软件工程：第六章 实现

导学目标

- 了解编码的目的、依据以及分类等
- 掌握软件测试的基本概念等
- 掌握单元测试、集成测试、验收测试等
- 掌握白盒技术和黑盒技术方法
- 了解软件调试的方法和软件可靠性的基本概念

第一节 编码

6.1.1 基本概念

- 实现 => 编码+ 测试
- 编码：把软件设计的结果翻译成某种程序设计语言书写的程序
- 测试：在软件投入生产性运营之前，尽可能多的发现软件中的错误。目前软件测试仍然是保证软件质量的关键步骤。
- 调试：通过测试发现错误后还需要进行错误改正。调试是测试阶段最困难的工作

6.1.2 程序设计语言

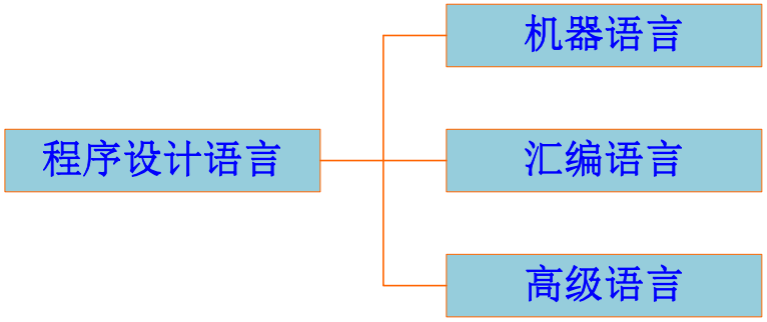
目的

- 把模块的过程性描述翻译为用选定的程序设计语言书写的源程序

依据

- 编码的依据主要是**概要设计**和**详细设计**的**说明文档**

程序设计语言分类



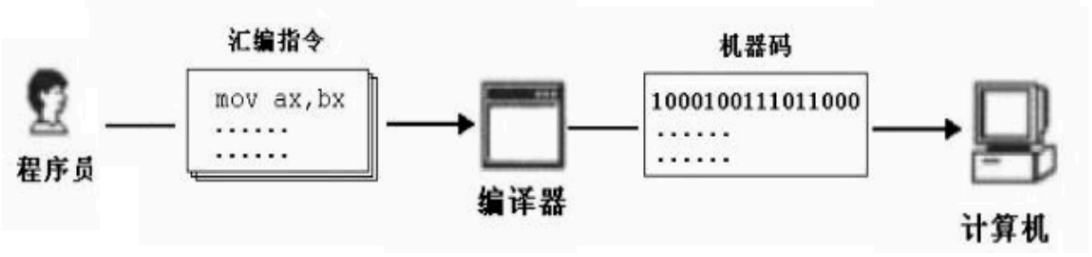
机器语言

- 机器语言是一种**二进制代码**表示的**低级语言**
- 示例
  - 1011011000000000：加法
  - 1011010100000000：减法
- 优点
  - 计算机可以直接识别的
- 缺点

- 不易理解和使用、重用性较差

汇编语言

- 汇编语言是使用**助记符**表示的低级语言，需要经过**汇编程序**翻译成机器语言才能执行
- 示例
  - 机器指令：1000100111011000
  - 汇编指令：MOV AX，BX
- 优点
  - 比机器语言易读写、易调试和修改
  - 执行速度快、占内存少
  - 针对硬件编制
- 缺点
  - 不能编写复杂程序
  - 依赖于机型、不通用、不可移植
- 汇编语言的执行过程



高级语言

- 高级语言是**面向用户**的，接近于自然语言的
- 示例
  - ```
    for（int i=0;i<10;i++）{println（i）}
    ```
- 优点
  - 编码效率高
  - 通用性强、兼容性好，便于移植
- 缺点
  - 运行效率低
  - 对硬件的操作不如汇编

语言选择的标准

- 系统用户的要求
  - 如果开发系统由用户维护，通常要求用他们熟悉的语言书写
- 可以使用的编译程序
  - 运行目标系统环境可提供编译程序限制可选用语言的范围
- 可以得到的软件工具
  - 有支持程序开发的软件工具可以利用
- 工程规模
    ```

    - 规模庞大，现有语言不适用，设计实现供该工程项目使用程序设计语言
- 程序员的知识
    - 如果和其他标准不矛盾，应选择程序员熟悉的语言
- 软件可移植性要求
- 软件的应用领域

6.1.3 编码规则

编码风格

- 逻辑简明清晰、易读易懂是重要标准
- 应该遵循5个方面的规则
    - 程序内部的文档
    - 数据说明
    - 语句构造
    - 输入输出
    - 效率

1、程序内部文档

1. 恰当的描述符
    - 选取含义鲜明的名义
        - 如：`void printScore（ ）`
    - 命名规则要一致
    - 缩写规则要一致
        - 如：`message`缩写为`msg`
2. 适当的注解
    - 序言性注解
    - 中间注解
3. 良好的视觉组织
    - 空格
    - 空行
    - 缩进

2、数据说明

- 数据说明在编写程序时确定
- 数据说明的次序应该标准化（按数据结构或者数据类型说明）
    - 常量->简单变量->数组->公用数据块->文件
    - 整形->实型->字符->逻辑
- 多个变量名在一个语句说明，按字母顺序排列
- 复杂数据结构用注解说明实现方法和特点

3、语句构造

- 不要把多个语句写在同一行

