软件工程概论
Software Engineering

刘伟

liuwei@xidian.edu.cn

88204608

CH8. Testing the programs

Content

- Software Faults and Failures
- Testing Issues
- Unit Testing
- Integration Testing
- Testing Object-Oriented System
- Testing Planning
- Automated Testing Tools
- When to Stop Testing

8.1 Software Faults and Failures

How does software fail?

- Wrong requirement: not what the customer wants
- Missing requirement
- Requirement impossible to implement
- Faulty design
- Faulty code
- Improperly implemented design

8.1 Software Faults and Failures

Terminology术语

- Fault identification（错误确定）: what fault caused the failure判定什么错误导致故障?
- Fault correction（错误纠正）: change the system to correct fault改动系统以修正错误
- Fault removal（错误去除）: take out the fault去除错误

8.1 Software Faults and Failures

Types of faults

- Algorithmic fault算法错误
- Syntax fault语法错误
- Computation and precision fault计算和精度错误
- Documentation fault文档错误
- Stress or overload fault强度或过载错误
- Capacity or boundary fault能力或边界错误
- Timing or coordination fault计时或协调错误
- Throughput or performance fault吞吐量或性能错误
- Recovery fault恢复错误
- Hardware and system software fault硬件和系统错误
- Standards and procedures fault标准和规程错误

8.1 Software Faults and Failures

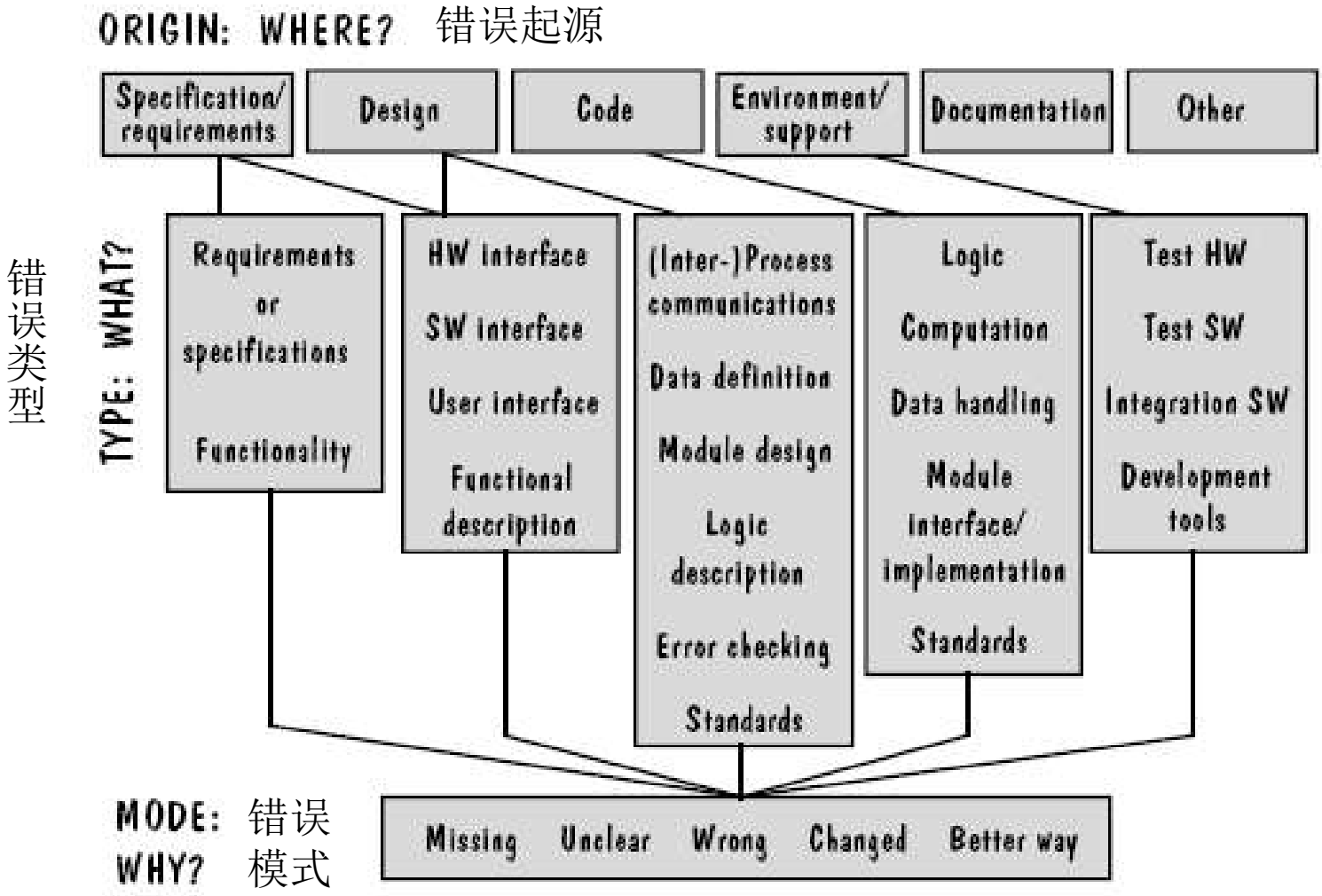
IBM 正交故障分类

Table 8.1. IBM orthogonal defect classification

Fault type错误类型	*Meaning含义*
Function功能	Fault that affects capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure
Interface接口	Fault in interacting with other components or drivers via calls, macros, control blocks or parameter lists
Checking检查	Fault in program logic that fails to validate data and values properly before they are used
Assignment赋值	Fault in data structure or code block initialization.
Timing/serialization 计时/串行	Fault that involves timing of shared and real-time resources
Build/package/merge 构建/打包/合并	Fault that occurs because of problems in repositories, management changes, or version control
Documentation文档	Fault that affects publications and maintenance notes
Algorithm算法	Fault involving efficiency or correctness of algorithm or data structure but not design

