# 第6章　软件测试

- 软件测试基础
- 白盒测试
- 黑盒测试
- 白盒测试和黑盒测试的比较
- 测试用例的有效性

# 软件测试基础

## 软件测试背景

例1：在某撑杆跳横杆自动调节系统中，如果运动员撑杆跳的高度达到6米，则横杆自动升高5厘米。

```
if (Height == 6)
{
    // 横杆自动升高5厘米
    Height  *= 1.05;
}
```

例2：在某嵌入式控制系统中，需反复检测各标志位的状况。

```
int i;
for (i = 0; i < 100000; i++)
{
    //  检测各标志位
}
```

# 软件测试基础

## 软件测试背景

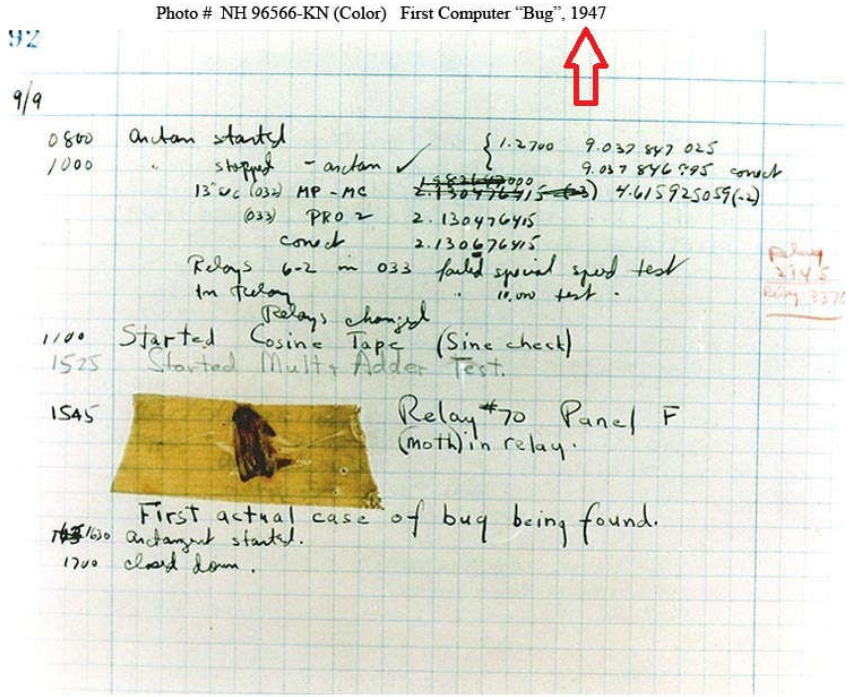
例3：买了一部新手机，（在7日内）如何进行有效测试？

- 检查手机屏幕、外观是否有损坏？是否能正常开机、启动、关机？安装的APP，是否能正常使用？是否能正常接打电话？能够正常上网？
- 能否连续长时间充电？ 能够长时间游戏？追剧？还是上乐学？低电量时手机性能怎么样？
- 同学们还能想到什么.....

# 软件测试基础

## 软件测试——找“bug”的由来

*Grace Murray Hoper*




Lieutenant Grace Hopper is part of a team that finds a moth that is bugging up the Mark II Aiken Relay Calculator at Harvard. After debugging the system, the moth is affixed to the computer log, where Hopper notes: "First actual case of bug being found.“

*——https://www.history.navy.mil/today-in-history/september-9.html*

# 软件测试基础

对于软件测试的定义，有如下不同的描述：

- **IEEE（1983）**：使用人工或自动运行测试系统的过程，其目的在于检验系统是否满足用户需求，或找出预期结果与实际运行结果间的差别，发现程序错误。
- **Glen Myers**：软件测试为了发现错误而执行程序的过程。
- 从软件质量和可靠性角度理解，软件测试是为保证软件质量、提高软件可靠性的活动，它应用测试理论和技术，发现程序中的错误和缺陷而实施的过程。

# 软件测试基础

软件测试过程主要包括测试对象和测试方法两部分内容。

测试对象分为程序代码和文档。

文档分为技术文档和用户文档。

- 对技术文档，检查设计方案是否符合需求；
- 对技术文档，检查设计方案是否正确；
- 对用户手册，提供的系统安装过程说明是否详尽；
- 对用户手册，提供的系统操作方法、过程是否正确；
- 对用户手册，提供的示例数据是否准确；
- 对用户手册，提供的信息是否完整、详尽且无二义性.

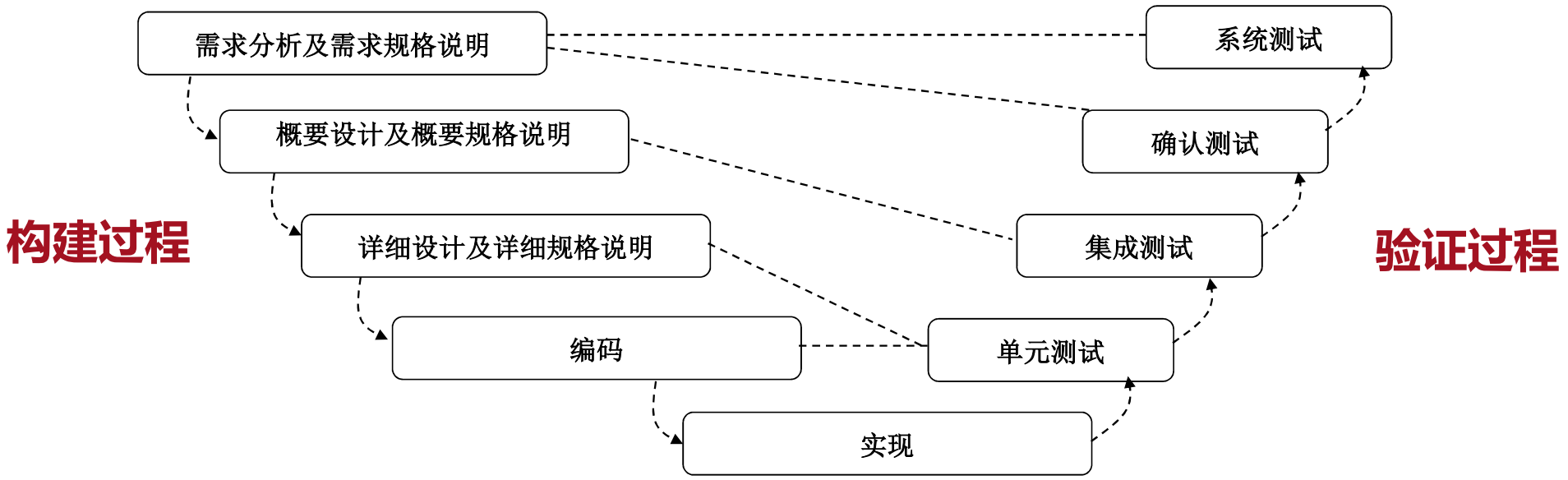
## 测试过程模型


传统软件工程把测试作为独立阶段，直至代码结束才开始测试过程。但这时已经错过发现软件系统结构、业务逻辑、数据设计中存在严重问题的时机，等到代码结束再去查找，将更难以发现这些问题，对这些问题的修改将付出更高的代价。

