

软件质量与评测技术

Software Quality & Evaluation Technology

计算机学院 单纯
sherryshan@bit.edu.cn
2025年11月

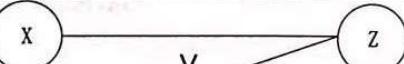
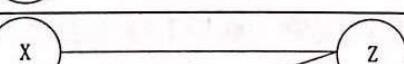
测试设计技术

Test Design Techniques

计算机学院 单纯
sherryshan@bit.edu.cn
2025年12月17日

3.1.6.5 因果图法 (6)

■ 因果图的关系符号

符号名	图例	释义
恒等		<ul style="list-style-type: none">• 若原因出现，则结果出现• 若原因不出现，则结果也不出现
非		<ul style="list-style-type: none">• 若原因出现，则结果不出现• 若原因不出现，则结果出现
与		<ul style="list-style-type: none">• 若几个原因都出现，结果才出现• 若其中有1个或更多原因不出现，则结果不出现
或		<ul style="list-style-type: none">• 若几个原因中有1个或更多出现，则结果出现• 若几个原因都不出现，则结果不出现
与非		<ul style="list-style-type: none">• 若几个原因中有1个或更多不出现，则结果出现• 若所有原因均出现，则结果不出现
或非		<ul style="list-style-type: none">• 若所有原因全部不出现，则结果出现• 若其中有1个或更多原因出现，则结果不出现

3.1.6.5 因果图法（7）

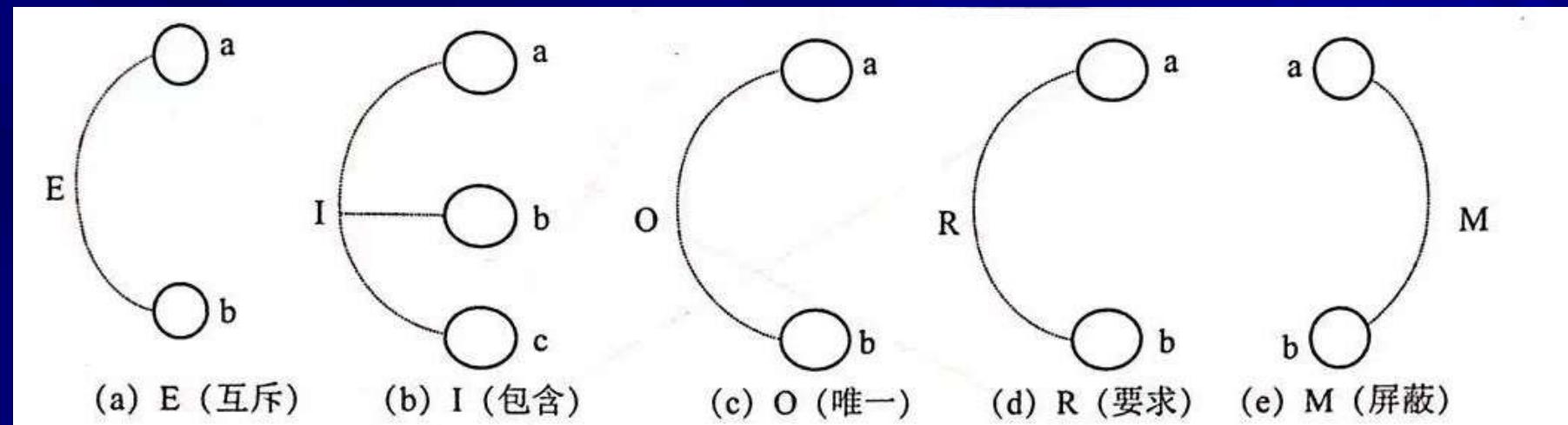
■ 因果图的约束符号

- 为了表示原因与原因之间，结果与结果之间可能存在的约束条件，在因果图中可以附加一些表示约束条件的符号

约束名	释义
E (互斥)	表示 a、b 两个原因不会同时成立，两个中最多有一个可能成立
I (包含)	表示 a、b、c 这 3 个原因中至少有一个必须成立
O (唯一)	表示 a 和 b 当中必须有一个，且仅有一个成立
R (要求)	表示当 a 出现时，b 必须也出现。a 出现时不可能 b 不出现
M (屏蔽)	表示当 a 是 1 时，b 必须是 0。而当 a 为 0 时，b 的值不定

3.1.6.5 因果图法 (8)

■ 因果图的约束符号 (续) — 图例



实例1（1）

■ 薪金计算 — 描述

- 年薪制员工：严重过失，扣年终风险金的4%；过失，扣年终风险金的2%
- 非年薪制员工：严重过失，扣当月薪资的8%；过失，扣当月薪资的4%

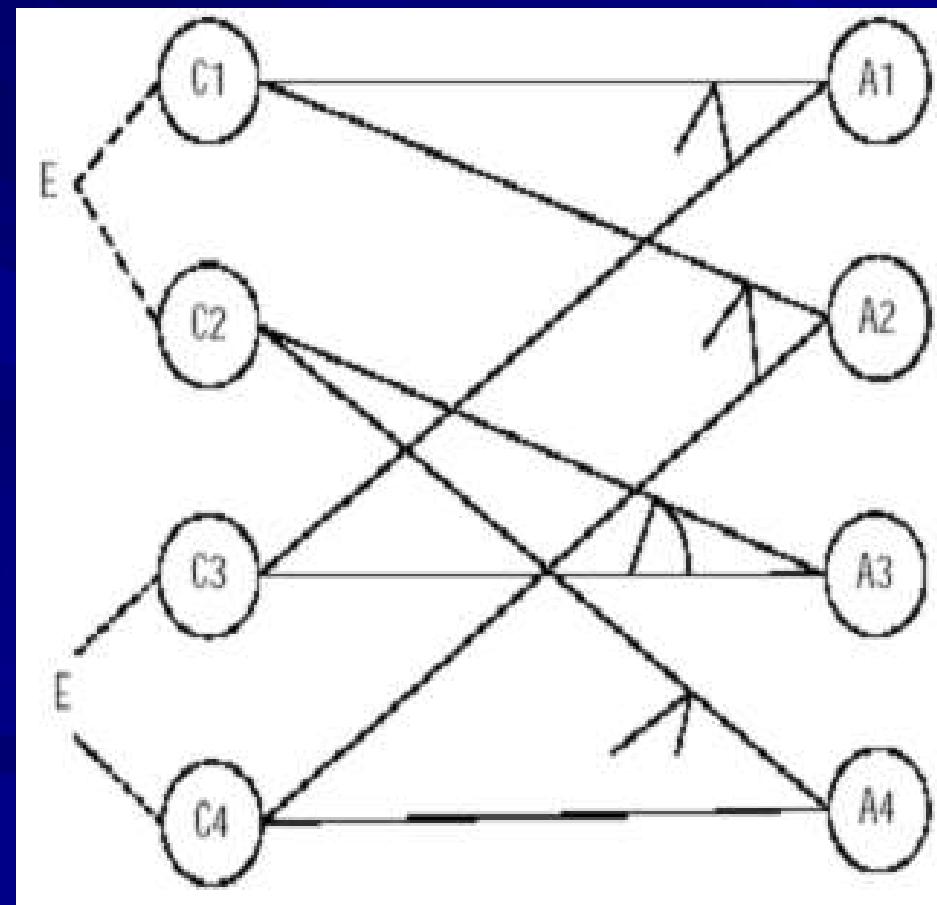
实例1（2）

■ 薪金计算（续） — 原因和结果

原因	结果
C1-年薪制员工	A1-扣年终风险金的 4%
C2-非年薪制员工	A2-扣年终风险金的 2%
C3-严重过失	A3-扣当月薪资的 8%
C4-过失	A4-扣当月薪资的 4%

实例1 (3)

■ 薪金计算 (续) — 因果图



实例1 (4)

■ 薪金计算（续） — 判定表

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
C2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
C3	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
C4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
A1						0	0			0	1					
A2						0	0			1	0					
A3						0	1			0	0					
A4						1	0			0	0					
TC						Y	Y			Y	Y					

■ 判定表中TC标记为Y的每一列就是测试用例

实例2 (1)

■ 自动售货机 — 产品说明书

- 有一个处理单价为1元5角钱的盒装饮料的自动售货机软件。若投入1元5角硬币，按下“可乐”、“雪碧”、或“红茶”按钮，相应的饮料就送出来。若投入的是2元硬币，在送出饮料的同时退还5角硬币

实例2 (2)

■ 自动售货机（续）

– 分析

■ 原因

- 投入1元5角硬币
- 投入2元硬币
- 按“可乐”按钮
- 按“雪碧”按钮
- 按“红茶”按钮

■ 中间状态

- 已投币
- 已按钮

实例2 (3)