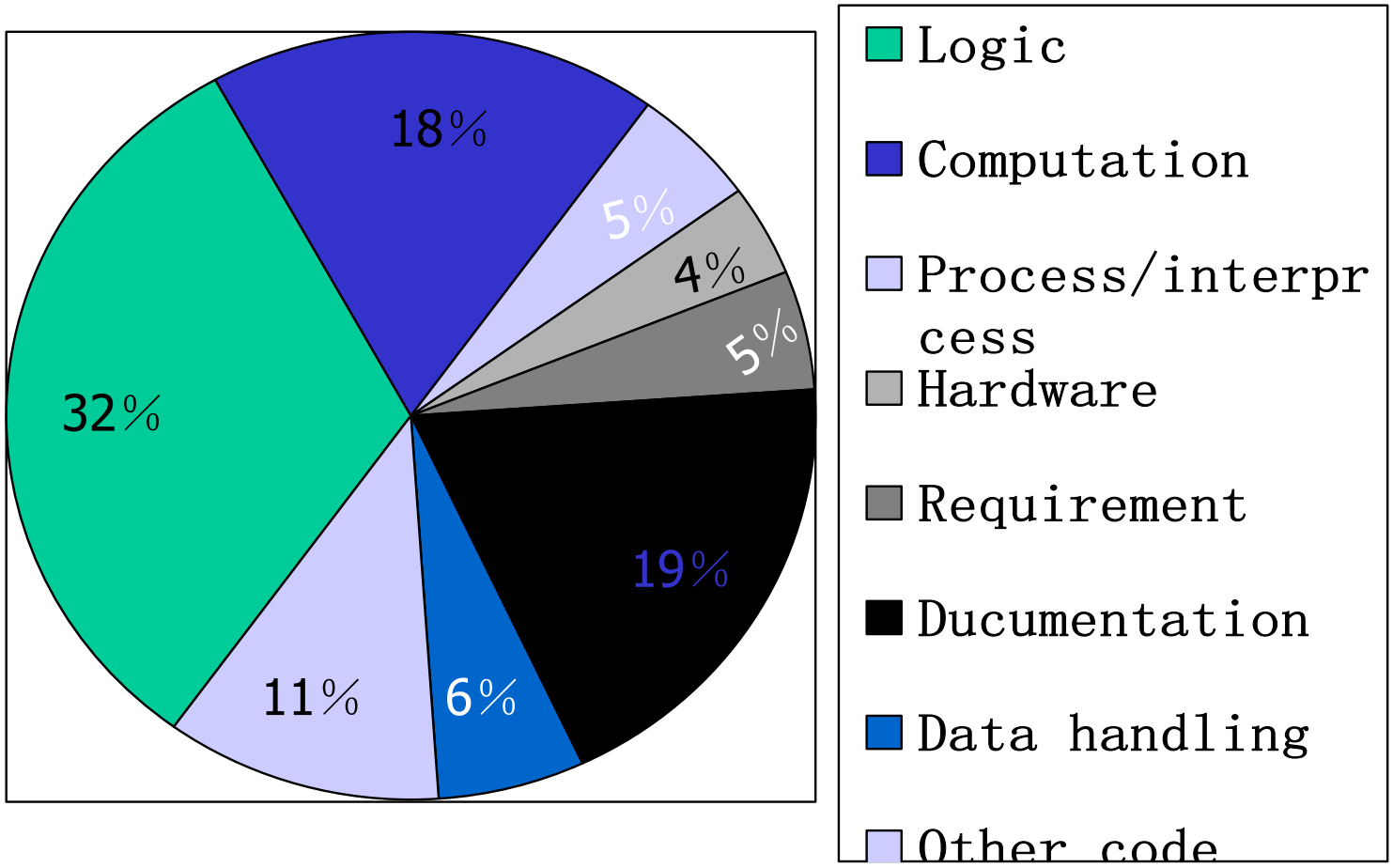
8.1 Software Faults and Failures

HP Fault Classification



8.1 Software Faults and Failures

Faults for One HP's Division Grady 1997



8.2 Testing Issues

- Test organization测试的组成
- Attitudes toward testing测试态度
- Who performs the tests由谁进行测试
- Views of the test objects对测试对象的看法

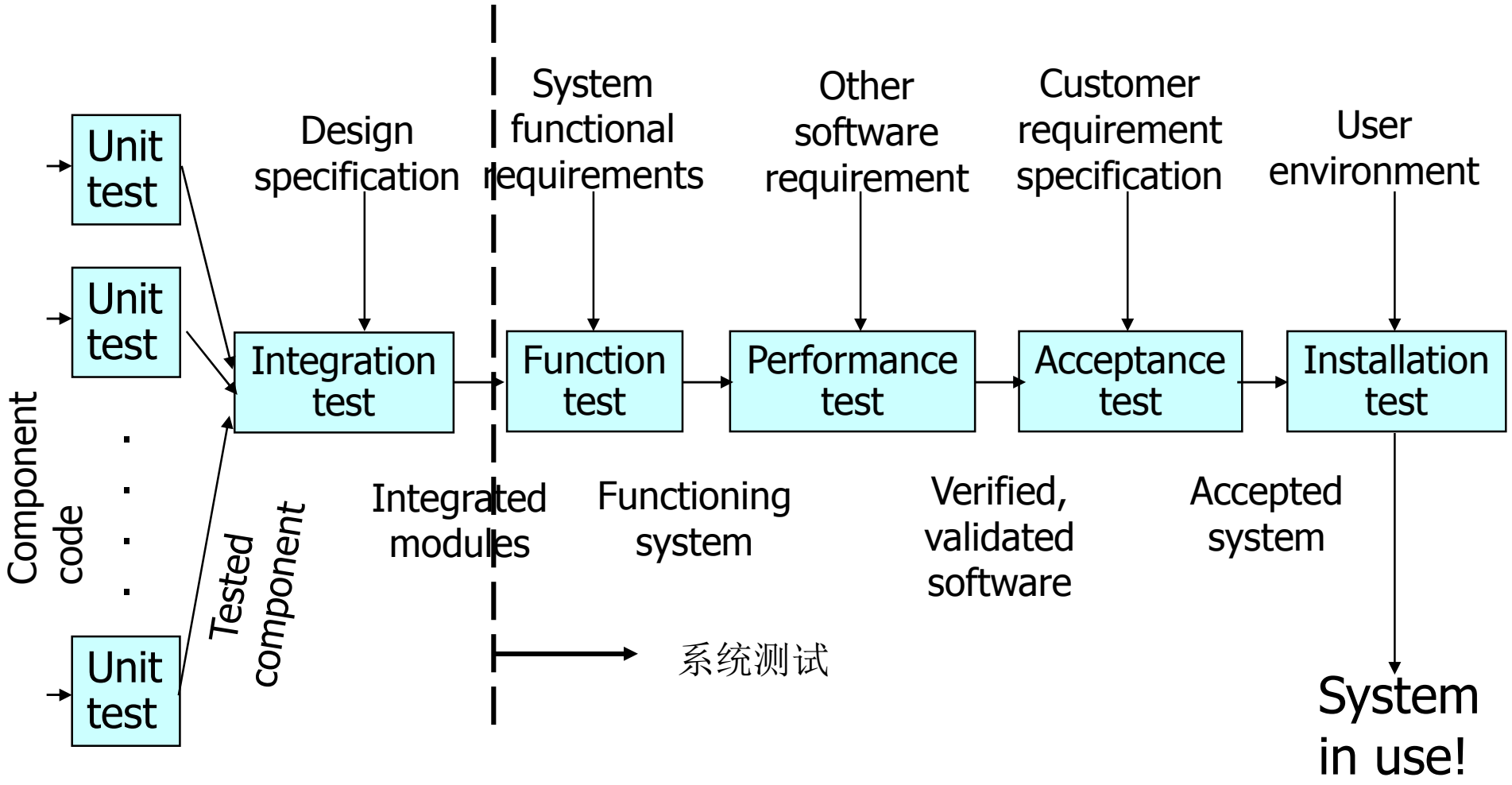
8.2 Testing Issues

Several Test Stages for large system

- Module test, component test, unit test
- Integration test
- Function test
- Performance test
- Acceptance test
- Installation test