# 软件测试基础

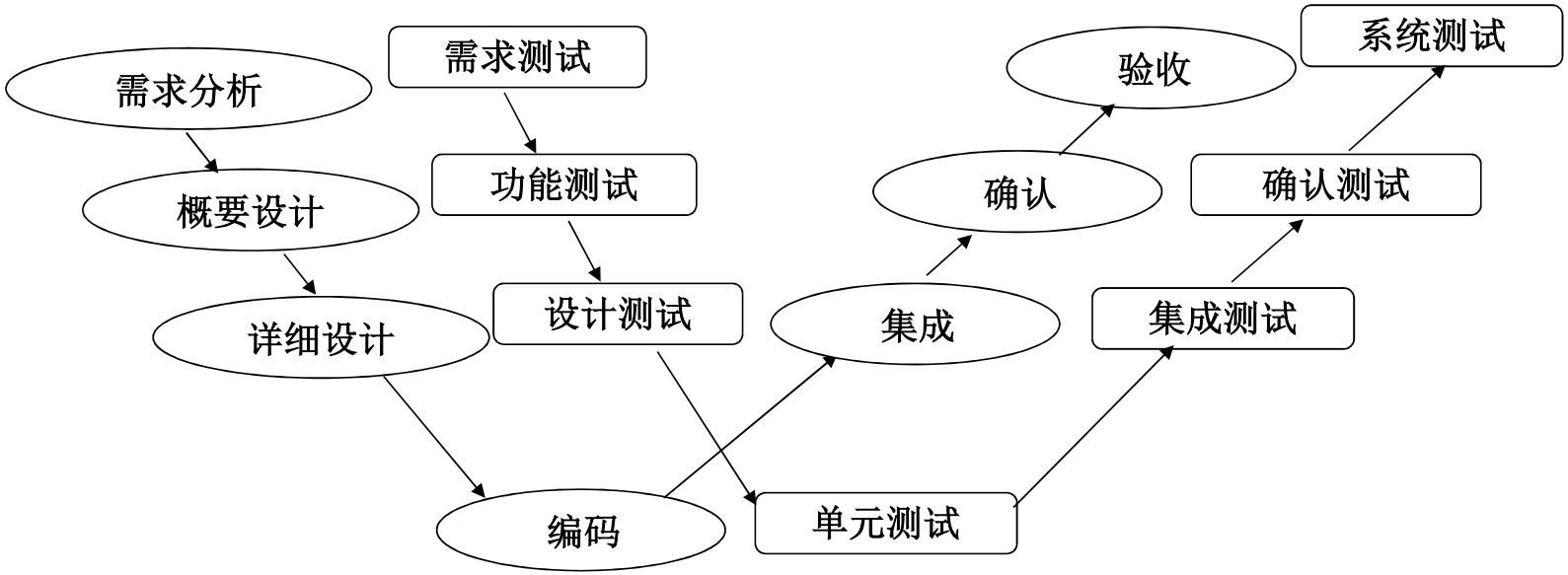
## 1. 软件测试过程模型——V模型

V模型的重要价值在于，它定义了软件测试如何与软件工程各阶段相融合，它清楚地描述了各级别软件测试与软件开发各阶段的对应关系。



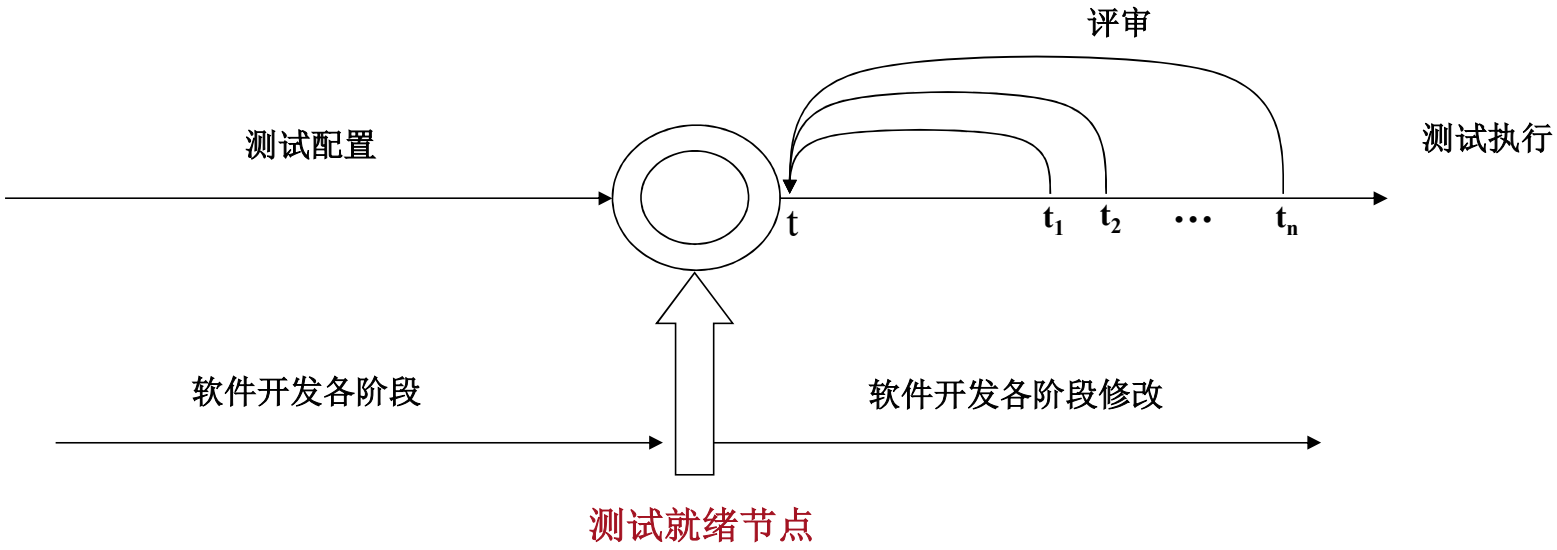
## 2.软件测试过程模型——W模型

W模型的重要贡献在于，明确软件开发各阶段都要进行测试，而不仅仅是在编码结束后才开始。这样，测试的对象不仅是代码，还可以是文档（需求规格说明、设计规格说明等）。



## 3. 软件测试过程模型——H模型

软件测试的H模型是对W模型在更高层次上的线性抽象。它明确表示，在任何一个开发流程，只要有必要，并且测试配置已准备就绪，就能进行测试活动。



# 软件测试基础

## 基本的软件测试技术

- 静态测试与动态测试
- 白盒测试与黑盒测试
- 测试策略

## 软件测试技术分类（一）——静态测试


静态测试的测试对象包括源程序和文档。项目开发过程中产生大量的规格说明，对这些规格说明的技术审查和管理复审，以及对文档的测试数据都属于静态测试。

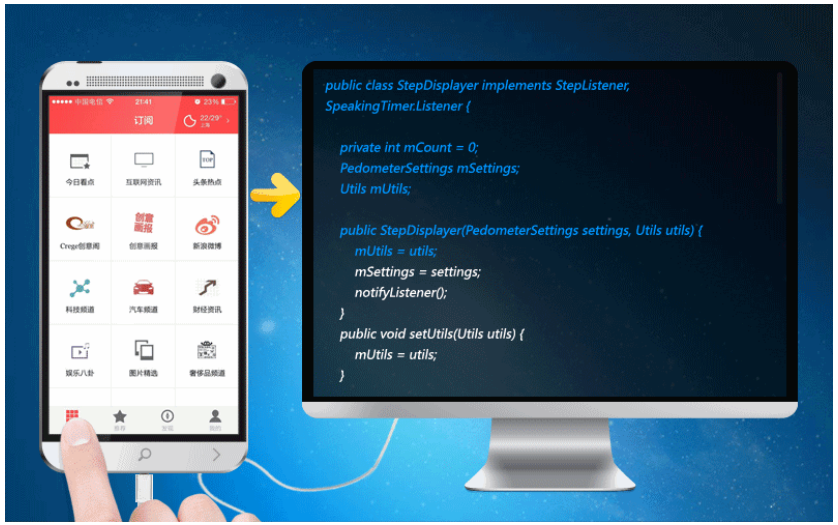
静态测试通过人工分析或程序正确性证明的方式来确认程序正确性。

非正式 ——— 正式

互审　　走查　　审查会议

## 软件测试技术分类（一）——动态测试

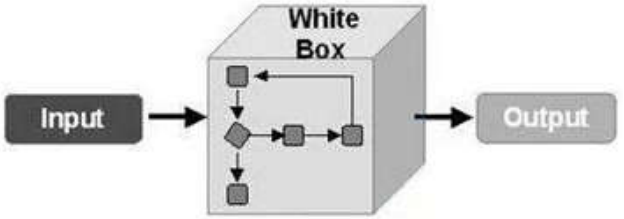
动态测试的测试对象针对源程序。就源程序来讲，静态测试是指不运行程序就找出程序中存在的错误，动态测试是通过运行程序而发现存在的错误和问题。



# 软件测试基础

## 软件测试技术分类（二）——白盒测试

白盒测试又称为结构测试、基于覆盖的测试。它是针对模块内部逻辑结构进行的测试。它面对程序内部的实现细节，分别对语句、条件、条件组合、循环等控制结构、异常、错误处理等特殊流程设计测试用例。



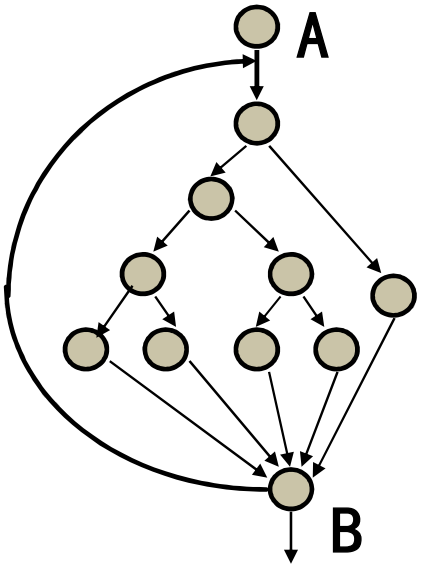

- 白盒测试中的穷尽测试

例：含5个分支，循环次数≤20，从A到B的可能路径：

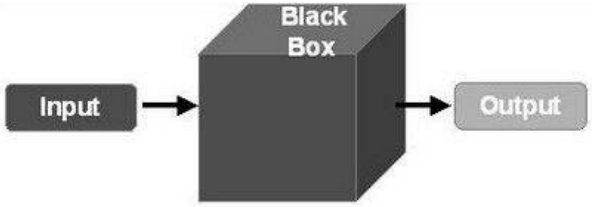
$5^1 + 5^2 + \cdots + 5^{19} + 5^{20} \approx 1.2 \times 10^{14}$

