CS10102302
软件测试基础

叶晨 教授级高工

计算机科学与技术系 电子与信息工程学院

同济大学

同济大学

个人介绍 – 叶晨

2003年本科毕业于同济大学自动化专业，留校后在大学生创新基地带教
2015年博士毕业于同济大学计算机应用，研究方向：智能交通、车联网
目前的研究方向：机器学习、图像处理、大数据分析及其在工业智能领域的应用
科研领域涉及无人驾驶汽车、智能交通系统、医学辅助诊断、创意汉字生成等

指导学生获奖

“挑战杯”全国一等奖，“小挑”全国银奖，“互联网+”上海市金奖
大学生程序设计竞赛、电子设计竞赛等多项学科竞赛，国际级/国家级一等奖20+

指导创新项目

国创10项，上创8项（2007～）
项目获奖：国家级一等奖、上海市特等奖/一等奖
本科生授权发明专利10项

教学提纲

1. 软件测试概述
2. 软件测试类型
3. 白盒测试方法
4. 黑盒测试方法

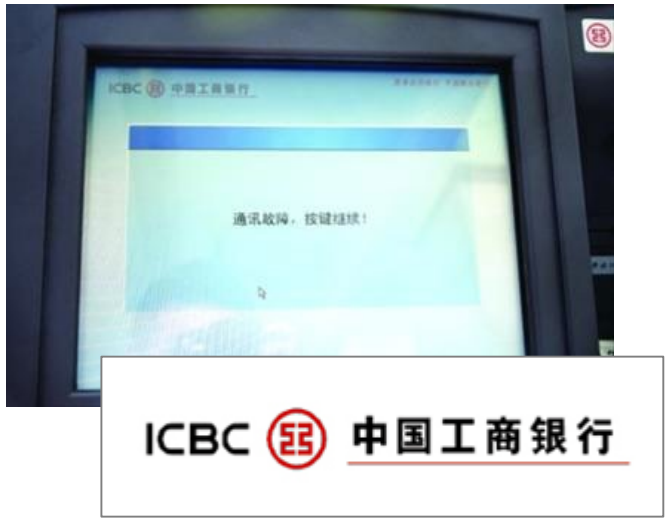
教学提纲

1 软件测试概述

- 软件缺陷术语
- 软件测试概念
- 软件测试基本原则
- 软件测试团队

关于软件质量

The average software product released on the market is not error free.



术语解释

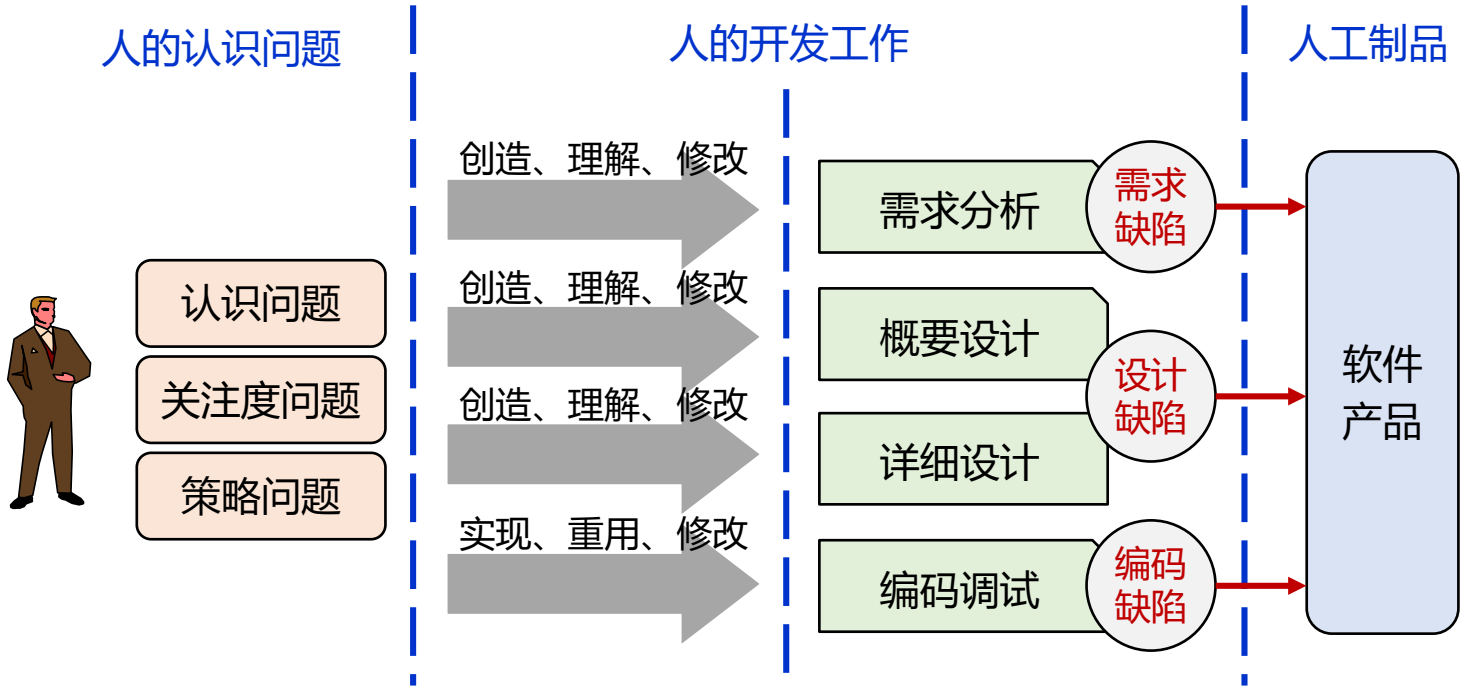
错误（Error）：在软件生存期内的不希望或不可接受的人为错误，其结果是导致软件缺陷的产生。

缺陷（Defect）：软件缺陷是存在于软件产品之中的那些不希望或不可接受的偏差，其结果是软件运行于某一特定条件时出现故障。

故障（Fault）：软件运行过程中出现的一种不希望或不可接受的内部状态，若无适当措施（容错）加以及时处理，便产生软件失效。

失效（Failure）：软件运行时产生的一种不希望或不可接受的外部行为结果。

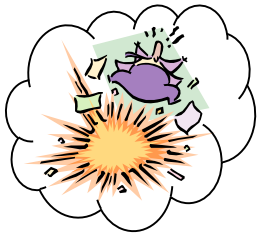
软件缺陷的产生


人的认识问题
人的开发工作
人工制品
认识问题
关注度问题
策略问题
创造、理解、修改
创造、理解、修改
创造、理解、修改
实现、重用、修改
需求分析
概要设计
详细设计
编码调试
需求缺陷
设计缺陷
编码缺陷
软件产品

软件缺陷的演化

举例：爱国者导弹

1991年2月，一次拦截失利
造成28名美国士兵丧生



软件
系统

计时用系统时钟（1/10秒）
并以整数表达

缺少防范措施

缺陷 → 激活 → 故障 → 演变 → 失效

寄存器存储导致误差
$(0.000000095)_{10}$

$0.000000095 \times 100h \times 60 \times 60 \times 10 = 0.34s$

软件测试

测试 —— 发现问题 —— 修复

什么是软件测试?

软件测试的目的是什么?

软件测试的定义

正向思维：验证软件正常工作

- 评价一个程序或系统的特性或能力并确定是否达到预期的结果。
- 在设计规定的环境下运行软件的所有功能，直至全部通过。

逆向思维：假定软件有缺陷

- 测试是为了发现错误而针对某个程序或系统的执行过程。
- 寻找容易犯错地方和系统薄弱环节，试图破坏系统直至找不出问题。

软件测试的目的

直接目标：发现软件错误

- 以最少的人力、物力和时间找出软件中潜在的各种错误和缺陷，通过修正这些错误和缺陷提高软件质量，回避软件发布后由于潜在的软件错误和缺陷造成的隐患所带来的商业风险。

期望目标：检查系统是否满足需求

- 测试是对软件质量的度量和评估，以验证软件的质量满足客户需求的程度，为用户选择和接受软件提供有力的依据。

附带目标：改进软件过程

- 通过分析错误产生的原因，可以帮助发现当前开发所采用的软件过程缺陷，从而进行软件过程改进；通过分析整理测试结果，可以修正软件开发规则，为软件可靠性分析提供依据。

测试的局限性

测试的不彻底性

- 测试只能说明错误的存在，但不能说明错误不存在
- 经过测试后的软件不能保证没有缺陷和错误

测试的不完备性

- 测试无法覆盖到每个应该测试的内容
- 不可能测试到软件的全部输入与响应
- 不可能测试到全部的程序分支的执行路径

测试作用的间接性

- 测试不能直接提高软件质量，软件质量的提高要依靠开发
- 测试通过早期发现缺陷并督促修正缺陷来间接地提高软件质量

软件测试心理学

人类行为总是倾向于具有高度目标性，
确立一个正确的目标具有重要的心理学影响。

- ✓ 如果测试的目的是要证明程序中没有错误，就会在潜意识中倾向于选择使程序出错可能性较小的测试数据。
- ✓ 如果目的是要证明程序中存在错误，就会选择一些易于发现程序所含错误的测试数据。

- ✓ 当人们开始一项工作时，如果已经知道它是不可行或无法实现的，表现就会很糟糕。
- ✓ 要证明软件中不存在错误是不可能的，如果把测试看成是发现错误的过程，那么测试就是可完成的任务。

