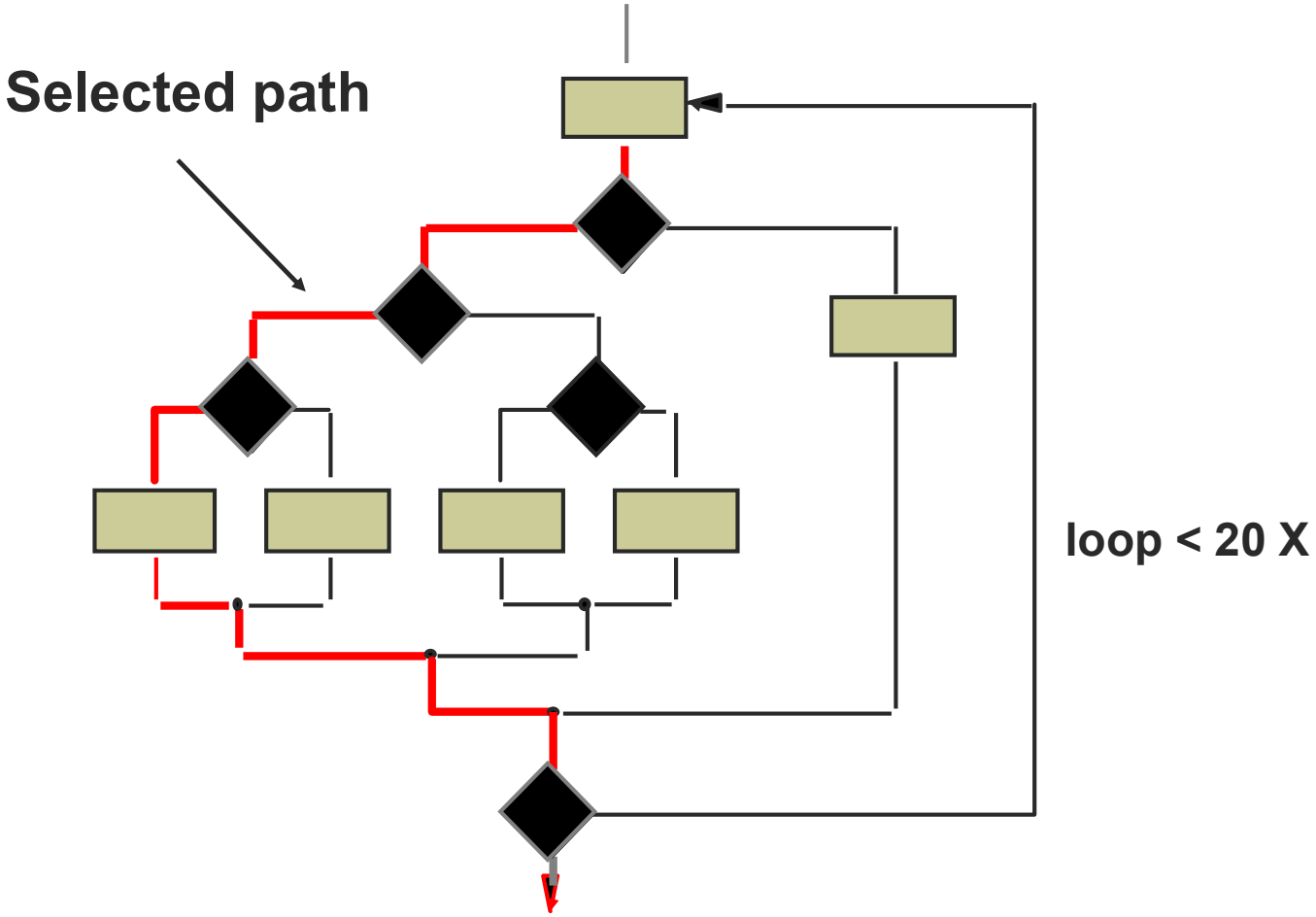



**There are $10^{14}$ possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!**

# 测试技术的作用

- 测试用例的选择：用最少的用例发现最多的缺陷
  - 测试是有代价的，要耗费桩程序、驱动、人力等成本，测试成本随着测试用例数量增多会直线上升
  - 测试并不要求发现所有缺陷，这是成本不允许的
    - 除非使用代码分析器，否则被测试的代码不超过60%
  - 测试技术就是帮助设计和选择测试用例的
    - 在同等的成本（测试用例数量）下，发现的缺陷越多越好

# Selective Testing



# 常见测试技术

- 随机测试
- 基于规格的技术 - 黑盒测试方法
- 基于代码的技术 - 白盒测试方法
- 特定测试技术

# 随机测试

- 随机测试（ 随机测试Ad hoc Testing）是一种基于软件工程师直觉和经验的技术，也许是实践中使用最为广泛的测试技术 [SWEBOK2004] 。

- 随机测试根据软件工程师的技能、直觉和对类似程序的经验[Myers1979]，从所有可能的输入值中选择输入子集，建立测试用例。

# 函数int add(int x, int y)的测试用例

| ID | 输入 | | 预期输出 |
|---|---|---|---|
| 1 | x=100 | y=20 | 120 |
| 2 | x= 2147483640 | y=100 | 超越最大值异常 |

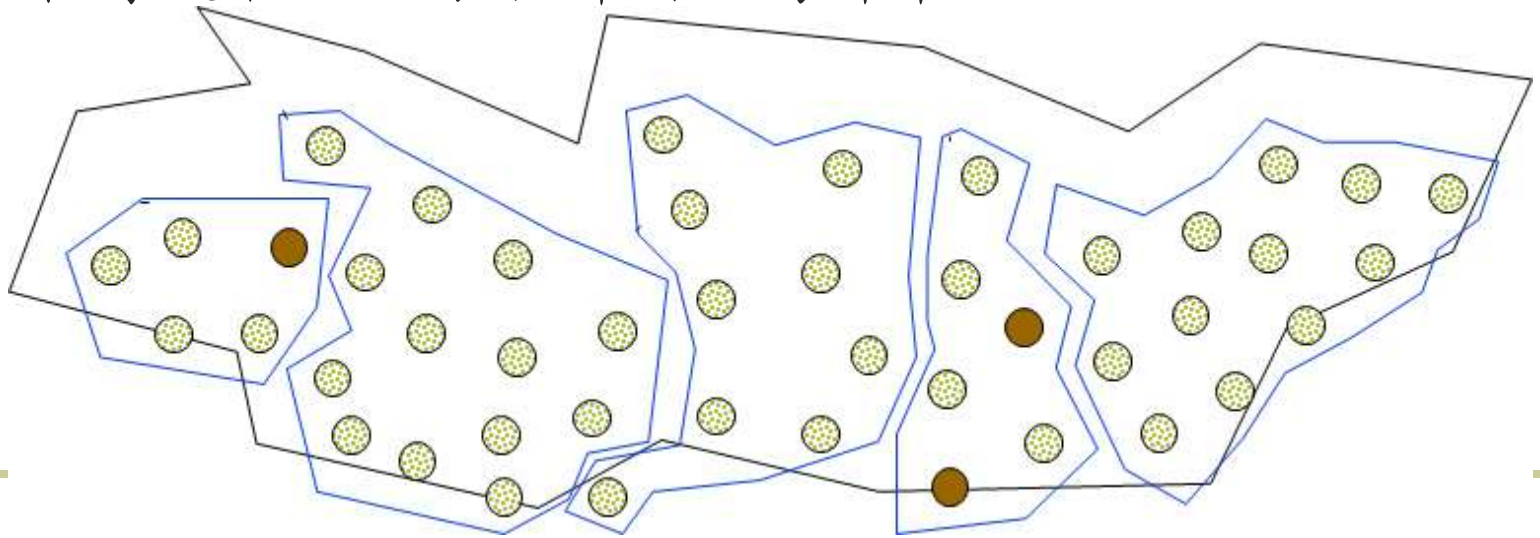
经验

# 黑盒测试

- 功能测试根据软件所需的功能和所实现的功能选择测试数据，分析测试的充分性。
- 把测试对象看作一个黑盒子，完全基于输入和输出数据来判定测试对象的正确性。
  - 等价类划分
  - 边界值分析
  - 基于决策表的方法
  - 基于状态转换的方法
- 测试使用测试对象的规格说明来设计输入和预期输出