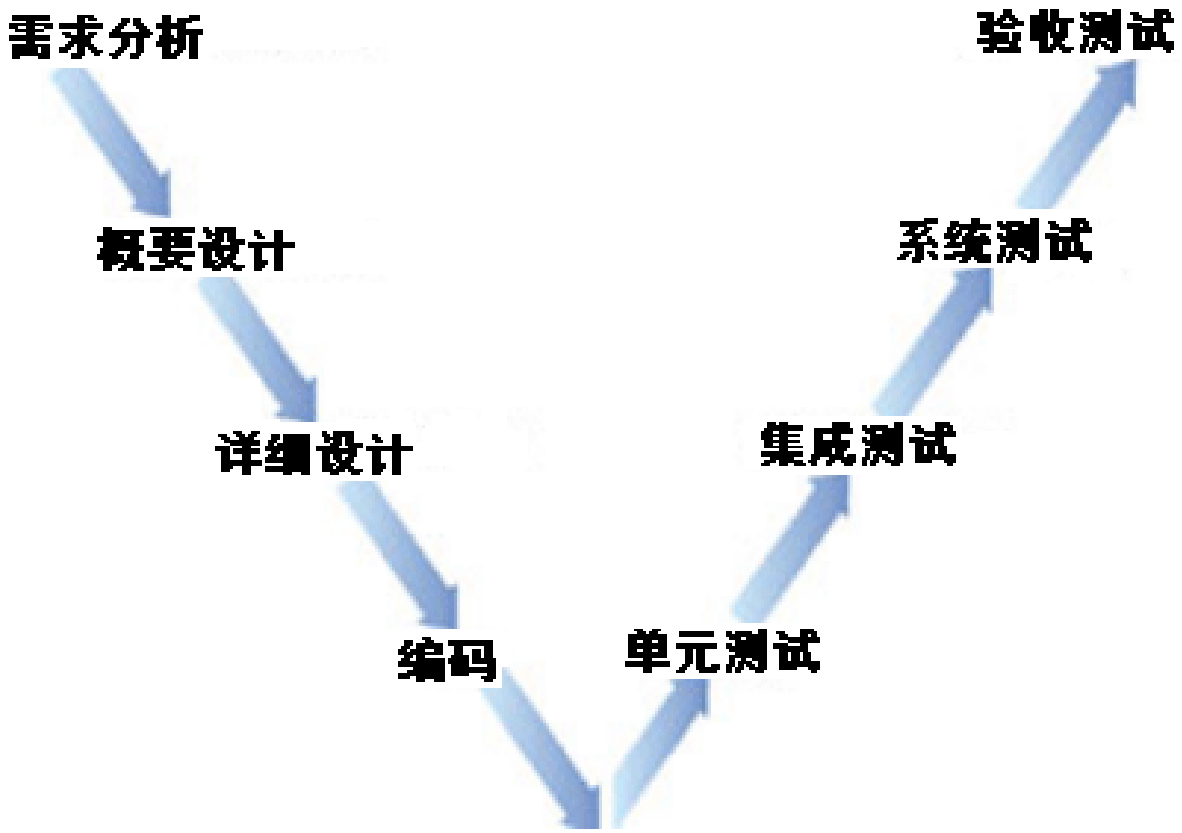
8.2 Testing Issues

Testing steps



8.2 Testing Issues

测试V模型



8.2 Testing Issues

测试定义

- 测试是为了发现程序中的错误而执行程序的过程
- 好的测试方案是极可能发现迄今为止尚未发现的错误的测试方案；
- 成功的测试是发现了迄今为止尚未发现的错误的测试 。

8.2 Testing Issues

测试的意义和几点说明

- 至今为止，测试仍然是软件质量保证的最重要手段。
- 测试是证实软件需求说明的功能是否实现，是否达到预期的指标的最有效手段。
- 软件测试耗时费力，追求用最小的测试代价获得最大的测试效果。
- 测试是为了发现错误，不是为了证明程序无错误。
- 测试不能证明程序中没有错误。
- 测试的可信度（dependability)问题

8.2 Testing Issues

Testing Principles测试指导原则1

- 所有的测试都应追溯到用户需求，从用户角度看，最严重的错误是不能满足用户需求。
- 制定测试计划，并严格执行，排除随意性。测试计划在需求分析阶段就开始了，详细的测试用例在设计阶段确定。
- Pareto原则：所发现错误的80%很可能源于程序模块的20%中。
- 测试应当从“小规模”开始，逐步转向“大规模”
- 穷举测试是不可能的（Exhaustive testing）。
- 由独立的第三方或专门的测试小组进行独立测试。

8.2 Testing Issues

Testing Principles测试指导原则2

- 测试用例由输入数据和相应的预期输出组成。
- 测试用例不仅选用合理的输入数据，还要选择不合理的。
- 不仅检查程序是否做了应该做的事，还应该检查是否不应该做的。
- 长期保留测试用例，以便进行回归测试和维护。

8.2 Testing Issues

测试技术的分类

- 静态测试
  - 代码会审 code inspection
  - 走查 walk-through
  - 办公桌检查 desk checking
  - 例如：Yourdon 结构化走通、IBM的Fagan检查。
- 动态测试
  - 黑盒测试
  - 白盒测试
  - 穷举和选择测试。

8.2 Testing Issues

静态测试

- 定义 - 人工方式进行的代码复审。
- 目的 - 检查程序静态结构，找出编译不能发现的错误和人的主观认识上偏差。
- 范围 - 需求定义、设计文档、源代码
- 特点
  - 研究表明，对于某些类型的错误，静态测试更有效 。
  - 经验表明，组织良好的代码复审可以发现程序中30%到70%的编码和逻辑设计错误。
  - 不存在错误定位问题。

8.2 Testing Issues

动态测试

- 定义 - 在设定的测试数据上执行被测试程序的过程
- 目的 - 通过执行程序代码动态地验证结果的正确性
- 三个过程：
  - 设计测试用例（test case)
  - 执行被测试程序
  - 分析执行结果并发现错误
- 三个要素：程序、测试数据、需求定义

8.2 Testing Issues

黑盒测试（功能测试）

- 定义：已知产品应该具有的功能，通过测试检验其每个功能是否都能够正常使用。又称功能测试
- 用途：把程序看成一个黑盒子，仅仅考虑输入和输出的对应关系和程序接口，完全不考虑它的内部结构和处理过程。一般用于综合测试、系统测试等

8.2 Testing Issues

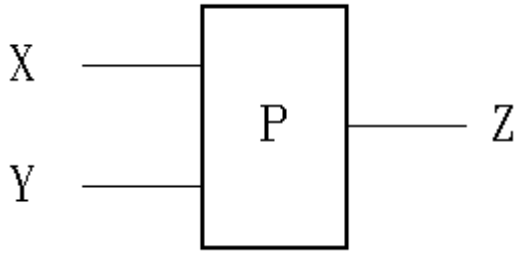
黑盒测试（功能测试）

- 黑盒测试方法是在程序接口上进行测试，主要发现：
  - 是否有不正确或遗漏了的功能?
  - 在接口上，输入能否正确地接受? 能否输出正确的结果?
  - 是否有数据结构错误或外部信息（如数据文件）访问错误?
  - 性能上是否能够满足要求?
  - 是否有初始化或终止性错误?