一定程度的独立测试通常可以更高效地发现软件缺陷，但独立不能替代对软件的熟悉和经验。

软件测试原则

原则1：测试用例中一个必需部分是对预期输出或结果的定义。

原则2：编程人员应当避免测试自己编写的程序。

原则3：编写软件组织不应测试自己编写的软件。

原则4：应当彻底检查每个测试的执行结果。

原则5：测试用例不仅根据有效的和预期的输入情况，也应根据无效的和未预料到的输入情况。

原则6：检查程序是否“未做应该做的”仅是测试的一半，另一半是检查是否“做了不应该做的”。

原则7：应避免测试用例用后即弃，除非是一次性软件。

原则8：计划测试工作时不应默许假定不会发现错误。

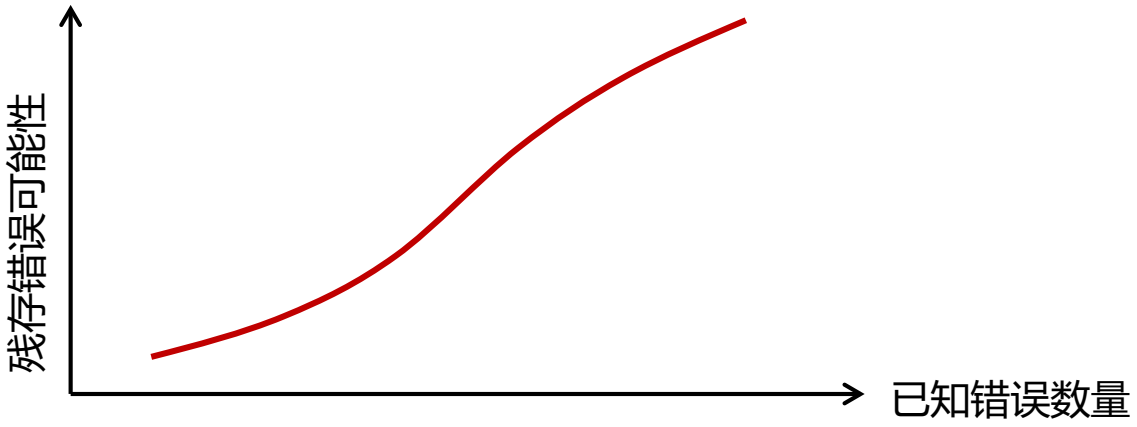
原则9：程序某部分存在更多错误的可能性，与该部分已发现错误的数量成正比。

原则10：软件测试是一项极富创造性、极具智力挑战性的工作。

缺陷的集群性

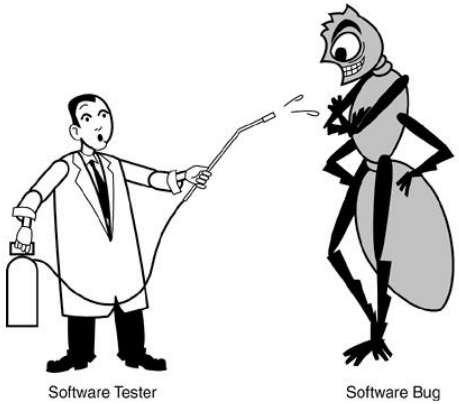
软件错误具有聚集性，对存在错误的部分应重点测试。

- 80/20原则：80%的软件错误存在于20%的代码行中
- 经验表明：测试后程序中残留的错误数目与该程序中已检出的错误成正比。



杀虫剂悖论

用同样的测试用例多次重复进行测试，最后将不再能够发现新的缺陷。



如同给果树喷撒农药，为了杀灭害虫只打一种杀虫药，虫子就会有抗体而变得适应，于是杀虫剂将不再发挥作用。

测试用例需要定期评审和修改，同时要不断增加新的不同测试用例来测试软件的不同部分，从而发现更多潜在的缺陷。

软件测试团队的任务

软件测试与质量保证合二为一

① 发现软件程序、系统或产品中所有的问题
② 尽早地发现问题
③ 督促开发人员尽快地解决程序中的缺陷
④ 帮助项目管理人员制定合理的开发计划
⑤ 对问题进行分析、分类总结和跟踪
⑥ 帮助改善开发流程，提高产品开发效率
⑦ 提高程序编写的规范性、易读性、可维护性等

你认为软件测试人员需要具备哪些素质？

你认为软件测试人员需要具备哪些素质？

耐心、细心、沟通协调、责任心
逆向思维、追求完美
站在用户的角度看问题

测试人员角色

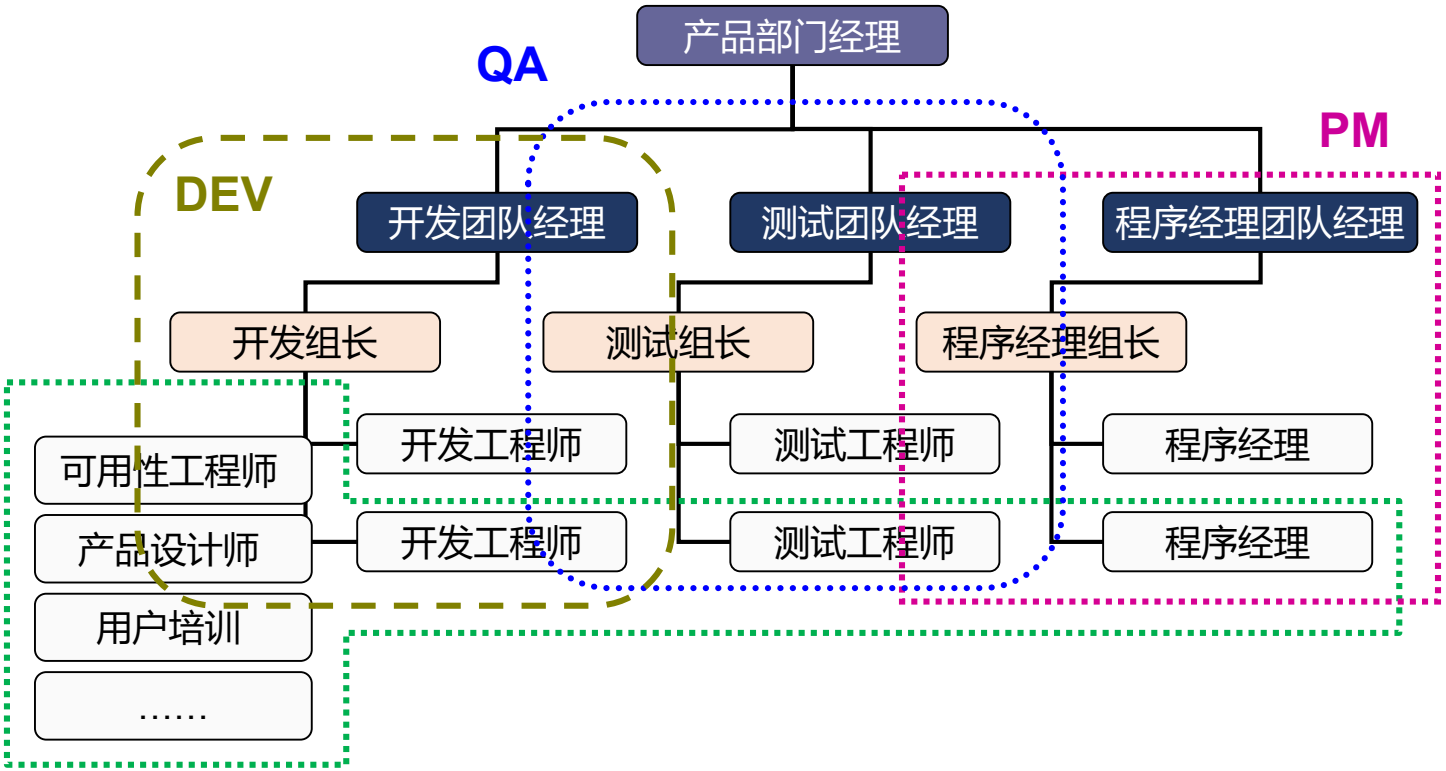
测试经理：建立和完善测试流程以及部门管理体制，审核测试项目并分配资源，监控和协调各项目的测试工作，负责与其他部门的协调和沟通工作。

测试组长：制定测试项目计划（包括人员、进度、软硬件环境和流程等），实施软件测试，跟踪和报告计划执行情况，负责测试用例质量，管理测试小组并提供技术指导。

测试工程师：理解软件产品的要求，对其进行测试以便发现软件中的错误，验证软件是否满足规格说明所描述的需求，编写相应的测试方案和测试用例。

测试工具工程师：编写软件测试工具，并利用所编写的测试工具对软件进行测试，或者为测试工程师开发测试工具。

举例：微软研发团队


产品部门经理
QA
PM
DEV
开发团队经理
测试团队经理
程序经理团队经理
开发组长
测试组长
程序经理组长
开发工程师
测试工程师
程序经理
可用性工程师
产品设计师
开发工程师
测试工程师
程序经理
用户培训
……

微软测试工程师的一天

凌晨 产品构建完成后，测试编译自动开始；如果测试编译成功，基本验证测试BVT（Basic Verification Test）自动开始。

1. 早晨上班，检查测试包与BVT结果的Email。如果有BVT错误，在第一时间分析原因，隔离错误代码并报告最高级别的缺陷（开发人员应当日修改）。
2. 在缺陷管理系统中检查Bug情况，验证分配给自己的且开发人员已修改的Bug。
3. 关闭Bug，并针对该Bug修正所影响范围，执行回归测试。
4. 验证最近开发的测试脚本执行结果，如果发现新错误，则报告Bug并进行调试，解决脚本中的问题。
5. 开发新的测试规范或测试脚本，并使用个人创建的任务验证新开发的测试脚本。
6. 用已通过的脚本来验证所对应开发人员的新版本程序，尽量发现任何严重问题。
7. 改进与提高自动化测试系统的功能。
8. 参与产品规格说明书、测试用例的评审会议。
9. 复审测试同伴编写的脚本和相关文档。
10. 回答与项目相关的其他各种问题。

