软件工程基础

第六章 软件测试与质量保证

苏 生
83201311(Tel)
susheng@uestc.edu.cn

信息与软件工程学院
电子科技大学

本章学习目标

1. 掌握基本的黑盒与白盒测试方法
2. 掌握软件测试策略
3. 掌握软件质量保证概念与过程

第六章 软件测试与质量管理

6.1 软件测试的概念

6.2 软件测试技术

6.3 软件测试策略

6.4 软件质量保证

6.1 软件测试的概念

- 软件测试
  - 描述一种用来促进鉴定软件的正确性、完整性、安全性等软件质量的过程。
  - 在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。
  - 使用人工操作或者软件自动运行的方式来检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别的过程。
  - 以最少的人力、资源投入，在最短的时间内完成测试，发现软件系统的缺陷，保证软件的优良品质，把人为因素的影响减少到最小，则是软件公司探索和追求的目标。

软件测试的概念

- 测试用例
  - 为某个特殊目标而编制的一组测试输入、执行条件以及预期结果，以便测试某个程序路径或核实是否满足某个特定需求。
  - 包括测试目标、测试环境、输入数据、测试步骤、预期结果、测试脚本等，并形成文档

功能性测试用例格式

功能A描述		
用例目的		
前提条件		
测试步骤		
输入/动作	期望的输出/相应	实际情况
示例：典型值…		
示例：边界值…		
示例：异常值…		

第六章 软件测试与质量管理

6.1 软件测试的概念

6.2 软件测试技术

6.3 软件测试策略

6.4 软件质量保证

软件测试技术

- 黑盒测试
  - 将软件当作内部不可见盒子
  - 行为测试，侧重于软件的功能需求
  - 发现功能、接口、数据访问、行为等错误
- 白盒测试
  - 将软件当作内部可见、透明的盒子
  - 查看代码功能和实现逻辑路径

黑盒测试

等价类划分

一、概念

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。

二、思想

把所有可能的输入数据，即程序的输入域划分成若干部分，然后从每一部分中选取少数有代表性的数据做为测试用例

黑盒测试

原因：由于实现穷举测试的不可能性，只有从大量的可能数据中选取一部分作为测试用例。

效果：经过类别划分后，每一类的代表性数据在测试中的作用都等价于这一类中的其他值。

手段：在设计测试用例时，在需求说明的基础上划分等价类，列出等价表，从而确定测试用例。

等价类的类型

- 有效等价类
  - 对规格说明而言，有意义、合理的输入数据所组成的集合；
  - 检验程序是否实现了规格说明预先规定的功能和性能。
- 无效等价类
  - 对规格说明而言，无意义的、不合理的输入数据所组成的集合；
  - 检查被测对象的功能和性能的实现是否有不符合规格说明要求的地方。

如何划分等价类－1

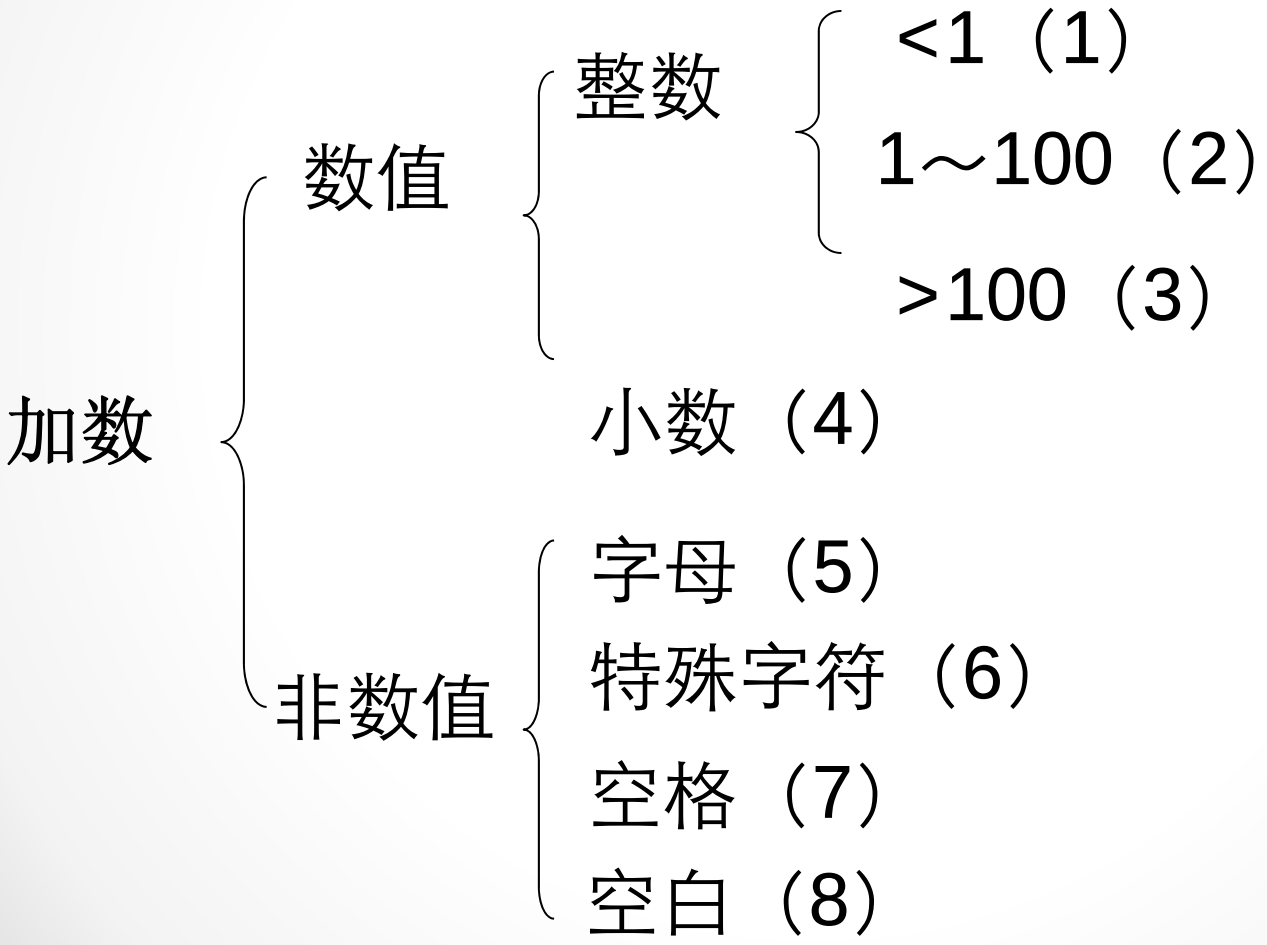
- 如何划分？——先从程序的规格说明书中找出各个输入条件，再为每个输入条件划分两个或多个等价类，形成若干的互不相交的子集。

- 加法器程序（程序功能计算两个**1～100**之间整数的和）的等价类，给出测试用例.

如何划分等价类—2

- 刚才给出的 测试用例 都是整数，如果输入的是小数、字符怎么办？
- 只考虑了输入数据的范围，没有考虑输入数据的类型。

考虑输入数据类型和范围



等价类的划分原则

（1）按照区间划分——在输入条件规定了取值范围或值的个数的情况下，可以确定一个有效等价类和两个无效等价类。

例：**程序输入条件为小于100大于10的整数x，则有效等价类为10＜x＜100，两个无效等价类为x≤10和x≥100。**

（2）按照数值划分——在规定了一组输入数据（假设包括 n个 输入值），并且程序要对每一个输入值分别进行处理的情况下，可确定 n 个有效等价类（每个值确定一个有效等价类）和一个无效等价类（所有不允许的输入值的集合）。

例：**程序输入x取值于一个固定的枚举类型{1,3,7,15}，且程序 中对这4个数值分别进行了处理，则有效等价类为x=1、x=3、 x=7、x=15，无效等价类为x≠1,3,7,15的值的集合。**

（3）按照数值集合划分——在输入条件规定了输入值的集合（集合中的输入值具有同等效果）或规定了“必须如何”的条件下，可以确定一个有效等价类和一个无效等价类（该集合有效值之外）。

例：程序输入用户口令的长度必须是4位的串，可以确定一个有效等价类是串的长度为4，一个无效等价类长度不为4。

（4）按照限制条件或规则划分——在规定了输入数据必须遵守的规则或限制条件的情况下，可确定一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

例：程序输入条件为取值为奇数的整数**x**，则有效等价类为**x**的值为奇数的整数，无效等价类为**x**的值不为奇数的整数。

等价类划分法设计测试用例的步骤

（1）确定等价类

（2）建立等价类表，列出所有划分出的等价类

（3）从划分出的等价类中按以下的3个原则设计测试用例：

A 为每一个等价类规定一个唯一的编号

B 设计一个新的测试用例，使其尽可能多的覆盖尚未被覆盖的有效等价类，重复这一步，直到所有的有效等价类都被覆盖为止。

C 设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类，重复这一步，直到所有的无效等价类都被覆盖为止。

案例1：

1、某城市电话号码由三部分组成，分别是：

地区码—— 空白或三位数字；

前　缀—— 非‘0’或非‘1’开头的三位数字；

后　缀—— 4位数字。

假定被测程序能接受一切符合上述规定的电话号码， 拒绝所有不符合规定的电话号码。

（1）划分等价类、列出等价类表

输入条件	有效等价类	编号	无效等价类	编号
地区码	空白	1	有非数字字符	5
			少于3位数字	6
	3位数字	2	多于3位数字	7
前缀	200~999	3	有非数字字符	8
			起始为’0’	9
			起始为’1’	10
			少于3位数字	11
			多于3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
			多于4位数字	15

测试用例编号	输入数据			预期输出	覆盖等价类
	地区码	前缀	后缀		
1	空白	321	4567	接受（有效）	1, 3, 4
2	123	805	9876	接受（有效）	2, 3, 4
3	20A	321	4567	拒绝（无效）	5
4	33	234	5678	拒绝（无效）	6
5	1234	234	4567	拒绝（无效）	7
6	123	2B3	1234	拒绝（无效）	8
7	123	013	1234	拒绝（无效）	9
8	123	123	1234	拒绝（无效）	10
9	123	23	1234	拒绝（无效）	11
10	123	2345	1234	拒绝（无效）	12
11	123	234	1B34	拒绝（无效）	13
12	123	234	34	拒绝（无效）	14
13	123	234	23345	拒绝（无效）	15

黑盒测试

边界条件测试

从长期的测试工作经验得知，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

缺陷分析示例

```
int data[10];
for(int i=0;i<=10;i++)
    data[i] = 1;
```

将不等式的“<”错写成“<=”