执行时间：设测试一次需要2ms，

则穷尽测试需要约3万年



## 软件测试技术分类（二）——黑盒测试

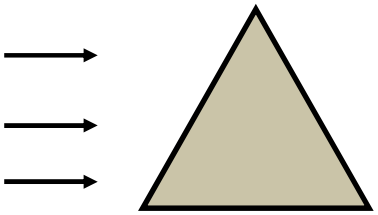



黑盒测试是把模块作为一个整体进行测试。它不关心程序逻辑实现的具体细节，而是关注模块的输入（接口）、输出（运行结果）。因而它测试的是模块功能是否符合设计，运行时是否能（被）正确调用。

- 黑盒测试中的穷尽测试

例:输入三条边长用黑盒测试中的穷尽测试，可采用的测试用例数(设字长32位)：

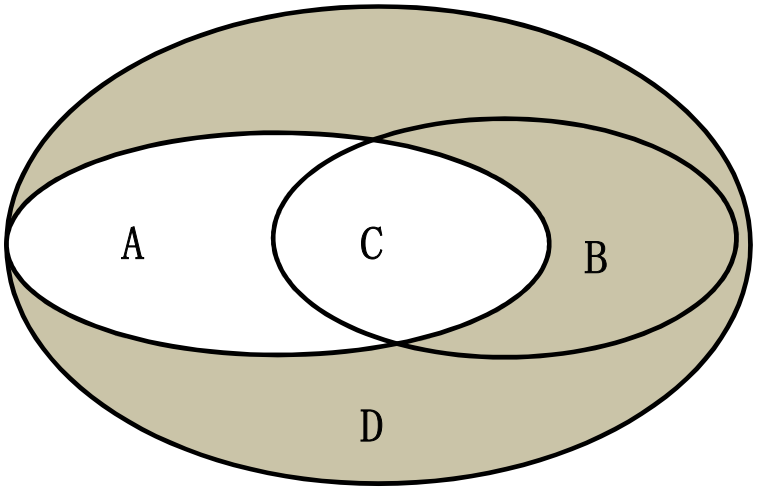
$$2^{32} \times 2^{32} \times 2^{32} \approx 2^{96}$$



执行时间：设测试一次需1毫秒，共需约$2^{61}$年。

# 软件测试基础

在实际测试过程中，无论是黑盒测试还是白盒测试都不能进行穷尽测试，所以软件测试不可能发现程序中存在的所有错误。为此，需精心组织测试方案、设计测试用例，力争用尽可能少的测试用例，发现尽可能多的错误。




黑盒测试与白盒测试能发现错误的关系

A 只能用黑盒测试发现的错误；

B 只能用白盒测试发现的错误；

C 两种方法都能发现的错误；

D 两种方法都不能发现的错误。

# 软件测试基础

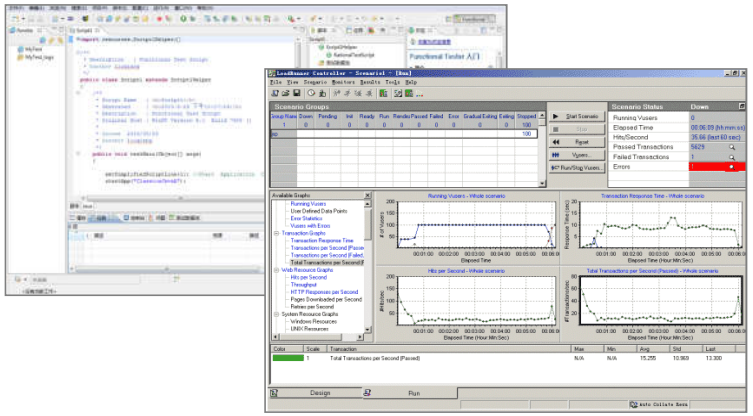
## 软件测试的局限性

### 测试的不彻底性

- 测试只能说明程序中错误的存在，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。

## 白盒测试技术——逻辑覆盖

- 逻辑覆盖准则

## 白盒测试技术——逻辑覆盖

- 逻辑覆盖准则

| 覆盖强度 | 名 称 | 覆盖标准 |
| --- | --- | --- |
| 1 | 语句覆盖 | 程序中的每条语句都至少执行一次。 |
| 2 | 判定覆盖 | 程序中的所有判定的每个分支都至少执行一次。 |
| 3 | 条件覆盖 | 程序中每个判定的各个子关系表达式的取值都至少执行一次。 |
| 4 | 判定/条件覆盖 | 程序中的所有判定的每个分支都至少执行一次。同时，每个判定的各个子关系表达式的取值都至少执行一次。 |
| 5 | 条件组合覆盖 | 程序中每个判定中的各子关系表达式的取值组合都至少执行一次。 |

# 白盒测试

练习：

```
void Example(double x, double y, double z)
{
    if ((y > 1) && (z == 0))    x /= y;
    if ((y == 2) || (x == 1))   x++;
}
```

请分别给出满足语句覆盖、条件覆盖和条件组合覆盖的测试用例。

# 白盒测试

## 白盒测试——循环测试

循环测试是三种基本控制结构之一，循环测试的目的是检查循环结构的有效性。循环分为简单循环、嵌套循环、并列循环和非结构循环四类，如图所示。

对于最多为n次的简单循环，应做下列测试：

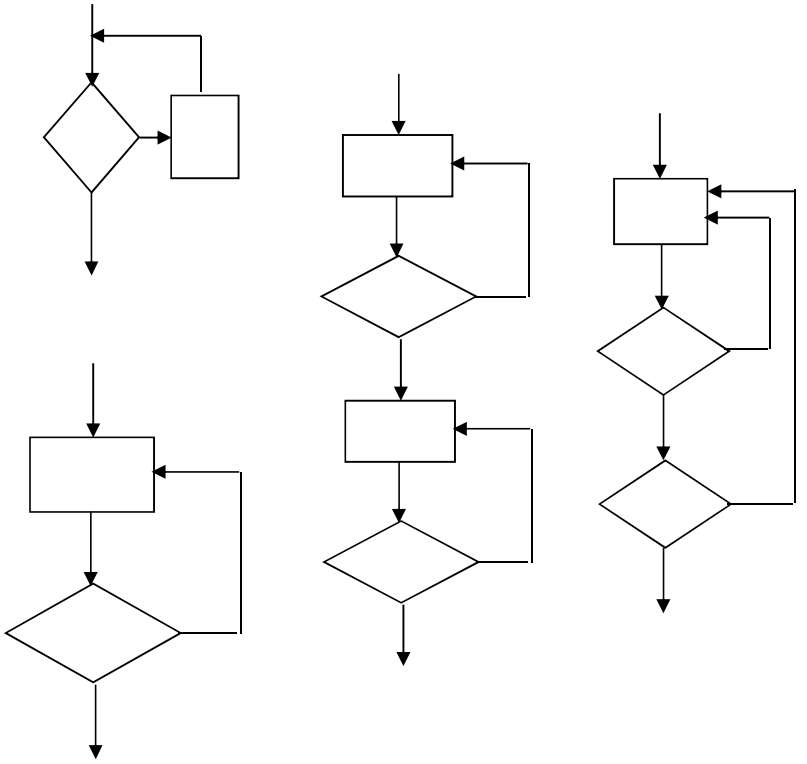
（1）完全跳过循环

（2）仅循环一次；

（3）循环两次；

（4）循环m次，m＜n；

（5）分别循环（n-1)次、n次、n＋1次。

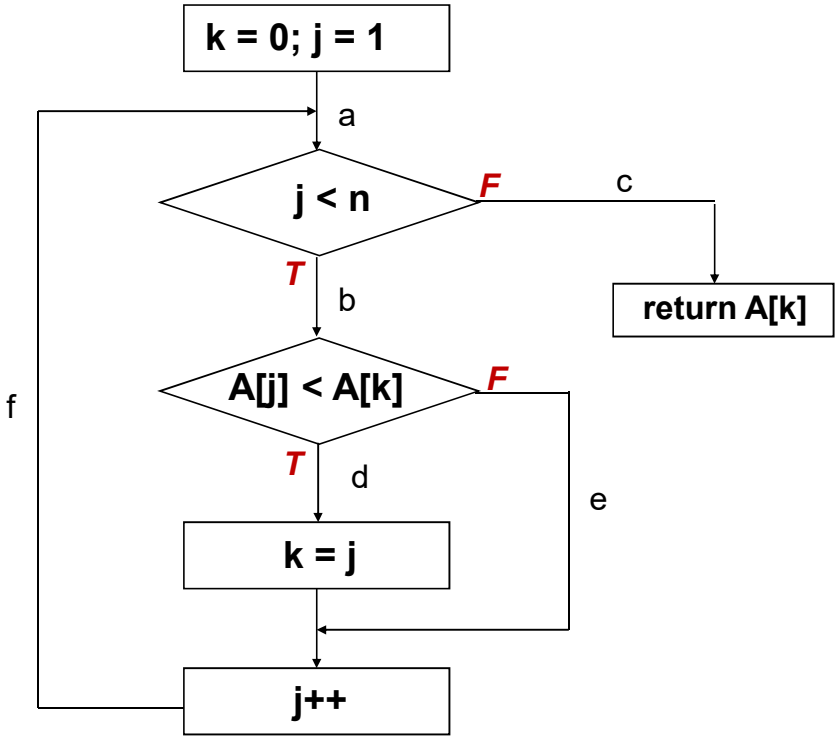


简单循环　　顺序循环　　嵌套循环

## 白盒测试——单循环测试　练习

问题：求整型数组A中元素的最小值。

```
k = i = 0;
for (j = i+1; j < n; j++)
{
        if (A[j] < A[k])
        {
            k = j;
        }
}
return  A[k];
```