■ 自动售货机（续）

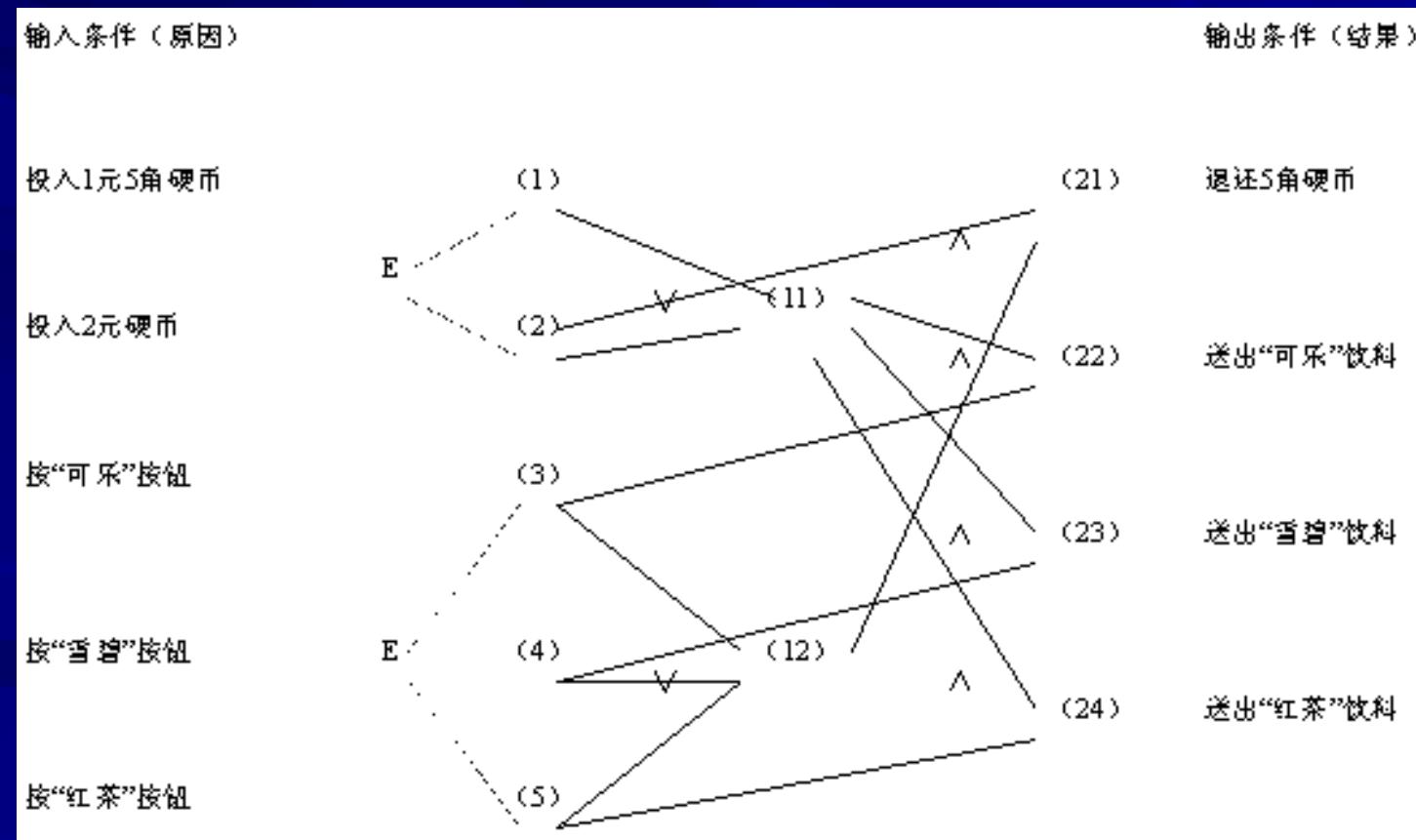
– 分析（续）

■ 结果

- 退还5角硬币
- 送出“可乐”饮料
- 送出“雪碧”饮料
- 送出“红茶”饮料

实例2 (4)

■ 自动售货机（续） — 因果图



实例2 (5)

■ 自动售货机（续） — 测试用例

			1	2	3	4	5	6	7	8	9	10	11
输入	投入 1 元 5 角硬币	(1)	1	1	1	1	0	0	0	0	0	0	0
	投入 2 元硬币	(2)	0	0	0	0	1	1	1	1	0	0	0
	按“可乐”按钮	(3)	1	0	0	0	1	0	0	0	1	0	0
	按“雪碧”按钮	(4)	0	1	0	0	0	1	0	0	0	1	0
	按“红茶”按钮	(5)	0	0	1	0	0	0	1	0	0	0	1
中间 结点	已投币	(11)	1	1	1	1	1	1	1	1	0	0	0
	已按钮	(12)	1	1	1	0	1	1	1	0	1	1	1
输出	退还 5 角硬币	(21)	0	0	0	0	1	1	1	0	0	0	0
	送出“可乐”饮料	(22)	1	0	0	0	1	0	0	0	0	0	0
	送出“雪碧”饮料	(23)	0	1	0	0	0	1	0	0	0	0	0
	送出“红茶”饮料	(24)	0	0	1	0	0	0	1	0	0	0	0

3.1.7 结论

- 所有从事软件测试和即将从事软件测试的人大都是从黑盒测试做起的，每种类型的软件有各自的特点，每种测试用例设计方法也有各自的特点，针对不同软件如何利用这些黑盒方法是非常重要的，它能极大地提高测试效率和测试覆盖度，认真掌握这些方法的原理，有效地提高测试水平，积累更多的测试经验，这是测试人员最宝贵的财富

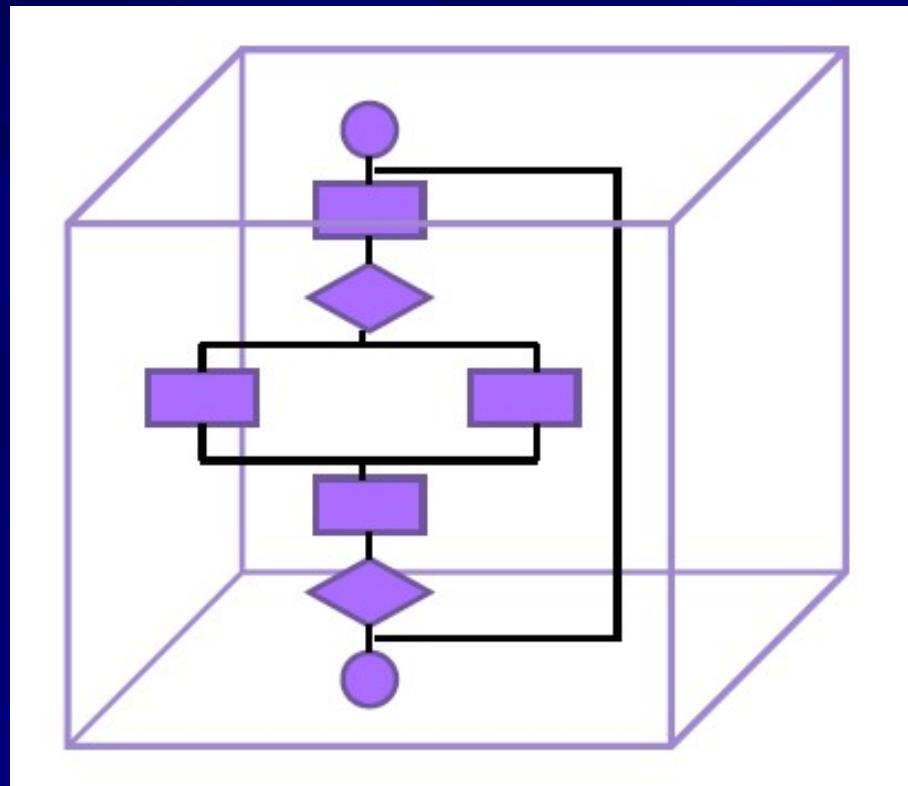
3.2 白盒测试用例设计技术

- 3.2.1 白盒测试的定义
- 3.2.2 白盒测试的作用
- 3.2.3 白盒测试的局限
- 3.2.4 黑盒测试 VS 白盒测试
- 3.2.5 白盒测试的内容
- 3.2.6 如何安排白盒测试
- 3.2.7 白盒测试用例设计技术

3.2.1 白盒测试的定义（1）

- 白盒测试是知道产品内部工作过程，可通过测试来检测产品内部动作是否按照规格说明书的规定正常进行

3.2.1 白盒测试的定义（2）



- 白盒测试依赖于代码、规格说明或其他源资料的信息
- 在白盒测试中，软件测试员可以访问程序员的代码，并通过检查代码来协助测试

3.2.1 白盒测试的定义（3）

■ 前提

- 知道软件产品内部工作过程

■ 目标

- 通过测试来检测软件产品内部动作是否按照规格说明书的规定正常进行

■ 重点

- 按照软件内部的结构测试程序中的每条通路是否都能按预定要求正确工作

3.2.2 白盒测试的作用（1）

■ 有了“黑盒”测试为什么还要“白盒”测试？

- 黑盒测试只能观察软件的外部表现，即使软件的输入输出都是正确的，却并不能说明软件就是正确的。因为程序有可能用错误的运算方式得出正确的结果，例如“负负得正，错错得对”，只有白盒测试才能发现真正的原因
- 白盒测试能发现程序里的隐患，像内存泄漏、误差累计问题。在这方面，黑盒测试存在严重的不足