缺陷分析示例

```
int m, d;
function multiply(int a, int c)
    m = a * c;
```

可能超出整数变量m（2的16或32次方）取值范围的界

边界条件类型情况

数值
字符
位置
数量

速度
地点
尺寸

边界条件特征情况

第一个 / 最后一个

开始 / 完成

空 / 满

最慢 / 最快

最大 / 最小

相邻 / 最远

最小值 / 最大值

超过 / 在内

最短 / 最长

最早 / 最迟

最高 / 最低

边界条件测试指导原则

1. 若输入条件指定为以a和b为边界的范围，则测试用例应该包括a和b，略大于和略小于a和b

2. 若输入条件指定为一组值，则测试用例应当执行其中的最大值和最小值，以及略大于和略小于最大值和最小值的值

3. 上述原则也适用于输出条件

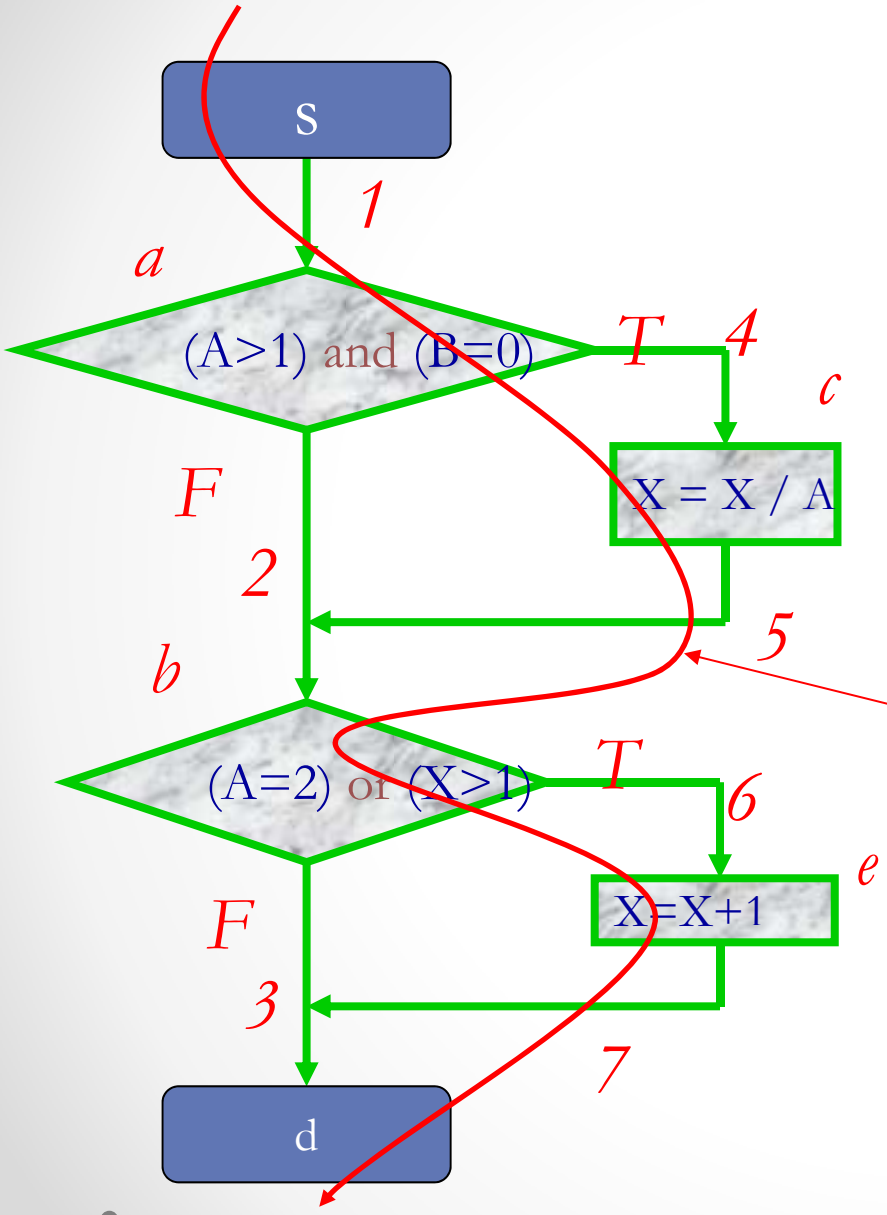
4. 上述原则也适用于程序内部预定义的边界值

白盒测试

- 判定结构测试
  - 语句覆盖
  - 判定覆盖
  - 条件覆盖
  - 判定-条件覆盖

语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得每一可执行语句至少执行一次。这种覆盖又称为点覆盖，它使得程序中每个可执行语句都得到执行，但它是最弱的逻辑覆盖准，效果有限，必须与其它方法交互使用。




```
PROCEDURE Example(A,B:real; X:real );
Begin
    IF (A>1) AND (B=0) THEN
        X:= X / A;
    IF ( A=2 ) OR (X>1) THEN
        X:=X+1
END;
```

I. A=2, B= 0, X=4 ---- sacbed

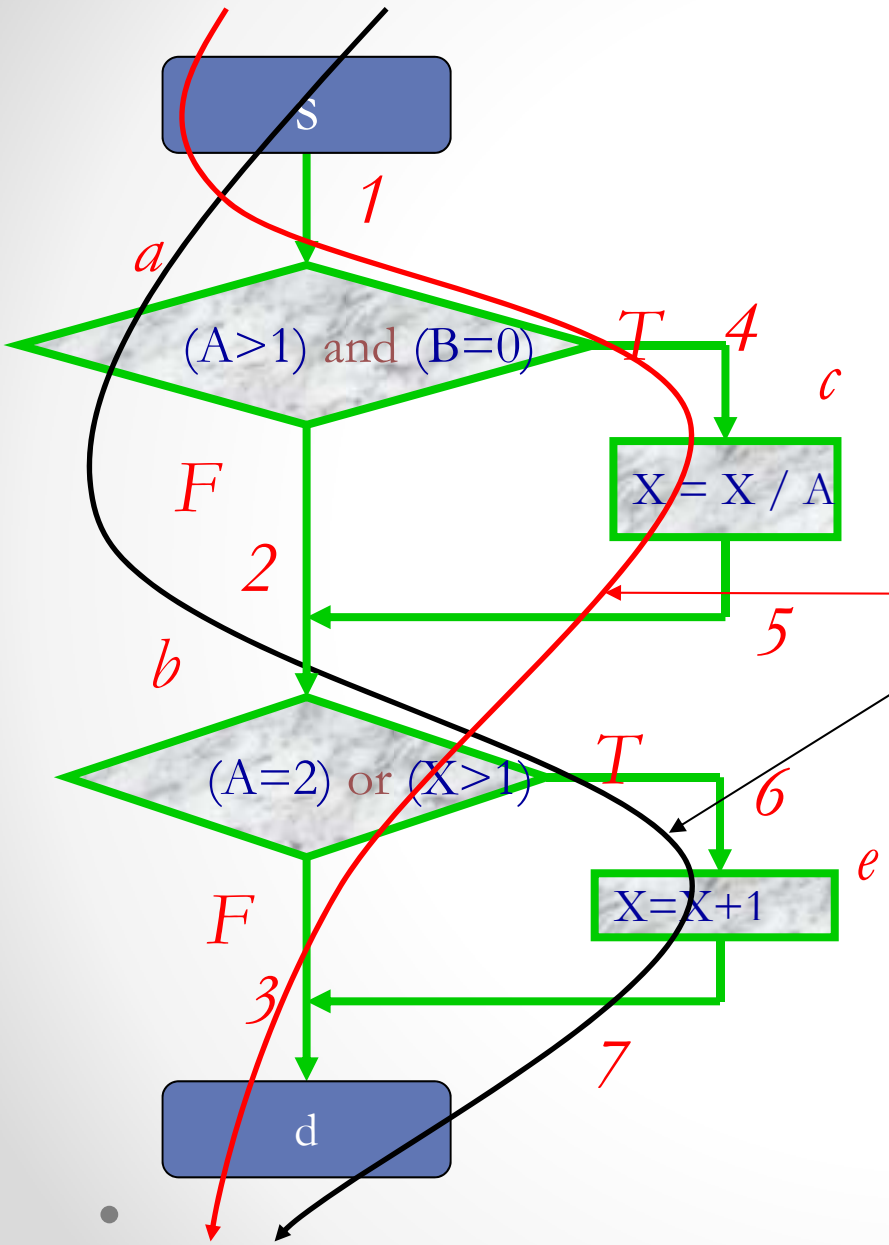
语句覆盖
所有的语句至少执行一次!
是最弱的逻辑覆盖

语句覆盖优缺点

- 优点：可以很直观地从源代码得到测试用例，无须细分每条判定表达式。
- 缺点：由于这种测试方法仅仅针对程序逻辑中显式存在的语句，但对于隐藏的条件和可能到达的隐式逻辑分支，是无法测试的。
- 例如：**IF (A>1) AND (B=0) THEN, “null else”,** 用例不需要考虑迫使 **A<1**的出错情况。

判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次，即程序中的每个分支至少执行一次 。判定覆盖又称为分支覆盖**(Branch coverage)**。
- 判定覆盖只比语句覆盖稍强一些，实际效果表明，只是判定覆盖，还不能保证一定能查出在判断的条件中存在的错误。因此，还需要更强的逻辑覆盖准则去检验判断内部条件。