# 白盒测试

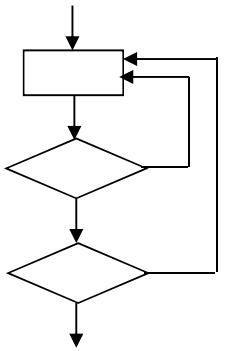
对于嵌套循环若生搬硬套简单循环的测试策略，可能使测试次数成几何级数增长。因此需要在确保循环测试有效性的前提下，采用如下的测试方法以减少测试次数：

（1）从最内层循环开始测试，此时所有外层循环都取最小值，内层循环按简单循环的测试策略测试。

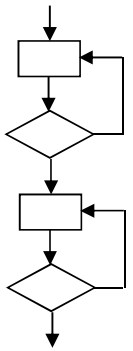
（2）由里向外，回退到上一层循环测试，这层循环的所有外层循环仍取最小值，由该层循环嵌套的那些循环取一些典型值。

（3）继续向外扩展，直至所有循环测试完毕。

（4）对于顺序循环分两种情况，若两个循环完全独立，采用简单循环的测试策略，反之，若第一循环的计数器用作第二循环的初值，即两循环不独立，需借鉴嵌套循环的测试策略。



嵌套循环



顺序循环

# 白盒测试

## 白盒测试——嵌套循环测试　练习

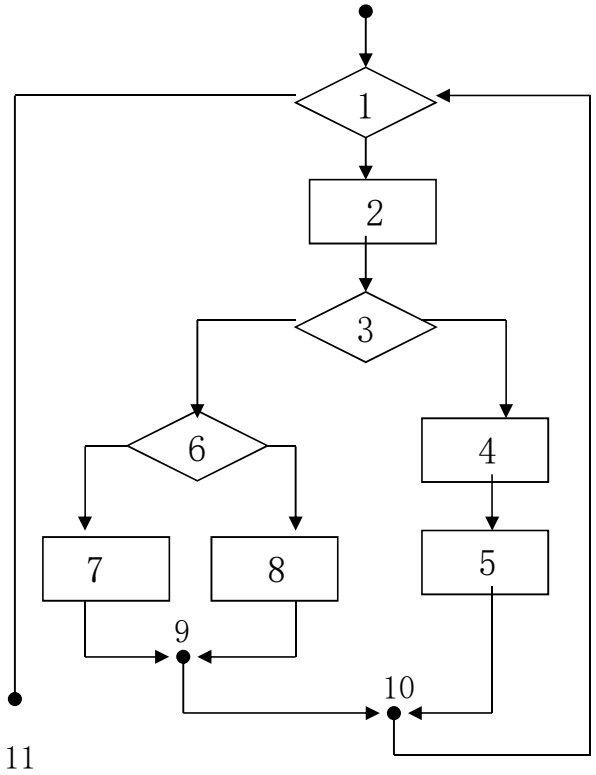
```
for (i = 0; i <= num; i++)
{
    while (j > 0)
    {
        j--;
    }
}
```

思考：

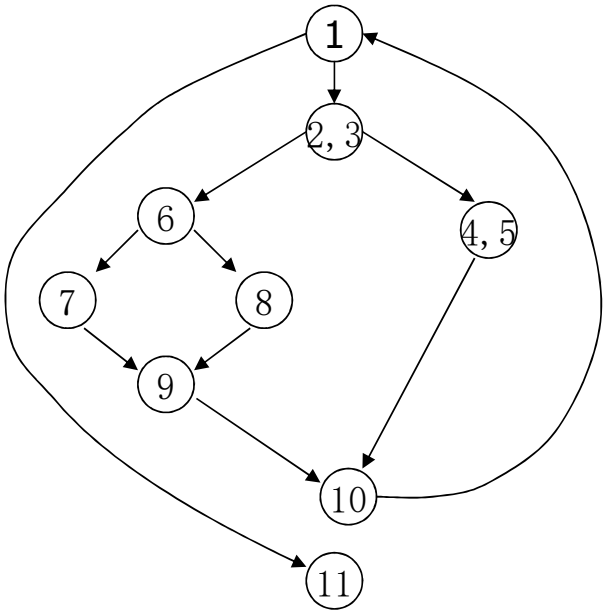
```
for (i = 0; i <= num1; i++)
{
    for (k = i; k <= num2; k++)
    {
        while (j > 0)
        {
            j--;
        }
    }
}
```