8.2 Testing Issues 黑盒测试（功能测试）

- 用黑盒测试发现程序中的错误，必须在所有可能的输入条件和输出条件中确定测试数据，来检查程序是否都能产生正确的输出，但这是不可能的。
- 举例：假设一个程序P有输入量X和Y及输出量Z，在字长为32位的计算机上运行。若X、Y取整数，按黑盒方法进行穷举测试：
  - 可能采用的测试数据组：

$$2^{32} \times 2^{32} = 2^{64}$$



  - 如果测试一组数据需要1毫秒，一年工作365× 24小时，完成所有测试需5亿年。

8.2 Testing Issues

白盒测试（结构测试）

- 定义
  - 已知产品内部的工作过程，通过测试检验产品内部动作是否都能按照需求定义的规定正常使用。
- 用途
  - 必须完全了解程序的内部结构和处理过程，才能按照程序内部的逻辑测试，以检验程序中每条路径是否正确，因此 一般用于规模较小的程序和单元测试.

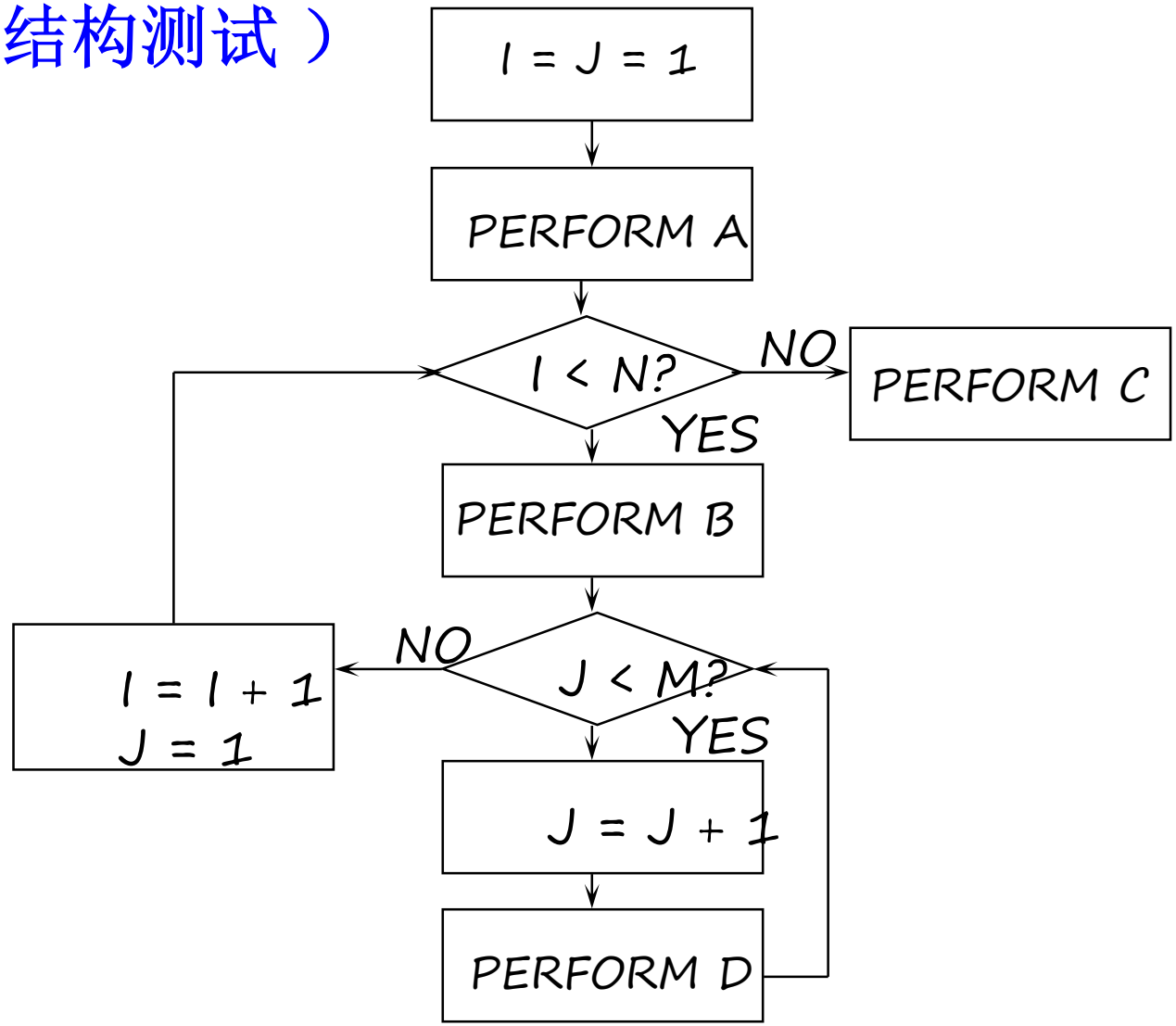
8.2 Testing Issues

白盒测试（结构测试）

- 使用白盒测试方法，主要对程序模块进行如下检查：
  - 对程序模块的所有独立的执行路径至少测试一次；
  - 对所有逻辑判定，取“真”与取“假”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等。

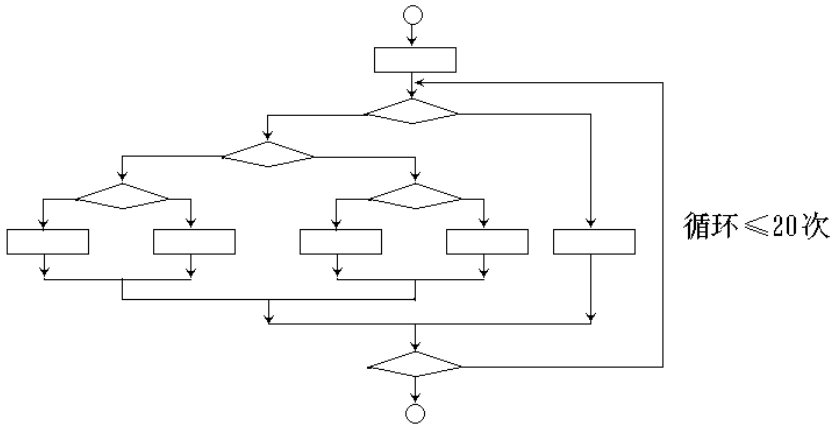
8.2 Testing Issues

白盒测试（结构测试）



8.2 Testing Issues（白盒测试（结构测试））

- 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。
- 举例：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，
    要把所有路径测试完，需3170年。



8.2 Testing Issues

穷举测试 (Exhaustive testing)