# 黑盒测试方法-等价类划分

- 等价类是指某个输入域的子集合。在该子集合中，各个输入数据对于揭露程序中的错误都是等效的
  - 有效等价类：是指对于程序的规格说明来说是合理的、有意义的输入数据构成的集合。利用有效等价类可检验程序是否实现了规格说明中所规定的功能和性能。
  - 无效等价类：是指对于程序的规格说明来说是不合理的、无意义的输入数据构成的集合。利用无效等价类可检验程序是否规避了各种错误与异常。



# 黑盒测试方法-等价类划分

使用等价划分法设计测试用例的关键在于划分输入数据的等价类，以下是有助于等价类划分的启发式规则：

1. 若规定了输入数据的个数，则类似地可以划分出一个有效的等价类和两个无效的等价类；
2. 若规定了输入值的范围，则要划分出一个有效的等价类（输入值在此范围），两个无效的等价类（输入值小于最小值或大于最大值）；
3. 若规定了输入数据的一组值，而且程序对不同输入值做不同的处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效的等价类（任意一个不允许的输入值）；
4. 若规定了输入数据必须遵循的规则，则可以划分出一个有效的等价类（符合规则）和若干个无效的等价类（从不同角度违反规则）；
5. 若规定了输入数据为整型，则可以划分出正整数、零和负整数三个有效的等价类；
6. 若程序的处理对象是表格，则应该使用空表，以及含一项或多项的表。

# 等价类划分示例

Sale类getChange(double payment)方法

规格

| | |
|---|---|
| 前置条件 | **payment>0**<br>**payment>= Sales.total** |
| 后置条件 | return= payment- Sales.total |

- 可以将其输入数据划分为三类:

1有效数据：符合前置条件;

2无效数据，破坏第一个前置条件： payment<=0

3无效数据，破坏第二个前置条件 payment<total

测试用例

| ID | 输入 | | 预期输出 |
|---|---|---|---|
| 1 | payment=100 | Total=50 | 50 |
| 2 | payment=-100 | total=20 | 输入数据无效 |
| 3 | payment=50 | total=100 | 输入数据无效 |

# 黑盒测试方法 - 边界值分析

- 是对等价类划分方法的补充。

- 经验表明错误最容易发生在各等价类的边界上，而不是发生等价类内部。因此针对边界情况设计测试用例，可以发现更多的缺陷

# 边界值分析示例

- Sale类getChange(double payment)方法的3个等价类

1有效数据：payment>0 && payment>=total;

2无效数据，破坏第一个前置条件： payment<=0

3无效数据，破坏第二个前置条件 payment<total

- *可以得到边界值*
  - *payment=-1,payment=0,payment=1;*
  - *payment=total，payment=total+1,payment=total-1;*

不同等价类的边界是互相重合的!

# 边界值分析示例

- *getChange(double payment)方法的边界值分析测试用例*

| ID | 输入 | | 预期输出 |
|---|---|---|---|
| 1 | payment=50 | total=50 | 0 |
| 2 | payment=50 | total=49 | 1 |
| 3 | payment=50 | total=51 | 输入数据无效 |
| 4 | payment=1 | total=1 | 0 |
| 5 | payment=0 | total=0 | 0 |
| 6 | payment=-1 | total=10 | 输入数据无效 |

# 黑盒测试方法 – 基于决策表的方法

- 用于测试以复杂逻辑判断为规格的测试对象
  - 决策表既能保证测试的完备性，又能保证成本最小
  - 决策表的每一列规则选项都是一个等价类

# 基于决策表的测试方法示例

- getGift()方法

| 条件和行动 | 规则 | | |
|---|---|---|---|
| **prePoint** | <1000 | <2000 | **<5000** |
| **postPoint** | >=1000 | >=2000 | **>=5000** |
| **Gift Event Level 1** | X | | |
| **Gift Event Level 2** | | X | |
| **Gift Event Level 3** | | | **X** |

- 测试用例

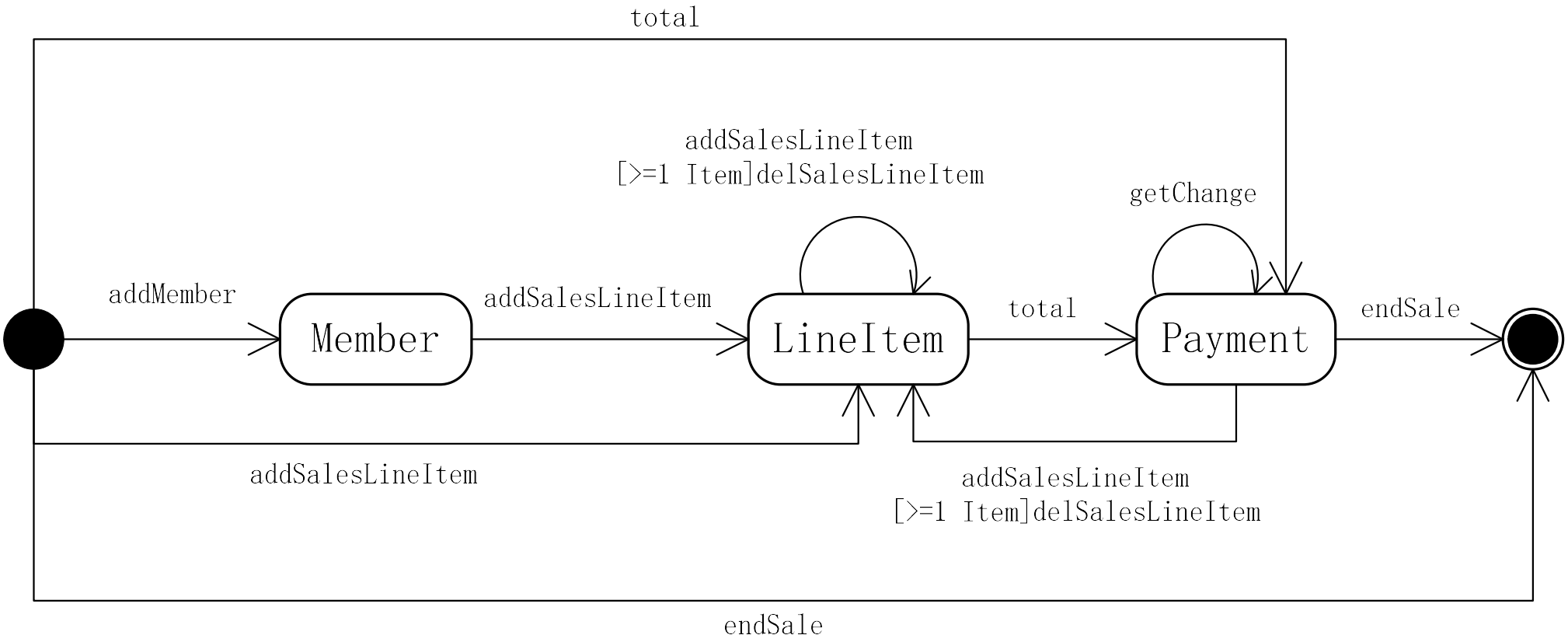
| ID | 输入 | | 预期输出 |
|---|---|---|---|
| 1 | prePoint=500 | postPoint=1500 | GiftLevel=1 |
| 2 | prePoint=500 | postPoint=2500 | GiftLevel=2 |
| 3 | prePoint=500 | postPoint=5500 | GiftLevel=3 |