# 白盒测试

## 白盒测试——路径测试




待测试程序

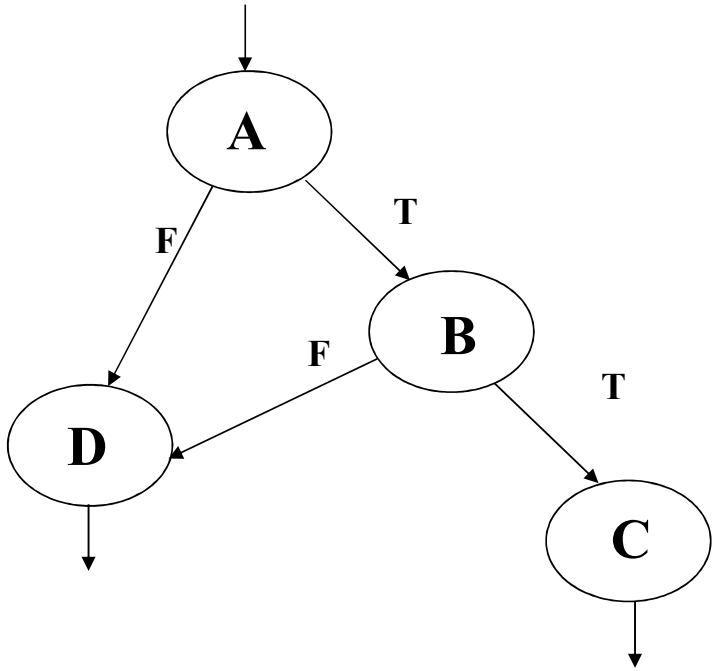



用流图表示的待测试程序

# 白盒测试

流图中“与”节点的逻辑表示

```
if  (A and B)
   then  C
   else  D
```

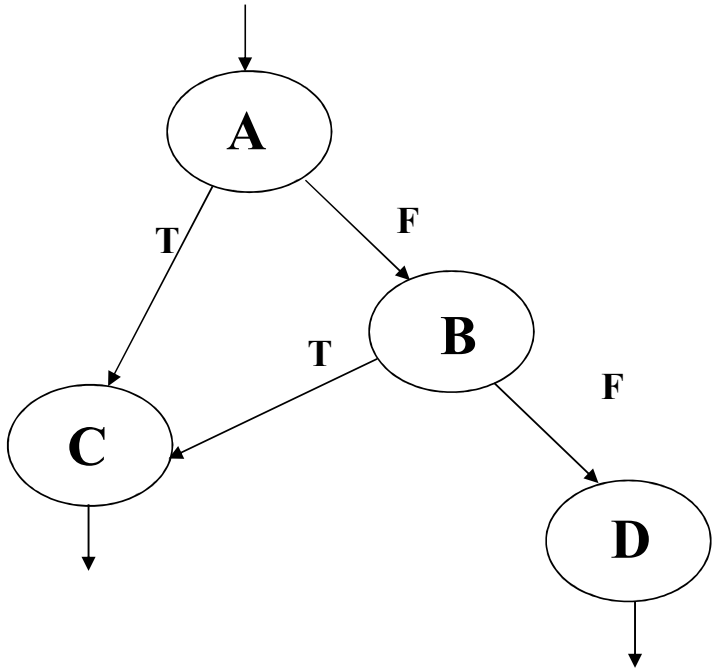



拆分“与”关系逻辑表达式的目的，是在对“**A and B**”进行测试时，确保不会忽略对表达式**B**的测试。

# 白盒测试

流图中“或”节点的逻辑表示

```
if  (A or B)
   then  C
   else  D
```




拆分“或”关系逻辑表达式的目的，是在对“**A or B**”进行测试时，确保不会忽略对表达式**B**的测试。

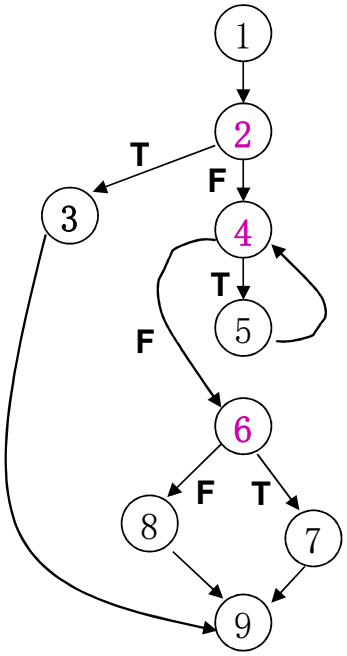
# 白盒测试

## 路径测试练习

```
double GetMean(const string& path) {
    int score, sum = 0, num = 0;          } // (1)
    ifstream ScoreFile(path);

    if (!ScoreFile)   // (2)
       throw("File is NOT Found!");   // (3)
    // Calculate the total score
    while (ScoreFile >> score) // (4)
    {
        sum += score;                     } // (5)
        num++;
    }
    // Compute the mean and print the result
    if (num > 0) // (6)
        return (double)sum / num; // (7)
    else
        throw("No scores found in file"); // (8)
} // (9)
```

1. 画出流图




2. 计算独立路径数:

V(G) = 封闭区域数目

V(G) = 边数 - 节点数 + 2

V(G) = 判断节点数 + 1

3. 给出独立路径及对应的测试用例

Path1: (1) (2) (3) (9)

Path2: (1) (2) (4) (6) (7) (9)

Path3: (1) (2) (4) (6) (8) (9)

Path4: (1) (2) (4) (5) (6) (8) (9)

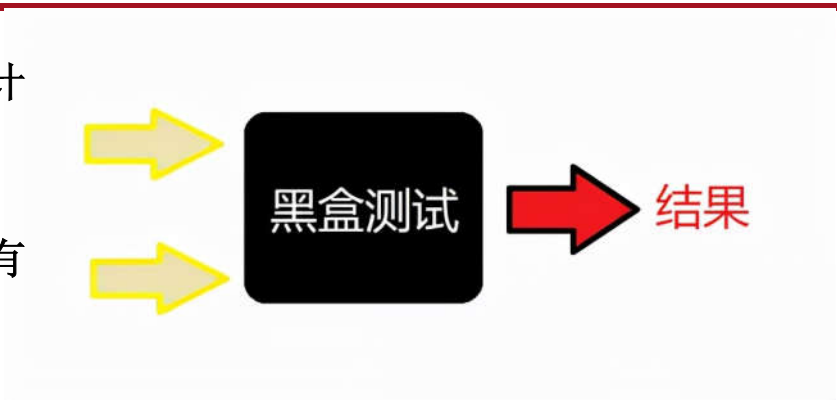
# 白盒测试

## 白盒测试方法的比较

| 方法 | 语句覆盖 | 判定覆盖 | 条件覆盖 | 条件/判定覆盖 | 条件组合覆盖 | 路径覆盖 |
|---|---|---|---|---|---|---|
| 优点 | 最基本、简单的覆盖形式。 | 简单，无需细分每个条件。 | 对容易出错的条件进行的测试。 | 兼顾判定和判定中各条件的取值判断。 | 对程序进行较彻底的测试，覆盖面广。 | 清晰、测试用例有效。 |
| 缺点 | 对于判定、条件、路径都没有涉及，是粗粒度的覆盖。 | 往往大部分的判定语句是由多个逻辑条件组合而成，仅仅判断其组合条件的结果，而忽视每个条件的取值情况，必然会遗漏部分测试场景。 | 测试用例较多，同时也不能完全涵盖判定覆盖。 | 测试用例较多，同时也不能完全涵盖路径覆盖。 | 设计大量复杂的测试用例，工作量较大。 | 不能替代条件组合覆盖。 |

# 黑盒测试

黑盒测试又称功能测试或行为测试，它主要根据设计说明中的功能设计来测试程序能否按预期实现。




黑盒测试的目的是尽量发现系统功能中的错误。常见的系统功能错误有以下几类：

- 功能不正确或不完整；
- 界面或接口错误；
- 数据结构错误；
- 访问外部数据库错误；
- 性能不满足需求；
- 初始化或终止系统时的错误。

# 黑盒测试

## 黑盒测试方法

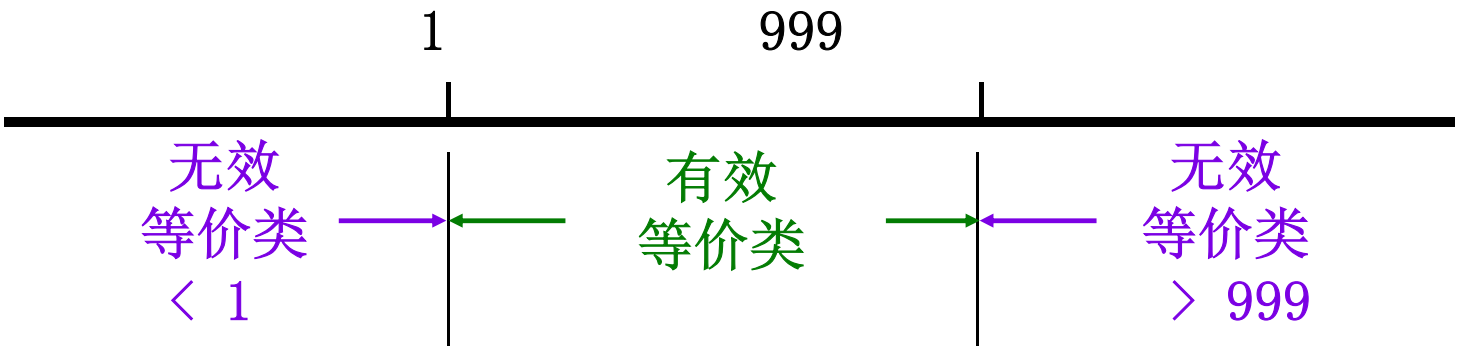
# 黑盒测试

## 黑盒测试——划分等价类的规则

对于常见的数据分析，如特殊数值、区间值、布尔值等，结合各类经验，有以下原则辅助确定等价类：

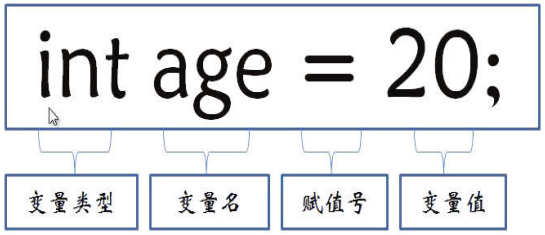
⑴ 如果定义了输入数据的取值范围（如**[a. b]**），则可划分一个有效等价类（**[a. b]**间的数据集）和两个无效等价类（**-∞, a**）以及（**b, +∞**）。

例 输入条件规定：项数可从1到999



# 黑盒测试

## 黑盒测试——划分等价类的规则




⑵ 如果规定了输入数据的个数（如**N**个），则可以划分出一个有效等价类（**1**～**N**之间）和两个无效等价类（**0**个）或（**N+M**个数据）。

例如，在C语言中，对变量标识符规定“以字母或者下划线开头的字符串”。那么所有以字母开头的串构成一个有效等价类、以下划线开头的串构成另一个有效等价类，而不在这两个等价类集合中（不以字母或下划线开头）的串归于无效等价类。