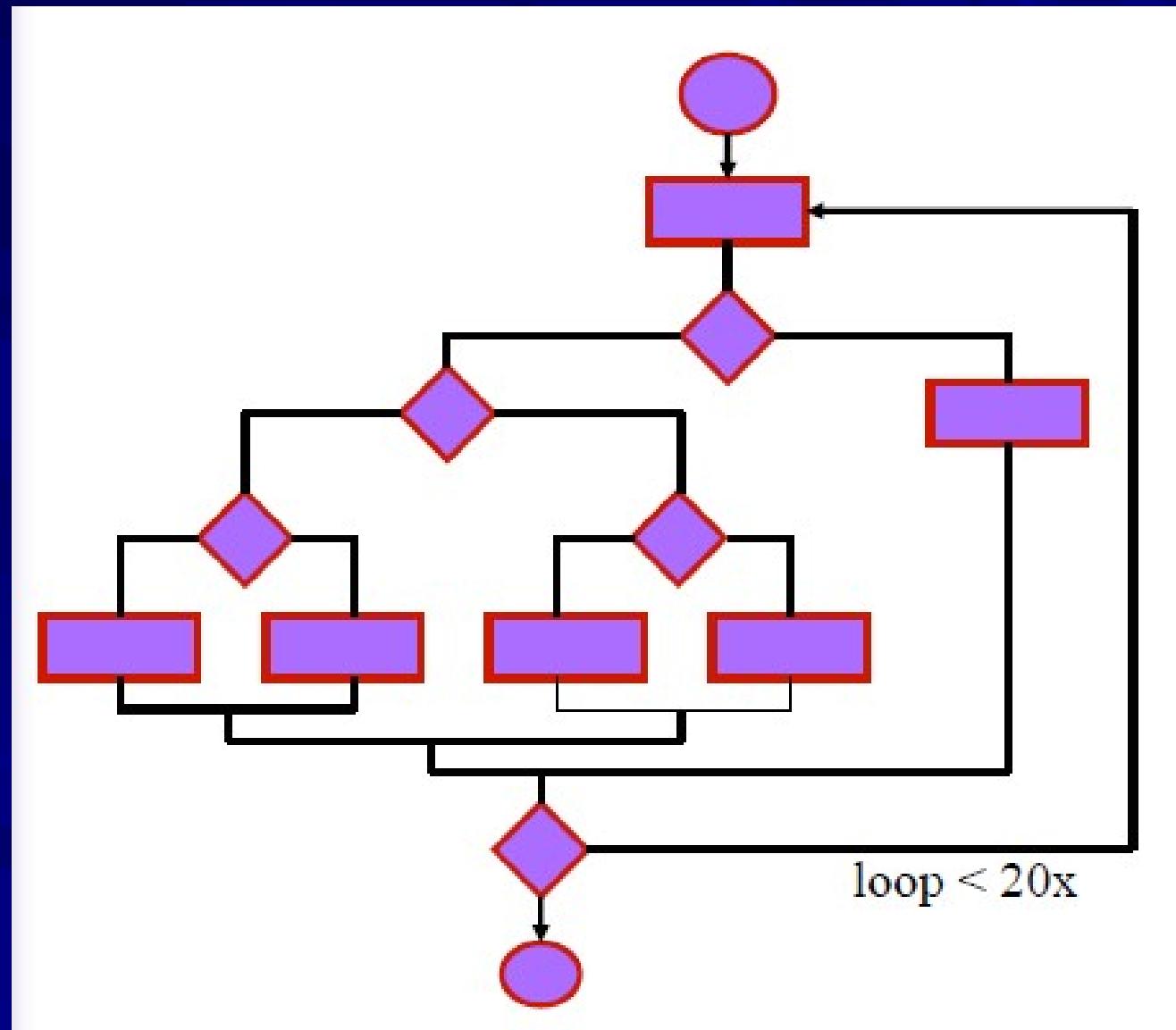
3.2.2 白盒测试的作用（2）

- 软件人员使用白盒测试方法，主要想对程序模块进行如下的检查
 - 对程序模块的所有独立执行路径至少测试一次
 - 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次
 - 在循环的边界和运行界限内执行循环体
 - 测试内部数据结构的有效性

3.2.3 白盒测试的局限（1）

- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。给出一个小程序的流程图，它包括了一个执行 20 次的循环
- 包含的不同执行路径数达 5^{20} 条，对每一条路径进行测试需要1毫秒，假定一年工作 365×24 小时，要想把所有路径测试完，需 3170 年

3.2.3 白盒测试的局限 (2)



3.2.4 黑盒测试 VS 白盒测试 (1)

■ 黑盒测试

- 不涉及程序结构
- 用软件规格说明生成测试用例
- 某些代码段得不到测试
- 可适用于从单元测试到系统测试
- 需要用白盒测试加以补充

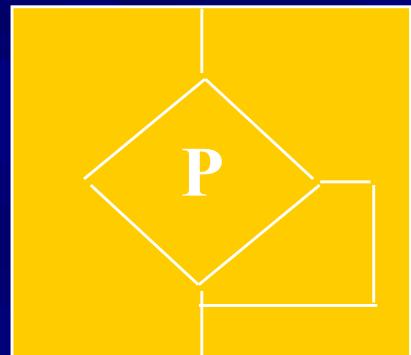
3.2.4 黑盒测试 VS 白盒测试 (2)

■ 白盒测试

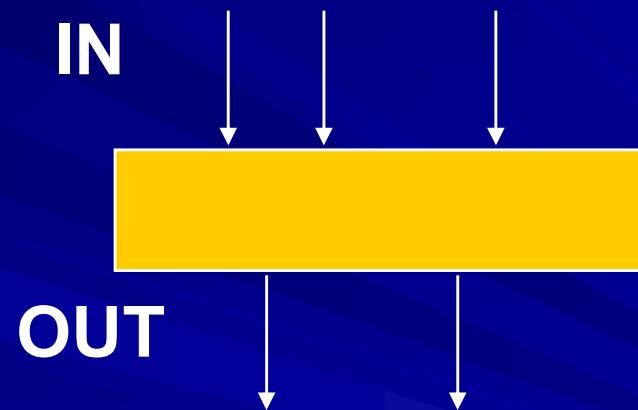
- 考查程序逻辑结构
- 用程序结构信息生成测试用例
- 通常适用于单元测试和集成测试

3.2.4 黑盒测试 VS 白盒测试 (3)

白盒测试



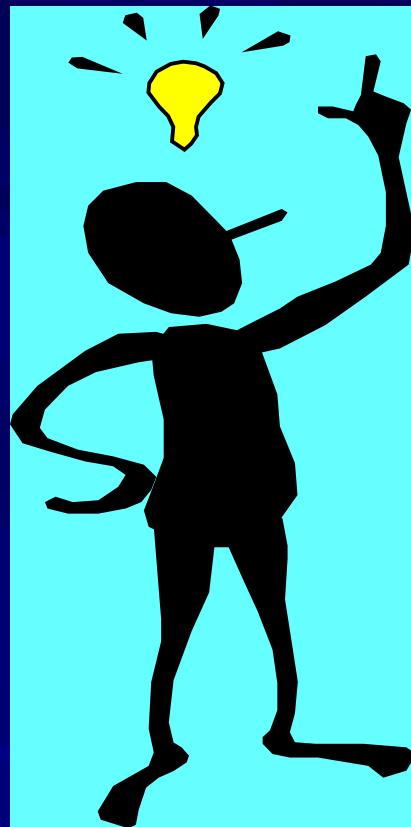
黑盒测试



3.2.4 黑盒测试 VS 白盒测试 (4)

	黑盒测试	白盒测试
优点	<ul style="list-style-type: none">●适用于各测试阶段●从产品功能角度测试●容易入手，生成测试数据	<ul style="list-style-type: none">●可以生成测试数据，使特定程序部分得到测试●有一定的充分性度量手段●可获得较多工具支持
缺点	<ul style="list-style-type: none">●某些代码段得不到测试●如果规格说明有误则无法发现●不易进行充分性度量	<ul style="list-style-type: none">●不易生成测试数据●无法对未实现规格说明的部分测试●工作量大，通常只用于单元测试，有引用局限
性质	是一种确认技术，回答“我们在构造一个正确的系统吗？”	是一种验证技术，回答“我们在正确地构造一个系统吗？”

3.2.5 白盒测试的内容



- 代码检查
- 静态结构分析
- 代码质量度量
- 功能确认与接口分析
- 逻辑覆盖率分析
- 性能与效率分析
- 内存分析

3.2.6 如何安排白盒测试

- 3.2.6.1 单元测试
- 3.2.6.2 集成测试
- 3.2.6.3 系统测试
- 3.2.6.4 验收测试
- 3.2.6.5 白盒测试综合策略

3.2.6.1 单元测试（1）

- 单元测试是在模块源程序代码编写完成之后进行的测试。只有通过了单元测试的模块，才可以把它们集成到一起进行集成测试。否则，即使集成测试通过了，投入使用 的软件也会像地基不牢的摩天大楼一样暗藏着很多不安全因素

3.2.6.1 单元测试（2）

- 由于在软件开发后期可能会因为需求变更或功能完善等原因对某个程序单元的代码做某些改动，所以可以把单元测试看成是从详细设计开始一直到系统构造完成贯穿于整个时期的一种活动

3.2.6.1 单元测试（3）

- 单元测试的定义和目标
- 调试
- 单元测试环境
- 单元测试策略
- 如何安排白盒测试

单元测试的定义和目标（1）

- 对于传统的结构化程序而言，程序单元是指程序中定义的函数或子程序，单元测试就是对函数或子程序进行的测试
- 对于面向对象的程序而言，程序单元是指特定的一个具体的类或相关的多个类，单元测试是对类的测试；但有时候，在一个类特别复杂时，就会把方法作为一个单元进行测试

单元测试的定义和目标（2）

- 单元测试是针对软件设计的最小单位——程序模块，进行正确性检验的测试工作，其目的在于
 - 验证代码是与设计相符合的
 - 跟踪需求和设计的实现
 - 发现设计和需求中存在的缺陷
 - 发现在编码过程中引入的错误
- 在单元测试活动中，软件的每个单元应在与程序的其他部分相隔离的情况下进行测试