# 黑盒测试方法 – 基于状态转换的方法

- 专门针对复杂测试对象。该类复杂测试对象对输入数据的反应是多样的，还需要依赖自身的状态才能决定。
- 通常要先为对象建立状态图，描述测试对象的状态集合、 输入集合和输入导致的状态转换集合。
- 以状态图基础，可建立测试对象的转换表。
- 状态转换表每一行都应该被设计为测试用例。

# 基于状态转换的方法：示例

- 类Sales
  - 建立状态图



# 基于状态转换的方法：示例

- 测试用例线索：状态转移矩阵

| 输入 | | 预期输出状态 |
|---|---|---|
| 方法 | 当前状态 | |
| addMember | Start | Member |
| | Member | 非法 |
| | LineItem | 非法 |
| | Payment | 非法 |
| | End | 非法 |
| addSalesLineItem | Start | LineItem |
| | Member | LineItem |
| | LineItem | LineItem |
| | Payment | LineItem |
| | End | 非法 |
| [>=1 Item] delSalesLineItem | …… | …… |
| total | …… | …… |
| getChange | …… | …… |
| endSale | …… | …… |

# 基于状态转换的方法：示例测试用例

| ID | 输入 | | 预期输出 |
|---|---|---|---|
| | 前置语句 | 方法 | |
| 1 | s=new Sales(); | s.addMember(2); | No Exception |
| 2 | s=new Sales();<br>s.addMember(1); | | MemberLable Invalid Time |
| 3 | s=new Sales();<br>s.addSalesLineItem(1); | | |
| 4 | s=new Sales();<br>s.addSalesLineItem(1);<br>s.total(); | | |
| 5 | s=new Sales();<br>s.addSalesLineItem(1);<br>s.total();<br>s.getChange(100);<br>s.endSale(); | | Sales dose not Exists |
| ..... | | | |

## 穷尽功能测试----例

一个程序需3个整型的输入数据，若计算机的字长为16位，则每个数据可能取的值有$2^{16}$个，3个数的排练组合共有

$$2^{16} \times 2^{16} \times 2^{16} = 2^{48} \approx 3 \times 10^{14} \text{（种）}$$

若每执行一次需1毫秒，则需1万年。

# 白盒测试

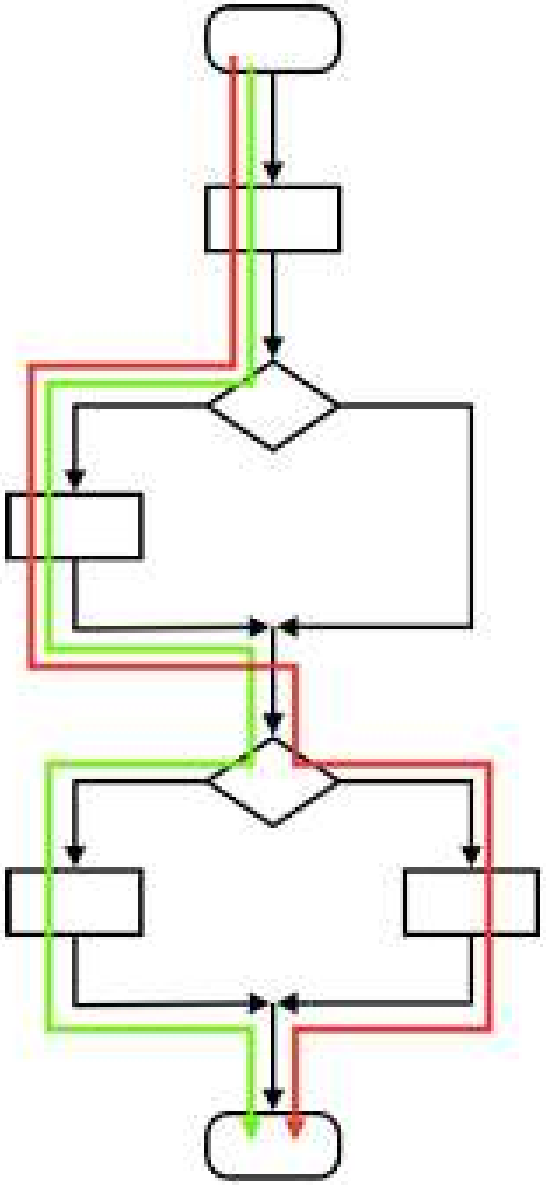
- 将测试对象看作透明的，按照测试对象内部的程序结构来设计测试用例进行测试工作
  - 语句覆盖：确保被测试对象的每一行程序代码都至少执行一次
  - 条件覆盖：确保程序中每个判断的每个结果都至少满足一次
  - 路径覆盖：确保程序中每条独立的执行路径都至少执行一次