```
student.name = “zs; student.age = 18;
```

- 尽量避免复杂的条件测试
- 尽量减少非条件的测试

```
if（not（a>b））
可替换为
if（a<=b）
```

- 避免大量使用嵌套循环和条件嵌套
- 利用括号使表达式运算次序更加清晰

```
int a[5] = {1，2，3，4，5};
int *p = &a ;
println（*p++）
```

- 避免使用goto语句

4、输入和输出

- 对所有输入数据都进行检验，保证输入有效
- 检查输入项重要组合合法性
- 保持输入格式简单
- 使用数据结束标记，不要求用户指定数据数目
- 提示交互式输入请求，如可用选择或边界数值
- 程序设计语言对格式有严格要求时，应保持输入格式一致
- 设计良好输出报表
- 给所有输出数据加标志

5、效率

1. 程序的运行时间
   - 简化算术和逻辑表达式
   - 嵌套循环，确定是否有语句可从内层往外移
   - 避免使用多维数组
   - 尽量避免使用指针和复杂的表
   - 使用执行时间短的算术运算
   - 不要混合使用不同的数据类型
   - 尽量使用整数运算和布尔表达式
2. 存储器效率
   - 大中型计算机考虑操作系统页式调度特点，将程序功能合理分块，每个模块或一组密切相关程序体积与每页容量相匹配，减少页面调度
   - 微型计算机关键是程序简单性，选择生成较短目标代码且存储压缩性能优良的编译程序
3. 输入输出效率
   - 所有输入 / 输出都应有缓冲，减少通信的额外开销
   - 对二级存储器（如磁盘）选用最简单访问方法

- 二级存储器的输入 / 输出以信息组为单位进行
- 如“超高效”输入 / 输出很难被理解，不采用

第二节 软件测试基础

6.2.1 相关概念

软件测试的目标

- 测试是为了**发现程序中的错误**而执行程序的过程
- 好的测试方案是**极有可能发现迄今尚未发现的尽可能多的错误**的测试
- 成功的测试是**发现了迄今尚未发现的错误**的测试

软件测试的定义

- 为了发现程序中的错误而执行程序的过程

软件测试只能查找出程序的错误，并不能证明程序中没有错误

为什么要测试?

- 软件开发过程必须伴有质量保证活动
- 软件测试时软件质量保证的关键因素
- 软件开发过程中的各个阶段都可能引入新的差错

软件测试的准则

- 所有测试应能追溯到用户需求
  - 测试的目的是发现错误，其中最严重的是不能满足用户需求的错误
- 应尽早地和不断地进行软件测试
  - 不应把软件测试仅看作是软件开发一独立阶段，应把它贯穿到软件开发各阶段中
- 把Pareto原理应用到软件测试中
  - Pareto原理说明，测试中发现80%的错误很可能是由程序中20%的模块造成的
- 测试应从小规模开始，逐步进行大规模测试
- **穷举测试是不可能的**
- **应该由第三方进行测试工作**

黑盒测试与白盒测试

- 黑盒测试（功能测试）：如果**知道产品应具有功能**，可通过测试来检验是否每个功能都能正常使用
- 白盒测试（结构测试）：如果**知道产品内部工作过程**可通过测试来检验产品内部动作是否按照规格说明书的规定正常进行

软件测试的一般过程

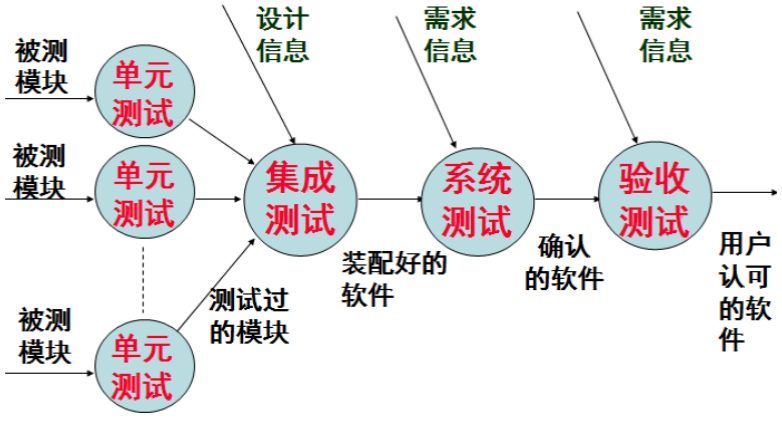
1. 确定测试方案
   - 预定要测试的功能
   - 应该输入的测试数据和预期结果
   - 设计测试用例
2. 设计测试方案的基本目标
   - 确定最可能发现某个错误或者某类错误的测试数据