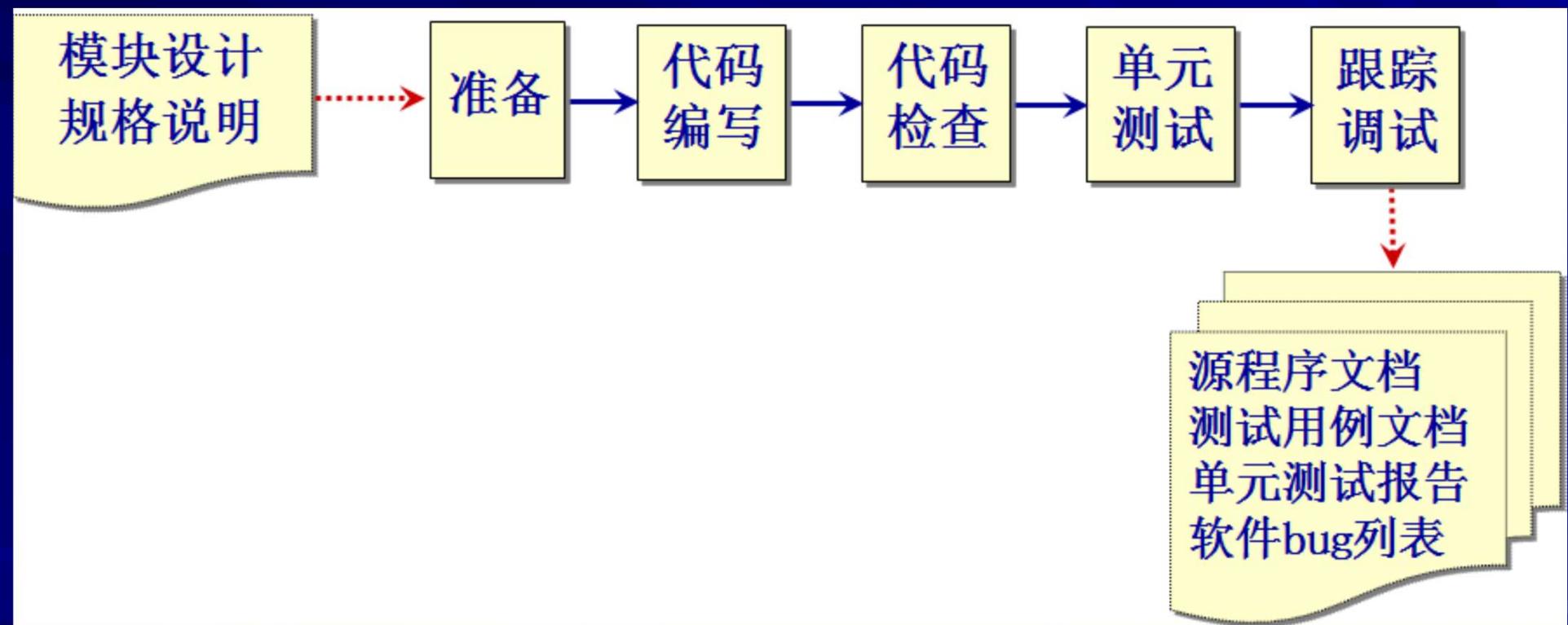
单元测试的定义和目标（3）

- 测试的主要工作分为两个步骤：人工检查和动态执行跟踪
 - 人工检查主要是保证代码算法的逻辑正确性、清晰性、规范性、一致性、算法高效性，并尽可能地发现程序中没有发现的错误
 - 动态执行跟踪就是通过设计测试用例，执行待测程序来跟踪比较实际结果与预期结果来发现错误

单元测试的定义和目标（4）

- 经验表明，使用人工静态检查法能够有效地发现30%~70%的逻辑设计和编码错误
- 但是代码中仍会有大量的隐蔽的错误无法通过视觉检查发现，必须通过跟踪调试法细心分析才能够捕捉到。所以，动态跟踪调试方法也成了单元测试的重点与难点

单元测试的定义和目标 (5)



单元测试的定义和目标（6）

- 单元测试与其他测试不同，单元测试可看做是编码工作的一部分，在编码的过程中考虑测试问题，得到的将是更优质的代码，因为在这时程序员对代码应该做些什么了解得最清楚。因此一般应该由程序员完成单元测试工作，并且在提交产品代码的同时也提交测试代码

调试 (1)

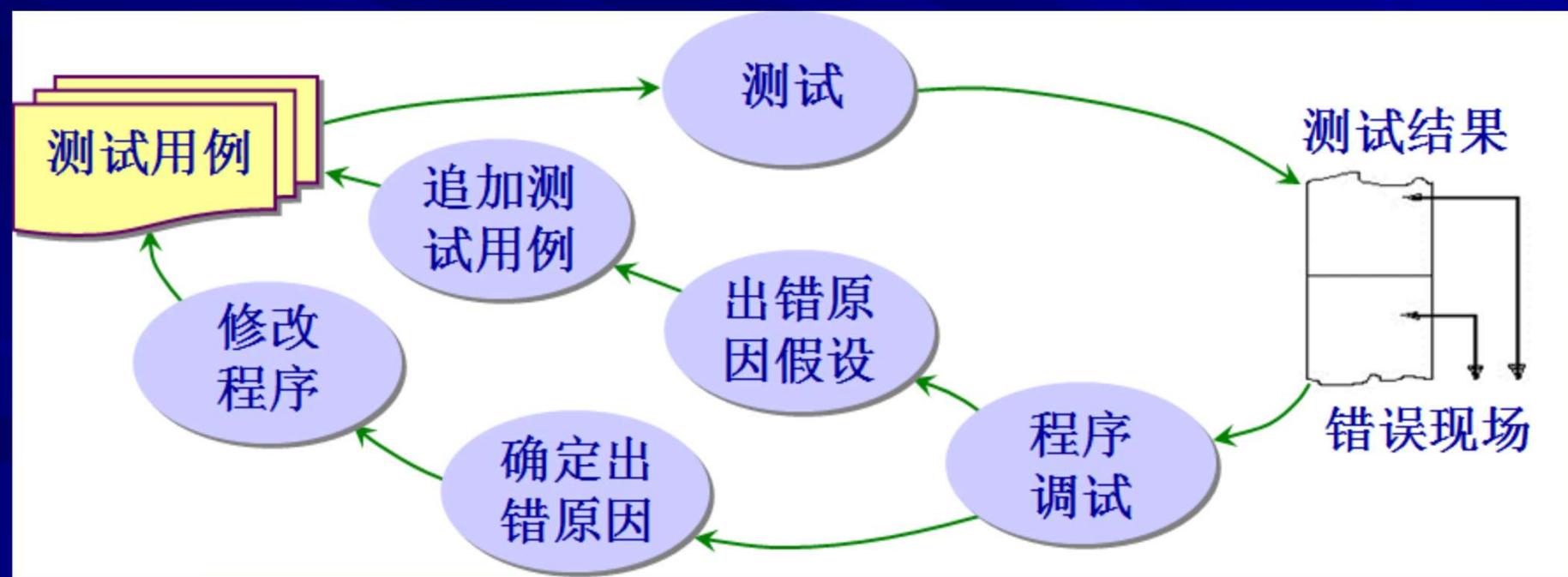
- 软件调试是在进行了成功的测试之后才开始的工作。它与软件测试不同，调试的任务是进一步诊断和改正程序中潜在的错误
- 调试活动由两部分组成
 - 确定程序中可疑错误的确切性质和位置
 - 对程序（设计、编码）进行修改，排除这个错误

调试 (2)

- 调试工作是一个具有很强技巧性的工作。
- 软件运行失效或出现问题，往往只是潜在错误的外部表现，而外部表现与内在原因之间常常没有明显的联系。如果要找出真正的原因，排除潜在的错误，不是一件易事
- 调试是通过现象，找出原因的一个思维分析的过程

调试 (3)

- 调试不是测试，但是，它是作为测试的后继工作而出现的
- 调试的执行步骤



调试 (4)

■ 调试的执行步骤（续）

- 从错误的外部表现形式入手，确定程序中出错位置
- 研究有关部分的程序，找出错误的内在原因。
有两种可能
 - 错误原因不能肯定，先做某种假设，再设计测试用例来证实这个假设
 - 找到错误原因，分析相关程序和数据结构，界定修改范围
- 修改设计和代码，以排除这个错误

调试 (5)

■ 调试的执行步骤（续）

- 重复进行暴露了这个错误的原始测试或某些有关测试，以确认
 - 该错误是否被排除
 - 是否引进了新的错误
- 如果所做修正无效，则撤消这次修改，重复上述过程，直到找到一个有效的解决办法为止

调试 (6)

■ 从技术角度来看，查找错误的难度在于

- 现象与原因所处的位置可能相距甚远
- 当其他错误得到纠正时，这一错误所表现出的现象可能会暂时消失，但并未实际排除
- 现象实际上是由一些非错误原因（例如，舍入不精确）引起的
- 现象可能是由于一些不容易发现的人为错误引起的
- 错误是由于时序问题引起的，与处理过程无关

调试 (7)