软件测试人员

核心素质：责任心

测试技术能力

- 掌握理论、方法、工具
- 开发测试工具、自动化测试框架、测试脚本等
- 掌握操作系统、网络、数据库、中间件等知识

非技术能力

- 沟通能力，协作精神
- 自信心、耐心、专心
- 洞察力、记忆力
- 怀疑精神，创造性
- 反向思维与发散思维
- 自我督促、幽默感

专业领域经验

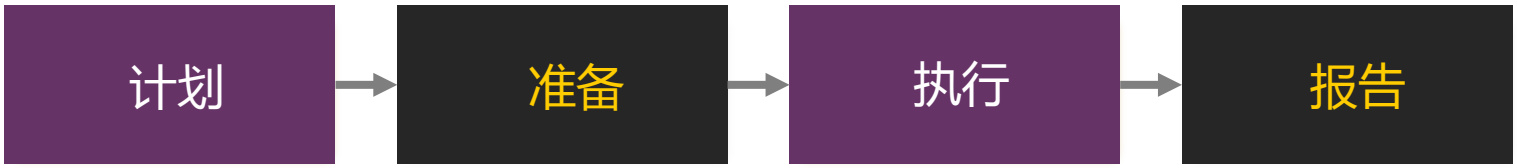
- 了解业务知识
- 积累实践经验

教学提纲

2 软件测试类型

- 软件测试活动
- 软件测试类型
- 软件测试模型
- 测试用例设计

软件测试过程



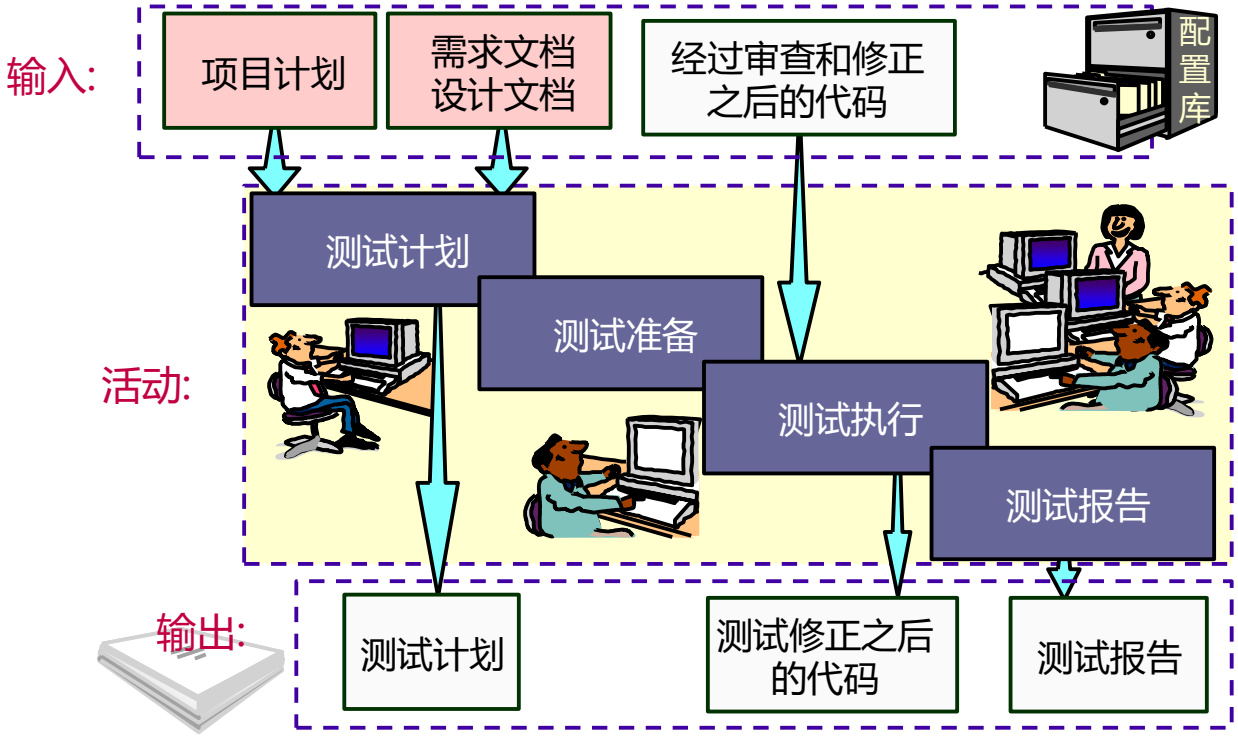
- 识别测试需求
- 分析质量风险
- 拟定测试方案
- 制定测试计划

- 组织测试团队
- 设计测试用例
- 开发工具和脚本
- 准备测试数据

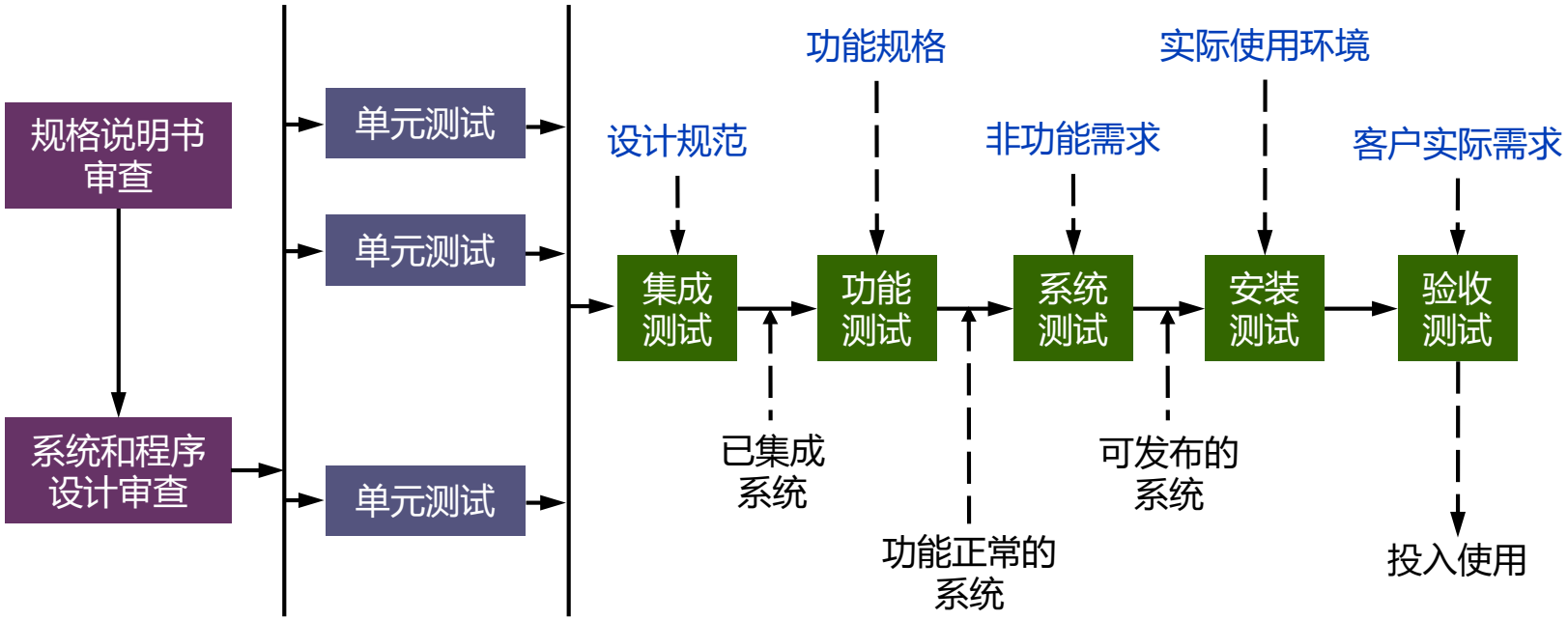
- 获得测试版本
- 执行和实施测试
- 记录测试结果
- 跟踪和管理缺陷

- 分析测试结果
- 评价测试工作
- 提交测试报告

软件测试活动


输入:
项目计划
需求文档
设计文档
经过审查和修正之后的代码
配置库
活动:
测试计划
测试准备
测试执行
测试报告
输出:
测试计划
测试修正之后的代码
测试报告

软件测试阶段


规格说明书
审查
系统和程序
设计审查
单元测试
单元测试
单元测试
设计规范
功能规格
非功能需求
实际使用环境
客户实际需求
集成
测试
功能
测试
系统
测试
安装
测试
验收
测试
已集成
系统
功能正常的
系统
可发布的
系统
投入使用

软件测试类型


测试对象角度
单元测试、集成测试、系统测试、验收测试
测试技术角度
黑盒测试（功能测试）、白盒测试（结构测试）
程序执行角度
静态测试、动态测试
人工干预角度
手工测试、自动化测试

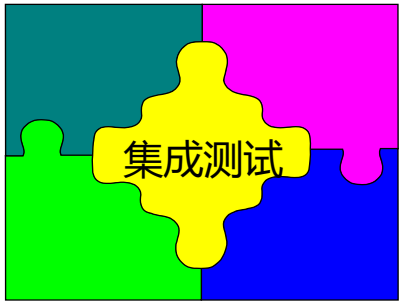
单元测试

单元测试（Unit Testing）是对软件基本组成单元进行的测试，其测试对象是软件设计的最小单位（模块或者类）。

单元测试一般由编写该单元代码的开发人员执行，用于检测被测代码的功能是否正确。

集成测试

集成测试（Integration Testing）是在单元测试的基础上，将所有模块按照总体设计的要求组装成为子系统或系统进行的测试。



- 一次性集成方式：分别测试每个单元，再一次性将所有单元组装在一起进行测试。
- 渐增式集成方式：先对某几个单元进行测试，然后将这些单元逐步组装成较大的系统，在组装过程中边连接边测试。