# 黑盒测试

## 黑盒测试——划分等价类的规则


(3) 如果规定输入数据是特殊值，则特殊值集合是有效等价类，其余取值构成一个无效等价类。

(4) 如果输入数据是布尔量，则可划分出一个有效等价类和一个无效等价类。

(5) 如果定义了输入的数据规则，则可划分出一个符合规则的有效等价类和一个违反规则的无效等价类。

(6) 如果输入的数据是整型，则可划分负数、零和正数三个有效等价类。

(7) 对于上述各自划分的有效等价类和无效等价类，可以根据不同角度、规则、程序处理方式等各方面入手，再细分为若干有效或无效的等价子类。

# 黑盒测试

## 黑盒测试——划分等价类的步骤

第一步：根据输入数据，划分待测问题的等价类，并对每个等价类进行编号；

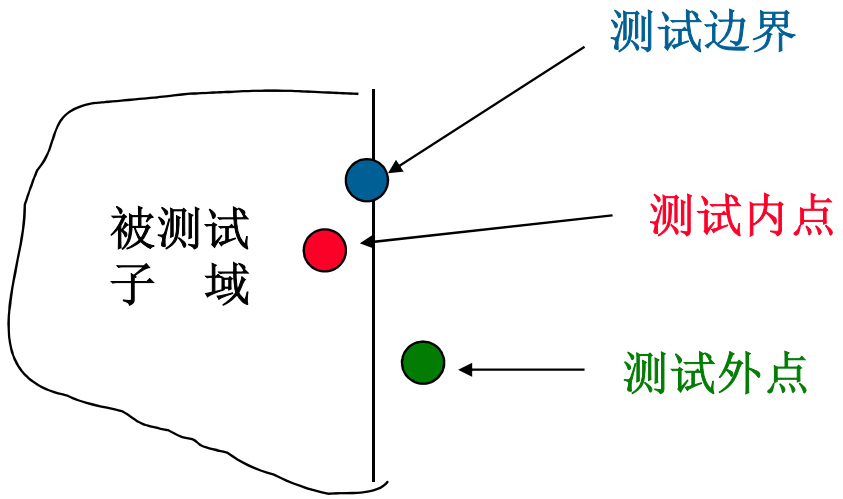
第二步：优化等价类（合并或拆分），并对每个等价类设计对应的测试用例。

练习：设有一个档案管理系统，要求用户输入以年月表示的日期。假设日期限定在1990年1月～2049年12月，并规定日期由6位数字字符组成，前4位表示年，后2位表示月。现用等价类划分法设计测试用例，对“日期检查”模块进行功能测试。

# 黑盒测试

## 黑盒测试——边界值分析

边界值分析法是对等价分类技术的补充，即在一个等价类中不是任选一个元素作为此等价类的代表进行测试，而是选择此等价类边界上的值。



# 黑盒测试

## 黑盒测试——边界值分析法

例：在做三角形计算时，要输入三角形的三个边长：A、B和C。 我们应注意到这三个数值应当满足

A＞0、B＞0、C＞0、

A＋B＞C、A＋C＞B、B＋C＞A，才能构成三角形。但如果把六个不等式中的任何一个大于号“＞”错写成大于等于号“≥”，那就不能构成三角形。问题恰出现在容易被疏忽的边界附近。

# 黑盒测试

## 边界值分析设计测试用例原则

(1) 如果输入数据给定了范围，则对于范围的边界，定义比边界值少1、边界值、比边界值多1的数值设计测试用例。例如[a, b]整数区域，则取a-1、a、a+1，b-1, b, b+1作为测试用例。

例1: 邮件收费规定 1～5 kg收费2元，则应根据数据精度要求，设计如下不同的测试用例:

0.9，1，1.1, 4.9, 5，5.1 kg

或 0.99，1，1.01, 4.99, 5，5.01 kg

(2) 如果规定了输入数据的个数N，则设计0个数据、1个数据、2个数据、N－1个数据、N个数据和N＋1个数据等的测试用例。

例2:一个输入文件可有1～255个记录则可分别设计有：0个、 1个、2个、254个、255个、256个记录的输入文件。

# 黑盒测试

(3) 如果没有指定值区域，则应取计算机所能表达的最大值和最小值作为测试用例。例如：整数、浮点数。

(4) 如果输入数据是有序集，则取该集中第一、第二，倒数第二和最后一个元素作为测试用例。

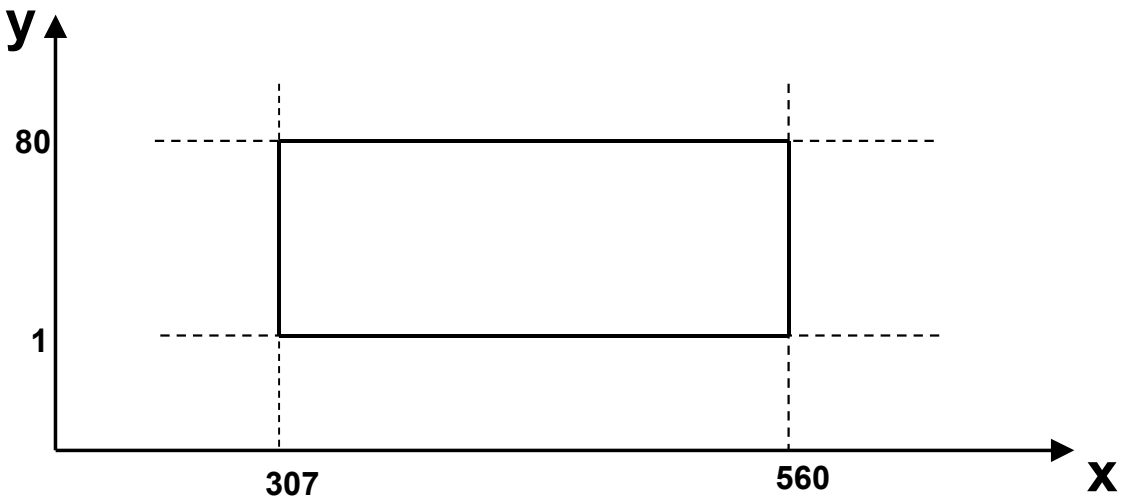
例：有两个输入变量 x(**307 ≤ x ≤ 560**) 和 y(**1 ≤ y ≤ 80**)的程序，请设计进行边界值分析测试的测试用例。

左边界：(306，50)、(307，50)、(308，50)

右边界：(559，50)、(560，50)、(561，50)

下边界：(400，0)、 (400，1)、(400，2)

上边界：(400，79)、(400，80)、(400，81)



# 黑盒测试

## 黑盒测试——错误推测法

# 黑盒测试

## 黑盒测试与白盒测试比较

| | 黑盒测试 | 白盒测试 |
|---|---|---|
| 优点 | ▶适用于各测试阶段<br>▶从产品功能角度测试<br>▶容易入手生成测试数据 | ▶可以构成测试数据使特定程序部分得到测试<br>▶有一定的充分性度量手段<br>▶可获得较多工具支持（反向工程） |
| 缺点 | ▶某些代码段得不到测试<br>▶如果规格说明有误则无法发现<br>▶不易进行充分性度量 | ▶不易生成测试数据<br>▶无法对未实现规格说明的部分测试<br>▶工作量大，通常只用于单元测试，有引用局限 |

# 软件测试策略

- 软件测试策略
- 调试
- 软件测试报告

# 软件测试策略

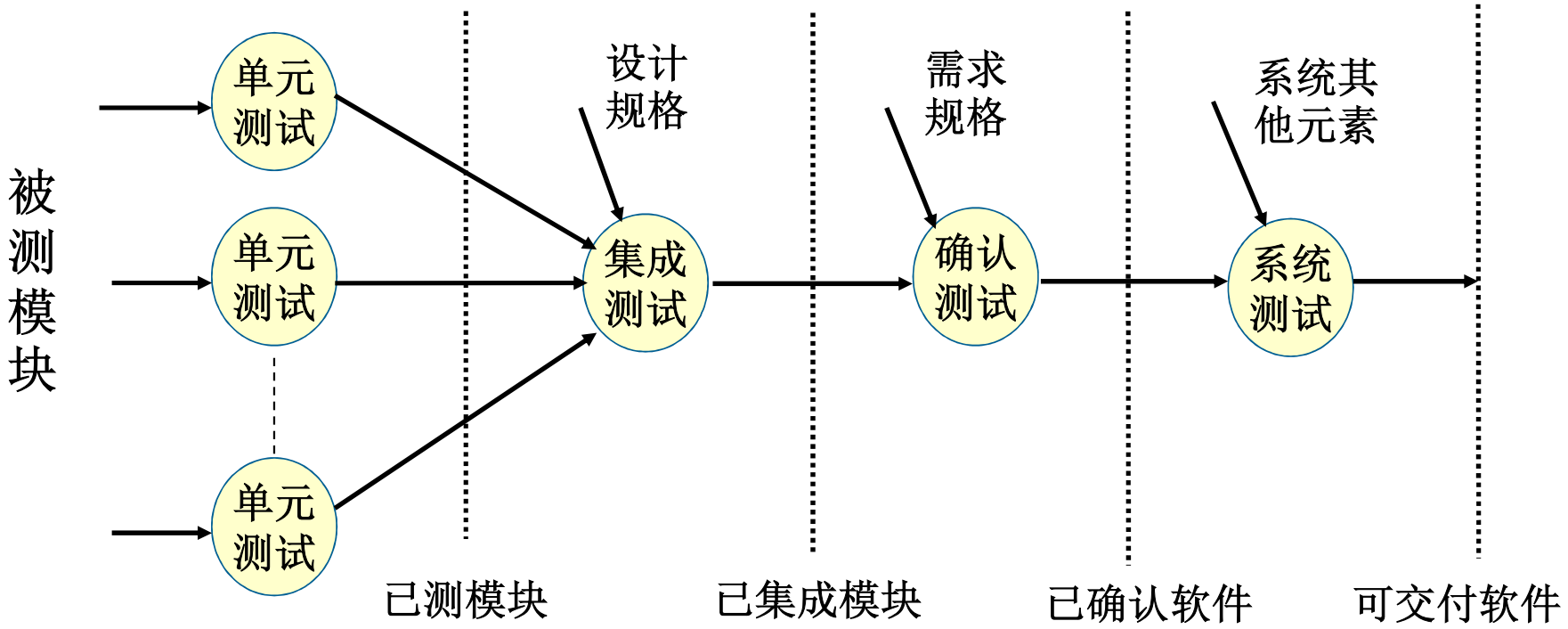
软件测试阶段完成的主要任务有两类：

- 一类是局部模块的测试，它是整个测试阶段的基石。
- 另一类是软件系统全局结构的测试，它构成整个测试系统的大厦。
- 从局部到全局，是经过一系列测试过程转换而成，它们包括单元测试、集成测试、确认测试和系统测试，每个测试过程都有各自不同的测试策略。