# 白盒测试

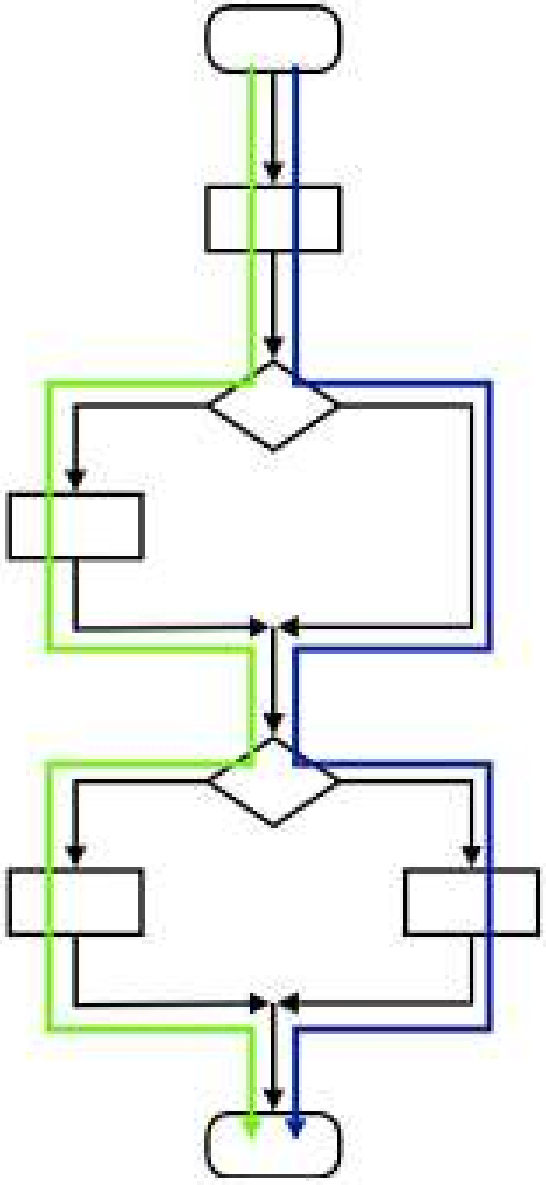
- 结构测试（白箱测试）：

  结构测试旨在充分地覆盖软件的结构，并以软件中的某类成分是否都已得到测试为准则来判断软件测试的充分性

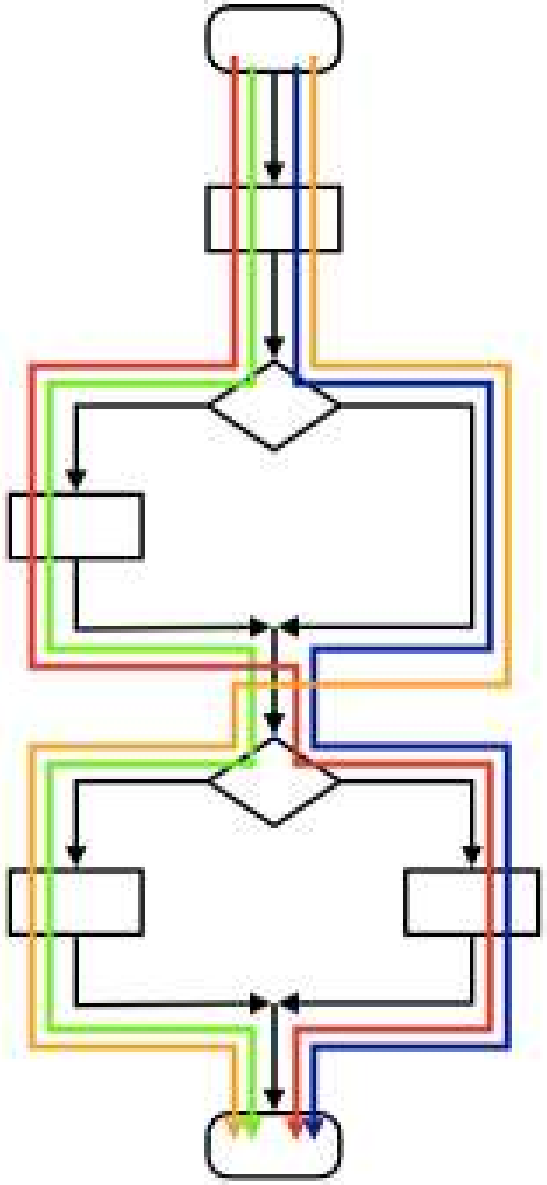
# 白盒测试



语句覆盖



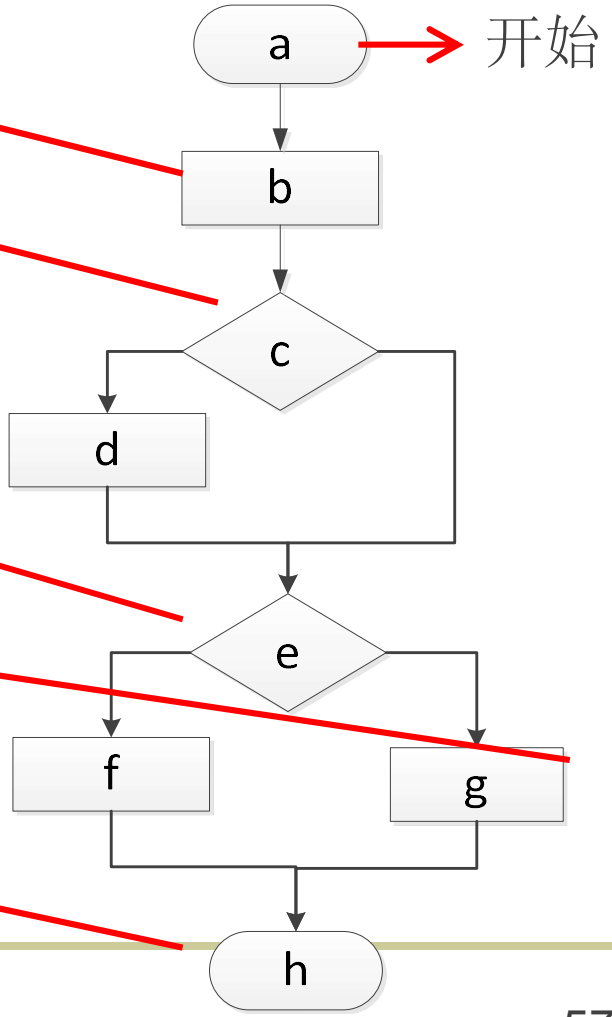
条件覆盖



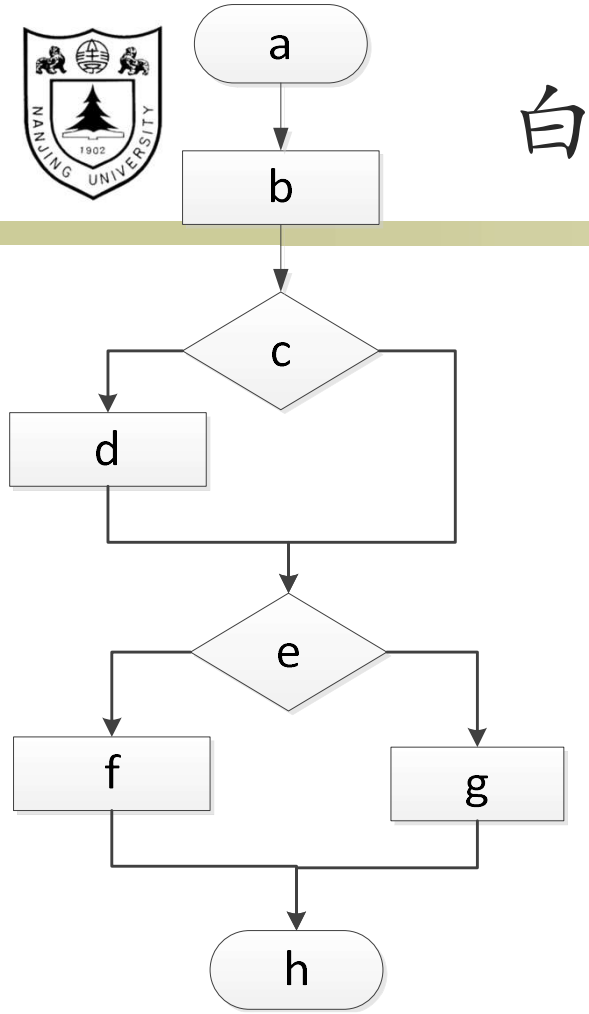
路径覆盖

# 白盒测试：示例

```
int getBonus ( boolean cashPayment , int consumption , boolean vip){
    //已有的bonus需要调整getBonus，不能直接使用属性bonus
    int preBonus=this.getBonus();
    if (cashPayment) {
      preBonus+=consumption;
    }
    if (vip) {
      preBonus*=1.5;
    } else {
      preBonus*=1.2;
    }
    return preBonus;
}
```



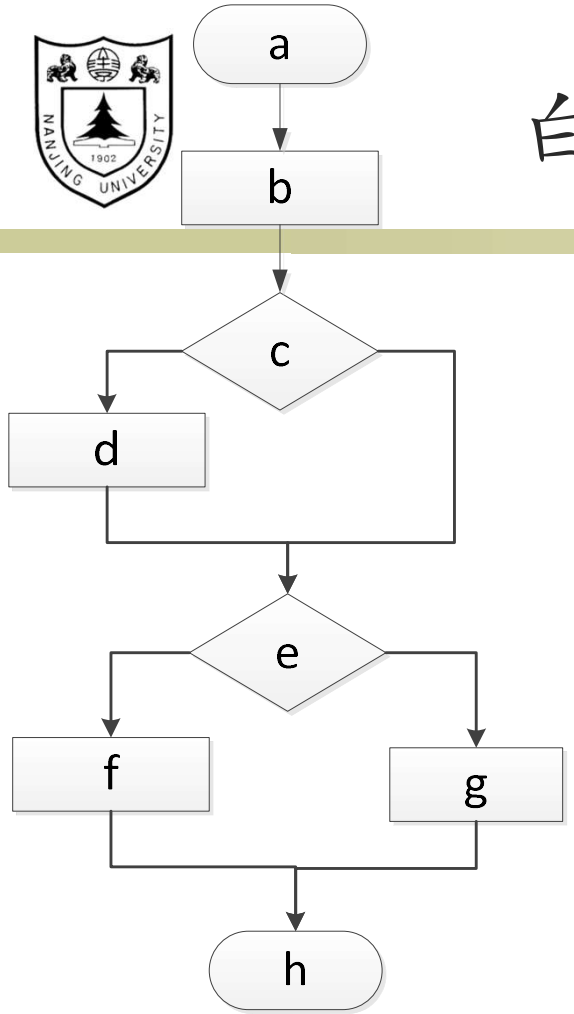
# 白盒测试：示例




- c: *if (cashPayment)*
- *d: c= true*
- *e：if(vip)*
- *f：e=true*
- *g：e=false*
- 2条路径可以做到语句覆盖
  - abcdegh：c=true,e=false
  - abcdefh:c=true,e=true

| ID | 输入 | | | | 预期输出 |
|---|---|---|---|---|---|
| 1 | getBonus()=100 | cashPayment=true | consumption=100 | vip=true | 300 |
| 2 | getBonus()=100 | cashPayment=true | consumption=100 | vip=false | 240 |