- （A>1）and (B=0) 为一个判定
- A>1 为一个条件

- 每个语句至少执行一次！
- 每个判定的每种可能都至少执行一次！
- 即每个判定的真假分支都至少执行一次！

I: A=3, B=0,X=1: sacbd

II: A=2, B=1,X=1: sabed

满足判定覆盖

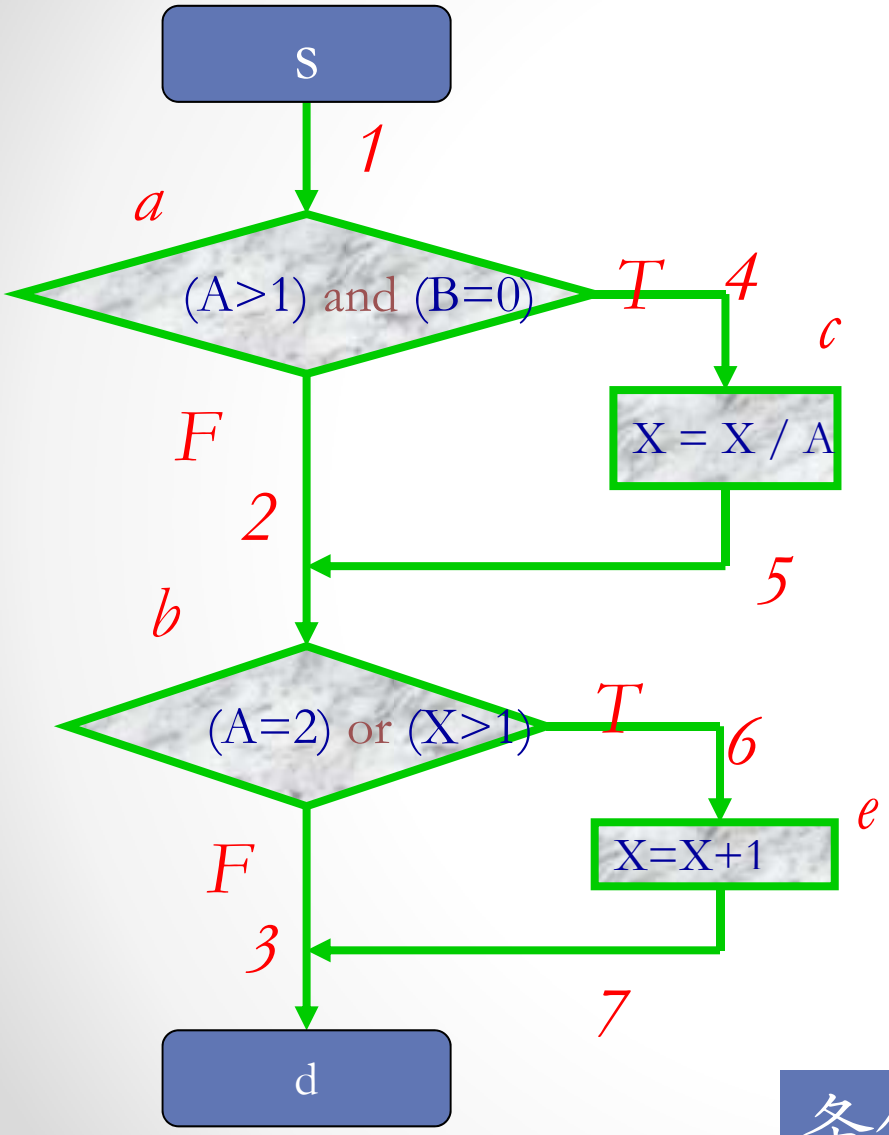
满足判定覆盖的测试用例一定满足语句覆盖：判定覆盖比语句覆盖强。但仍是弱的逻辑覆盖。

判定覆盖的优缺点

- 优点：判定覆盖比语句覆盖要多几乎一倍的测试路径，当然也就具有比语句覆盖更强的测试能力。同样判定覆盖也具有和语句覆盖一样的简单性，无须细分每个判定就可以得到测试用例。
- 缺点：往往大部分的判定语句是由多个逻辑条件组合而成（如，判定语句中包含**AND**、**OR**、**CASE**），若仅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。

条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。
- 条件覆盖深入到判定中的每个条件，但可能不能满足判定覆盖的要求。



每个语句至少执行一次，而且判定表达式中的每个条件都要取得各种可能的结果。

第一判定表达式:

设条件 A>1 取真 记为T1

假 F1

条件 B=0 取真 记为T2

假 F2

第二判定表达式:

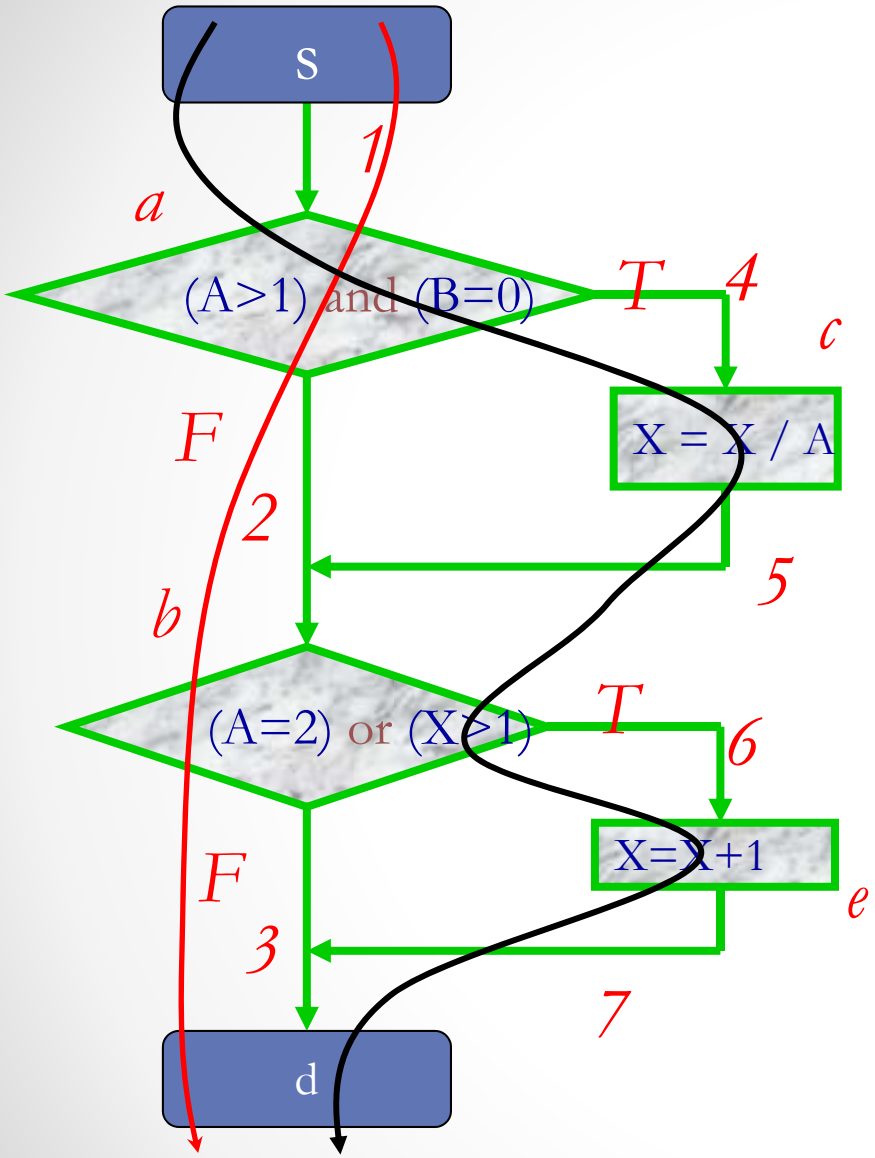
设条件 A=2 取真 记为T3

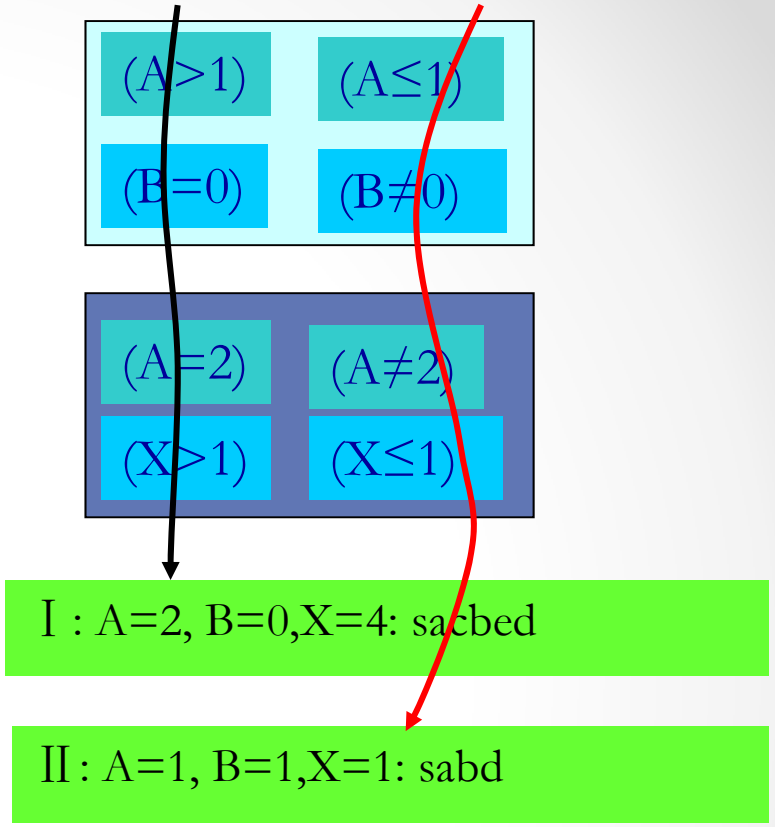
假 F3

条件 X>1 取真 记为T4

假 F4

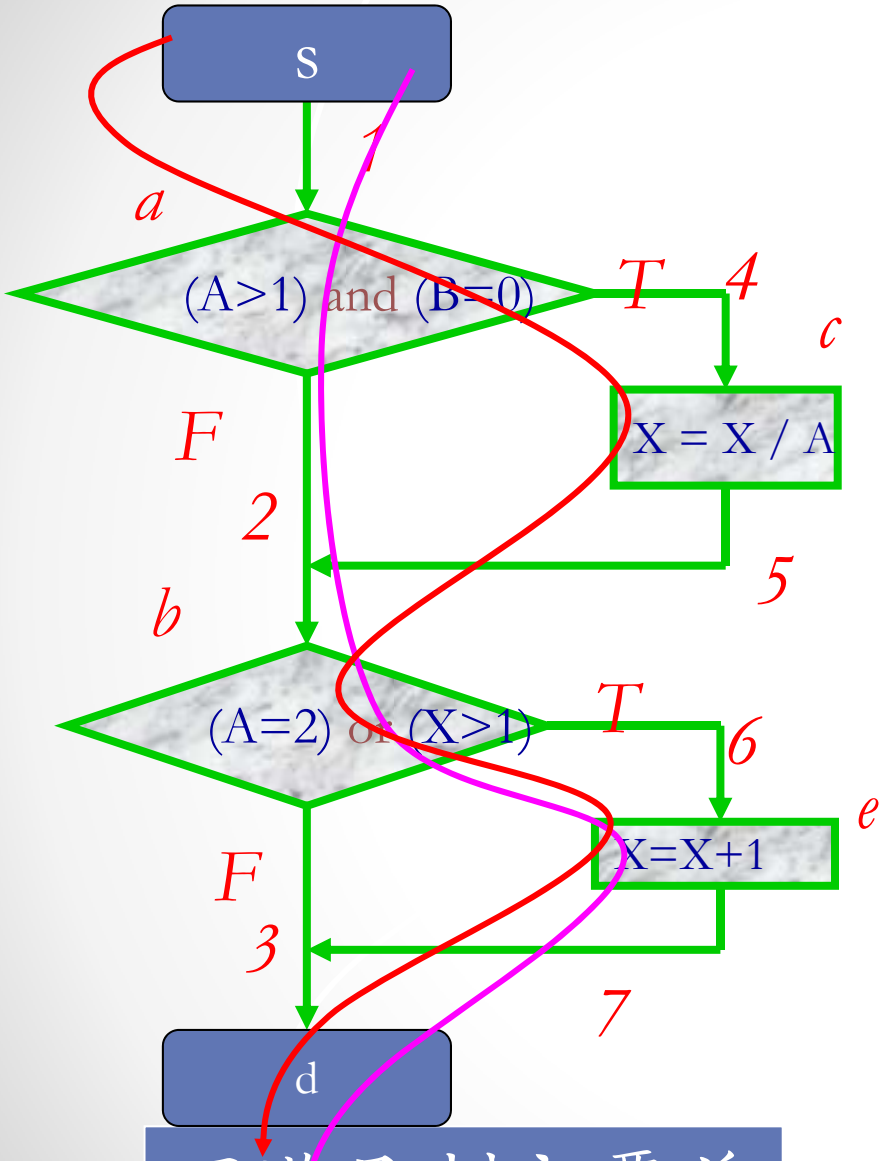
条件覆盖要求这8种值都要取到



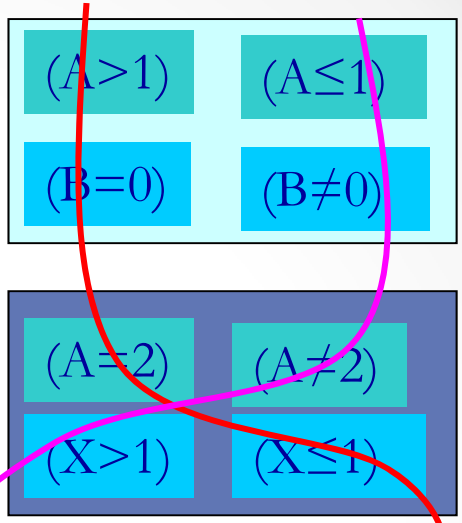


同时满足判定覆盖

测试用例			通过路径	满足的条件
A	B	X		
2	0	4	sacbed	T1,T2,T3,T4
1	1	1	sabd	F1,F2,F3,F4



不满足判定覆盖



Ⅲ: A=2, B=0,X=1: sacbed

Ⅳ: A=1, B=1,X=2: sabed

测试用例			通过路径	满足的条件
A	B	X		
2	0	1	sacbed	T1,T2,T3,F4
1	1	2	sabed	F1,F2,F3,T4

条件覆盖的优缺点

- 优点：

  显然条件覆盖比判定覆盖增加了对符合判定情况的测试，增加了测试路径。

- 缺点：

  要达到条件覆盖，需要足够多的测试用例，但条件覆盖并不能保证判定覆盖。条件覆盖只能保证每个条件至少有一次为真，而不考虑所有的判定结果。

判定覆盖和条件覆盖(DC)

条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却关心整个判定表达式的值。但也可能有相反的情况：虽然每个条件都取到了不同值，但判定表达式却始终只取一个值。

条件覆盖不一定包含判定覆盖
判定覆盖也不一定包含条件覆盖

判定-条件覆盖

- 既然判定条件不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖，就自然会提出一种能同时满足两种覆盖标准的逻辑覆盖，这就是判定--条件覆盖。
- 判定－条件覆盖就是设计足够的测试用例，使得判断中每个条件的所有可能取值至少执行一次，同时每个判断本身的所有可能判断结果至少执行一次。换言之，即是要求各个判断的所有可能的条件取值组合至少执行一次。

判定-条件覆盖的优缺点

- 优点：判定/条件覆盖满足判定覆盖准则和条件覆盖准则，弥补了二者的不足。

- 缺点：判定/条件覆盖准则的缺点是未考虑条件的组合情况。

白盒测试

- 循环测试
  - 简单循环
  - 嵌套循环
  - 串接循环

以简单循环的测试方式为基础，其他循环是简单循环的简单叠加。

① 零次循环：从循环入口到出口

② 一次循环：检查循环初始值

③ 二次循环：检查二次循环

④ m次循环：检查m次循环

⑤ 最大次数循环、比最大次数多一 次、少一次的循环。

例

求数组最小值

```
k = i;
for ( j = i+1;  j <= n;  j++ ) do
        if  ( A[j] < A[k] ) then
                k = j;
        end if
end for
```

循环	i	n	A[i]	A[i+1]	A[i+2]	k	路 径
0	1	1				i	
1	1	2	1	2		i	
			2	1		i+1	
2	1	3	1	2	3	i	
			2	3	1	i+2	
			3	2	1	i+2	
			3	1	2	i+1	

白盒测试

- 嵌套循环
  - 从最内层循环开始，将其他循环设置为最小值
  - 对最内层循环执行简单循环测试，外层循环迭代次数最小
  - 由内向外构造下一个循环的测试，使其他外层循环为最小值
  - 继续上述过程，知道所有循环测试完成

第六章 软件测试与质量管理

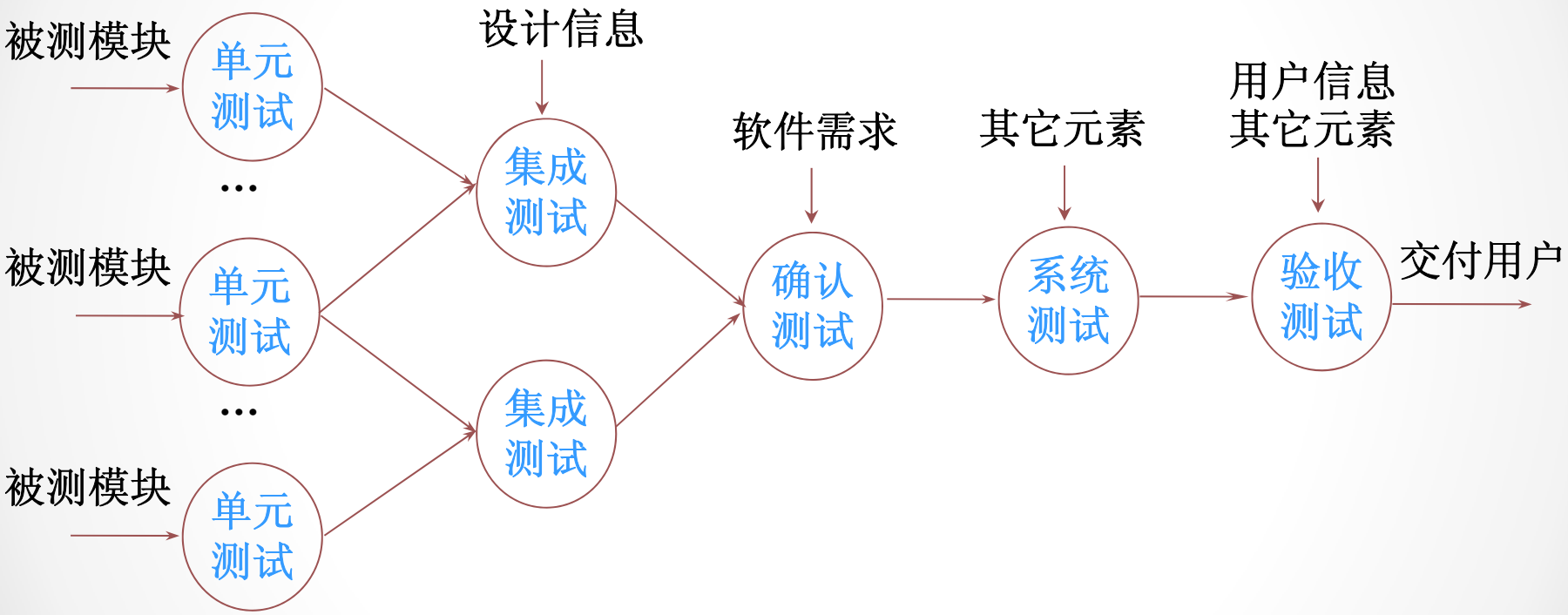
6.1 软件测试的概念

6.2 软件测试技术

6.3 软件测试策略

6.4 软件质量保证

软件测试策略

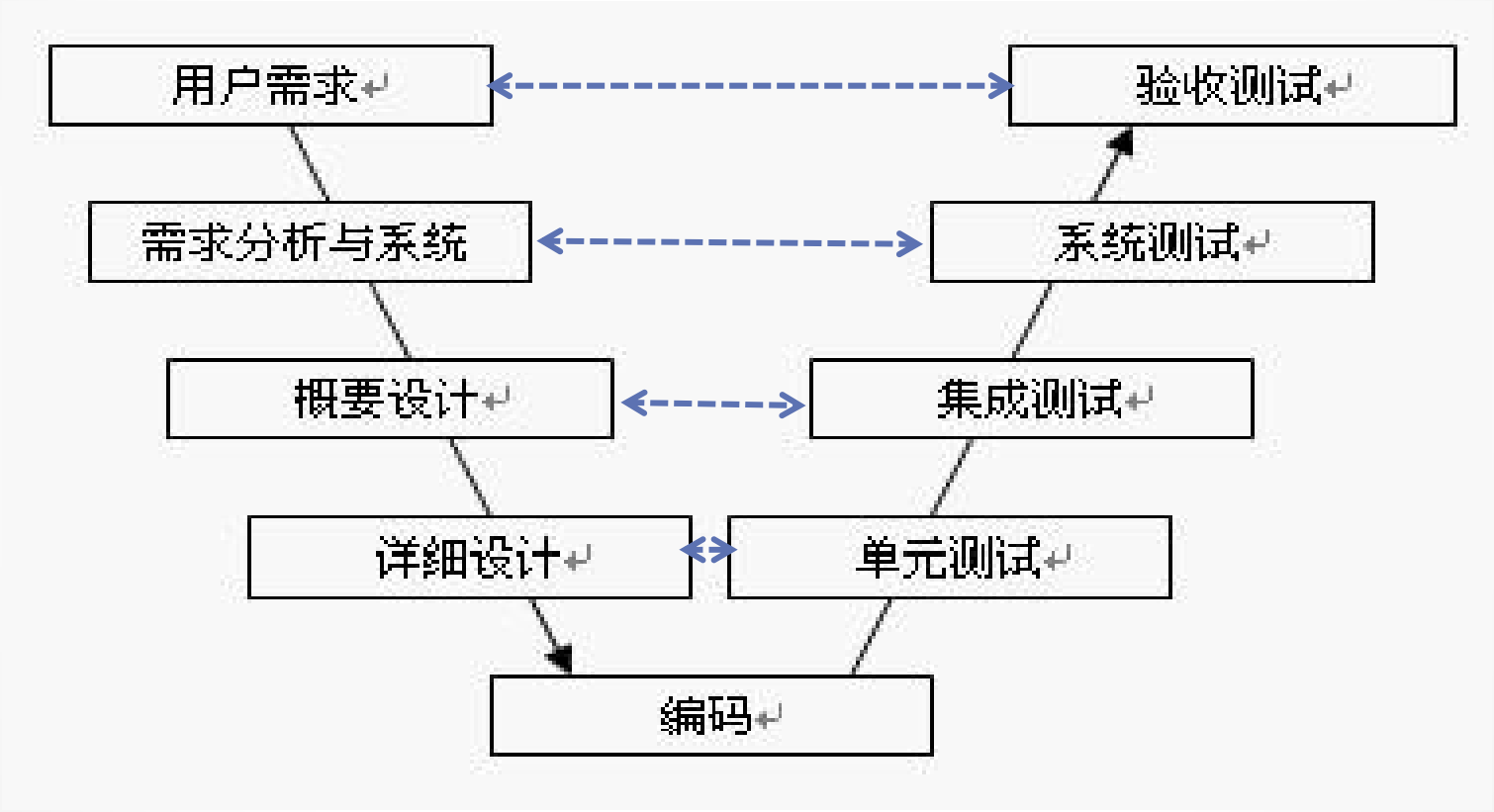


软件测试的过程

软件测试策略

- 单元测试：针对每个单元的测试， 以确保每个模块能正常工作为目标。
- 集成测试：对已测试过的模块进行组装，进行集成测试。目的在于检验与软件设计相关的程序结构问题。
- 确认（有效性）测试：是检验所开发的软件能否满足所有功能和性能需求的最后手段。
- 系统测试：检验软件产品能否与系统的其他部分（比如，硬件、数据库及操作人员）协调工作。
- 验收（用户）测试：检验软件产品质量的最后一道工序。主要突出用户的作用，同时软件开发人员也应有一定程度的参与。