集成测试的对象是模块间的接口，其目的是找出在模块接口上，包括系统体系结构上的问题。

功能测试

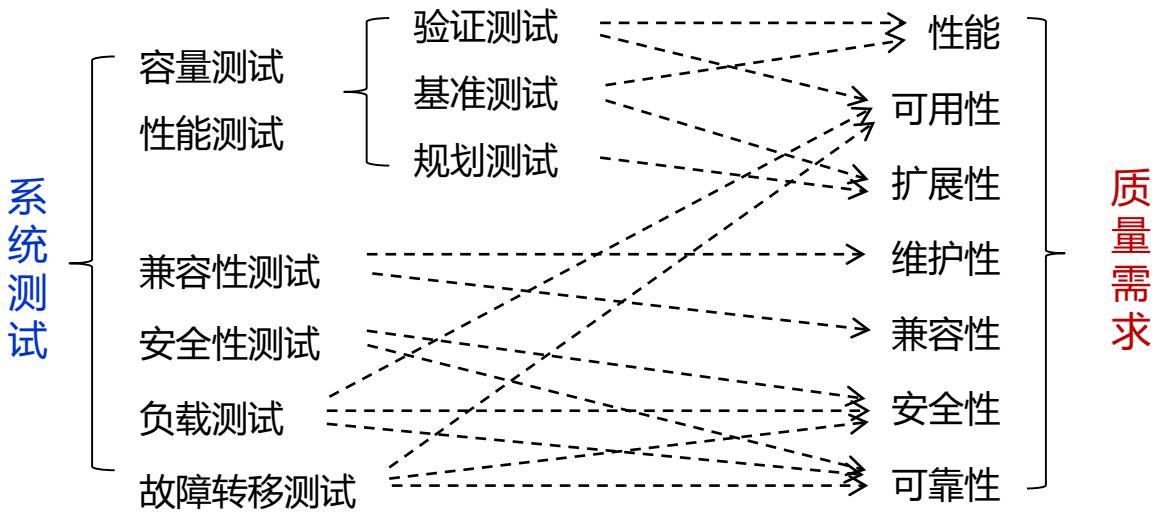
功能测试（Functional Testing）是在已知产品所应具有的功能基础上，从用户角度来进行功能验证，以确认每个功能是否都能正常使用。

- 界面
- 数据
- 操作
- 逻辑
- 接口

举例：清华大学网络学堂

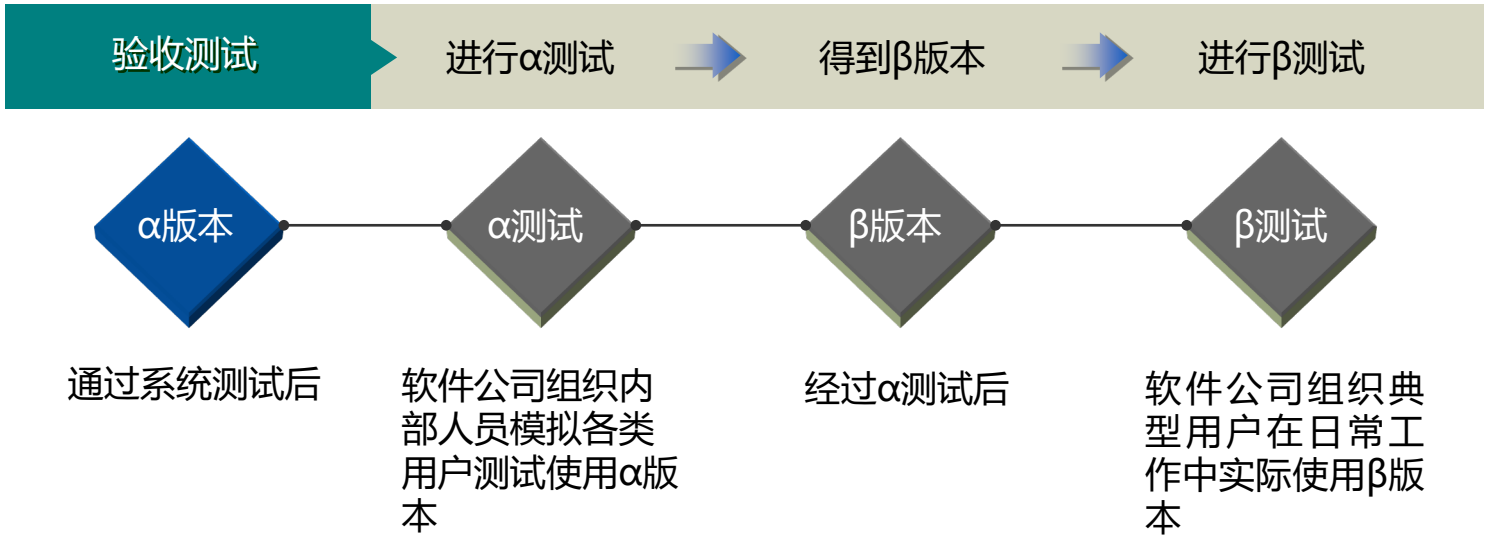
系统测试

系统测试（System Testing）是在实际运行环境或模拟实际运行环境下，针对系统的非功能特性所进行的测试，包括负载测试、性能测试、恢复测试、安全测试和可靠性测试等。



验收测试

验收测试是在软件产品完成了功能测试和系统测试之后、产品发布之前所进行的软件测试活动，其目的是验证软件的功能和性能是否能够满足用户所期望的要求。



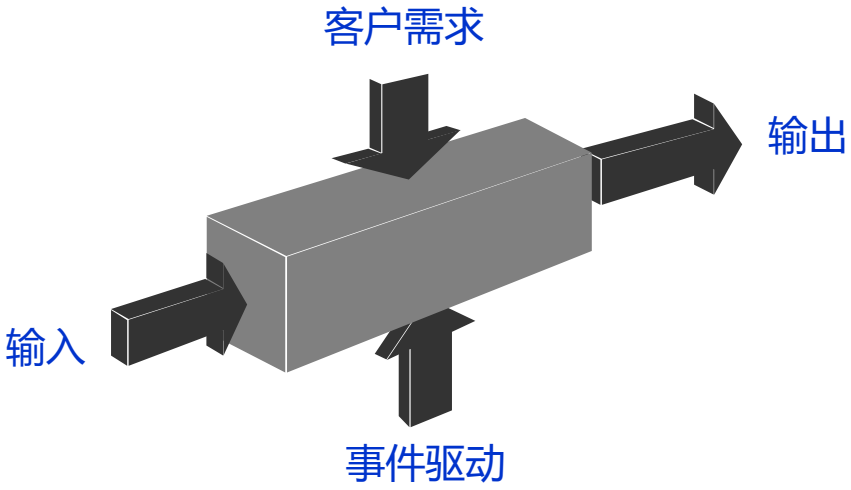
安装测试

安装测试是系统验收之后，需要在目标环境中进行安装，其目的是保证应用程序能够被成功地安装。

- 应用程序是否可以成功地安装在以前从未安装过的环境中?
- 应用程序是否可以成功地安装在以前已有的环境中?
- 配置信息定义正确吗?
- 考虑到以前的配置信息吗?
- 在线文档安装正确吗?
- 安装应用程序是否会影响其他的应用程序吗?
- 安装程序是否可以检测到资源的情况并做出适当的反应?

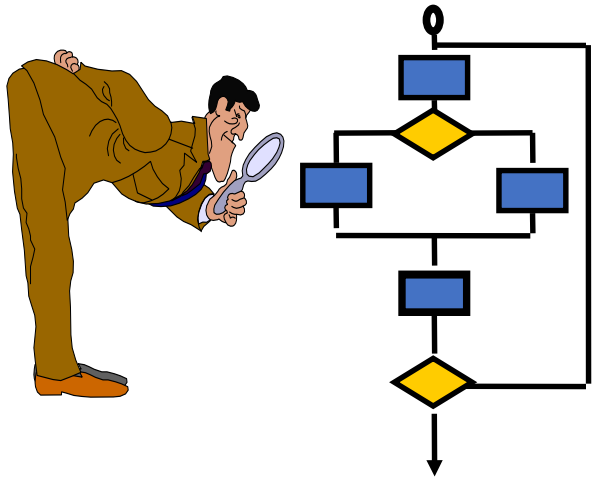
黑盒测试

黑盒测试（Black Box Testing）：又称功能测试，它将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符合它的功能说明。



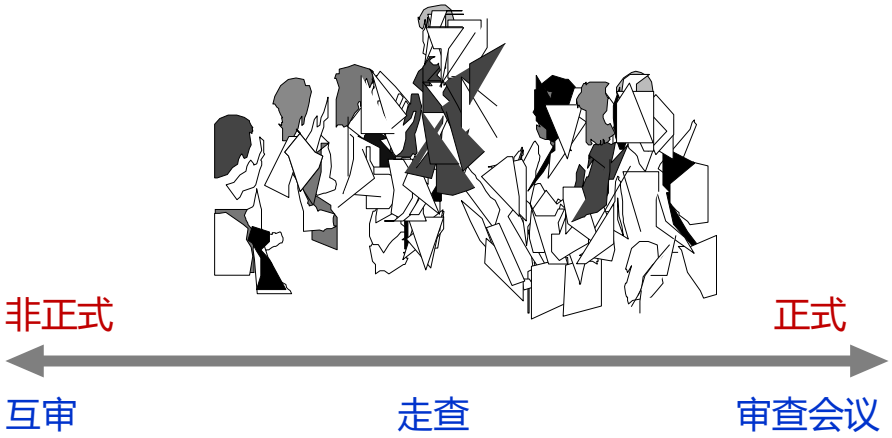
白盒测试

白盒测试（White Box Testing）：又称结构测试，它把测试对象看做一个透明的盒子，允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。



静态测试与动态测试

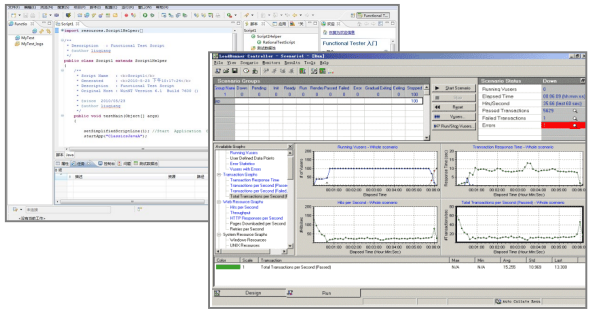
静态测试：通过人工分析或程序正确性证明的方式来确认程序正确性。



动态测试：通过动态分析和程序测试等方法来检查程序执行状态，以确认程序是否有问题。

手工测试与自动化测试

- 手工测试：测试人员根据测试大纲中所描述的测试步骤和方法，手工地输入测试数据并记录测试结果。
- 自动化测试：相对于手工测试而言，主要是通过所开发的软件测试工具、脚本等手段，按照测试工程师的预定计划对软件产品进行的自动测试。