■ 从技术角度来看，查找错误的难度在于（续）

- 现象是由于难于精确再现的输入状态（例如，实时应用中输入顺序不确定）引起
- 现象可能是周期出现的。在软、硬件结合的嵌入式系统中常常遇到

单元测试环境（1）

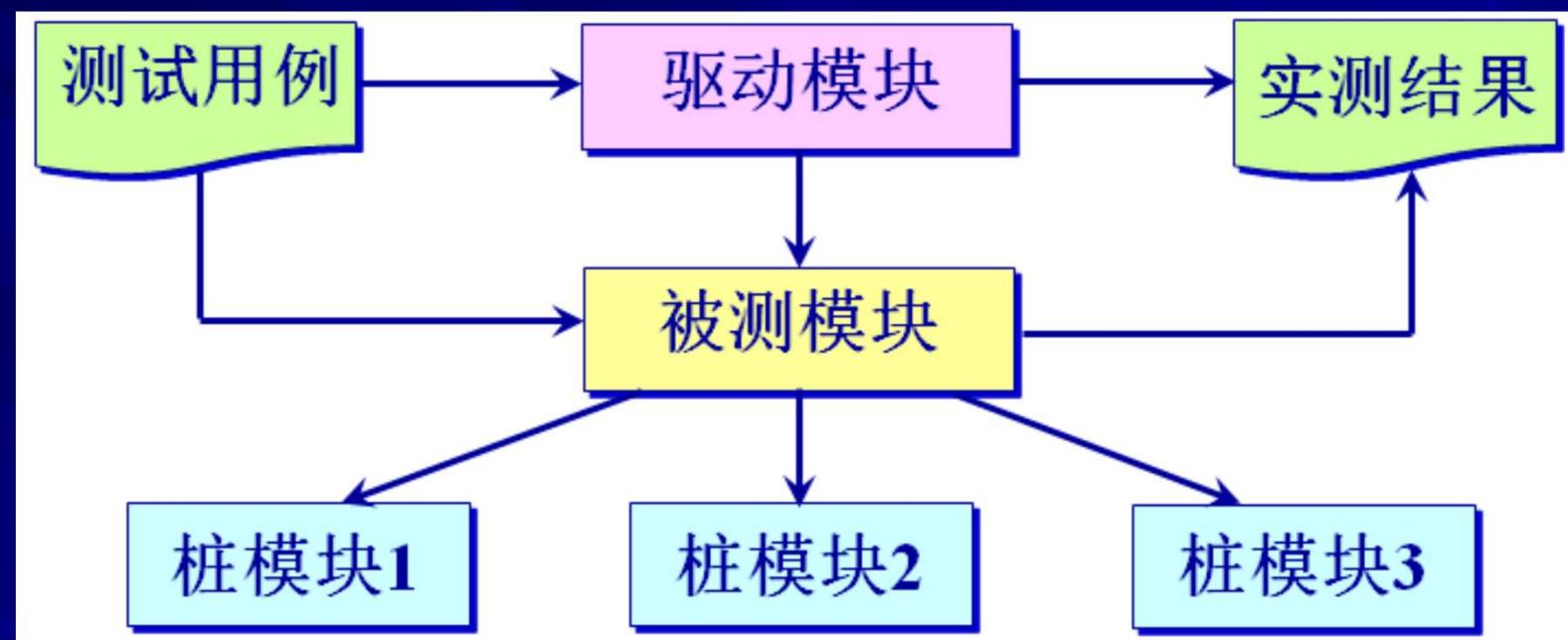
- 由于一个模块或一个方法（操作）并不是一个独立的程序，在考虑测试它时要同时考虑它和外界的联系，因此要用到一些辅助模块，来模拟与被测模块相联系的其他模块

单元测试环境（2）

■ 一般把这些辅助模块分为两种

- 驱动模块（driver）：相当于被测模块的控制程序。它接收测试数据，把这些数据传送给被测模块，被测模块执行它本身的功能，然后输出实际测试结果
- 桩模块（stub）：用于代替被测模块调用的子模块。桩模块可以进行少量的数据操作，不需要实现子模块的所有功能，但要根据需要来实现或代替子模块的部分功能

单元测试环境 (3)



单元测试环境 (4)

- 驱动模块和桩模块是辅助测试用的，不属于应交付的软件产品，但它需要一定的开发费用
- 若驱动模块和桩模块比较简单，实际开销相对低些。提高模块的内聚性可以简化单元测试。如果每个模块只完成一个功能，所需测试用例数目将显著减少，模块中的错误也更容易发现

单元测试环境（5）

- 对于包或子系统，可以根据所编写的测试用例编写一个测试模块类来做驱动模块，用于测试包中所有的待测试模块。最好不要在每个类中用一个测试函数的方法来测试跟踪类中所有的方法

单元测试策略 (1)

■ 自顶向下的单元测试策略

- 从最顶层的单元开始，把顶层调用的单元用桩模块代替，对顶层模块做单元测试
- 对下一层单元进行测试时，使用上面已测试的单元做驱动模块，并为被测模块编写新的桩模块
- 依次类推，直到全部单元测试结束

单元测试策略（2）

- 这种测试策略的优点是：可以在集成测试之前为系统提供早期的集成途径
- 由于详细设计一般都是自顶向下进行的，这样自顶向下的单元测试测试策略在顺序上与详细设计一致。因此，测试工作可以与详细设计和编码工作重叠进行
- 这种测试策略的缺点是：单元测试被桩模块控制，随着单元测试的不断进行，测试过程也会变得越来越复杂

单元测试策略（3）

■ 自底向上的单元测试策略

- 先对调用它的最底层单元进行测试，使用驱动模块来代替调用它的上层单元
- 对上一层单元进行测试时，用已经被测试过的模块做桩模块，并为被测单元编写新的驱动模块
- 依次类推，直到全部单元测试结束

单元测试策略（4）

- 这种策略的优点是：不需要单独设计桩模块；无需依赖结构设计，可以直接从功能设计中获取测试用例，可以为系统提供早期的集成途径
- 这种策略的缺点是：自底向上的单元测试不能和详细设计、编码同步进行

单元测试策略（5）

■ 孤立测试

- 这种测试的策略不需要考虑每个模块与其他模块之间的关系，分别为每个模块单独设计桩模块和驱动模块，逐一完成所有单元模块的测试
- 这种测试策略的优点是：方法简单、易操作，能够达到高覆盖率。因为一次测试只需要测试一个单元，其驱动模块比自底向上的驱动模块设计简单，而其桩模块的设计也比自顶向下策略中使用的桩模块简单

单元测试策略（6）

■ 孤立测试（续）

- 另外，各模块之间不存在依赖性，所以单元测试可以并行进行
- 这种测试策略的缺点是：不能为集成测试提供早期的集成途径。依赖结构设计信息，需要设计多个桩模块和驱动模块，增加了额外的测试成本

单元测试策略（7）

■ 综合测试

- 在单元测试过程中，为了有效地减少开发桩模块的工作量，可以考虑综合自底向上测试策略和孤立测试策略

单元测试策略 (8)

```
void funcA ( int a, int b )
{
    if ( max (a, b) < 0 )
        printf ( “所有输入值为负数!\n” );
    else
        printf ( “输入值中至少有一个大于或
等于 0 的整数!\n” );
}

int max(int a, int b) {
    if (a >= b) return a;
    else return b;
}
```

单元测试策略 (9)

■ 为了减轻桩模块工作量

- 首先采用由底向上的测试策略对max函数进行单元测试
- 然后借鉴孤立测试策略，直接使用max做桩来测试funcA函数
- 在设计用例时不关注max本身怎么执行，而只关注该桩要返回一个小于零和大于等于零的值，以验证funcA能否在这两种情况下输出需要的信息

如何安排白盒测试

■ 单元测试

- 编码阶段，依据详细设计说明书
- 代码检查
- 功能确认与接口分析
- 覆盖率分析
- 内存分析

3.2.6.2 集成测试

- 集成测试的定义和目标
- 集成测试的策略
- 如何安排白盒测试

集成测试的定义和目标（1）

- 集成测试的目标是根据实际情况对程序模块采用适当的集成测试策略组装起来，对系统的接口以及集成后的功能进行正确性检验的测试工作

集成测试的定义和目标（2）

■ 这时需要考虑的问题是

- 在把各个模块连接起来的时候，穿越模块接口的数据是否会丢失
- 一个模块的功能是否会对另一个模块的功能产生不利的影响
- 各个子功能组合起来，能否达到预期要求的父功能
- 全局数据结构是否有问题
- 单个模块的误差累积起来，是否会放大，从而达到不能接受的程度

集成测试的定义和目标（3）

- 软件集成测试是依据概要设计说明和集成测试计划进行的。通常要根据具体情况采取不同的集成测试策略将多个模块組裝成子系统或系统，测试构成被测应用程序的各个模块能否以正确、稳定、一致的方式交互，即验证其是否符合软件开发过程中的概要设计说明的要求

集成测试的定义和目标（4）

- 对于传统软件来说，按集成程度不同，可以把集成测试分为 3 个层次，即
 - 模块内集成测试
 - 子系统内集成测试
 - 子系统间集成测试
- 对于面向对象的应用系统来说，按集成程度不同，可以把集成测试分为2个层次，即
 - 类内集成测试
 - 类间集成测试

集成测试的定义和目标（5）

- 在搭建集成测试环境时，可以从以下几个方面进行考虑
 - 硬件环境。在集成测试时，应尽可能考虑实际的使用环境。如果实际使用环境不可用，才考虑可替代的环境或在模拟环境下进行
 - 操作系统环境。在对软件进行集成测试时不但要考虑不同机型，而且要考虑到实际环境中安装的各种具体的操作系统环境，并仔细地选择测试用例

集成测试的定义和目标（6）

- 在搭建集成测试环境时，可以从以下几个方面进行考虑（续）
 - 数据库环境。在搭建集成测试所使用的数据库环境时要从性能、版本、容量等多方面考虑，选择适用的数据库产品
 - 网络环境。一般用户所使用的网络环境都是以太网。可以利用开发组织的内部网络环境作为集成测试的网络环境。特殊环境要求除外（如有的软件运行需要无线设备）

集成测试的定义和目标（7）

- 在搭建集成测试环境时，可以从以下几个方面进行考虑（续）
 - 测试工具运行环境。有些集成测试必须借助测试工具才能够完成，因此也需要搭建一个测试工具能够运行的环境
 - 其他环境。如Web应用所需要的Web服务器环境、浏览器环境等。这就要求测试人员根据具体要求进行搭建

集成测试的策略（1）

- 软件集成有很多种方式，每一种方式有其自身的优缺点，因此要根据软件系统的实际特点来选择合适的集成测试策略
- 在实际集成测试的过程中，可以将多种集成测试的策略结合起来，完成对被测软件的集成测试