# 软件测试策略

## 测试步骤及策略

所有测试过程都应采用综合测试策略；即先作静态分析，制订测试计划，再进行动态测试。



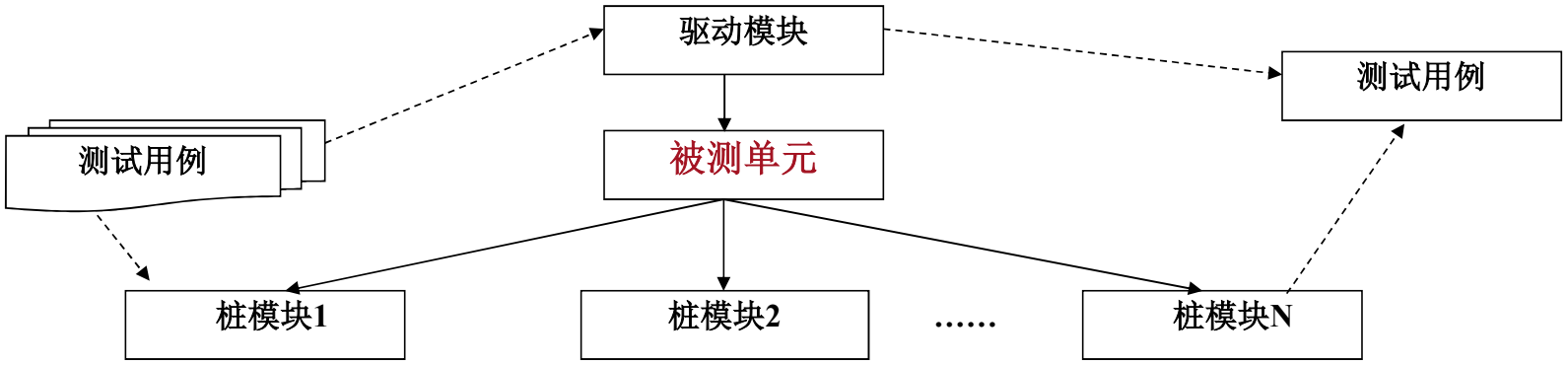
# 软件测试策略

## 1. 单元测试——测试策略

在如今的互联网时代，软件迭代的速度越来越快，研发的职责也越来越多。DevOps的理念是“you build it, you run it”，研发/测试合二为一的趋势可被理解为：”you build it, you test it”。

单元是软件测试过程中最小的测试单位，它有两个基本属性：

- 单元是可以独立编译的最小组件（函数、接口、类）；
- 单元是由个人开发的软件组件（菜单、界面）。



# 软件测试策略

## 1. 单元测试——MOCK

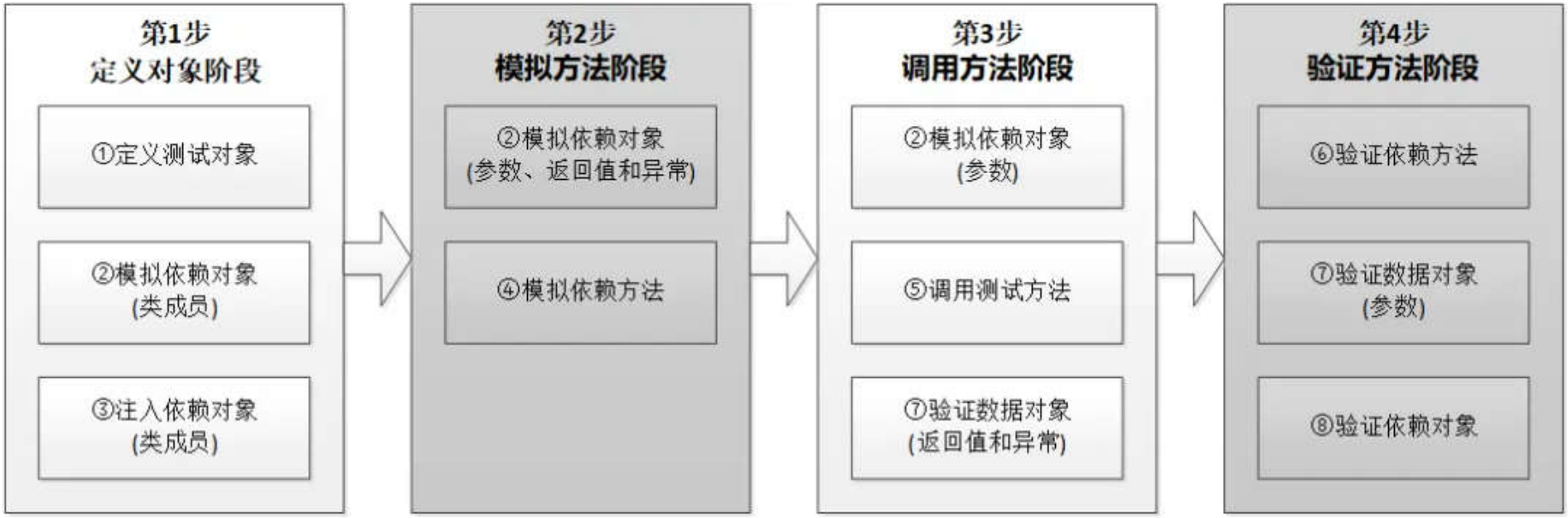
Mock测试是在测试过程中对于某些不容易构造，或者不易获取的对象，用一个虚拟对象（即Mock对象）来创建，以便进行测试的方法。

- 真实对象有难以确定的行为，会产生不可预测的结果；
- 真实对象很难被创建，如具体的Web容器；
- 真实对象的某些行为很难触发，如网络错误；
- 真实情况令程序的运行速度很慢；
- 真实对象有用户界面；
- 测试需要询问真实对象它是如何被调用的。

# 软件测试策略

## 1. 单元测试——MOCK

- 单元测试编写流程——对象间依赖是必然存在的



# 软件测试策略

## 2. 集成测试——测试策略

集成测试的重点，是将各模块按照软件系统结构的定义，将软件模块组装在一起。如何实施组装过程，使得系统既能快速集成，又能准确发现在组装过程中出现的错误和问题呢？这需要正确的集成测试方法。

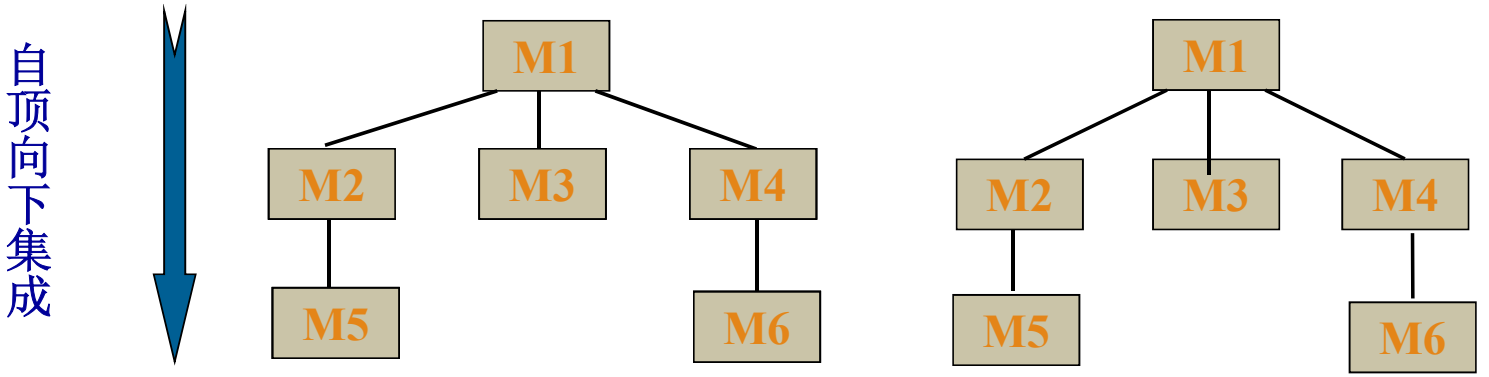
### (1) 非渐增式集成

非渐增式集成是一次集成过程，即先按照各模块在系统结构中的位置，设计驱动模块或桩模块进行辅助测试。然后将测试合格的各模块按照系统结构一次性完成系统集成的过程。

### (2) 渐增式集成

渐增式集成不同于非渐增式集成，它不是将模块一次性组装，而是按照系统结构逐渐将模块依次集成到系统中。从集成的过程来看，渐增式集成过程分为自底向上的集成和自顶向下的集成。