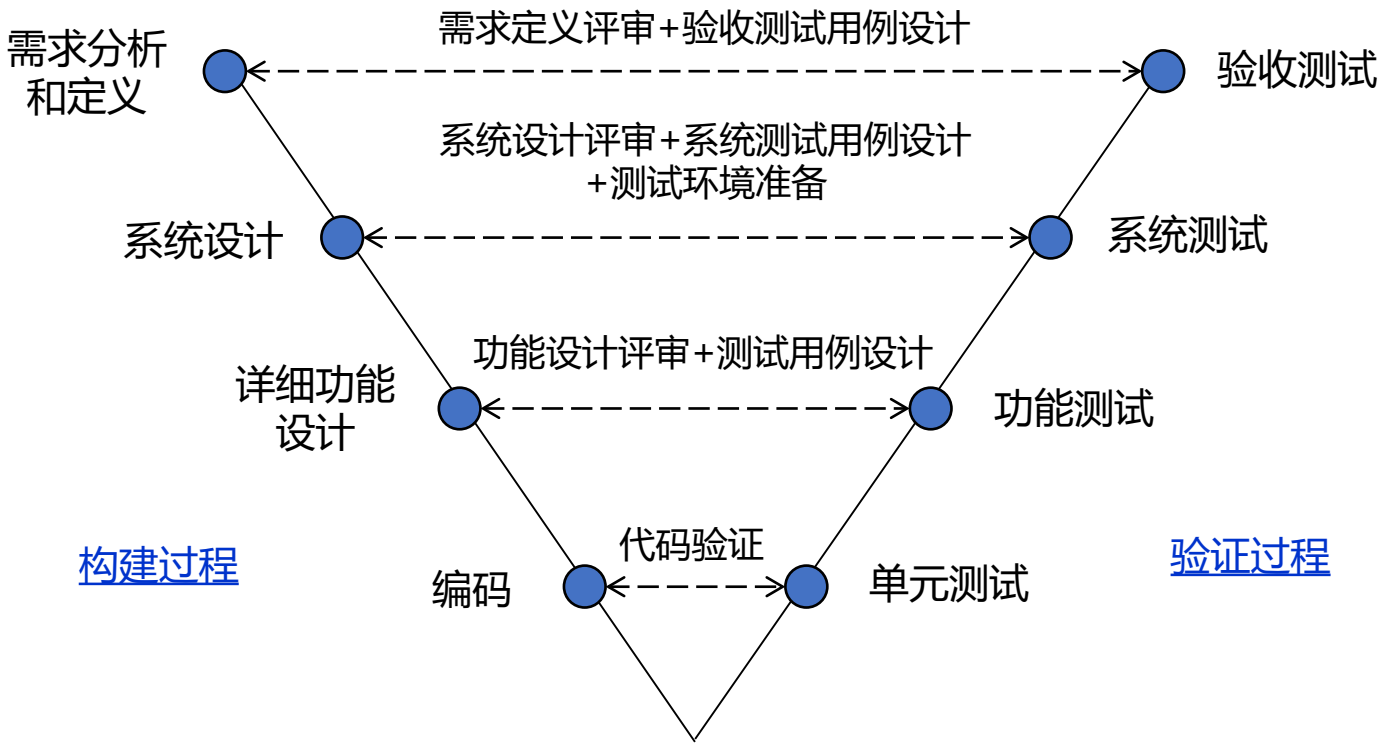


手工测试与自动化测试

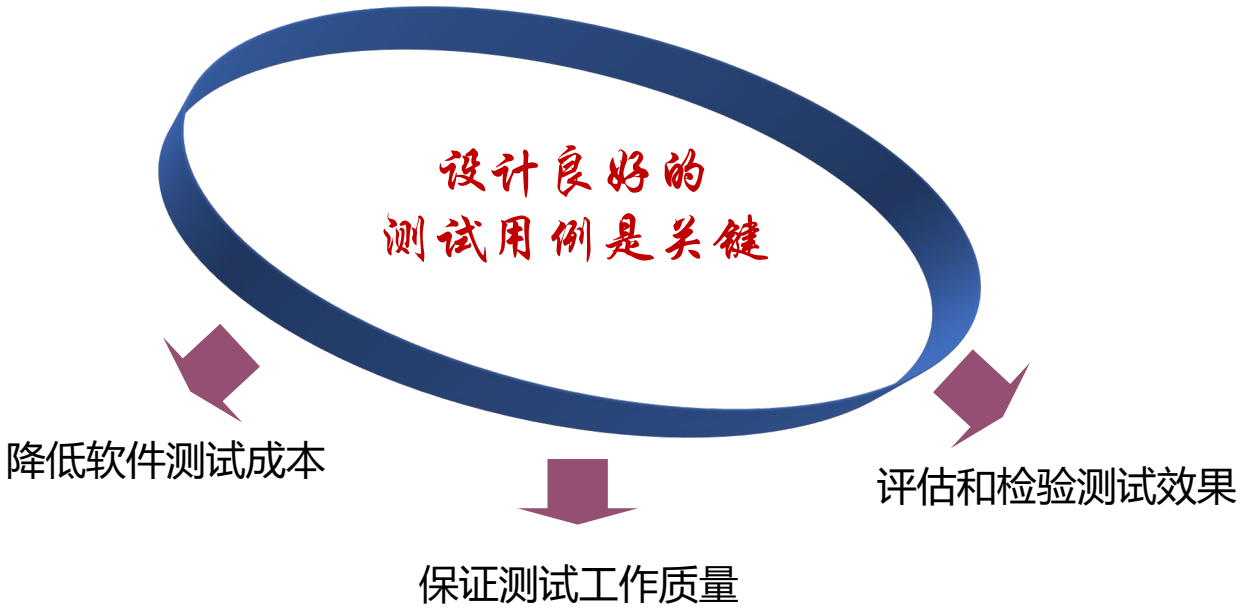
自动化测试只是对手工测试的一种补充，但绝不能代替手工测试，二者有各自的特点。

- 在系统功能逻辑测试、验收测试、适用性测试、涉及物理交互性测试时，多采用黑盒测试的手工测试方法；
- 单元测试、集成测试、系统负载或性能、稳定性、可靠性测试等比较适合采用自动化测试；
- 对那种不稳定软件的测试、开发周期很短的软件、一次性的软件等不适合自动化测试；
- 工具本身并没有想象力和灵活性，一般自动化测试只能发现15~30%的缺陷，而手工测试可以发现70~85%的缺陷；自动化测试工具在进行功能测试时，其准确的含义是回归测试工具。

V 模型


需求定义评审+验收测试用例设计
需求分析
和定义
验收测试
系统设计评审+系统测试用例设计
+测试环境准备
系统设计
系统测试
功能设计评审+测试用例设计
详细功能
设计
功能测试
代码验证
构建过程
编码
单元测试
验证过程

测试用例的重要性



测试用例的重要性

指导人们系统地进行测试

- 临时性发挥也许会有灵感出现，但是多数情况下会感觉思维混乱，甚至一些功能根本没有测到而另一些功能已经重复测过几遍。
- 测试用例可以帮助你理清头绪，进行比较系统的测试，不会有太多的重复，也不会让你的测试工作产生遗漏。

有效发现缺陷，提高测试效率

- 测试不可能是完备的而且受到时间约束，测试用例可以帮助你分清先后主次，从而更有效地组织测试工作。
- 编写测试用例之后需要标识重要程度和优先级，以便在时间紧迫的情况下有重点地开展测试工作。

测试用例的重要性

作为评估和检验的度量标准

- 测试用例的通过率和软件缺陷的数量是检验软件质量的量化标准，通过对测试用例的分析和改进可以逐步完善软件质量，不断提高测试的水平。
- 测试用例也可以用于衡量测试人员的工作量、进度和效率，从而更有效地管理和规划测试工作。

积累和传递测试的经验与知识

- 测试用例不是简单地描述一种具体实现，而是描述处理具体问题的思路。设计和维护测试用例有助于人们不断积累经验和知识，通过复用测试用例可以做到任何人实现无品质差异的测试。

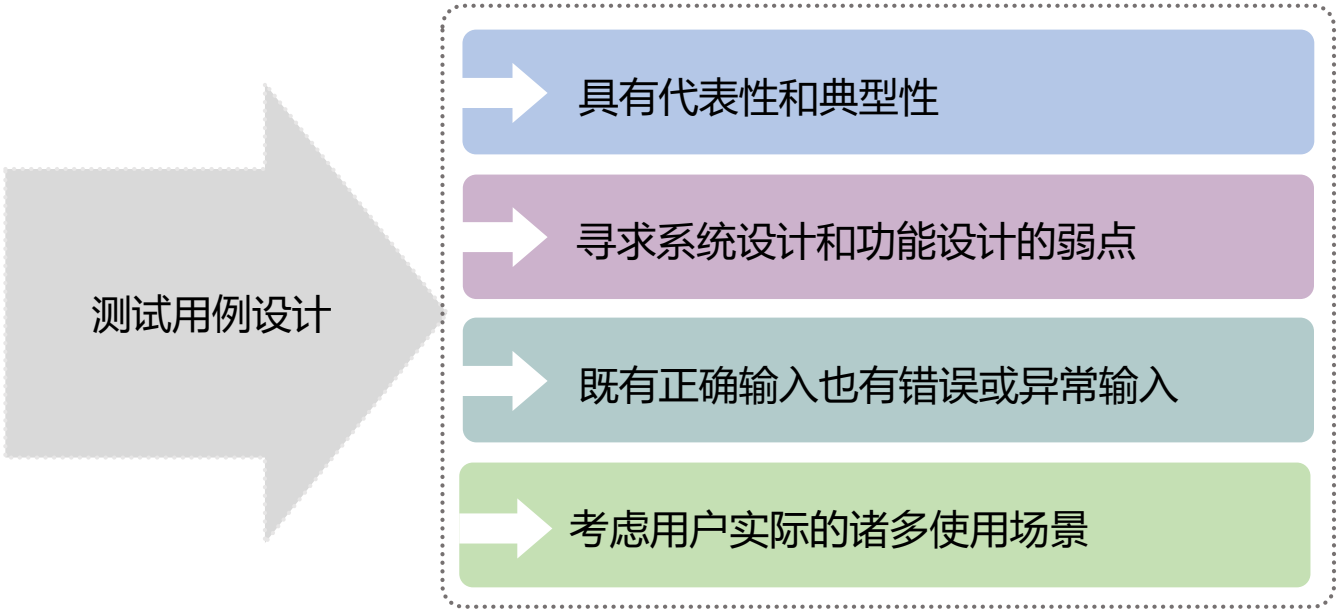
测试用例

测试集
测试用例值的集合

- 测试用例值：完成被测软件的某个执行所需的输入值
- 期望结果：当且仅当程序满足其期望行为，执行测试时产生的结果
- 前缀值：将软件置于合适的状态来接受测试用例值的任何必要的输入
- 后缀值：测试用例值被发送以后，需要被发送到软件的任何输入