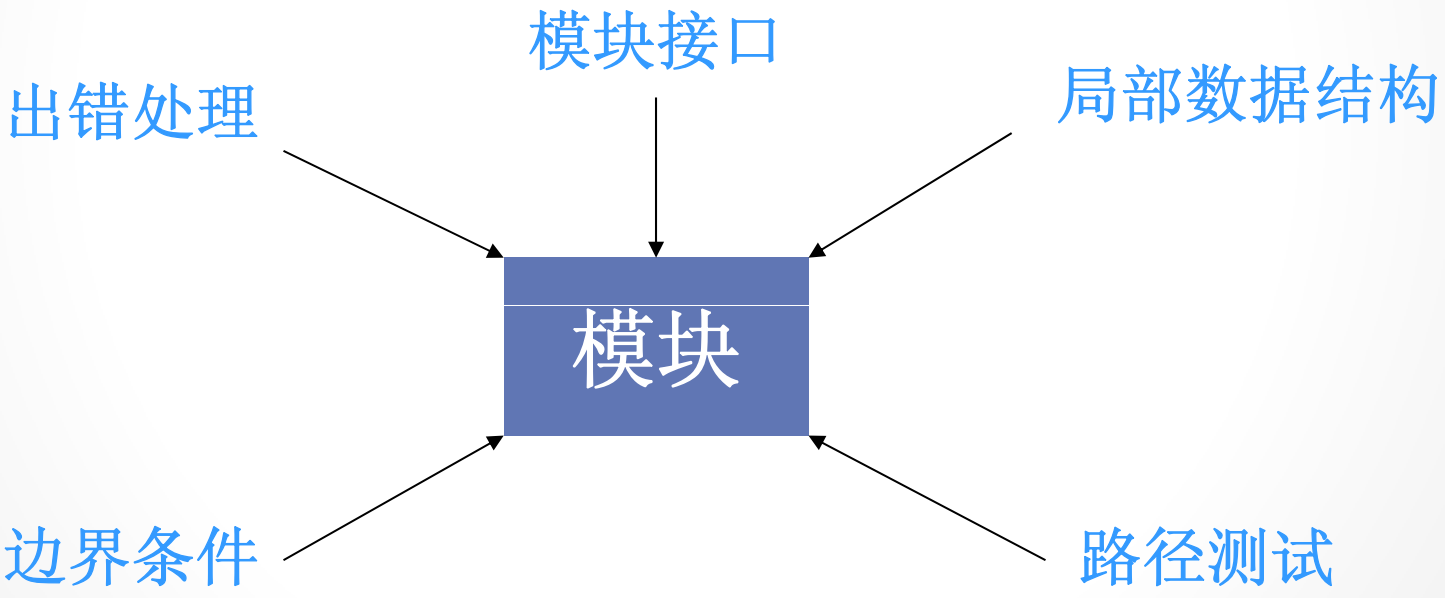
软件测试的过程模型



软件测试V模型

单元测试

- 单元测试针对每个程序的模块，主要测试5个方面的问题：——模块接口、局部数据结构、边界条件、独立的路径和错误处理。



单元测试—模块接口

- 这是对模块接口进行的测试，检查进出程序单元的数据流是否正确。模块接口测试必须在任何其它测试之前进行。
- 模块接口测试至少需要如下的测试项目：

（1）调用所测模块时的输入参数与模块的形式参数在个数、属性、顺序上是否匹配；

（2）所测模块调用子模块时，它输入给子模块的参数与子模块中的形式参数在个数、属性、顺序上是否匹配；

（3）是否修改了只做输入用的形式参数；

（4）调用标准函数的参数在个数、属性、顺序上是否正确；

（5）全局变量的定义在各模块中是否一致。

单元测试—局部数据结构

- 在模块工作过程中，必须测试模块内部的数据能否保持完整性，包括内部数据的内容、形式及相互关系不发生错误。
- 对于局部数据结构，应该在单元测试中注意发现以下几类错误：

（1）不正确的或不一致的类型说明。

（2）错误的初始化或默认值。

（3）错误的变量名，如拼写错误或书写错误。

（4）下溢、上溢或者地址错误。

单元测试—路径测试

- 在单元测试中，最主要的测试是针对路径的测试。测试用例必须能够发现由于计算错误、不正确的判定或不正常的控制流而产生的错误。
- 常见的错误有：

  误解的或不正确的算术优先级；混合模式的运算；错误的初始化；精确度不够精确；表达式的不正确符号表示。
- 针对判定和条件覆盖，测试用例还要能够发现如下错误：

  不同数据类型的比较；不正确的逻辑操作或优先级；应当相等的地方由于精确度的错误而不能相等；不正确的判定或不正确的变量；不正确的或不存在的循环终止；当遇到分支循环时不能退出；不适当地修改循环变量。

单元测试—边界条件

- 边界测试是单元测试的最后一步，必须采用边界值分析方法来设计测试用例，认真仔细地测试为限制数据处理而设置的边界处，看模块是否能够正常工作。
- 一些可能与边界有关的数据类型如数值、字符、位置、数量、尺寸等，还要注意这些边界的首个、最后一个、最大值、最小值、最长、最短、最高、最低等特征。
- 在边界条件测试中，应设计测试用例检查以下情况：

（1）在n次循环的第0次、1次、n次是否有错误。

（2）运算或判断中取最大值、最小值时是否有错误。

（3）数据流、控制流中刚好等于、大于、小于确定的比较值是否出现错误。

单元测试—出错处理

- 测试出错处理的重点是模块在工作中发生了错误，其中的出错处理设施是否有效。
- 检验程序中的出错处理可能面对的情况有：

（1）对运行发生的错误描述难以理解。

（2）所报告的错误与实际遇到的错误不一致。

（3）出错后，在错误处理之前就引起系统的干预。

（4）例外条件的处理不正确。

（5）提供的错误信息不足，以至于无法找到错误的原因。

单元测试的执行过程

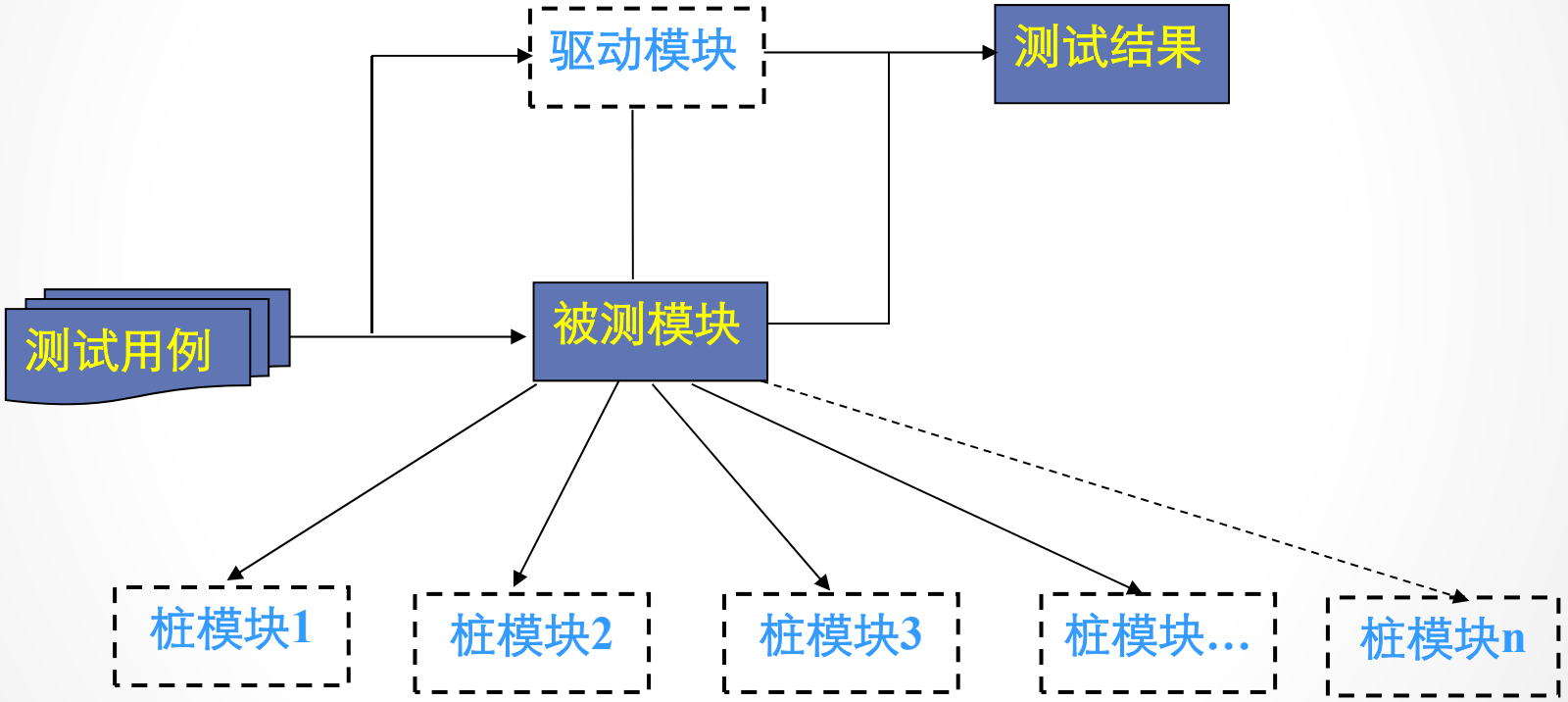
- 何时进行单元测试？单元测试常常是和代码编写工作同时进行的，在完成了程序编写、复查和语法正确性验证后，就应进行单元测试用例设计。
- 在单元测试时，如果模块不是独立的程序，需要设置一些辅助测试模块。辅助测试模块有两种：