# 软件测试策略




第一步，测试主控模块M1，设计桩模块S1、S2、S3，模拟被M1调用的M2、M3、M4。

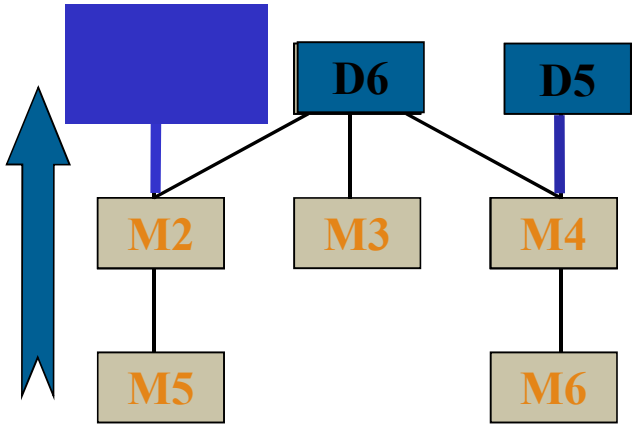
第二步，依次用M2、M3、M4替代桩模块S1、S2、S3，每替代一次进行一次测试。

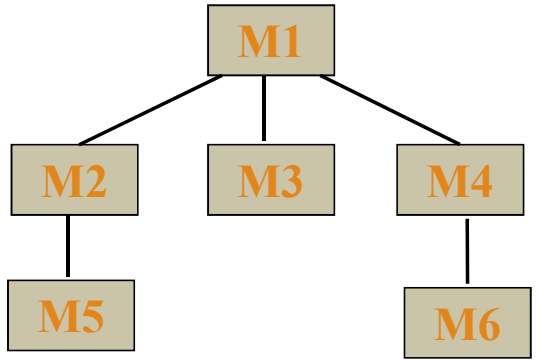
第三步，对由主控模块M1和模块M2、M3、M4构成的子系统进行测试，设计桩模块S4、S5。

第四步，依次用模块M5和M6替代桩模块S4、S5，并同时进行新的测试。组装测试完毕。

# 软件测试策略

自底向上集成




第一步，对最底层的模块M3、M5、M6进行测试,设计驱动模块D1、D2、D3来模拟调用。

第二步，用实际模块M2、M1和M4替换驱动模块D1、D2、D3。

第三步，设计驱动模块D4、D5和D6模拟调用，分别对新子系统进行测试。

第四步，把已测试的子系统按程序结构连接起来完成程序整体的组装测试。

# 软件测试策略

- 自顶而下增值

优点：能够尽早发现系统主控方面的问题。

缺点：无法验证桩模块是否完全模拟了下属模块的功能。

- 自底而上增值

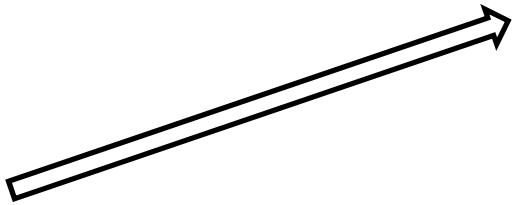
优点：驱动模块较易编写，且能够尽早查出底层涉及较复杂的算法和实际的I/O模块中的错误。

缺点：最后才能发现系统主控方面的问题。

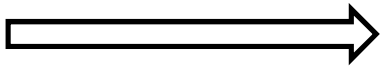
- 集成过程的原则

① 尽早测试关键模块。

② 尽早测试包含I/O的模块。



- 公共模块
- 关键路径上的模块
- 影响系统性能的模块



- 访问文件/数据库
- 与外部系统的数据交换
- 数据采集

# 软件测试策略

## 3. 确认测试

确认测试也称为验收测试，它是指在模拟用户实际操作的环境下（或开发环境下），运用黑盒测试法验证软件的有效性是否符合需求。

回想在需求获取时：
- 用户的操作场景；
- 用户的操作习惯；
- 用户的操作流程；
- ……

确认测试完成后，可能产生两种情形：

(1) 软件系统功能、性能、领域等要素满足用户需求规格说明，软件是有效的。

(2) 软件系统功能、性能、领域等要素存在不满足用户需求规格说明的某些方面，确认测试文档指出存在的错误或问题、对比预期结果给出测试分析报告。

# 软件测试策略

## 4. 系统测试

系统测试是指软件系统作为整个计算机系统的一部分，与计算机系统的硬件系统、数据、外部其它软件、文档等要素相结合，在用户实际运行环境中进行的确认测试。

系统测试的内容：

- 功能测试
- 性能测试
- 压力测试
- 容量测试
- 安全测试
- 文档测试
- 恢复性测试
- 备份测试

# 软件测试策略

## 系统测试——压力测试

压力测试也成为强度测试，是指系统在各种资源超负荷得情况下对运行状况进行测试。

- 压力测试的基本思路：计算机系统资源匮乏或有限的条件下运行的测试，是测试系统在使用峰值时的性能。
- 压力测试的主要对象包括：内存、外部存储、网络负载、中断处理、缓冲区、事务队列、在线用户量等。

# 软件测试策略

## 系统测试——压力测试：PV模型（前端/后端）

- PV（page view）是指服务器页面的访问次数。当客户端打开一个网页或刷新一次网页，就计算一次PV。

每台服务器每秒处理请求的数量 =

((总PV*80%) / (24小时*60分*60秒*40%)) / 服务器数量

含义：表示在一天中，80%的请求发生40%的时间内。24小时的40%是9.6小时，有80%的请求发生一天的9.6个小时当中（可以理解为：白天请求多，晚上请求少）。

提问：如何建设一个网站，使之能分别支持10万、100万、5000万的PV访问量？

# 软件测试策略

## 测试步骤及策略比较

| 测试类型 | 目的 | 测试依据 | 测试方法 |
| --- | --- | --- | --- |
| 单元测试 | 发现模块内部逻辑和功能上的错误或缺陷，以及接口存在的问题 | 详细设计规格说明 | 白盒测试 |
| 集成测试 | 发现模块间调用关系，以及模块间接口方面问题 | 概要设计规格说明 | 以黑盒测试为主，结合使用白盒测试 |
| 验收测试 | 测试系统软件整体功能和性能，进行一系列整体的、有效性测试 | 需求规格说明 | 黑盒测试 |
| 系统测试 | 结合网络、服务器等硬件，及支持软件等综合环境进行测试 | 需求规格说明 | 黑盒测试 |

# 调试

软件测试的目的是发现错误，测试完成之后的软件调试目的是为了定位和修改错误，以保证软件运行的正确性和可靠性。

软件调试活动主要分为以下三部分内容：

(1) 确定软件系统出现错误的准确位置，这需要结合调试人员的经验和技巧。

(2) 对发现错误的修改，如分析修改的程序是否涉及全局数据、是否涉及面向对象机制（如虚函数）、是否影响修改函数的主调函数结果的正确性等。

(3) 对修改后的内容重新进行测试，特别是涉及到全局数据结构、文件结构、系统结构等内容的修改，还必须进行确认测试和系统测试。

*最后，修改对应的文档内容，并通知与之对应的人员。*

# 软件测试报告

## 测试报告文档


在软件测试各阶段完成之前，必须编写软件测试报告，并按照评审标准对软件测试报告进行评审。编写测试报告的目的是发现并消除其中存在的遗漏、错误和不足，使得测试用例、测试预期结果等内容符合标注及规范的要求。通过了评审的软件测试报告成为基线配置项，纳入项目管理的过程。

软件测试报告主要包括软件测试说明和软件测试报告两个部分 。

# 软件测试常见问题

## 与软件测试有关的几个基本问题

- 问题1：由于单元测试要编写测试驱动程序，非常麻烦，能否等到整个系统全部开发完成后，再一次性的统一进行单元测试？
- 问题2：如果完成单元测试后，将所有单元集成在一起组装成系统有何不妥？集成测试是否多此一举？
- 问题3：在集成测试时，对于已经进行了功能测试、性能测试的子系统，在系统测试时是否能跳过相同内容的测试？
- 问题4：能否将系统测试和确认测试“合二为一”？