验证值：查看测试用例值结果所要用到的值
结束命令：终止程序或返回到稳定状态所要用到的值

测试用例设计要求


测试用例设计
具有代表性和典型性
寻求系统设计和功能设计的弱点
既有正确输入也有错误或异常输入
考虑用户实际的诸多使用场景

案例讨论：纸杯测试

人们在日常生活中经常使用一次性纸杯，请根据自己的生活常识，提出尽可能多的测试用例，并进一步给出设计建议。

教学提纲

3 白盒测试方法

- 控制流图概念
- 语句覆盖和判定覆盖
- 条件覆盖和条件组合覆盖
- 路径覆盖

白盒测试方法

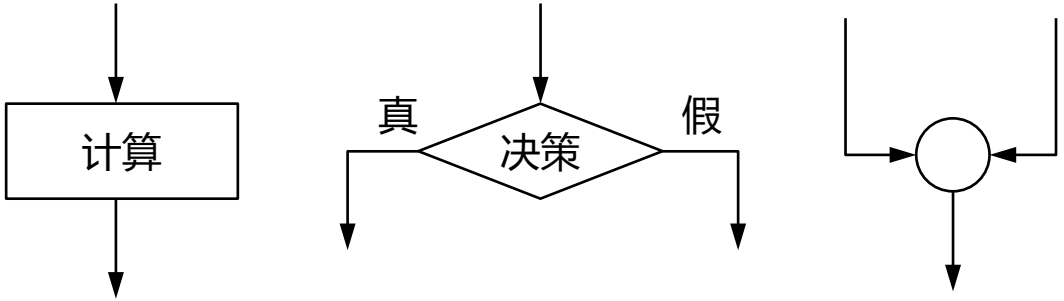
白盒测试需要测试人员阅读和理解被测对象的实现代码及其结构，适当地选择一些执行路径，并且观察所选择的路径是否产生了预期的结果。

控制流图

控制流图（CFG， Control Flow Graph）是一个过程或程序的抽象表示。

控制流图的基本符号：

- 矩形代表了连续的顺序计算，也称基本块
- 节点是语句或语句的一部分，边表示语句的控制流



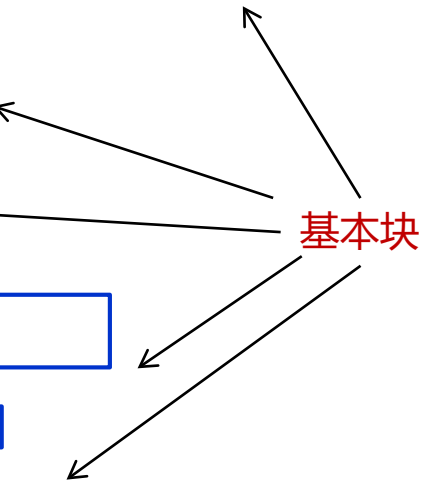
顺序计算　　判断节点　　合并节点

示例：控制流图

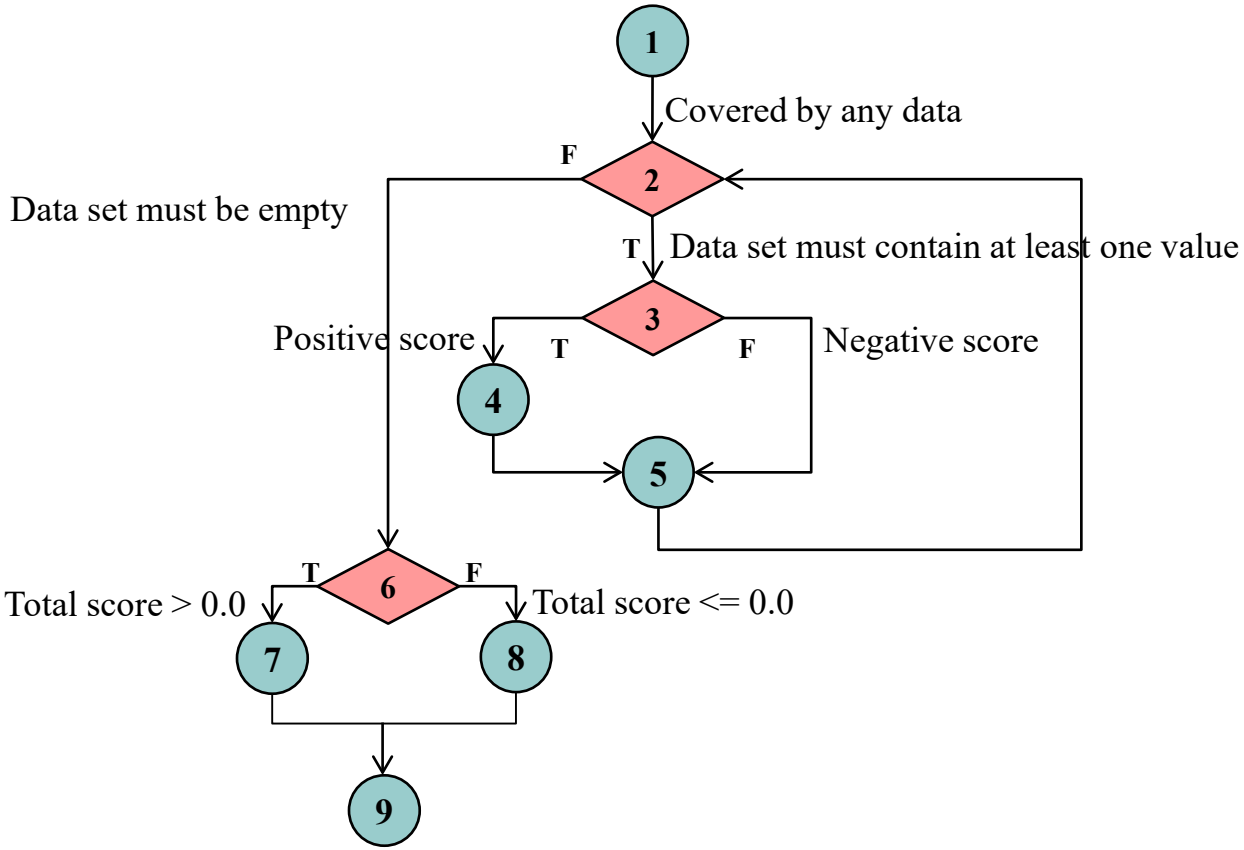
```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}
```

示例：控制流图

```
FindMean(float *mean, FILE *fp)
{
①   float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
②   while (!EOF(fp)) {
③     if (score > 0.0) {
④       sum += score;
        num++;
      }
⑤     fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
⑥   if (num > 0) {
⑦     *mean = sum/num;
      printf("The mean score is %f \n", mean);
    } else
⑧     printf("No scores found in file\n");
⑨ }
```

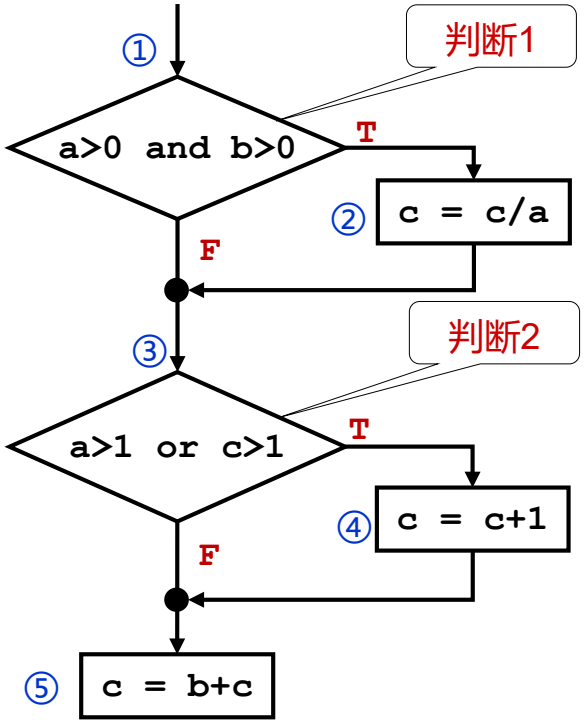


示例：控制流图


1
Covered by any data
F
2
Data set must be empty
T
Data set must contain at least one value
3
Positive score
T
F
Negative score
4
5
T
6
F
Total score > 0.0
Total score <= 0.0
7
8
9

示例描述

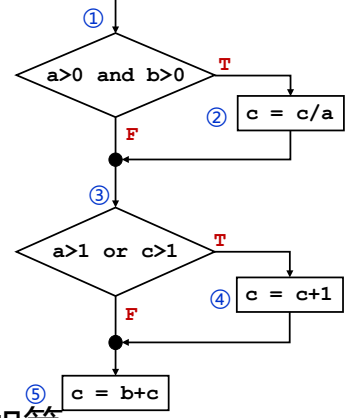
```
double func1(int a, int b, double c)
{
    if (a>0 && b>0) {
        c = c/a;
    }

    if (a>1 || c>1) {
        c = c+1;
    }

    c = b+c;

    return c;
}
```



语句覆盖

基本思想

程序中的每个可执行语句至少被执行一次。



测试用例：

输入：**a=2，b=1，c=6**

程序中三个可执行语句均被执行一次，满足语句覆盖标准。

问题分析：

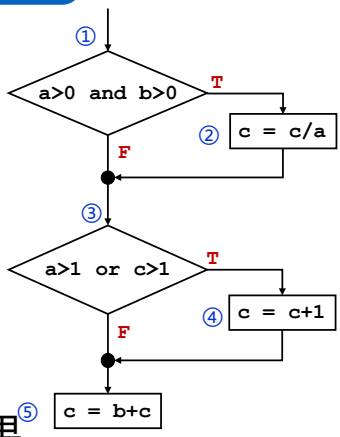
测试用例虽然覆盖可执行语句，但无法检查判断逻辑是否存在问题，例如第一个条件判断中“**&&**”被错误地写成“**||**”。