集成测试的策略（2）

■ 集成方式

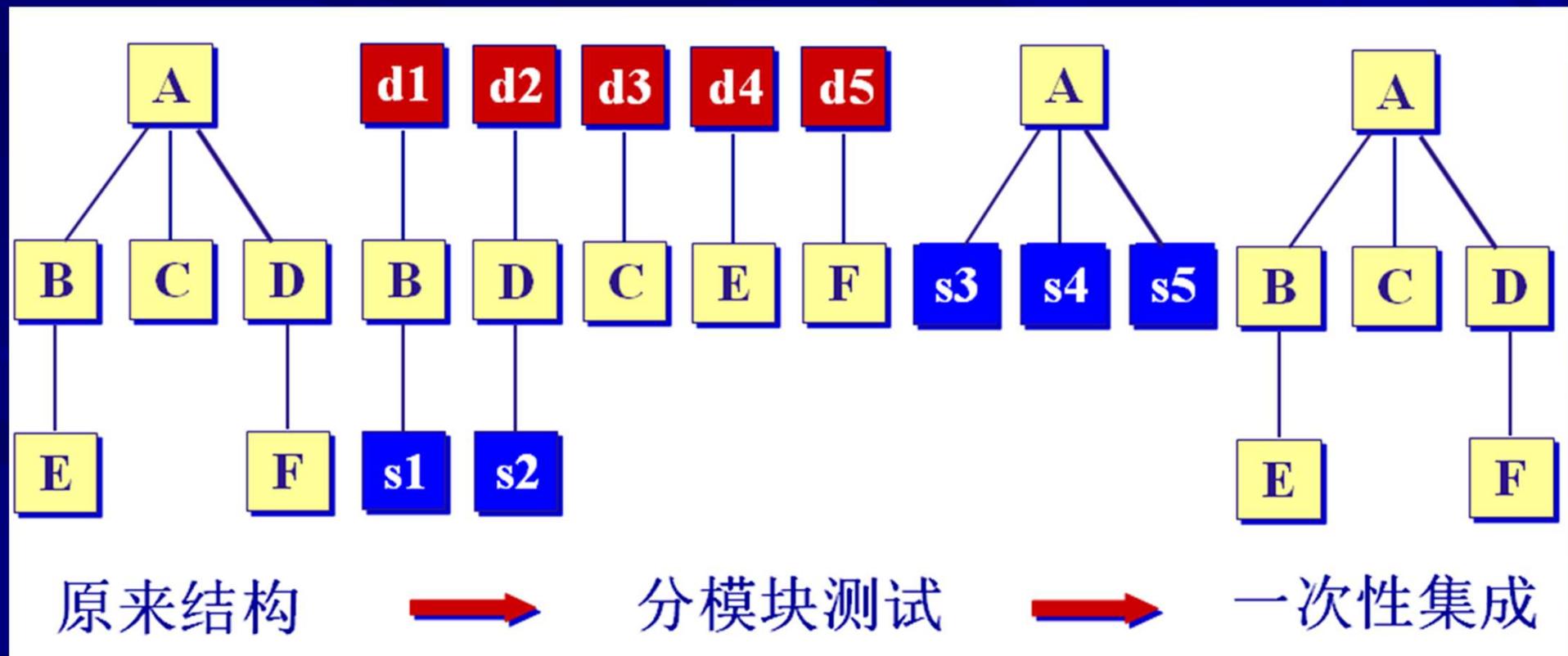
- 基于分解的集成
 - 一次性集成方式
 - 增量式集成方式
 - 自顶向下的增量式集成
 - 自底向上的增量式集成
 - 混合的增量式（三明治）集成
- 基于层次的集成

一次性集成方式（1）

- 这种集成方式是一种非增量式集成策略。它先对每个单元分别进行单元测试，然后把所有单元组装在一起进行测试，最终得到要求的软件系统

一次性集成方式 (2)

- 模块d1, d2, d3, d4, d5是为单元测试建立的驱动模块，s1, s2, s3, s4, s5是为单元测试建立的桩模块



一次性集成方式 (3)

- 优点是只要极少数的驱动和桩模块；需要的测试用例最少；方法简单；多个测试人员可以并行工作，对人力、物力资源利用率较高
- 缺点是一次试运行成功的可能性不大；错误定位和修改困难；有许多接口错误很容易躲过测试

自顶向下的增量式集成（1）

- 这种集成方式采用与设计一样的顺序，沿系统结构的控制层次自顶向下逐步组装各个单元。它首先将测试活动集中于顶层的构件，然后逐层向下，测试处于低层的构件
- 自顶向下的集成方式可以采用深度优先策略和广度优先策略

自顶向下的增量式集成（2）

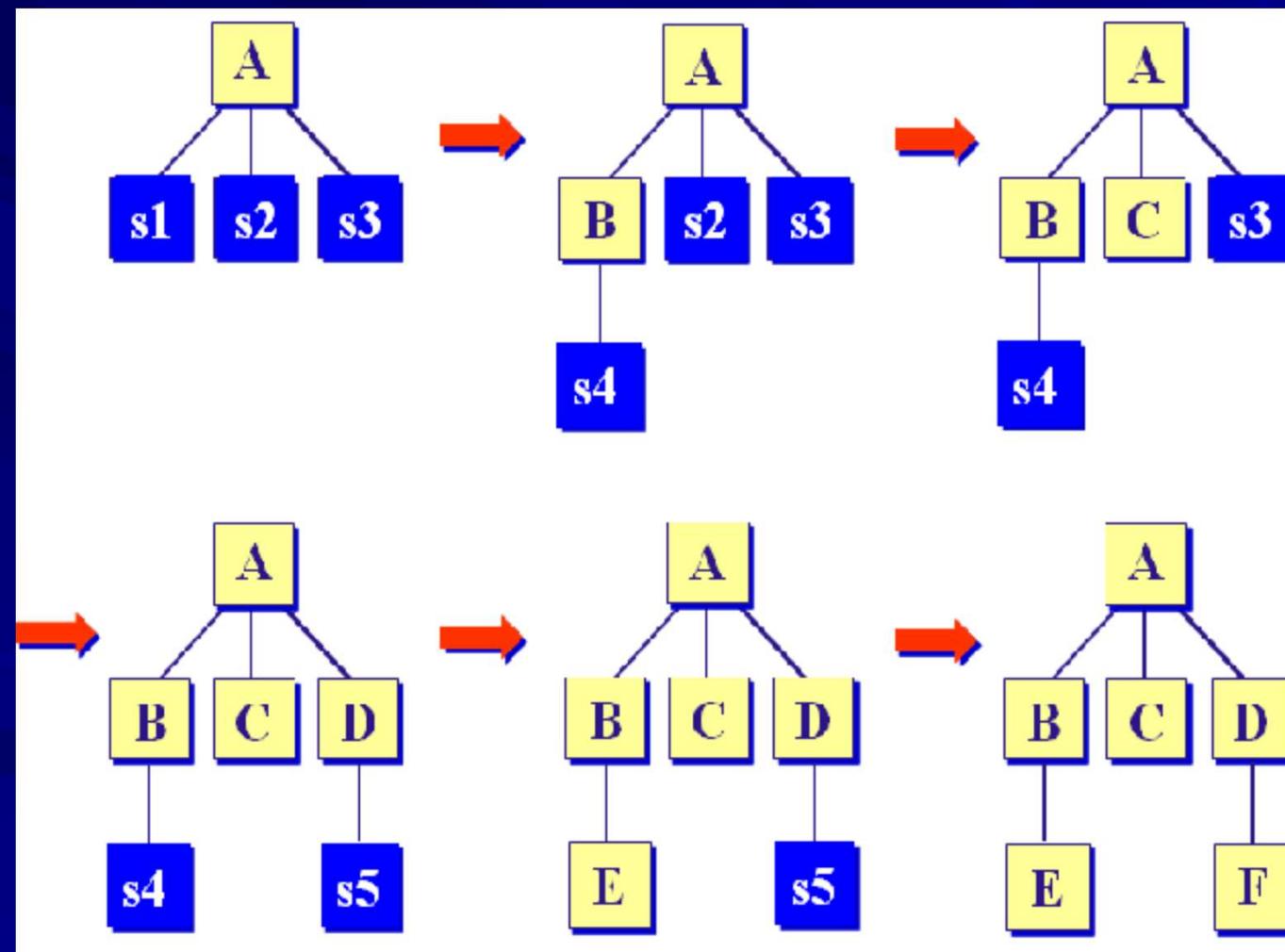
- 优点是在增量式集成的过程中较早地验证了主要的控制和判断点；减少了驱动模块开发的费用；由于集成顺序和设计顺序一致，可以和设计并行进行；支持故障隔离

自顶向下的增量式集成（3）

- 缺点是建立桩模块的成本较高；底层构件中的一个无法预计的需求可能会导致许多顶层构件的修改；底层构件行为的验证被推迟了；随着底层模块的不断增加，整个系统越来越复杂，导致底层模块的测试不充分，尤其是那些被复用的模块

自顶向下的增量式集成 (4)