（1）驱动模块**(Drive)** 用来模拟被测试模块的上一级模块，相当于被测模块的主程序。它接收数据，将相关数据传送给被测模块，启动被测模块，并打印出相应的结果。

（2）桩模块**(Stub)** 用来模拟被测模块工作过程中所调用的模块。它们一般只进行很少的数据处理。

- 驱动模块和桩模块都是额外的开销，虽然在单元测试中必须编写，但并不需要作为最终的产品提供给用户。

单元测试的执行过程



集成测试

- 集成测试就是将软件集成起来后进行测试。又称为子系统测试、组装测试、部件测试等。
- 集成测试主要可以检查诸如两个模块单独运行正常，但集成起来运行可能出现问题的情况。
- 集成测试是一种范围很广的测试，当向下细化时，就成为单元测试。

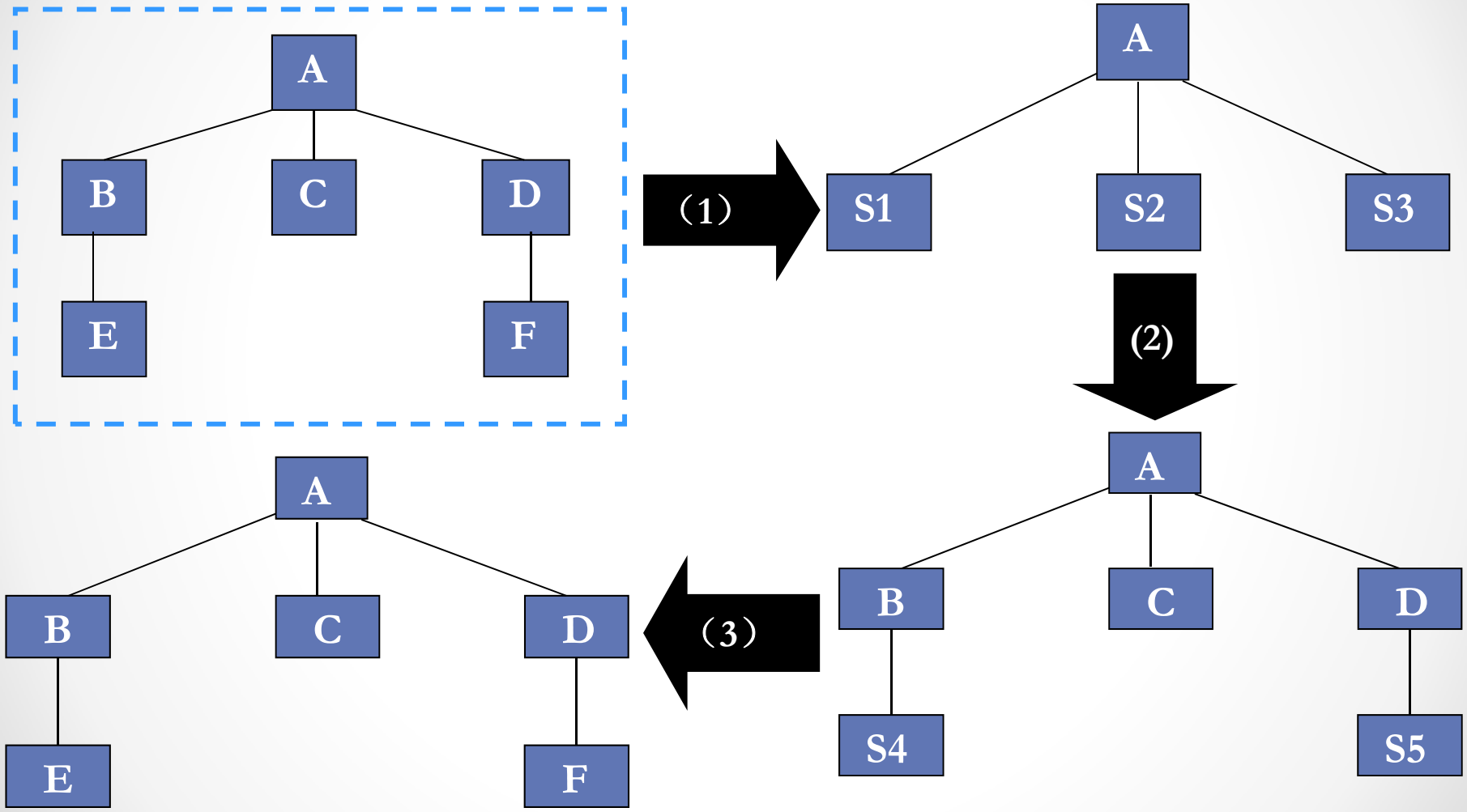
集成测试的主要方法

- 自顶向下的集成方法
- 自底向上的集成方法
- 冒烟方法

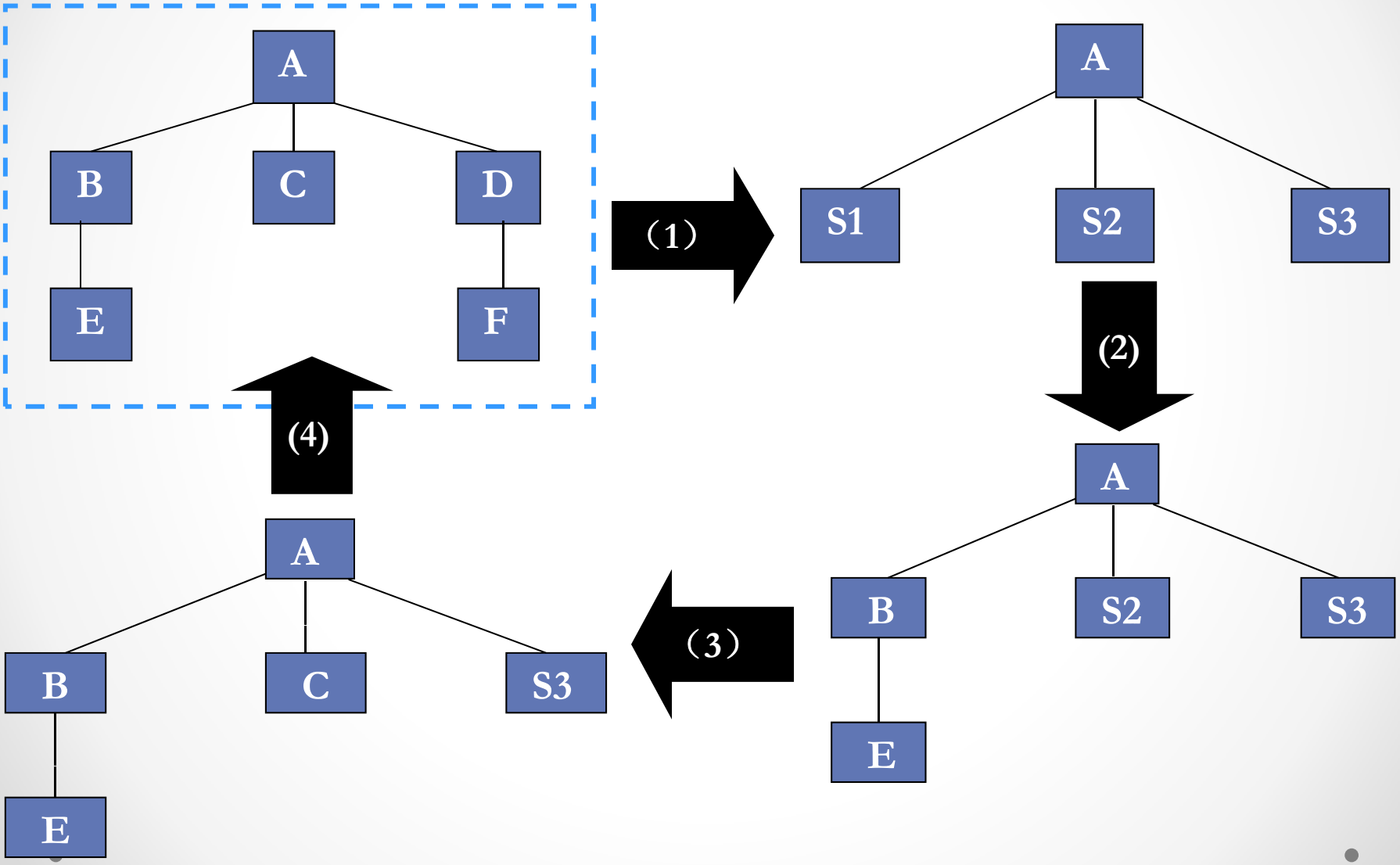
自顶向下的集成方法

- 这种组装方式将模块按系统程序结构，沿控制层次自顶向下进行集成。从属于主控模块的按深度优先方式（纵向）或者广度优先方式（横向）集成到结构中去。
- 自顶向下的集成方式在测试过程中较早地验证了主要的控制和判断点。
- 选用按深度方向集成的方式，可以首先实现和验证一个完整的软件功能。
- 缺点是桩的开发量较大