语句覆盖是最弱的逻辑覆盖准则。

判定覆盖（分支覆盖）

基本思想

程序中每个判断的取真分支和取假分支至少经历一次，即判断真假值均被满足。



测试用例：

输入：a=2,b=1,c=6，覆盖判断1的 T分支和判断2的 T分支

输入：a=-2,b=-1,c=-3，覆盖判断1的 F分支和判断2的 F分支

问题分析：

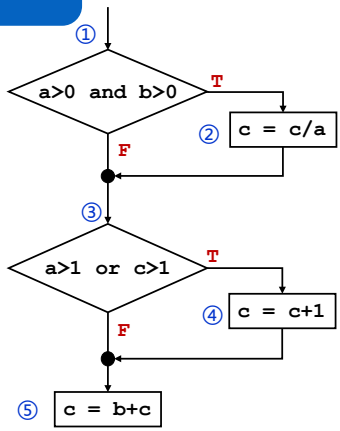
由于大部分判定语句是由多个逻辑条件组合而成，若仅判断其整个最终结果，而忽略每个条件的取值情况，必然会遗漏部分测试路径。

判定覆盖具有比语句覆盖更强的测试能力，但仍是弱的逻辑覆盖。

条件覆盖

基本思想

每个判断中每个条件的可能取值至少满足一次。



测试用例：

输入： a=2,b=-1,c=-2，覆盖 a>0,b<=0,a>1,c<=1 条件

输入：a=-1,b=2,c=3，覆盖 a<=0,b>0,a<=1,c>1 条件

问题分析：

条件覆盖不一定包含判定覆盖，例如以上的测试用例就没有覆盖判断1的T分支和判断2的 F分支。

条件覆盖只能保证每个条件至少有一次为真，而不考虑整个结果。

判定条件覆盖

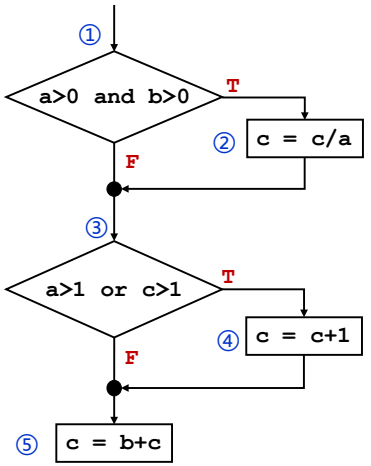
基本思想

判断条件中的所有条件可能取值至少执行一次，同时所有判断的可能结果至少执行一次。

测试用例：

输入：a=2,b=1,c=6，覆盖 a>0,b>0,a>1,c>1 且判断均为T

输入：a=-1,b=-2,c=-3，覆盖 a<=0,b<=0,a<=1,c<=1且判断均为F



问题分析：

判定条件覆盖能够同时满足判定、条件两种覆盖标准。

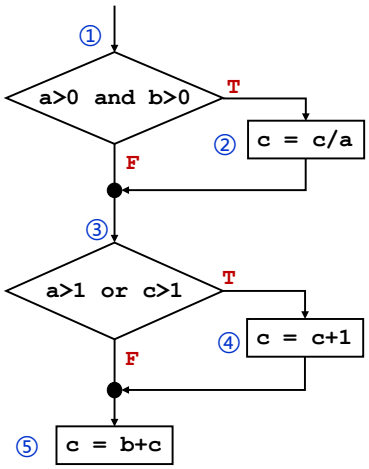
没有考虑条件的各种组合情况。

条件组合覆盖

基本思想

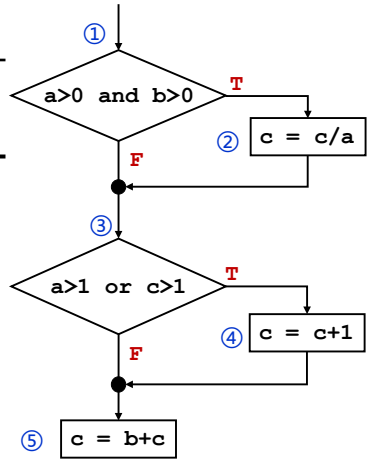
判断中每个条件的所有可能取值组合至少执行一次，并且每个判断本身的结果也至少执行一次。

组合编号	覆盖条件取值	判定条件取值	判定-条件组合
1	a>0，b>0	判断1取T	a>0，b>0，判断1取T
2	a>0，b<=0	判断1取F	a>0，b<=0，判断1取F
3	a<=0，b>0	判断1取F	a<=0，b>0，判断1取F
4	a<=0，b<=0	判断1取F	a<=0，b<=0，判断1取F
5	a>1，c>1	判断2取T	a>1，c>1，判断2取T
6	a>1，c<=1	判断2取T	a>1，c<=1，判断2取T
7	a<=1，c>1	判断2取T	a<=1，c>1，判断2取T
8	a<=1，c<=1	判断2取F	a<=1，c<=1，判断2取F



条件组合覆盖

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0,b>0 a>1,c>1	1-2-4	1, 5
输入：a=2, b=-1, c=-2	a>0,b<=0 a>1,c<=1	1-3-4	2, 6
输入：a=-1, b=2, c=3	a<=0,b>0 a<=1,c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4, 8



问题分析：

条件组合覆盖准则满足判定覆盖、条件覆盖和判定条件覆盖准则。

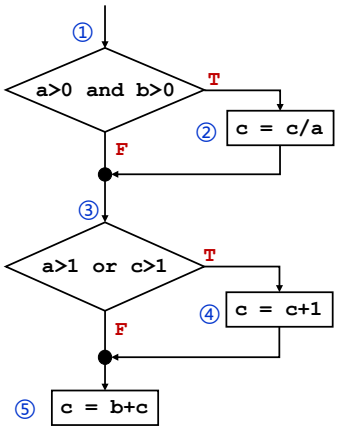
覆盖了所有组合，但覆盖路径有限，上面示例中1-2-5没覆盖。

路径覆盖

基本思想

覆盖程序中的所有可能的执行路径。

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0,b>0 a>1,c>1	1-2-4	1, 5
输入：a=1, b=1, c=-3	a>0,b>0 a<=1,c<=1	1-2-5	1, 8
输入：a=-1, b=2, c=3	a<=0,b>0 a<=1,c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4, 8



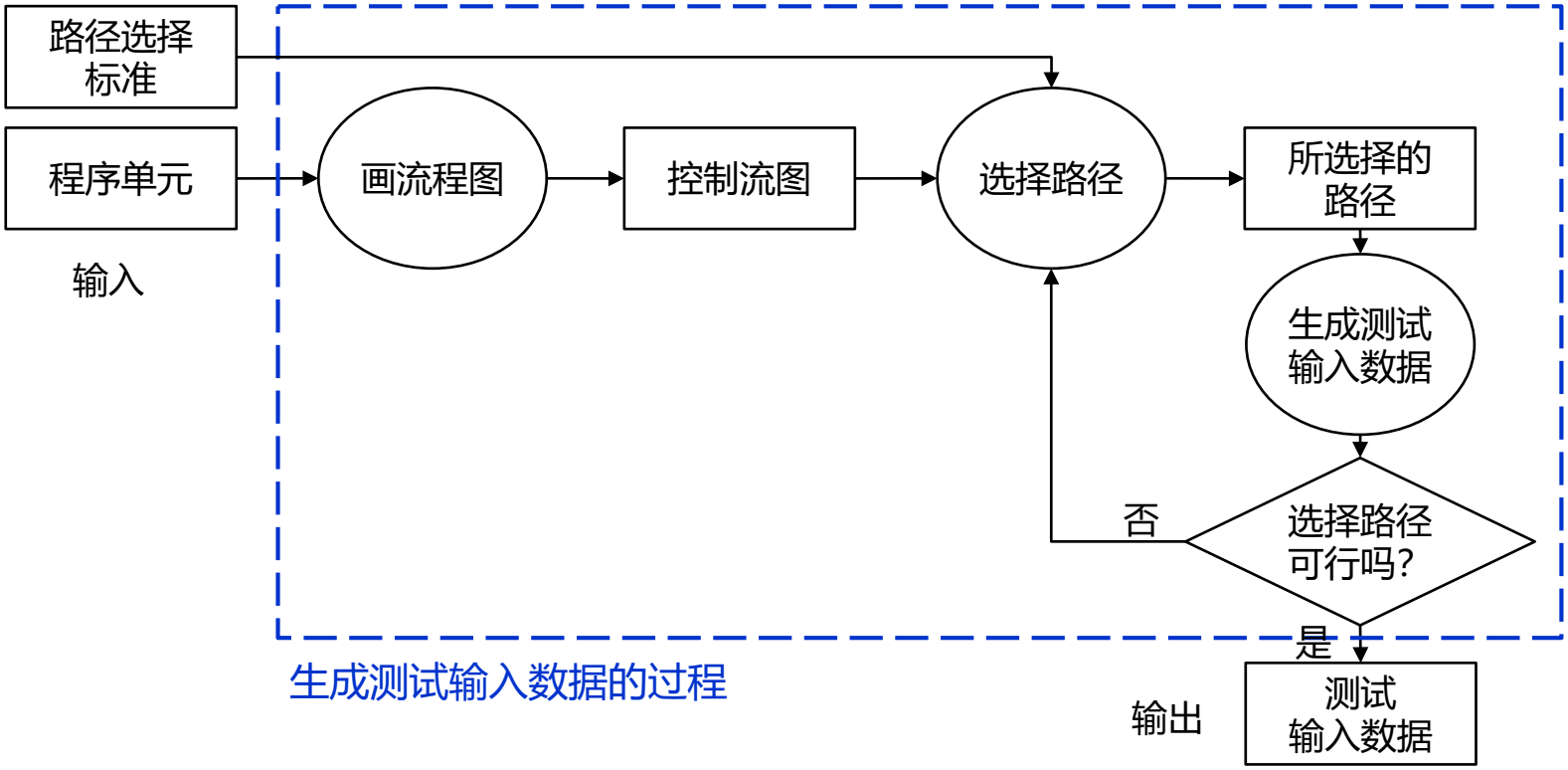
路径覆盖

问题分析：前面的测试用例完全覆盖所有路径，但没有覆盖所有条件组合。

下面结合条件组合和路径覆盖两种方法重新设计测试用例：

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0,b>0 a>1,c>1	1-2-4	1, 5
输入：a=1, b=1, c=-3	a>0,b>0 a<=1,c<=1	1-2-5	1, 8
输入：a=2, b=-1, c=-2	a>0,b<=0 a>1,c<=1	1-3-4	2, 6
输入：a=-1, b=2, c=3	a<=0,b>0 a<=1,c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4, 8