■ 广度优先的自顶向下增量式集成

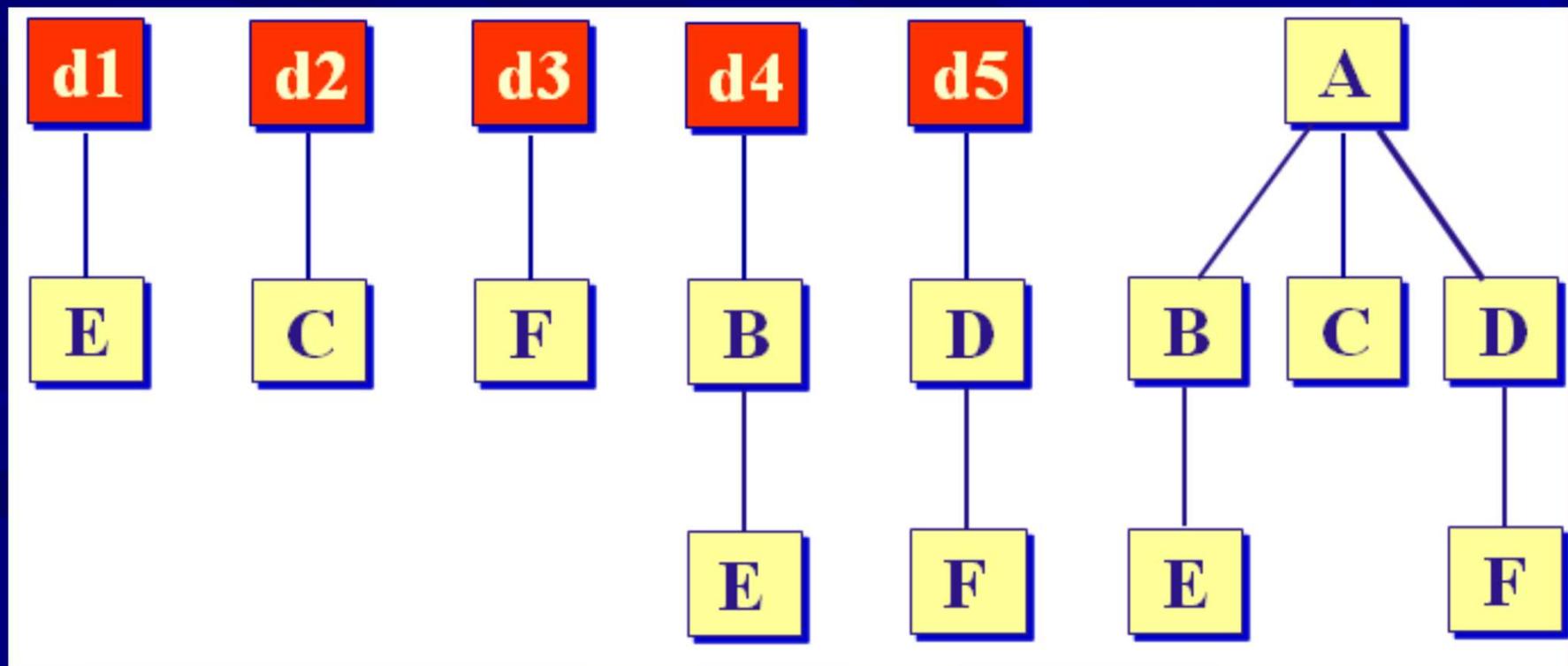


自底向上的增量式集成 (1)

- 这种集成方式是从程序模块结构的最底层的模块开始集成和测试
- 采用这种集成方式，对于一个给定层次的指定模块，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，所以不再需要桩模块
- 在模块的测试过程中想要从子模块得到信息可以通过直接运行子模块得到

自底向上的增量式集成 (2)

- 自底向上的增量式集成的步骤如图所示
 - 其中，d1, d2, d3, d4, d5代表驱动模块，图形中的集成顺序为由左到右



自底向上的增量式集成（3）

■ 自底向上的增量式集成的步骤如下

- 从模块结构的底层模块开始测试，并为其建立驱动模块
- 用实际模块代替驱动模块，与已测试的直接下属子模块组装成一个更大的模块组进行测试
- 重复上面的工作直到系统的最顶层模块被加入到已测的系统中

自底向上的增量式集成 (4)

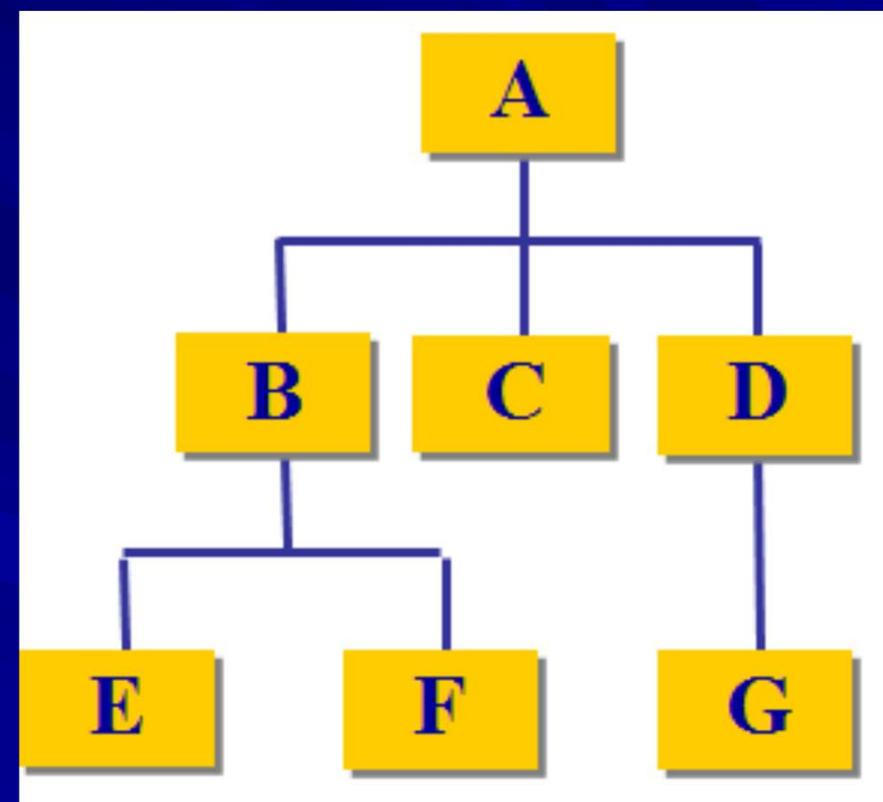
- 优点是对底层模块的行为能够早期验证；可以并行进行测试和集成，驱动模块的编写工作量远比桩模块的编写工作量小；支持故障隔离
- 缺点是对高层的验证被推迟到了最后，设计上的错误不能被及时发现；对于底层的一些异常很难覆盖
- 自顶向下增量的方式和自底向上增量的方式各有优缺点。一般来讲，一种方式的优点是另一种方式的缺点

混合的增量式（三明治）集成 (1)

- 这种集成方式把系统分成三层，中间一层为目标层。测试时对目标层上面各层使用由顶向下的集成策略，对目标层下面各层使用自底向上的集成策略，最后在目标层会合

混合的增量式（三明治）集成 (2)

- 以右图的程序结构为例，目标层为B、C、D。目标层上面是A，目标层下面是E、F、G。使用三明治集成的具体步骤如下



混合的增量式（三明治）集成 (3)

- 对目标层上面一层使用由顶向下集成策略，测试**A**，用桩模块代替**B、C、D**
- 再用实际模块**B、C、D**替换桩模块，用自顶向下集成策略与目标层上面的模块**A**集成，测试(**A, B, C, D**)，用桩模块代替**B**的下属模块**E**、**F**和**D**的下属模块**G**
- 对目标层下面的模块使用自底向上集成策略，测试底层模块**E、F、G**，使用驱动模块代替上一层模块**B、D**

混合的增量式（三明治）集成 (4)

- 采用自底向上集成策略把目标层下面的模块与目标层集成，测试 (B, E, F) 和 (D, G) 用驱动模块代替A
- 把三层集成到一起，测试 (A, B, C, D, E, F)

混合的增量式（三明治）集成 (5)

- 优点是集合了由顶向下和自底向上的两种集成策略的优点，且对中间层能够尽早进行比较充分的测试；而且该策略的并行度比较高
- 缺点是中间层如果选取不恰当，可能会有比较大的驱动模块和桩模块的工作量

基于层次的集成 (1)

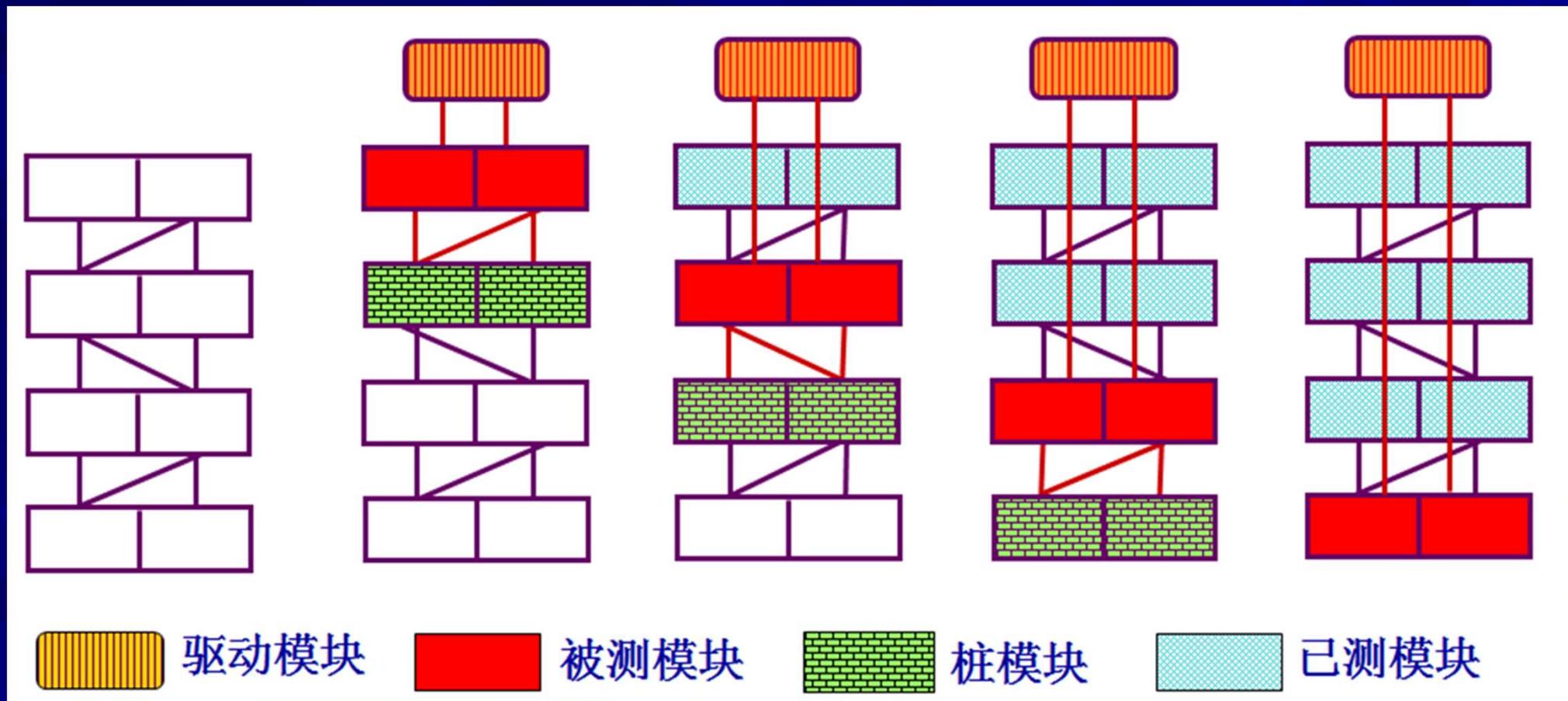
- 分层集成是针对具有层次式体系结构的应用软件使用的一种集成策略
- 分层集成的具体步骤如下
 - 划分系统的层次
 - 确定每个层次内部的集成策略
 - 确定层次间的集成策略