- 定义
  - 包含所有可能情况的测试。
  - 对于黑盒测试，必须对所有输入数据的各种可能值的排列组合都进行测试；
  - 对于白盒测试，程序中每条可能的路径在每种可能的输入数据下至少执行一次。
- 穷举测试是不可能的。
  - 例一：要对C编译系统进行黑盒穷举测试，一方面要编出所有能够想象出来的合法程序让它编译；另一方面又要编出一切不合法的程序，考察它能否指出程序的非法性质。显然，这两类（合法和不合法）程序的数量是无限的。

8.2 Testing Issues

选择测试

- 仅选择一些具有代表的、典型的测试用例，进行有限的测试。
- 以最少的测试用例发现最多的程序错误。

8.3 Unit Testing

- 单元测试主要评价模块的五个特性：
  - 模块接口
  - 局部数据类型
  - 重要的执行通路
  - 出错处理通路
  - 影响以上特性的边界条件
- 单元测试过程
  - 代码审查
  - 设计驱动软件（驱动模块driver）以代替上级模块
  - 设计存根软件 （桩模块stub）以代替下级模块

8.3 Unit Testing

Examining the Code代码检查

- Two types of code review两种代码评审方法
  - Code Walkthroughs代码走读
  - Code Inspections代码检查
- Success of Code Reviews代码复审的成功
  - P344（269）

典型的审查准备时间和会议时间

Table 8.2. Typical inspection preparation and meeting times.

Development artifact开发工作	*Preparation time准备时间*	*Meeting time会议时间*
Requirements document	25 pages per hour	12 pages per hour
Functional specification	45 pages per hour	15 pages per hour
Logic specification	50 pages per hour	20 pages per hour
Source code	150 lines of code per hour	75 lines of code per hour
User documents	35 pages per hour	20 pages per hour

发现活动中找到的错误

Table 8.3. Faults found during discovery activities.

Discovery activity发现活动	*Faults found per thousand lines of code每千行代码发现的错误*
Requirements review	2.5
Design review	5.0
Code inspection	10.0
Integration test	3.0
Acceptance test	2.0

8.3 Unit Testing

Proving code correct证明代码正确性

- Formal proof techniques形式证明技术
- Symbolic execution符号执行
- Automated theorem-proving自动定理证明

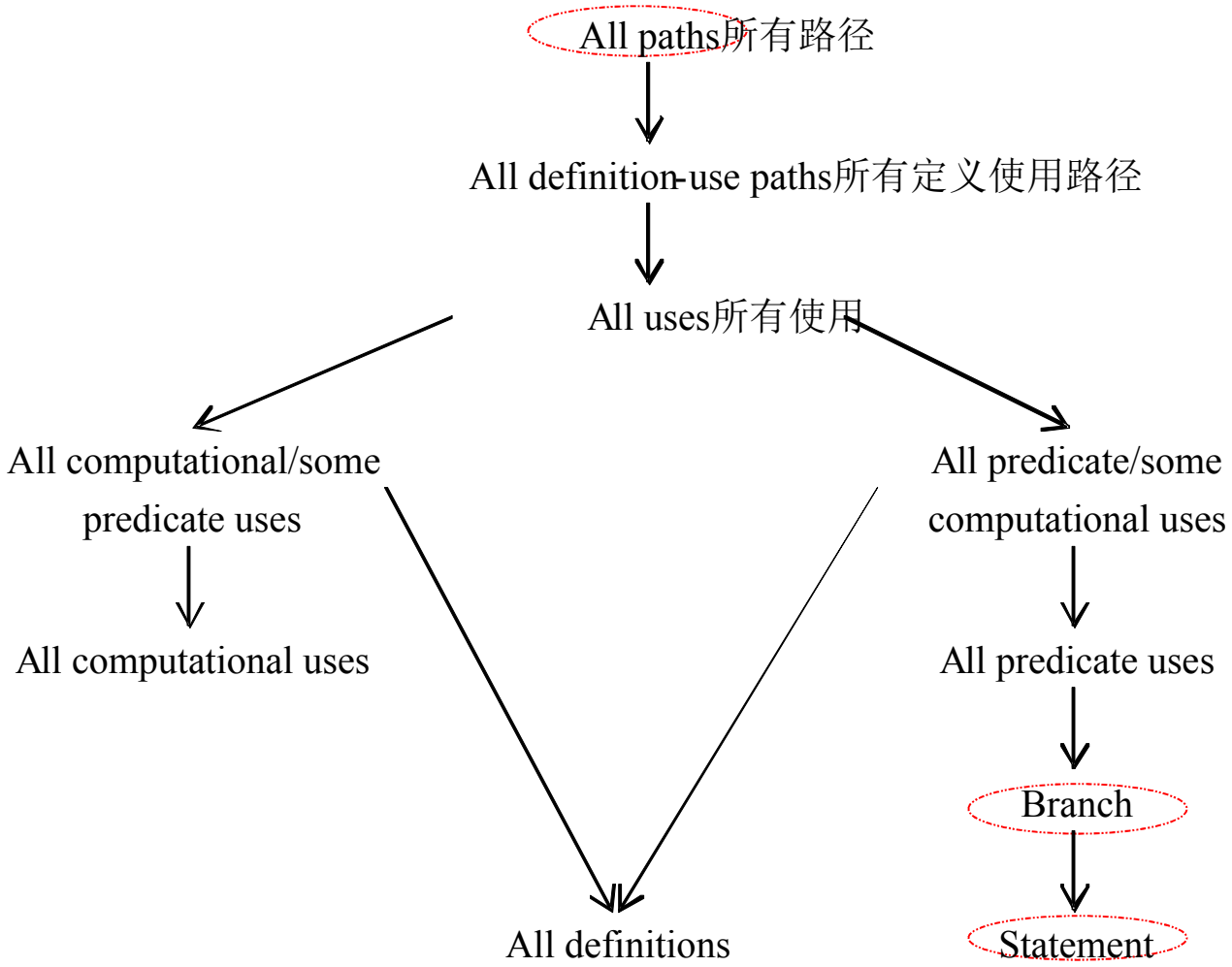
8.3 Unit Testing

Test thoroughness测试的彻底性

- Statement testing语句测试
- Branch testing分支测试
- Path testing路径测试
- Definition-use testing定义使用测试
- All-uses testing所有使用测试
- All-predicate-uses/some-computational-uses testing所有判定使用/某些计算使用的测试
- All-computational-uses/some-predicate-uses testing所有计算使用/某些判定使用的测试