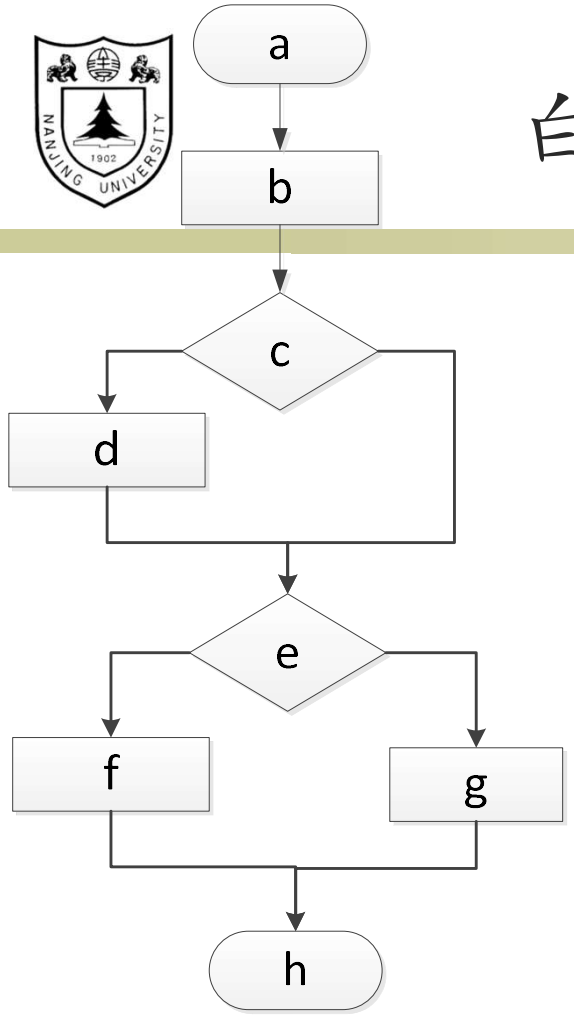
# 白盒测试：示例




- c: *if (cashPayment)*
- *e：if(vip)*
- 2条路径可以做到条件覆盖
  - abcdefh：c=true,e=true
  - abcegh:c=false,e=false

| ID | 输入 | | | | 预期输出 |
|---|---|---|---|---|---|
| 1 | getBonus()=100 | **cashPayment=true** | **consumption=100** | vip=true | 300 |
| 2 | getBonus()=100 | **cashPayment=false** | **consumption=100** | vip=false | 120 |

# 白盒测试：示例




- c: *if (cashPayment)*
- *e：if(vip)*
- 4条路径可以做到路径覆盖
  - abcdefh：c=true,e=true
  - abcdegh: c=true,e=false
  - abcefh: c=false, e=true
  - abcegh:c=false,e=false

| ID | 输入 | | | | 预期输出 |
|---|---|---|---|---|---|
| 1 | getBonus()=100 | cashPayment=true | consumption=100 | vip=true | 300 |
| 2 | getBonus()=100 | cashPayment=false | consumption=100 | vip=false | 120 |
| 3 | getBonus()=100 | cashPayment=true | consumption=100 | vip=false | 240 |
| 4 | getBonus()=100 | cashPayment=false | consumption=100 | vip=true | 150 |

- 逻辑覆盖
  - 语句覆盖
  - 判定覆盖
  - 条件覆盖
  - 判定/条件覆盖
  - 条件组合覆盖
  - 路径覆盖

# 白盒测试

现给出如下程序:

```
#include<stdio.h>
main(){
   float A, B, X;
   scanf("%f %f %f", &A, &B, &X);
   if (A>1)&&(B==0) X=X/A;
   if (A==2)||(X>1) X=X+1;
   printf("%f", X)}
```

设计该程序的测试数据以分别满足语句覆盖、判定覆盖、条件覆盖、条件组合覆盖和路径覆概的逻辑覆盖标准。

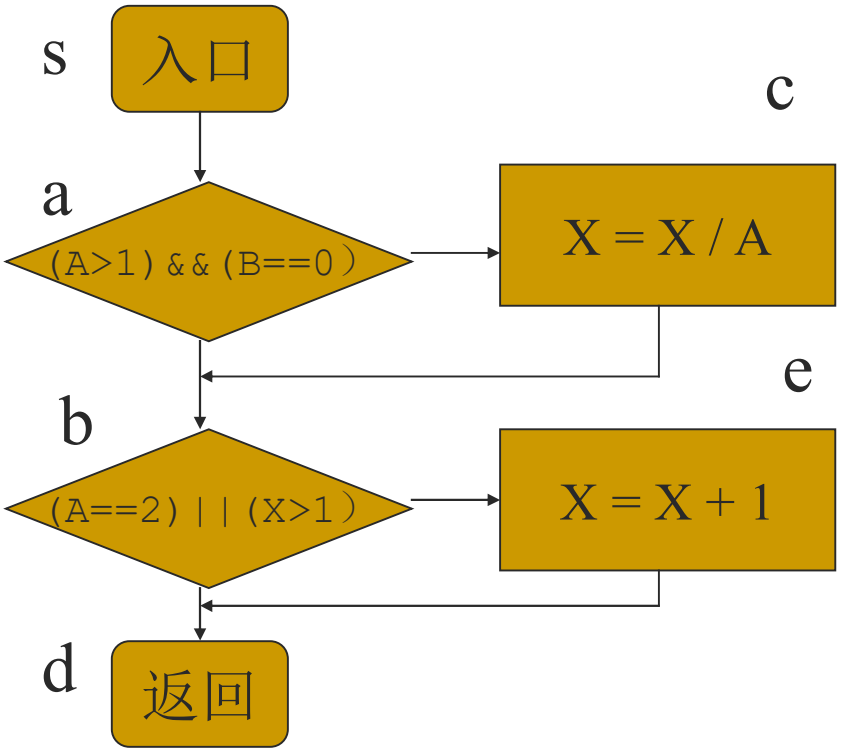
**语句覆盖：** 选取足够多的测试数据，使被测试程序中每个语句至少执行一次。

为使每个语句都执行一次，程序的执行路径应是sacbed：

A=2， B=0， X=4

问题：若把b点的判定错写为：

(A==2)||(X<1)