广度优先方式



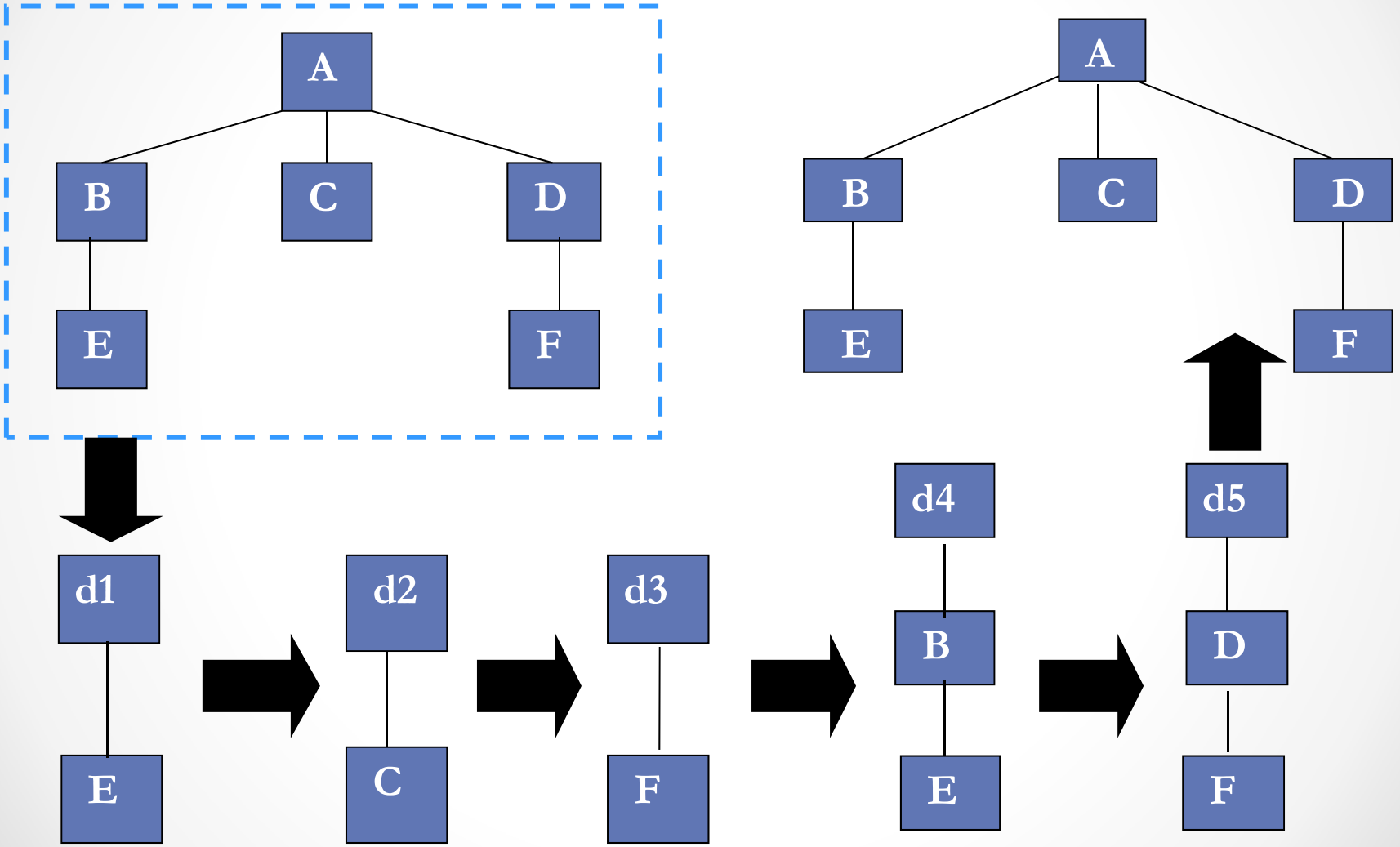
深度优先方式



自底向上的集成方法

- 自底向上集成方法是从软件结构最底层的模块开始，按照接口依赖关系逐层向上集成以进行测试。
- 由于是从最底层开始集成，对于一个给定层次的模块，它的子模块（包括子模块的所有下属模块）已经集成并测试完成，所以不再需要使用桩模块进行辅助测试。在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到。
- 自底向上的集成方法的优点是每个模块调用其他底层模块都已经测试，不需要桩模块；
- 缺点是每个模块都必须编写驱动模块；缺陷的隔离和定位不如自顶向下。

自底向上的集成方法



- 值得注意的是，在实际工作中，常常是综合使用自底向上和自顶向下的集成方法。
- 例如，按进度选择优先测试已经完成的模块
  - 如果已完成的模块所调用的模块没有完成，就采用自顶向下的方法，打桩进行测试
  - 如果已经完成模块的上层模块没有完成，可以采用自底向上集成方式。

冒烟方法

- 将已经转换为代码的软件构件集成为构造（**build**）。一个构造包括所有的数据文件、库、可复用的模块以及实现一个或多个产品功能所需的工程化构件。
- 设计一系列测试以暴露影响构造正确地完成其功能的错误。其目的是为了发现极有可能造成项目延迟的业务阻塞（**show stopper**）错误。
- 每天将该构造与其他构造，以及整个软件产品集成起来进行冒烟测试。这种集成方法可以是自顶向下，也可以自底向上。

回归测试

- 什么是回归测试？

  ——在集成测试策略的环境中，回归测试是对某些已经进行过的测试的某些子集再重新进行一遍，以保证上述改变不会传播无法预料的副作用或引发新的问题。

  ——在更广的环境里，回归测试就是用来保证（由于测试或其他原因的）改动不会带来不可预料的行为或另外的错误。

- 回归测试可以通过重新执行所有的测试用例的一个子集人工地进行，也可以使用自动化的捕获回放工具来进行。

回归测试

- 回归测试集包括三种不同类型的测试用例：

（1）能够测试软件的所有功能的代表性测试用例

（2）专门针对可能会被修改而影响软件功能的附加测试

（3）针对修改过的软件成分的测试

系统测试

- 系统测试是从用户使用的角度来进行的测试，主要工作是将完成了集成测试的系统放在真实的运行环境下进行测试，用于功能确认和验证
- 系统测试基本上使用黑盒测试方法
- 系统测试的依据主要是软件需求规格说明

系统测试的主要内容

- 功能性测试
- 可靠性测试
- 压力测试
- 性能测试
- 恢复测试
- 安全测试
- 其他的系统测试还包括配置测试、兼容性测试、本地化测试、文档测试、易用性测试等

功能测试

功能测试是在规定的一段时间内运行软件系统的所有功能，以验证这个软件系统有无严重错误。

可靠性测试

如果系统需求说明书中有对可靠性的要求，则需进行可靠性测试。

① 平均失效间隔时间 MTBF **(Mean Time Between Failures)** 是否超过规定时限?

② 因故障而停机的时间 MTTR **(Mean Time To Repairs)** 在一年中应不超过多少时间。

压力测试

压力测试是要检查在系统运行环境异常数量、频率或资源测试系统可以运行到何种程度的测试。

例如：

- 把输入数据速率提高一个数量级，确定输入功能将如何响应。
- 设计需要占用最大存储量或其它资源的测试用例进行测试。

- 设计出在虚拟存储管理机制中引起“颠簸”的测试用例进行测试。
- 设计出会对磁盘常驻内存的数据过度访问的测试用例进行测试。

- 压力测试的一个变种是敏感性测试。
- 在程序有效数据界限内一个小范围内的一组数据可能引起极端的或不平稳的错误处理出现，或者导致极度的性能下降的情况发生。此测试用以发现可能引起这种不稳定性或不正常处理的某些数据组合。

性能测试

性能测试是要检查系统是否满足在需求说明书中规定的性能。特别是对于实时系统或嵌入式系统。

性能测试常常需要与压力测试结合起来进行，并常常要求同时进行硬件和软件检测。

通常，对软件性能的检测表现在以下几个方面：响应时间、吞吐量、辅助存储区，例如缓冲区，工作区的大小等、处理精度，等等。

恢复测试

恢复测试是要证实在克服硬件故障(包括掉电、硬件或网络出错等)后，系统能否正常地继续进行工作，并不对系统造成任何损害。

- 为此，可采用各种人工干预的手段，模拟硬件故障，故意造成软件出错。并由此检查：
  - 错误探测功能——系统能否发现硬件失效与故障；

- 能否切换或启动备用的硬件；
- 在故障发生时能否保护正在运行的作业和系统状态；
- 在系统恢复后能否从最后记录下来的无错误状态开始继续执行作业；
- 掉电测试：其目的是测试软件系统在发生电源中断时能否保护当时的状态且不毁坏数据，然后在电源恢复时从保留的断点处重新进行操作。

启动/停止测试

这类测试的目的是验证在机器启动及关机阶段，软件系统正确处理的能力。

这类测试包括：

- 反复启动软件系统（例如，操作系统自举、网络的启动、应用程序的调用等）
- 在尽可能多的情况下关机。

配置测试

- 这类测试是要检查计算机系统内各个设备或各种资源之间的相互联结和功能分配中的错误。
- 它主要包括以下几种：
  - 配置命令测试：验证全部配置命令的可操作性（有效性）；特别对最大配置和最小配置要进行测试。软件配置和硬件配置都要测试。

- 循环配置测试：证明对每个设备物理与逻辑的，逻辑与功能的每次循环置换配置都能正常工作
- 修复测试：检查每种配置状态及哪个设备是坏的。并用自动的或手工的方式进行配置状态间的转换。

安全性测试

安全性测试是要检验在系统中已经存在的系统安全性、保密性措施是否发挥作用，有无漏洞。

- 力图破坏系统的保护机构以进入系统的主要方法有以下几种：
  - 正面攻击或从侧面、背面攻击系统中易受损坏的那些部分；
  - 以系统输入为突破口，利用输入的容错性进行正面攻击；

- 申请和占用过多的资源压垮系统，以破坏安全措施，从而进入系统；
- 故意使系统出错，利用系统恢复的过程，窃取用户口令及其它有用的信息；
- 通过浏览残留在计算机各种资源中的垃圾（无用信息），以获取如口令，安全码，译码关键字等信息；
- 浏览全局数据，期望从中找到进入系统的关键字
- 浏览那些逻辑上不存在，但物理上还存在的各种记录和资料等。

验收测试

- 在通过了系统的有效性测试及软件配置审查之后，就应开始系统的验收测试。
- 验收测试是以用户为主的测试。软件开发人员和QA（质量保证）人员也应参加。
- 由用户参加设计测试用例，使用生产中的实际数据进行测试。

- 在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。
- 确认测试应交付的文档有：
  - 确认测试分析报告
  - 最终的用户手册和操作手册
  - 项目开发总结报告。

验收测试的主要形式

- 根据合同进行的验收测试
- 用户验收测试
- 现场测试

α测试和β测试

- 在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。
- α测试是由一个用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的测试。

- α测试的目的是评价软件产品的FLURPS（即功能、局域化、可使用性、可靠性、性能和支持）。尤其注重产品的界面和特色。
- α测试可以从软件产品编码结束之时开始，或在模块（子系统）测试完成之后开始，也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。

- β测试是由软件的多个用户在实际使用环境下进行的测试。这些用户返回有关错误信息给开发者。
- 测试时，开发者通常不在测试现场。因而，β测试是在开发者无法控制的环境下进行的软件现场应用。
- 在β测试中，由用户记下遇到的所有问题，包括真实的以及主观认定的，定期向开发者报告。

- β测试主要衡量产品的**FLURPS**。着重于产品的支持性，包括文档、客户培训和支持产品生产能力
- 只有当α测试达到一定的可靠程度时，才能开始β测试。它处在整个测试的最后阶段。同时，产品的所有手册文本也应该在此阶段完全定稿。

软件测试的组织

- 测试团队的组建（**5**种模式）
- 各测试阶段中采用的模式
- 测试中的人员及其承担的任务
  - 测试经理
  - 测试设计人员
  - 测试自动化人员
  - 测试管理员
  - 测试人员