8.3 Unit Testing

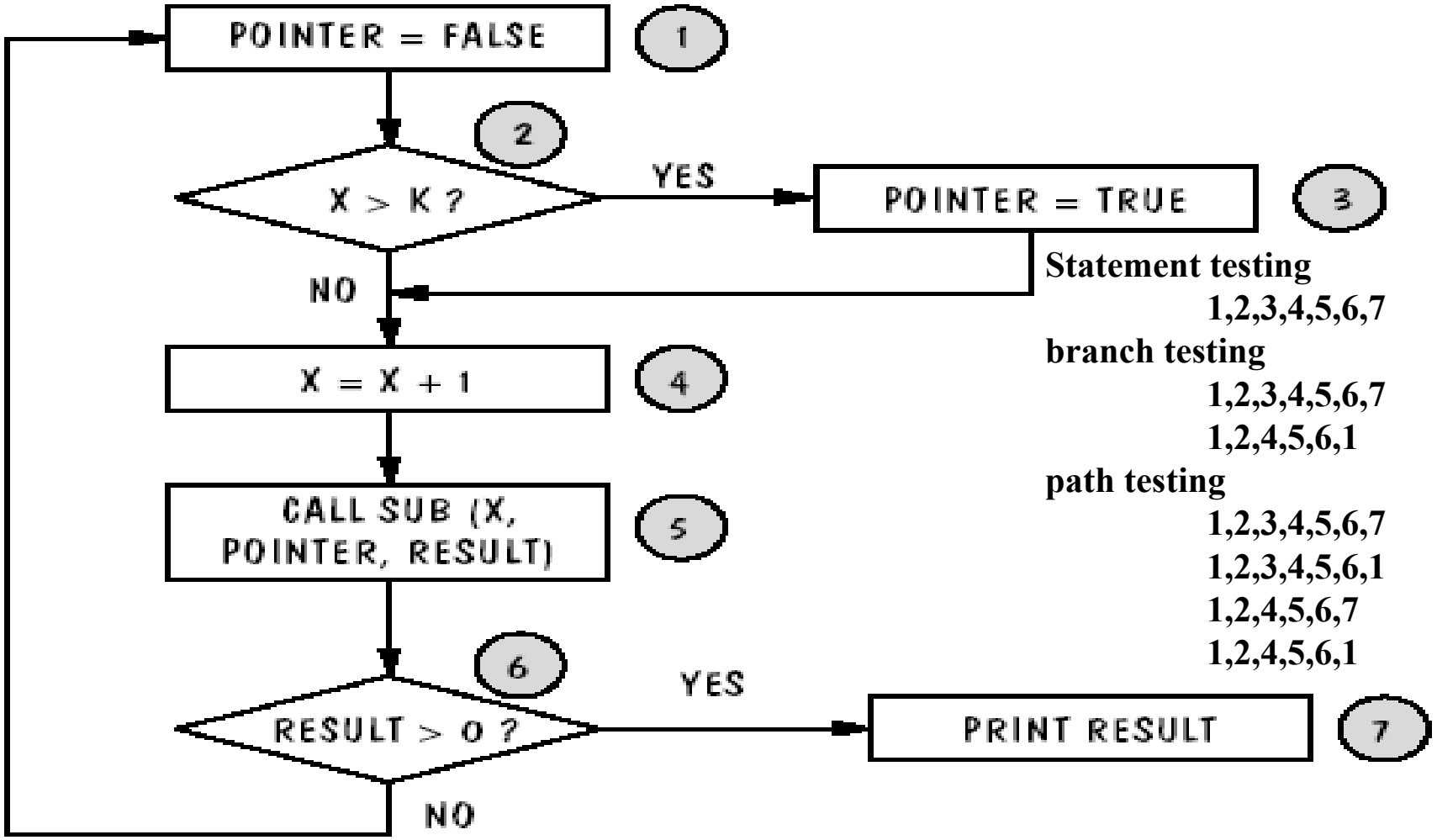
Relative strengths of test strategies



- Ntafos shows
  - random testing (not test strategies) found 79.5% faults
  - branch testing found 85.5%, and
  - all-uses testing found 90%
- Stronger strategy means more test cases

8.3 Unit Testing

Logic Flow逻辑流



Statement testing
1,2,3,4,5,6,7
branch testing
1,2,3,4,5,6,7
1,2,4,5,6,1
path testing
1,2,3,4,5,6,7
1,2,3,4,5,6,1
1,2,4,5,6,7
1,2,4,5,6,1

8.3 Unit Testing - Comparing techniques

由错误起源发现错误的百分比

Table 8.5. Fault discovery percentages by fault origin.

Discovery technique	*Requirements*	*Design*	*Coding*	*Documentation*
Prototyping	40	35	35	15
Requirements review	40	15	0	5
Design review	15	55	0	15
Code inspection	20	40	65	25
Unit testing	1	5	20	0

错误发现技术的效果

Table 8.6. Effectiveness of fault discovery techniques. (Jones 1991)

	Requirements faults	***Design faults***	***Code faults***	***Documentation faults***
Reviews *评审*	Fair	Excellent	Excellent	Good
Prototypes *原型化*	Good	Fair	Fair	Not applicable
Testing *测试*	Poor	Poor	Good	Fair
Correctness Proofs *正确性证明*	Poor	Poor	Fair	Fair

8.4 Integration testing集成测试

- Bottom-up自底向上
- Top-down自顶向下
- Big-bang一次性
- Sandwich testing三明治测试
- Modified top-down改进自顶向下
- Modified sandwich改进三明治