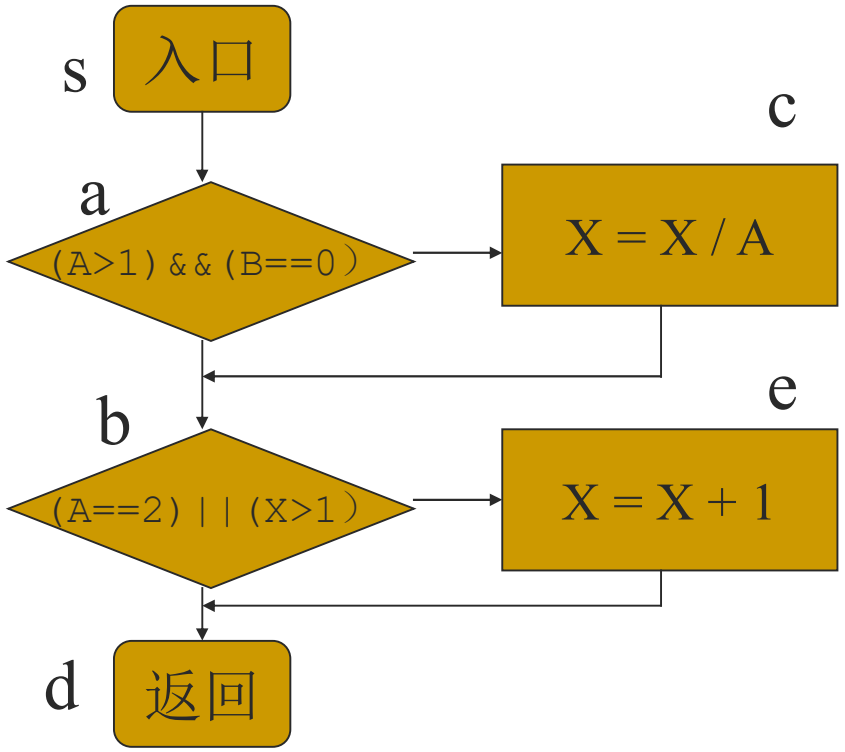


**判定覆盖：** 选取足够多的测试数据，使被测试程序中不仅每个语句至少执行一次，而且每个判定的每种可能的结果都至少执行一次。

能够分别覆盖路径sacbed和sabd或sacbd和sabed的两组测试数据，都满足判定覆盖标准：

（1）A=3，B=0，X=3（sacbd）

（2）A=2，B=1，X=1（sabed）



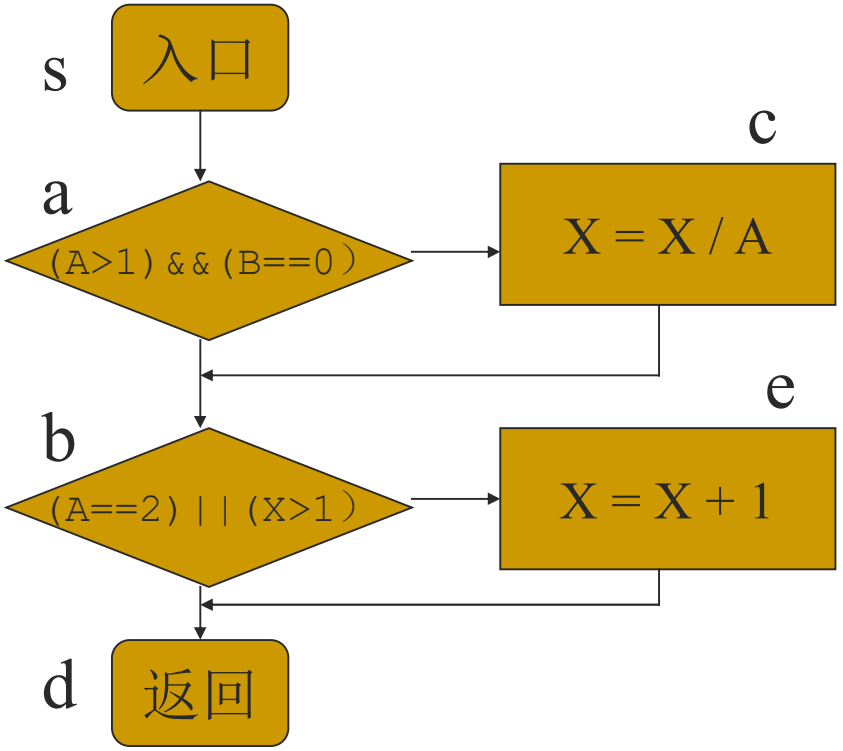
**条件覆盖：** 选取足够多的测试数据，使被测试程序中不仅每个语句至少执行一次，而且每个判定表达式中的每个条件都取到各种可能的结果。

在a点：A>1，A≤1，B=0，B≠0；

在b点：A=2，A ≠2，X>1，X ≤1。

（1）A=2，B=0，X=4 （sacbed）

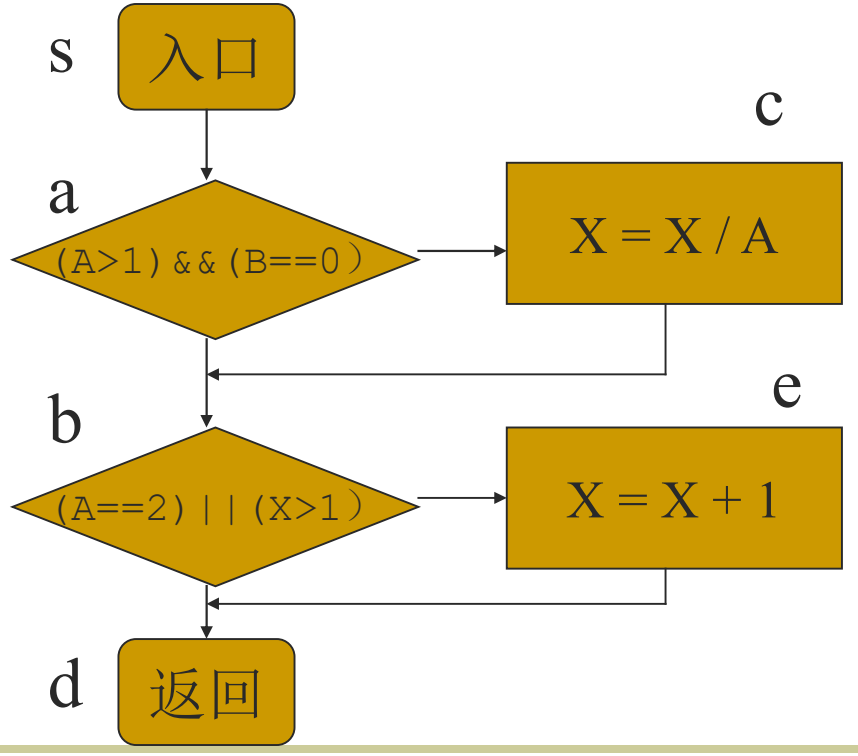
（2）A=1，B=1，X=1 （sabd）



判定/条件覆盖：选取足够多的测试数据，使得判定表达式中的每个条件都取到各种可能的结果，而且每个判定表达式也都取到各种可能的结果。

（1）A=2, B=0, X=4 (sacbed)

（2）A=1, B=1, X=1 (sabd)



**条件组合覆盖：** 选取足够多的测试数据，使得判定表达式中条件的各种可能组合都至少出现一次。

有八种可能的条件组合：

（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。

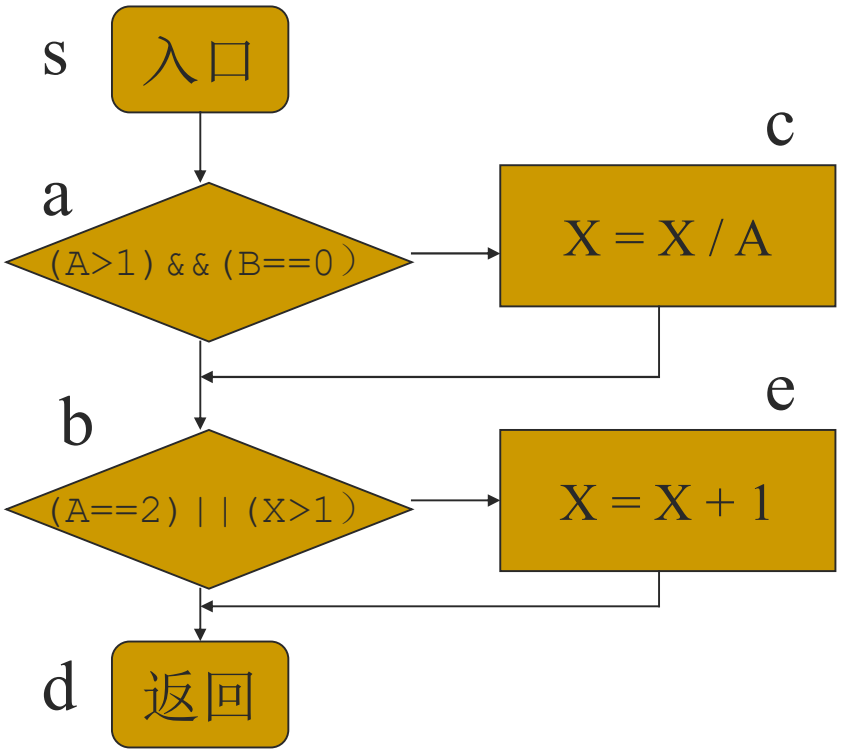
测试数据：

（1）A=2, B=0, X=4 (sacbed, 1,5)