3. 通常采用黑盒测试，白盒测试做补充
4. 逻辑覆盖
5. 等价值划分
6. 边界值分析（处理边界情况时程序最容易发生的错误）
7. 错误推测

软件测试步骤

1. 模块测试（单元测试）
    - 模块测试的目的是保证每个模块作为一个单元能正确运行，所发现的往往是编码或者详细设计的错误
2. 子系统测试
    - 把通过单元测试的模块组成一个子系统来测试，着重测试的是模块间的接口
3. 系统测试
    - 把经过测试的子系统装配成一个完整的系统来测试，所发现的往往是软件设计和需求说明中的错误
    - **无论是子系统测试还是系统测试，都包含检测和组装两层含义，通常称为集成测试**
4. 验收测试（确认测试）
    - 类似于系统测试，区别是需要用户参与，可能使用的是实际数据进行测试。目的是验证系统是否真正满足了用户需要，发现的问题往往是需求说明书中的错误
5. 平行测试
    - 同时运行新开发出来的系统和即将被它取代的旧系统，以便比较新旧两个系统的处理结果

软件测试步骤如图



第三节 软件测试过程分类

6.3.1 单元测试

基本说明

- 测试之前必须先通过编译程序检查并改正所有错误
- 用**详细设计**说明书做指南，对重要的执行通路进行测试
- 单元测试可以使用**白盒测试**
- 多个模块的测试可以**并行进行**

测试重点

1. 模块接口
    - 数据是否正确进出模块
2. 局部数据结构
    - 检查局部数据说明、初始化、默认值是否有问题
3. 重要执行通路
    - 重要执行路径是否有错误计算、不正确比较或不适当控制流
4. 出错处理通路
    - 错误描述是否能以理解
    - 记下的错误是否与实际错误不同
    - 错误处理之前，错误条件已引起系统干预
    - 错误处理不正确
    - 描述错误的信息不足以帮助定位错误
5. 边界条件
    - 是单元测试中最重要的任务

测试方法

> 结合使用

1. 代码审查（人工）
    - 先由编写人非正式地进行，再由审查小组正式进行，可以查出30%—70%的逻辑错误和编码错误
    - 编写者讲解，审查组审查
    - 预排法
2. 计算机测试
    - 开发驱动软件：作为主程序
    - 开发存根软件：代替被测试的模块所调用的模块

6.3.2 集成测试

基本说明

- 集成测试是测试和组装软件的系统化技术
- 主要目标是发现与接口有关的问题

由模块组装成程序有两种方法

- 非渐增式测试方法
    - 先分别测试每一个模块，再把所有的模块一次组装进行测试
- 渐增式测试方法
    - 把下一个要测试的模块和已经测试好的模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合进来测试

两种方法的优缺点

- 成本开销
    - 非渐增式测试方法分别测试每一个模块，需要编写的测试程序比较多，成本开销较**大**；

    - 渐增式测试方法利用已测试的模块作为部分测试程序，成本开销较**小**
- 发现错误时间
    - 渐增式可以较**快**的发现模块间接口错误
    - 非渐增式最后一次将所有模块组合在一起，因此发现错误较**晚**
- 错误定位
    - 非渐增式方法一下子把所有模块组合在一起，发现错误比较**难定位**；
    - 渐增式方法发现错误，则可以**容易定位**为刚加入的模块有错误
- 测试彻底程度
    - 渐增式测试方法是把已经测试好的模块和新加入的模块一起测试，已测试好的模块可以在新的条件下受到新的校验，测试更加彻底