基于层次的集成（2）

- 层次内部和层次之间的集成策略可以使用一次性集成、自顶向下集成、自底向上集成和三明治集成中的任何一种策略
- 这种集成策略的优点和缺点与其使用的层次间集成测试策略类似。它主要适用于有明显线性层次关系的软件系统
- 下图给出了一个分层集成的示意图。图中各层之间采用了自顶向下的集成策略

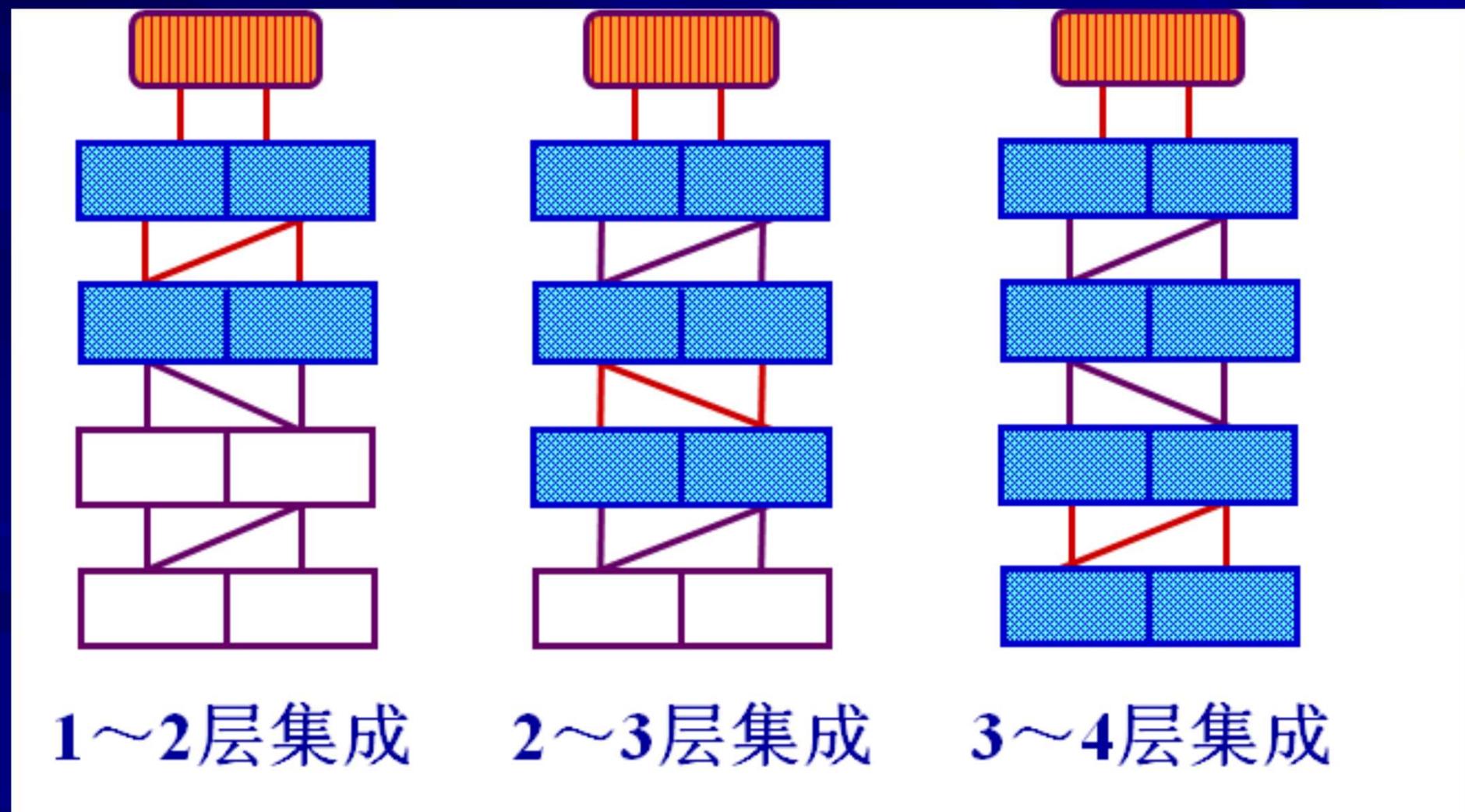
基于层次的集成 (3)

■ 层次内的集成



基于层次的集成 (4)

■ 层次间的集成



如何安排白盒测试

■ 集成测试

- 集成阶段，依据概要设计说明书
- 静态结构分析
- 代码质量度量
- 功能确认与接口分析
- 覆盖率分析
- 性能分析
- 内存分析

3.2.6.3 系统测试

- 系统测试的定义与目标
- 系统测试环境
- 如何安排白盒测试

系统测试的定义与目标

- 系统测试的目标在于通过与系统的需求规格说明进行比较，检查软件是否存在与系统规格不符合或与之矛盾的地方，以验证软件系统的功能和性能等满足规格说明所指定的要求
- 因此，测试设计人员应该主要根据需求规格说明来设计系统测试的测试用例

系统测试环境（1）

- 系统测试环境构建得是否合理、稳定和具有代表性，将直接影响到系统测试结果的真实性和正确性。部署合理的测试环境是达到测试目标的前提条件

系统测试环境（2）

- 不同（版本）的操作系统、不同（版本）的数据库，不同（版本）的网络服务器、应用服务器，再加上不同的系统架构等的组合，使得要构建的系统测试环境多种多样
- 测试人员可以通过构建系统测试环境库的方式来实现系统测试环境的复用，节省宝贵的测试时间

如何安排白盒测试

■ 系统测试

- 与软件所属的系统对接阶段，依据需求规格说明书
- 静态结构分析
- 功能确认与接口分析
- 覆盖率分析
- 性能分析
- 内存分析

3.2.6.4 验收测试

■ 如何安排白盒测试

- 验收阶段，依据软件任务书
- 以黑盒测试为主，根据需要辅以白盒测试

3.2.6.5 白盒测试综合策略（1）

- 在测试中，应尽量先用工具进行静态结构分析
- 测试中可采取先静态后动态的组合方式：先进行静态结构分析、代码检查和静态质量度量，再进行覆盖率测试
- 利用静态分析的结果作为引导，通过代码检查和动态测试的方式对静态结构分析结果进行进一步的确认，使测试工作更为有效

3.2.6.5 白盒测试综合策略（2）

- 覆盖率测试是白盒测试的重点，一般可使用基本路径测试法达到语句覆盖标准；对于软件的重点模块，应使用多种覆盖标准衡量代码的覆盖率

3.2.6.5 白盒测试综合策略（3）

- 在不同的测试阶段，测试的侧重点不同：在单元测试阶段，以代码检查、逻辑覆盖为主；在集成测试阶段，需要增加静态结构分析、静态质量度量；在系统测试阶段，应根据黑盒测试的结果，采取相应的白盒测试

3.2.7 白盒测试用例设计技术

- 3.2.7.1 静态白盒测试
- 3.2.7.2 动态白盒测试

3.2.7.1 静态白盒测试

■ 类别

- 代码检查（人工+工具）
- 静态结构分析（主要由软件工具自动进行）
- 软件质量度量（主要由软件工具自动进行）

课堂练习

```
main( )
{
int a[10], i, j, tmp;
for(i=0; i <=10; i++)
scanf("%d", &a[i]);
for(i=0; i <=10; i++)
for(j=i+1; j <= 10; j++)
if ( a[i] < a[j] ) {
a[i] = a[j];
}
for(i=0; i<=10; i++)
printf("%d ", a[i]);
}
```

参考答案

■ 数组的排序

1. 书写格式比较差，应有所缩进
2. 应有注释
3. {}应另起一行
4. 函数应有类型及**return, void**
5. 数组越界
6. 算法执行效率低，为n的平方
7. 气泡排序可降低复杂度
8. **tmp**变量应起到传递作用，否则数组内容被覆盖
9. **scanf, printf**应来自头文件，编译器不同

Thank You