总结：基于控制流的测试


路径选择
标准
程序单元
输入
画流程图
控制流图
选择路径
所选择的
路径
生成测试
输入数据
否
选择路径
可行吗?
是
测试
输入数据
输出
生成测试输入数据的过程

基本路径测试

基本路径测试是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。

具体测试步骤如下：

•绘制程序控制流图：描述程序控制流的一种图示方法；

•计算程序环路复杂度：环路复杂度是一种为程序逻辑复杂性提供的软件度量，可用于计算程序的基本独立路径数。

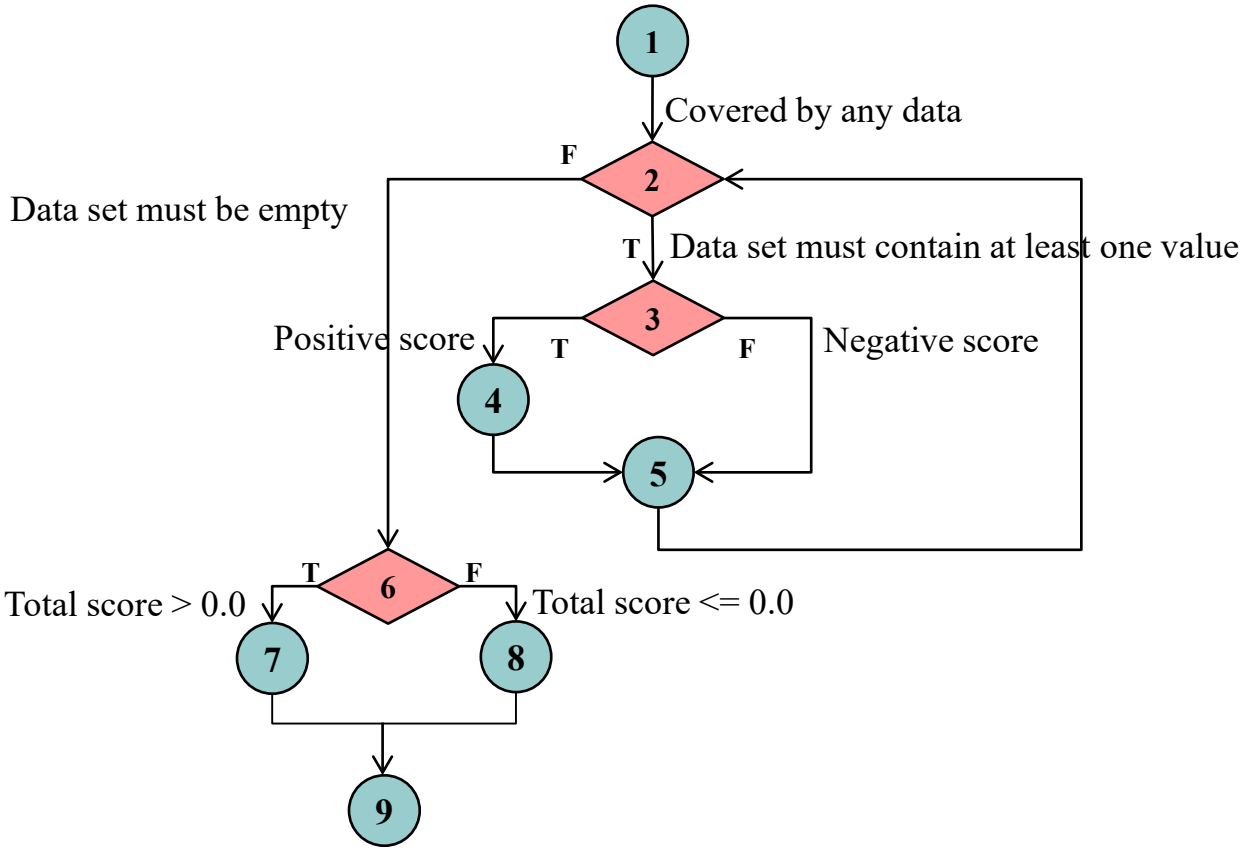
•确定基本路径：通过程序的控制流图导出基本路径集。

•设计测试用例：确保基本路径集中的每一条路径被执行一次。

示例：基本路径测试

```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}
```

示例：基本路径测试


1
Covered by any data
F
2
Data set must be empty
T
Data set must contain at least one value
3
Positive score
T
F
Negative score
4
5
T
6
F
Total score > 0.0
Total score <= 0.0
7
8
9

示例：基本路径测试

环路复杂性：

V(G) = 区域数目

V(G) = 边数 - 节点数 + 2

V(G) = 判断节点数 + 1

示例的结果为4，代表基本路径数

基本路径集：

P1：1 - 2 - 3 - 4 - 5 - 2 - •••

P2：1 - 2 - 3 - 5 - 2 - •••

P3：1 - 2 - 6 - 7 - 9

P4：1 - 2 - 6 - 8 - 9

示例：基本路径测试

设计测试用例：

① 输入：**fp≠NULL**，文件有数据**{1.0}**，覆盖路径**P1**

输出：打印信息“**The mean score is 1.0**”

② 输入：**fp≠NULL**，文件有数据**{-1.0}**，覆盖路径**P2**

输出：打印信息“**No scores found in file**”

③ 输入：**fp≠NULL**，文件无任何数据，覆盖路径**P3**

输出：该路径不可达

④ 输入：**fp≠NULL**，文件无任何数据，覆盖路径**P4**

输出：打印信息“**No scores found in file**”

循环测试

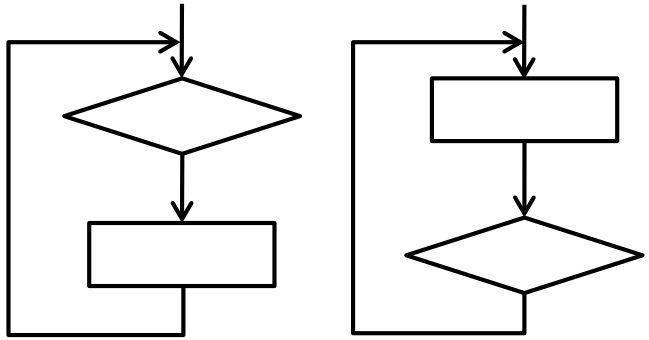
循环测试

•目的：检查循环结构的有效性

•类型：简单循环、嵌套循环、串接循环和非结构循环

简单循环（次数为 n）

- 完全跳过循环
- 只循环 1 次
- 只循环 2 次
- 循环 m （ m < n ）次
- 分别循环 n-1, n, n+1 次

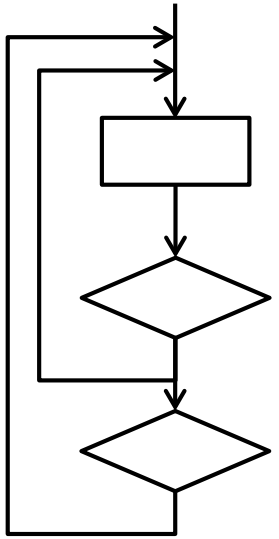


循环测试

嵌套循环

•从最内层循环开始，所有外层循环次数设为最小值；

- 对最内层循环按照简单循环方法进行测试；
- 由内向外进行下一个循环的测试，本层循环的所有外层循环仍取最小值，而由本层循环嵌套的循环取某些“典型”值；
- 重复上一步的过程，直到测试完所有循环。



串接循环

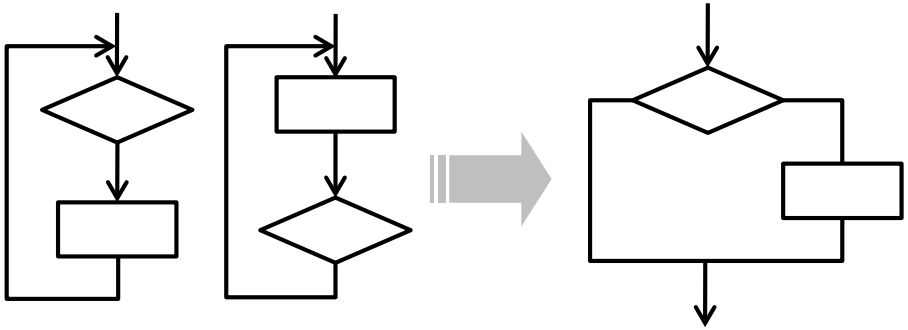
- 独立循环：分别采用简单循环的测试方法；
- 依赖性循环：采用嵌套循环的测试方法。

循环测试

Z路径覆盖下的循环测试

•这是路径覆盖的一种变体，将程序中的循环结构简化为选择结构的一种路径覆盖。

- 循环简化的目的是限制循环的次数，无论循环的形式和循环体实际执行的次数，简化后的循环测试只考虑执行循环体一次和零次（不执行）两种情况。



在循环简化的思路下，循环与判定分支的效果是一样的，即循环要么执行、要么跳过。

Reference

- 清华大学国家级精品课程 《软件工程》主讲人 刘强 副教授 刘璘 副教授
  - https://www.icourses.cn/sCourse/course_3016.html
  - https://www.xuetangx.com/course/THU08091000367/5883555?channel=learn_title

谢谢大家！

THANKS