- 测试进度
    - 使用非渐增式测试方法可以并行测试所有模块，因此能充分利用人力，加快进度

集成测试的两种策略

1、自顶向下集成

- 自顶向下的结合方法是一个日益为人们广泛采用的测试和组装软件的途径
- 由四个步骤完成
    1. 对主控制模块进行测试，测试时用存根程序代替所有直接附属于主控制模块的模块
    2. 根据选定的结合策略（深度优先或者宽度优先），每次用一个实际模块代替一个存根程序
    3. 在结合进一个模块的同时开始测试
    4. 为了保证加入的模块没有引入新的错误，可能需要进行回归测试
- 自顶向下的结合策略能够在测试的早期对主要的控制和关键抉择进行检验

2、自底向上集成

- 自底向上测试是从原子模块开始组装和测试的
- 由四个步骤完成
    - 把低层模块组合成某个特定软件的子功能的族
    - 写一个驱动程序，协调测试数据的输入与输出
    - 对由模块组成的子功能族进行测试
    - 去掉驱动程序，沿软件结构自下向上进行移动，把子功能族组合起来形成更大的子功能族

6.3.3 回归测试

- 重新执行已作过测试的某子集，保证变化没带来非预期副作用
- 三类不同的测试用例
    1. 检测软件全部功能的代表性测试用例
    2. 专门针对可能受修改影响的软件功能附加测试
    3. 针对被修改过软件功能测试

6.3.4 确认测试

也叫验收测试，目标是检验软件的有效性

- 确认：为了保证软件确实满足了用户需求而进行的一系列活动
- 验收：保证软件实现了某个特定要求的一系列活动

验收标准

- 软件的有效性：如果软件的功能和性能如同用户所合理期待的那样，就说明软件是有效的
- 需求分析阶段产生的**需求规则说明书**是软件有效性的标准，也是确认测试的基础

确认测试范围

- 某些已经测过的纯粹技术性的特点可能不需要再次测试
- 对用户特别感兴趣的功能和性能，可能要增加一些测试
- 通常使用生产中的实际数据进行测试
- 可能需要设计并执行和用户使用相关的测试
- 验收测试必须有用户积极参与或者以用户为主
- **验收测试一般使用黑盒测试法**

软件配置复查

- 是验收测试的重要内容，检查软件手册等

Alpha测试

- 用户对即将面市软件产品（称α版本）进行测试，开发者坐在用户旁边，随时记下错误情况和使用中问题，是受控环境下测试

Beta测试

- 在一个或者多个用户场所进行，开发者不在现场

第四节 软件测试技术

6.4.1 设计测试方案

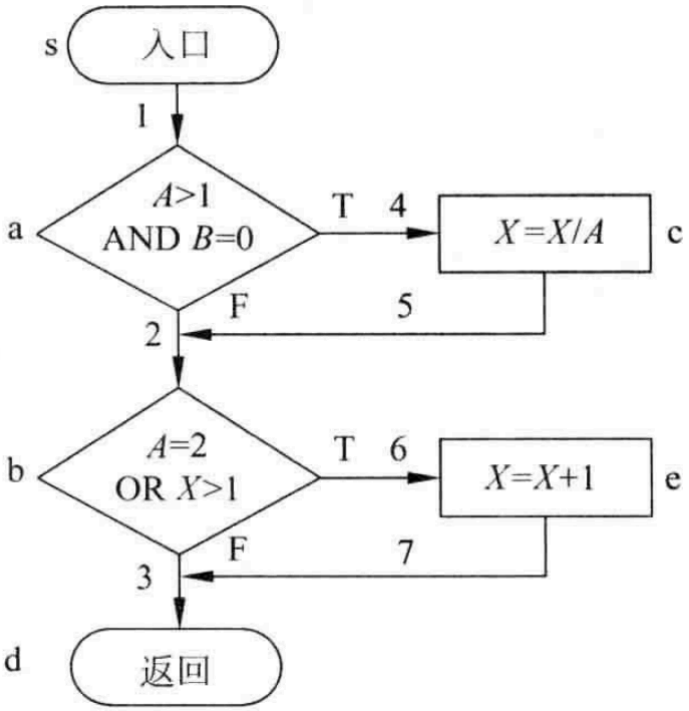
- **是测试阶段的关键性问题**
- **测试方案包括测试目的、输入数据和预期结果**
- **测试用例= 输入数据+ 预期结果**
- 测试方案的基本目标是确定一组最可能发现某个错误或者某类错误的测试数据，其中使用的典型技术是**白盒测试和黑盒测试**

6.4.2 白盒测试

常见的白盒测试技术

- 逻辑覆盖
- 控制结构测试

1、逻辑覆盖



(1) 语句覆盖

- 使程序中的每个语句至少执行一次
- 只需要设计一个测试用例
  - A=2，B=0，X=4 覆盖模块：sacbed 覆盖路径：1-4-5-6-7
- **语句覆盖只关心判断表达式的值，不关心表达式中每个条件取不同值的情况**
- **语句覆盖是最弱的逻辑覆盖**

(2) 判定覆盖

- 又叫分支覆盖，不仅每个语句至少执行一次，每个判定的真假分支至少执行一次
- 设计两组测试用例：