（2）A=2, B=1, X=1 (sabed, 2,6)

（3）A=1, B=0, X=2 (sabed, 3,7)

（4）A=1, B=1, X=1 (sabd, 4,8)



**路径覆盖：** 选取足够多的测试数据，使得程序的每条可能路径都至少执行一次（若程序图中有环，则每个环至少经过一次）。

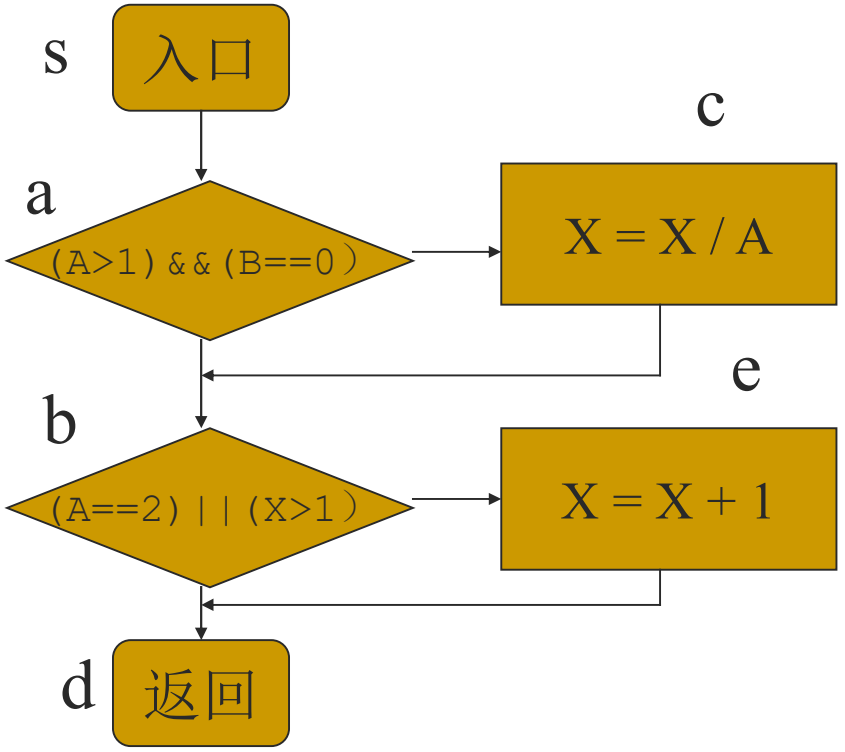
测试数据：

(1) A=1, B=1, X=1 (sabd)

(2) A=1, B=1, X=2 (sabed)

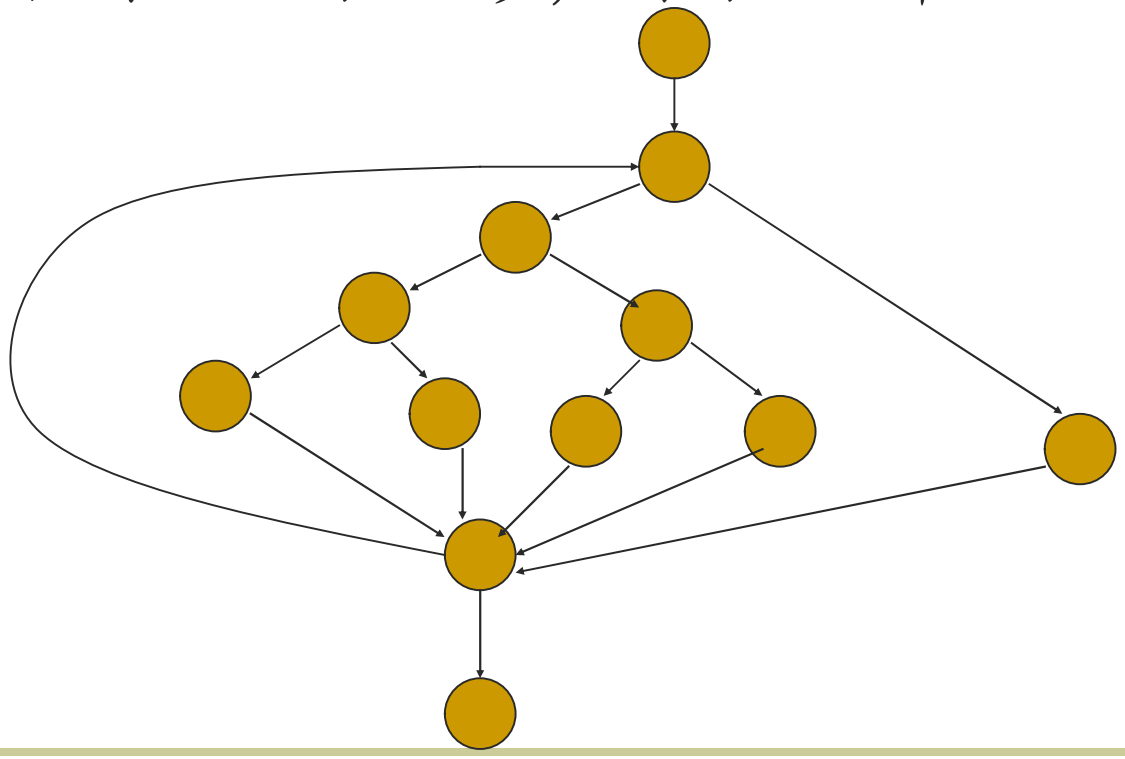
(3) A=3, B=0, X=1 (sacbd)

(4) A=2, B=0, X=4 (sacbed)



# 穷尽结构测试----例

一段程序对嵌套的IF语句执行20次，共有$5^{20} \approx 10^{14}$条通路，若每执行一次需1毫秒，则需3000年。



# 特定测试技术

1. 面向对象的测试;
2. 图形用户接口GUI测试;
3. 基于Web的测试;
4. 基于组件的测试
5. 并发程序的测试;
6. 协议遵从性测试;
7. 实时系统测试;
8. 极端要求安全性的系统的测试。

# 面向对象测试技术

- 1980slater ~1990s 发展
  - 用例/场景
  - 类
  - 继承/多态
- 常用
  - 7.5使用用例和场景设计系统测试用例；
  - 11.5基于协作设计类之间的集成测试用例；
  - 18.5基于状态图设计类的单元测试用例。

# 主要内容

- V&V基础
- 软件测试基础
- 软件测试的层次
- 软件测试技术
- 软件测试活动
- 软件测试的度量

# 测试活动

- 测试计划
- 测试设计
- 测试执行
- 测试评价

# 测试计划

- 明确软件测试的工作范围、资源与成本、基本策略、进度安排等。

- 明确标明测试的对象、测试的级别、测试的顺序、每个测试对象所应用的测试策略以及测试环境

# 测试设计

- 进一步明确需要被测试的对象，为被测对象设计测试用例集合。

- 测试用例的设计要综合考虑测试层次、被测对象特点和软件测试的目标，选择合适的测试技术。

# 测试执行

- 选择测试工具
- 搭建测试环境
- 装载测试用例
- 执行测试用例
- 记录测试结果

# 如何决定何时停止测试

管理者面临的问题:

- 当资源（时间和预算）消耗殆尽之时 (Wrong)
  - 没有关于测试效能的信息
  - 但是不能违反资源约束
- 当达到了某种覆盖准则
  - 没有对软件质量的保证 (Wrong)
  - 可以是一个合理和客观的标准
  - 可以被部分自动化
- 在达到质量标准时
  - 质量标准建立在已有的历史数据之上 (Right)

# 测试用例日志

| 测试用例ID | 种类 | 条件 | 期望结果 | 测试结果 | 测试对象ID |
|---|---|---|---|---|---|
| 输入测试用例ID | 输入测试种类 | 输入特殊的测试条件 | 测试用例预期的结果 | 记录“通过”或者“失败” | 被测试对象的ID |
| | | | | | |
| | | | | | |

# 缺陷报告

| 缺陷ID | 发现日期 | 测试脚本 | 测试用例 | 期望结果 | 实际结果 | 状态 | 严重性 | 优先级 | 缺陷类型 | 备注 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |
| | | | | | | | | | | |

# 测试评价

- 多数情况下，“成功”表示软件按期望运行，并且没有重大的非期望结果。
- 当测试结果特别重要时，需要召集一个正式的评审委员会来评价这些结果
- 测试评价完成之后，要发布测试报告
  - 图19-11
  - 图19-12

# 主要内容

- V&V基础
- 软件测试基础
- 软件测试的层次
- 软件测试技术
- 软件测试活动
- 软件测试的度量

# 测试度量

- 缺陷数据
  - 分类

- 测试覆盖率
  - 需求覆盖率 =被测试的需求数量 /需求总数;
  - 模块覆盖率 =被测试的模块数量 /模块总数;
  - 代码覆盖率 =被测试的代码行 /代码行数总数。

# 测试覆盖率与测试程序规模成反比

| 规模（代码行） | 测试用例数量 | 测试覆盖率 |
|---|---|---|
| 1 | 1 | 100% |
| 10 | 2 | 100% |
| 100 | 5 | 95% |
| 1000 | 15 | 75% |
| 10000 | 250 | 50% |
| 100000 | 4000 | 35% |
| 1000000 | 50000 | 25% |
| 10000000 | 350000 | 15% |

## 测试与排错（调试）的区别

- 测试是从已知的条件出发，使用预先定义的方法，并且有预期的测试结果。排错往往是从未知的初始条件（错误的性质，位置和范围）出发；
- 测试能够而且应该事先安排，事先设计和制定测试日程表，而排错的方法和所需的时间都不能事先确定；
- 测试是暴露程序员的过失，相反排错是帮助程序员纠正错误；

# 测试

- 测试应该是可预测的、机械的、强制的、严格的，排错要求随机应变、联想、实验、智力和自主；
- 测试的设计和实现在很大程度上可以忽略被测试对象的详细设计，但是没有详细设计的知识，排错是不可能的；
- 测试能由非程序员来做，而排错相反；
- 测试已经建立了它的理论基础，在理论上人们已知道，它能做什么和不能做什么，但是到目前为止，排错还没有一个经得起检验的理论方法。

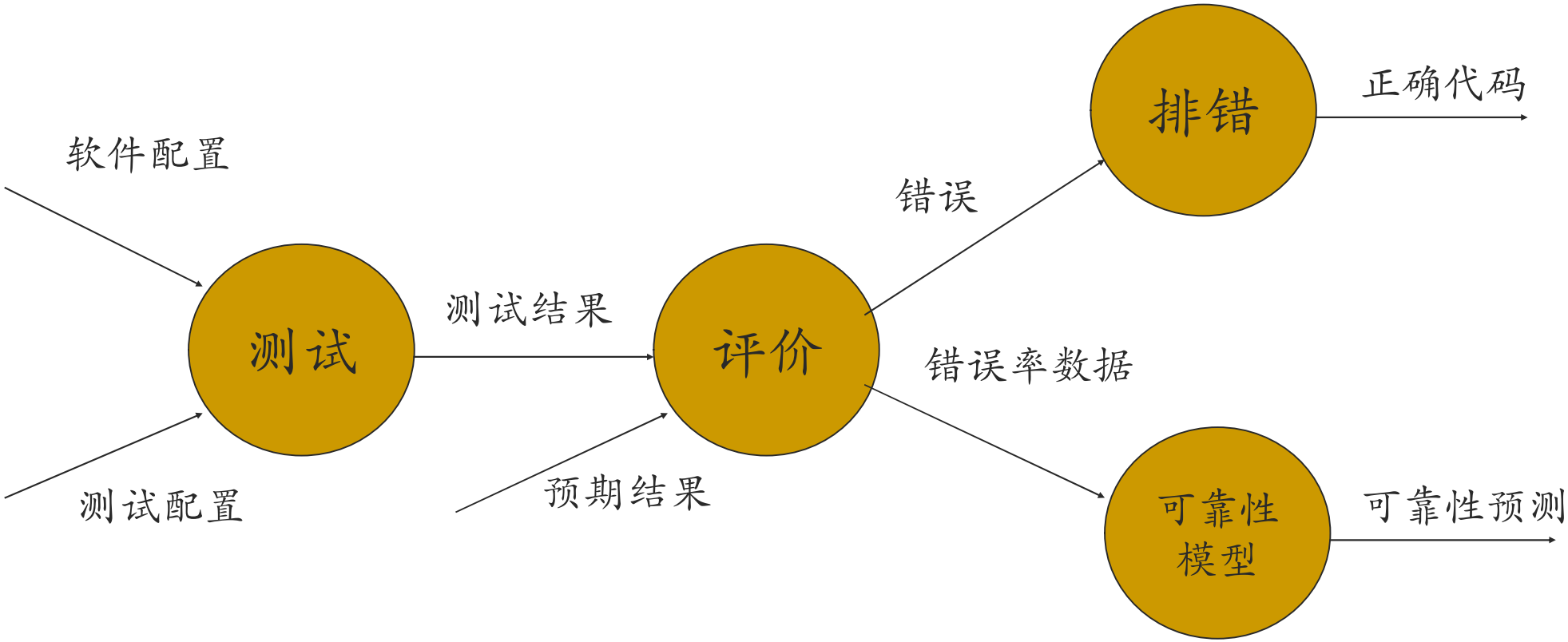
## 排错（调试）

- 排错是把一个软件错误的现象与其原因联系起来的人的思维过程，对这个过程人们还没有深入的了解。
- 虽然排错能够也应该是一个有条理的过程，它现在仍然具有很大的技巧成分，并且心理因素在排错过程占有一定的位置。
- 软件工程师在分析测试结果时，看到的往往是软件错误的征兆，错误的内部原因与错误的外部表现可能没有明显的关系。
- 排错三要素：分析、直觉、运气。

# 调试

- 排错（调试）方法
  - 原始方法：跟踪、插入打印语句。
  - 原因消除法：归纳法，演绎法。
  - 回溯法：从源程序中出现征兆的语句开始往回追踪。
  - 在排错方面最后的格言是：所有的办法都不行的时候，请别人来帮忙。

# 4、测试信息流



# 测试信息流

- 输入信息

  **软件配置：** 需求说明，设计说明，源程序清单等。

  **测试配置：** 测试计划和方案。

- 输出信息

  测试结果

  可靠性预测： 可能可靠（软件可靠性是可以接受的）

  所进行的测试尚不足以发现严重错误

  不可靠

# 总结

- 软件测试是V&V的重要手段，以发现缺陷为目标
- 软件测试工作依据其所测试的对象或目标而有所不同
- 软件测试技术用来追求软件测试工作的成本效益比
  - 最为常用的测试技术是随机测试、黑盒测试和白盒测试
- 软件测试的主要活动包括：测试计划、测试设计、测试执行和测试评价