第六章 软件测试与质量管理

6.1 软件测试的概念

6.2 软件测试技术

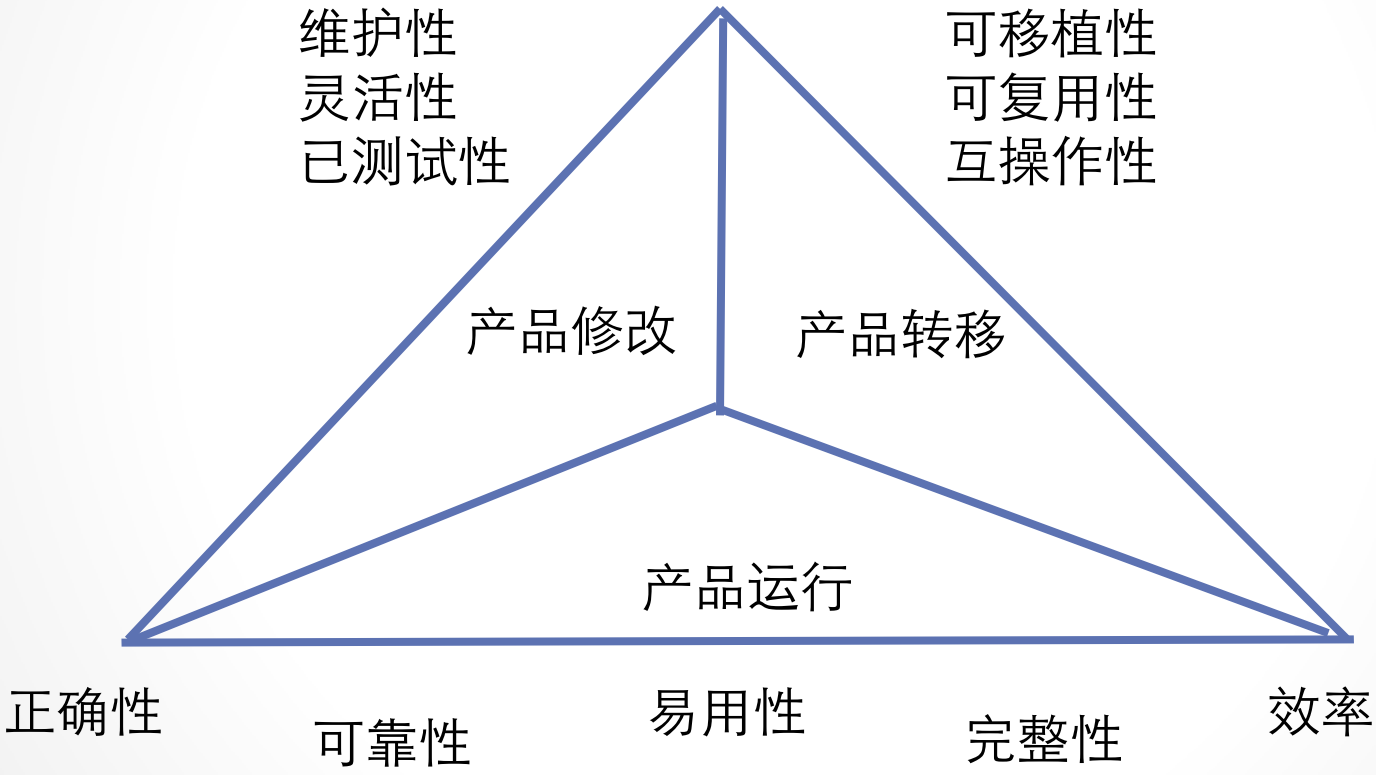
6.3 软件测试策略

6.4 软件质量保证

软件质量保证

- 软件测试的目的是发现软件的缺陷，从而保证软件的质量
- 明确表示是否符合功能和性能要求
- 用户满意度=合格的产品+好的质量+按预算和进度安排交付

如何度量质量



实现高质量软件的方法

- 软件工程方法
  - 合适的过程模型、需求分析方法、系统设计方法
- 项目管理技术
  - 项目计划、项目过程控制
- 质量控制（**Quality Control**）
  - 通过一系列技术确保每个工作产品符合其质量目标
- 质量保证（**Quality Assurance**）
  - 建立基础设置，支持软件工程方法，合理的项目管理，侧重过程与标准，如**ISO9000**、**CMM/CMMI**

技术评审

- 软件质量控制的一种方式
- 非正式评审
  - 同行审查：同事审查，临时会议或结对编程审查
  - 结对编程：两个人一起编写一段代码，提供即时和实时的工作同行审查
- 正式评审
  - 走查**(walkthrough)**和审查**(inspection)**
  - 会议：**3～5**人，制定计划，建立检查清单，限制时间和人员，对评审员必要的培训
  - 选择最有错误倾向的工作内容进行评审

软件质量保证的要素(策略)

- 标准：**IEEE, ISO**及其他标准
- 技术评审和工作过程审核
- 测试
- 错误/缺陷收集 与分析
- 变更管理（软件配置管理）
- 对软件技术与管理人员的教育
- 供应商管理
- 安全管理
- 风险管理

软件质量保证的目标、属性和要素

目标	属性	度量
需求质量	歧义	引起歧义地方的修改数量（如许多、大量）
	完备性	
	可理解性	节/小节的数量
	易变性	每项需求变更的数量
	可追溯性	不能追溯到设计/代码的需求数
	模型清晰性	UML模型数、每个模型中描述文字的页数、UML错误数
设计质量	体系结构完整性	是否存在现成的体系结构模型
	构件完备性	追溯到结构模型的构件数、过程设计的复杂性
	接口复杂性	挑选一个典型功能或内容的平均数、布局合理性
	模式	使用的模式数量

软件质量保证的目标、属性和要素

目标	属性	度量
代码质量	复杂性	环路复杂性
	可维护性	设计要素
	可理解性	内部注释的百分比 变量命名约定
	可重用性	可重用构件的百分比
	文档	可读性指标
质量控制效率	资源分配	每个活动花费的人员时间百分比
	完成率	实际完成时间与预算完成时间之比
	评审效率	参见评审度量
	测试效率	发现的错误及关键性问题数 改正一个所需的工作量 错误的根源

统计软件质量保证

	总 计		严 重		中 等		微 小	
错误	数量	百分比	数量	百分比	数量	百分比	数量	百分比
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
总计	942	100%	128	100%	379	100%	435	100%

ISO9001:2000标准

INFO

ISO 9001:2000标准

下面的大纲定义了ISO 9001:2000标准的基本要素。标准的有关详细信息可从国际标准化组织（www.iso.com）及其他Internet信息源（如www.praxiom.com）找到。

建立质量管理体系的要素。

建立、实施和改进质量体系。

制定质量方针，强调质量体系的重要性。

编制质量体系文件。

描述过程。

编制操作手册。

制定控制（更新）文件的方法。

制定记录保持的方法。

支持质量控制和质量保证。

提高所有利益相关者对质量重要性的认识。

注重客户满意度。

制定质量计划来描述目的、职责和权力。

制定所有利益相关者间的交流机制。

为质量管理体系建立评审机制。

确定评审方法和反馈机制。

制定跟踪程序。

确定质量资源（包括人员、培训、基础设施要素）。

建立控制机制。

针对策划。

针对客户需求。

针对技术活动（如分析、设计、试验）。

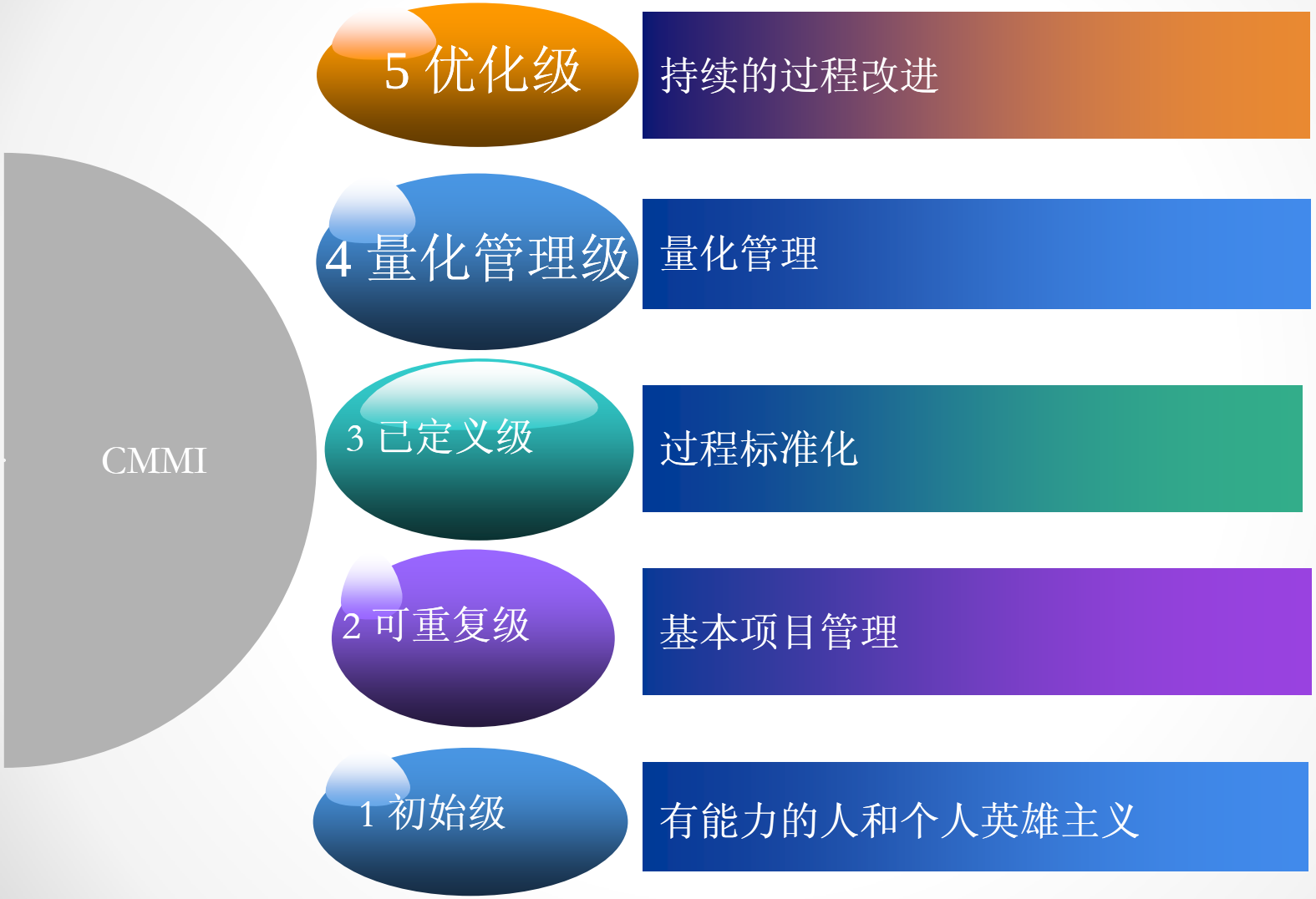
针对项目监测和管理。

制定补救措施。

评估质量数据和度量。

为持续的过程和质量改进制定措施。

成熟度模型标准（CMMI）



Question?