  - A=3，B=0，X=3 覆盖模块：sacbd 覆盖路径：1-4-5-3
  - A=2，B=1，X=1 覆盖模块：sabed 覆盖路径：1-2-6-7
- **两组测试用例可以覆盖所有判定路径的真假**
- **判定覆盖是弱的逻辑覆盖**

(3) 条件覆盖

- 不仅每个语句至少执行一次，而且使判定表达式中的每个条件都取到各种可能的结果
- 设计两组测试用例
  - A=2，B=0，X=4 满足条件：A>1，B=0，A=2，X>1 覆盖模块：sacbed 覆盖支路：1-4-5-6-7
  - A=1，B=1，X=1 满足条件：A<=1，B≠0，A≠2，X<=1 覆盖模块：sabd 覆盖支路：1-2-3
- 条件覆盖**一般**比判定覆盖要强
- 但是**也有可能**是相反的情况
- 满足条件覆盖，但是不满足判定覆盖

- A=2，B=0，X=1　满足条件：A>1，B=0，A=2，X<=1　覆盖模块：sacbed　　覆盖支路：1-4-5-6-7
- A=1，B=1，X=2　满足条件：A<=1，B≠0，A≠2，X>1　覆盖模块：sabed　覆盖支路：1-2-6-7

(4) 判定/条件覆盖

- 既满足判定覆盖，又满足条件覆盖
- 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值，而且每个判定表达式也都取到各种可能的结果
- 判定/条件覆盖**也不一定**比条件覆盖更强
- 设计测试用例
  - A=2，B=0，X=4　满足条件: A>1，B=0，A=2，X>1　　覆盖模块: sacbed 覆盖支路：1-4-5-6-7
  - A=1，B=1，X=1　满足条件: A<=1，B≠0，A<=2，X<=1 覆盖支路: 1-2-3
  - 该组测试用例就是条件覆盖所用的测试用例

(5) 条件组合覆盖

选取足够多的测试数据，使得每个判定表达式中条件的各种可能组合至少出现一次

- 8种组合方式如下：
  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
- 可设计4种测试用例：
  - A=2，B=0，X=4　满足条件【1、5】：A>1，B=0，A=2，X>1　　覆盖模块：sacbed　覆盖支路：1-4-5-6-7
  - A=2，B=1，X=1　满足条件【2、6】：A>1，B≠0，A=2，X<=1　覆盖模块:sabed　覆盖支路:1-2-6-7
  - A=1，B=0，X=2　满足条件【3、7】：A<=1，B=0，A≠2，X>1　覆盖模块：sabed　　覆盖支路：1-2-6-7
  - A=1，B=1，X=1　满足条件【4、8】：A<=1，B≠0，A≠2，X<=1 覆盖模块:sabd　　覆盖支路:1-2-3

(6) 点覆盖

- 使程序执行路径至少经过流图的**每一个节点**一次
- 点覆盖标准和语句覆盖标准相同

(7) 边覆盖

- 使程序执行路径至少经过流图的**每一条边**一次
- 通常边覆盖和判定覆盖是一致的

(8) 路径覆盖

- 使程序中**每条可能的路径都**至少执行一次

2、控制结构测试

- 现有的很多种白盒测试技术，是根据**程序的控制结构**设计测试数据的技术
- 常用的控制结构测试技术
  - 基本路径测试
  - 循环测试

(1) 基本路径测试

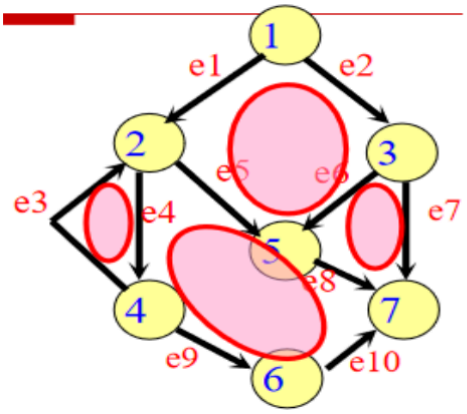
- Tom McCabe提出的一种白盒测试技术，步骤如下：
  1. 根据过程设计的结果画出相应的流图
  2. 计算流图的环形复杂度