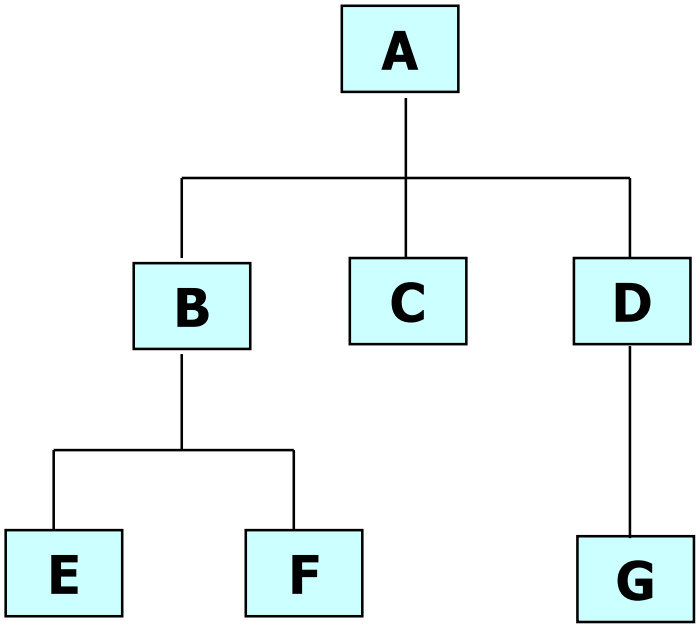
8.4 Integration Testing

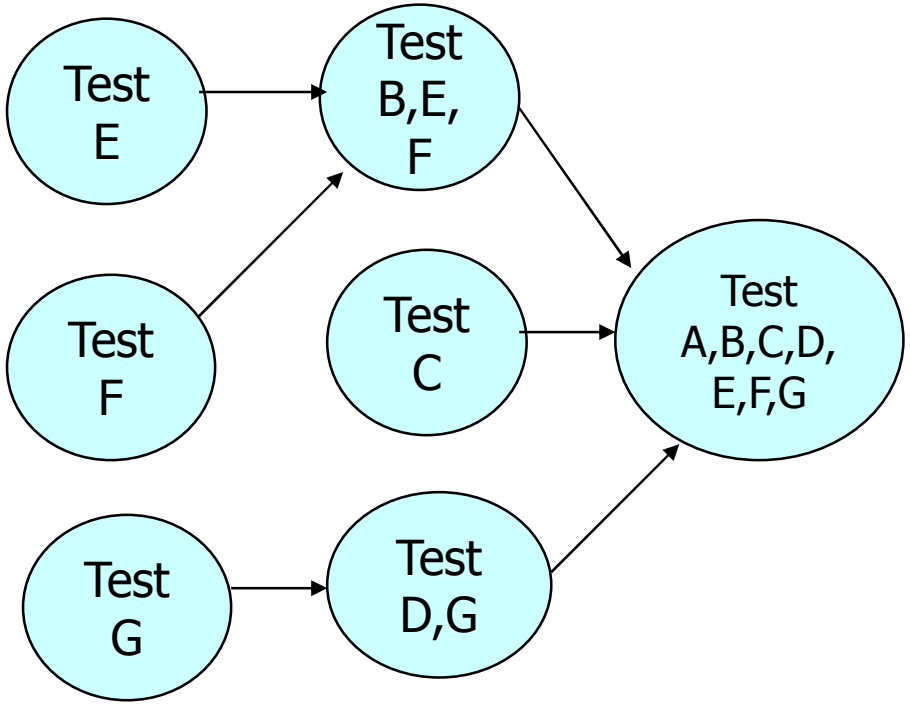
Bottom-up测试步骤

- 自底向上由一个叶模块开始，自底向上逐步添加新模块，组成程序的一个子系统或具有某一功能的模块族（群Cluster)
- 设计驱动程序，协调测试数据的输入和输出
- 测试
- 去掉驱动模块，沿软件结构自下而上移动，把有关的子系统结合，形成更大的子系统。

8.4 Integration Testing

Bottom-up测试示例





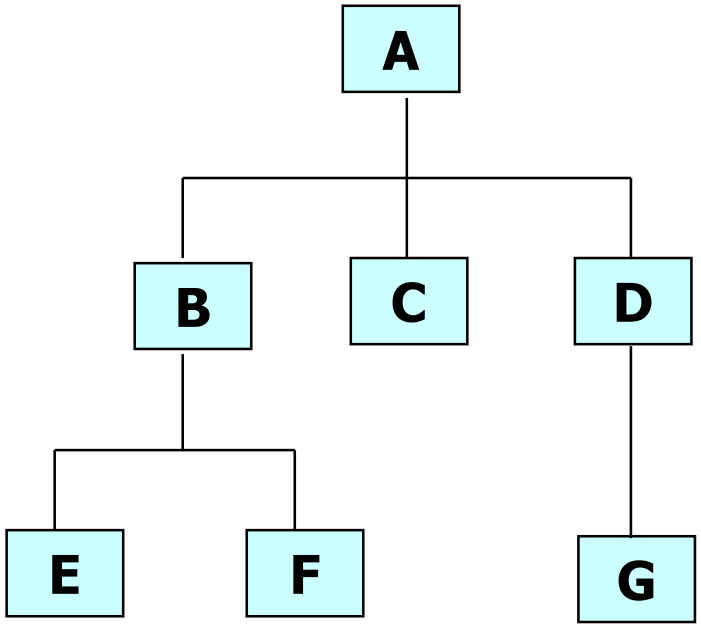
8.4 Integration Testing

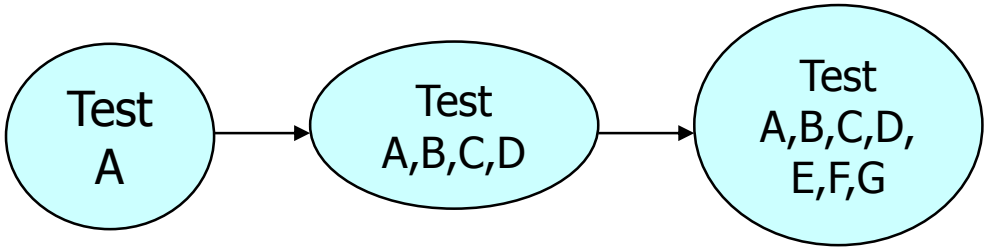
Top-down测试步骤

- 测试主控模块，由桩模块代替所有直属模块
- 根据所确定的组合策略增加一个模块，设计新的桩模块。
- 测试，如果在新的条件下发现新的错误，执行下一步；否则执行上一步。
- 回归测试（全部或部分地重复已做过的测试并返回）