```
V（G）＝区域数

V（G）＝E-N＋2          E为流图中边数，N为流图中节点数

V（G）＝P＋1            P为判定点数
```

  3. 确定线性独立路径的基本集合
     - **独立路径**：至少包含一条在定义该路径之前不曾用过的边
     - **环形复杂度**：独立路径基本集的上界
  4. 设计测试用例覆盖基本集合的路径
- 计算环形复杂度示例




```
e=10，n=7

V=e-n+2=5
V=区域数=5
V=判定节点数+1=5
```

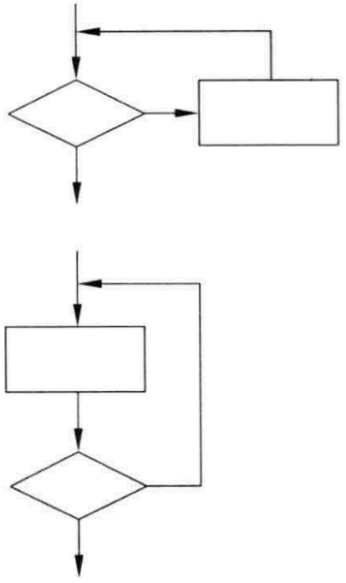
- 独立路径示例

```
路径1：1-2-4-6-7
路径2：1-2-4-2-5-7
路径3：1-2-5-7
路径4：1-3-7
路径5：1-3-5-7
```

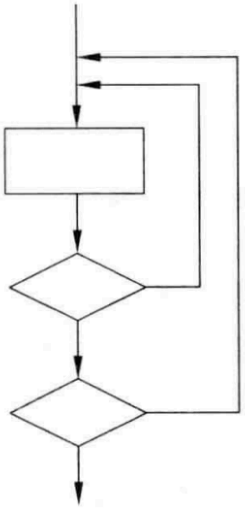
注意：一些独立路径无法独立测试，程序的正常流程不能形成独立执行该路径所需的数据组合，这种情况下这些路径必须作为其他路径的一部分来测试

(2) 循环测试

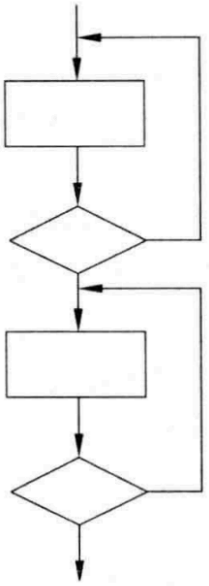
分为3种：简单循环、嵌套循环、串接循环



(a) 简单循环



(b) 嵌套循环



(c) 串接循环

简单循环

- 跳过循环
- 只通过循环一次
- 通过循环两次
- 通过m次，m<n-1（n是允许通过循环的最大次数）
- 通过n-1，n，n+1次

嵌套循环

- 从最内层循环开始，把所有其它层循环设置为最小值
- 对最内层循环做简单循环的全部测试
- 逐步外推，测试时保持所有外层循环变量取最小值，其它嵌套内层循环变量取“典型”值
- 反复进行，直到所有各层循环测试完毕

串接循环

- 各个循环互相独立，可用与简单循环相同方法进行测试
- 几个循环不是互相独立，需要使用测试嵌套循环

6.4.3 黑盒测试

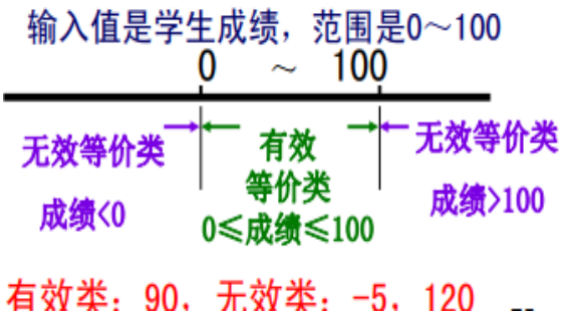
- **黑盒测试着重测试软件功能**
- **黑盒测试并不能取代白盒测试，它们之间是互补关系**
- **白盒测试在测试阶段的早期进行，黑盒测试主要用于测试过程的后期**
- 黑盒测试力图发现的错误类型
    1. 功能不正确或者遗漏了功能
    2. 界面错误
    3. 数据结构错误或外部数据库访问错误
    4. 性能错误
    5. 初始化和终止错误
- 设计黑盒测试方案应该考虑的问题
    1. 怎样测试功能的有效性
    2. 哪些类型的输入可以构成好的测试用例
    3. 系统是否对特定的输入值敏感
    4. 怎样划定数据类的边界
    5. 系统能够承受什么样的数据率和数据量
    6. 数据的特定组合将对系统运行产生什么影响
- **常见的黑盒测试技术**
    - 等价划分
    - 边界值分析
    - 错误推测

1、等价划分

- 把程序的输入域划分成若干数据类，从每一数据类选取少数有代表性数据做为测试用例
- 每类中的一个典型值在测试中的作用与这一类中所有其他值的作用相同
- 划分等价类的标准
    - 覆盖
    - 不相交
    - 代表性

等价类划分原则

1. 如果规定了输入值范围则可以划分出一个有效的等价类（输入值再此范围内）和两个无效等价类（输入值小于最小值或大于最大值）
    - 例如：学生成绩



有效类：90，无效类：-5，120 ...

2. 如果规定了输入数据的个数，类似的可以划分出一个有效的等价类和两个无效的等价类
    - 例如：整形数组的访问

```
int arr [4] ;

实际arr数组的范围          arr[0]-arr[3]
一个有效的等价类：         arr[1]
两个无效的等价类：         arr[-1]、arr[4]
```

3. 如果规定了输入数据的一组值，而且程序对不同输入值做不同处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效等价类（任一个不允许的输入值）
    - 例如：星期值的枚举

```
enum Weekday{Sun、Mon、Tue、Wed、Thu、Fri、Sat}

7个有效等价类：Sun、Mon、Tue、Wed、Thu、Fri、Sat
一个无效类：monta
```

4. 如果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和若干个无效等价类
    - 例如学生的年龄:

```
int age;  （0< age <=100）

一个有效类：  18
多个无效类：-1 ，200，3.5等
```

5. 如果规定了输入数据是整形，则可以划分出正整数、负整数、0 三个有效等价类
    - 例如规定数据输入数据num为整形

```
int num;

num取值范围：-65535-65535
三个有效类：-1，0，1
多个无效类：-70000，70000
```

6. 如果处理的对象是表格，则应该使用空表以及含一项或者多项的表

确认测试用例

1. 建立等价类表，列出所有划分出等价类，为每一等价类规定一唯一编号
2. 设计一新测试用例，尽可能多覆盖尚未被覆盖有效等价类，重复，直到所有有效等价类被覆盖
3. 设计一新测试用例，仅覆盖一尚未被覆盖无效等价类，重复，直到所有无效等价类被覆盖

等价划分实例：

- 某城市电话号码有三部分组成
  - 地区码：空白或三位数字
  - 前缀：非0或1开头的4位数字；
  - 后缀：4位数字
- 解：

输入条件	有效等价类	无效等价类
地区码	空白（1） 3位数字（2）	有非数字字符（5）
		少于3位数字（6）
		多余3位数字（7）
前缀	从2000到9999之间的4位数字（3）	有非数字字符（8）
		起始位为‘0‘（9）
		起始位为‘1‘（10）
		少于4位数字（11）
		多余4位数字（12）
后缀	4位数字（4）	有非数字字符（13）
		少于4位数字（14）
		多余4位数字（115）

2、边界值分析

- 处理边界时，程序最容易发生错误
- 着重测试边界情况，选取测试数据为刚刚等于，刚刚小于，刚刚大于的边界值

3、错误推断

- 靠经验和直觉推测程序可能存在错误，有针对性编写检查这些错误的测试用例

三种技术策略使用方法

- 优先使用边界值分析
- 必要时使用等价划分和错误推断补充测试方案

6.4.4 总结

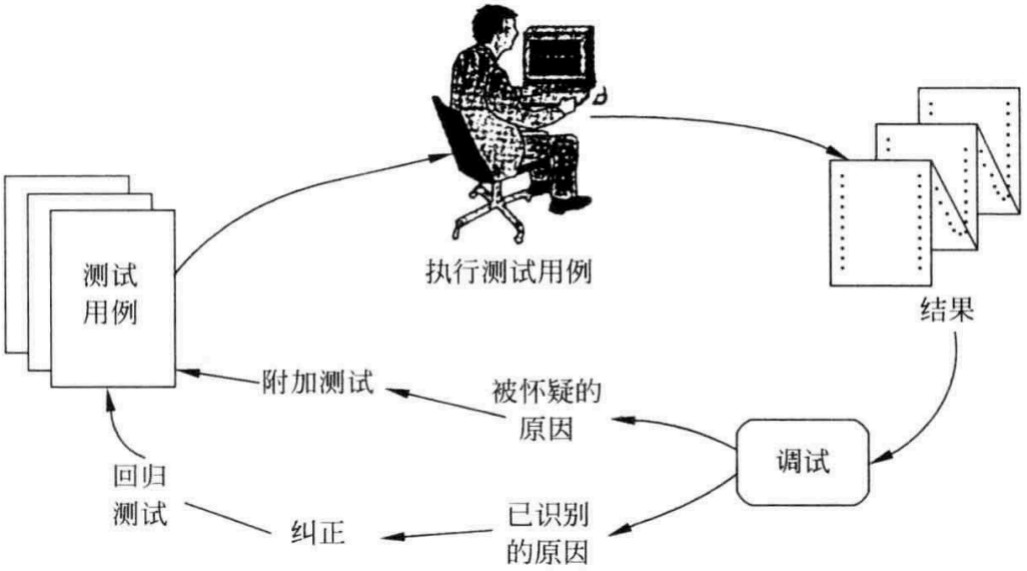
黑盒测试和白盒测试的比较

- 黑盒测试是站在用户的角度，根据需求规则说明书对软件的外部功能进行测试
- 白盒测试是根据程序的内部逻辑进行测试
- 无论黑盒测试还是白盒测试都不能进行穷尽测试

第五节 调试

软件调试是在进行了成功的测试之后才开始的工作。它与软件测试不同，调试的任务是**进一步诊断和改正**程序中潜在的错误

软件调试过程图



软件调试过程步骤

1. 执行测试用例
2. 评估结果，找出错误的内在原因
3. 找到则改正错误，并进行回归测试
4. 未找到，加测试用例证明猜测原因

软件调试过程图

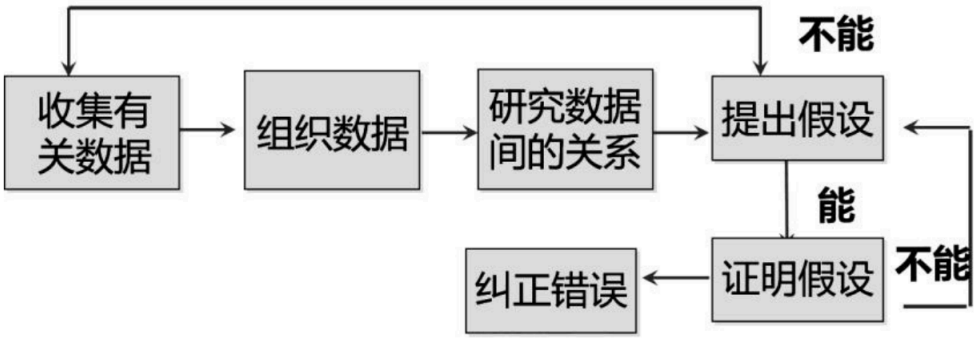
- 蛮干法
  - 将内存内容打印出来分析
  - 程序特定部位设置打印语句
  - 使用调试工具
- 回溯法
  - 确定最先发现“症状”位置。人工沿程序控制流程向回追踪源代码，直到找到错误根源或确定错误产生范围
- 原因排除法
  - 对分查找法、归纳法、演绎法

对分查找法

- 如果已知每个变量在程序内若干关键点正确值，用赋值或输入语句在程序中点附近“注入”正确值，运行程序检查输出
- 正确，错误原因在程序上半部分；反之，在程序后半部分
- 反复使用，将程序出错范围缩小到容易诊断的程度

归纳法

- 归纳法是一种从特殊推断一般的逻辑方法
- 归纳法调试的想法是：从一些线索着手，通过分析它们之间的关系来找出错误
- 归纳法的步骤



演绎法

- 演绎法是从一般原理和前提出发，经过排除和精化的过程来推导出结论的逻辑方法
- 根据已有测试用例，设想所有可能出错原因
- 逐个排除不正确的
- 验证余下假设确是出错原因

第六节 软件的可靠性

基本概念

- **可靠性**：程序在给定时间间隔及环境条件下，按规格说明书的规定，成功运行的概率
- **可用性**：给定的时间点，按规格说明书规定，成功运行概率导学目标