8.4 Integration Testing

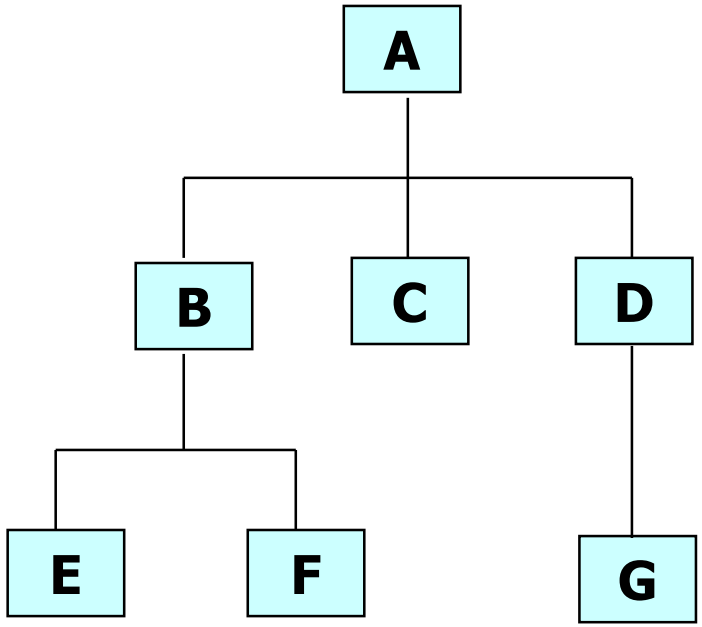
Top-down测试示例

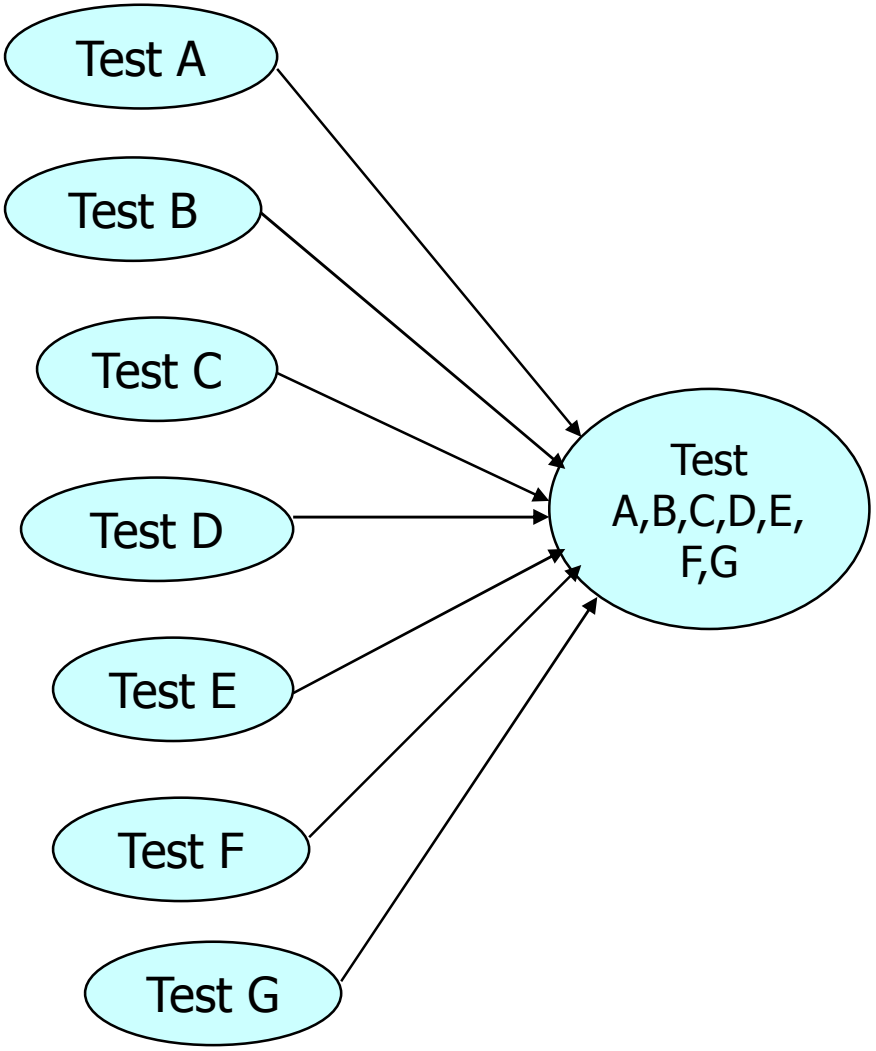




8.4 Integration Testing

Big-bang测试示例





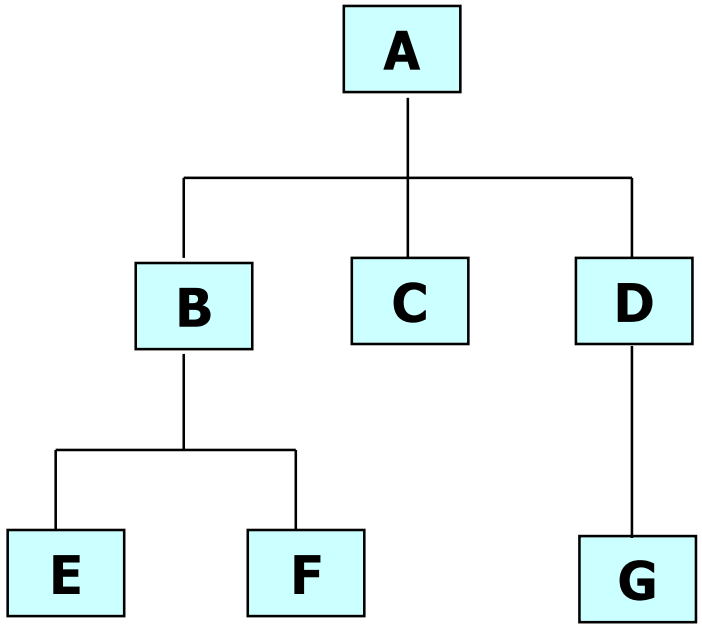
8.4 Integration Testing

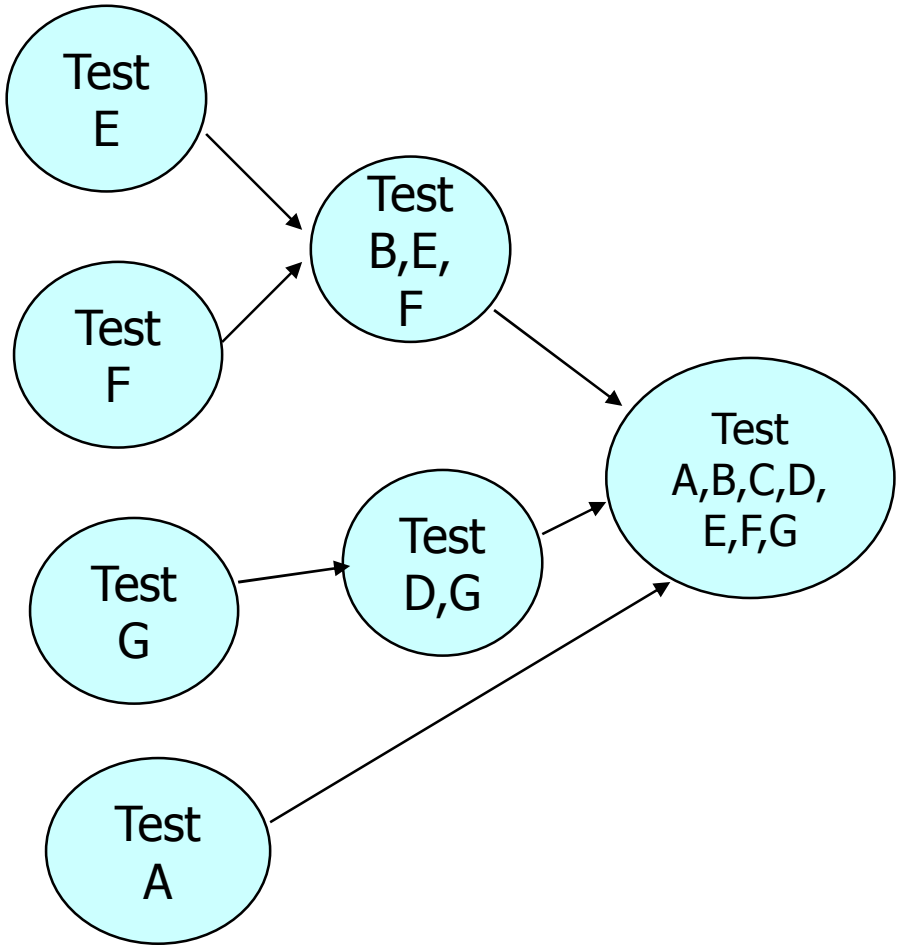
Sandwich testing测试步骤

- 对上层模块采用自顶向下，较早显示程序的总体轮廓。
- 目标层（中间层）的选择问题
- 对某些关键模块（如具有I/O功能的模块、功能重要或含有特殊算法的模块）或子系统采用自底向上组装和测试，以便容易地产生测试用例或减少重复测试次数。

8.4 Integration Testing

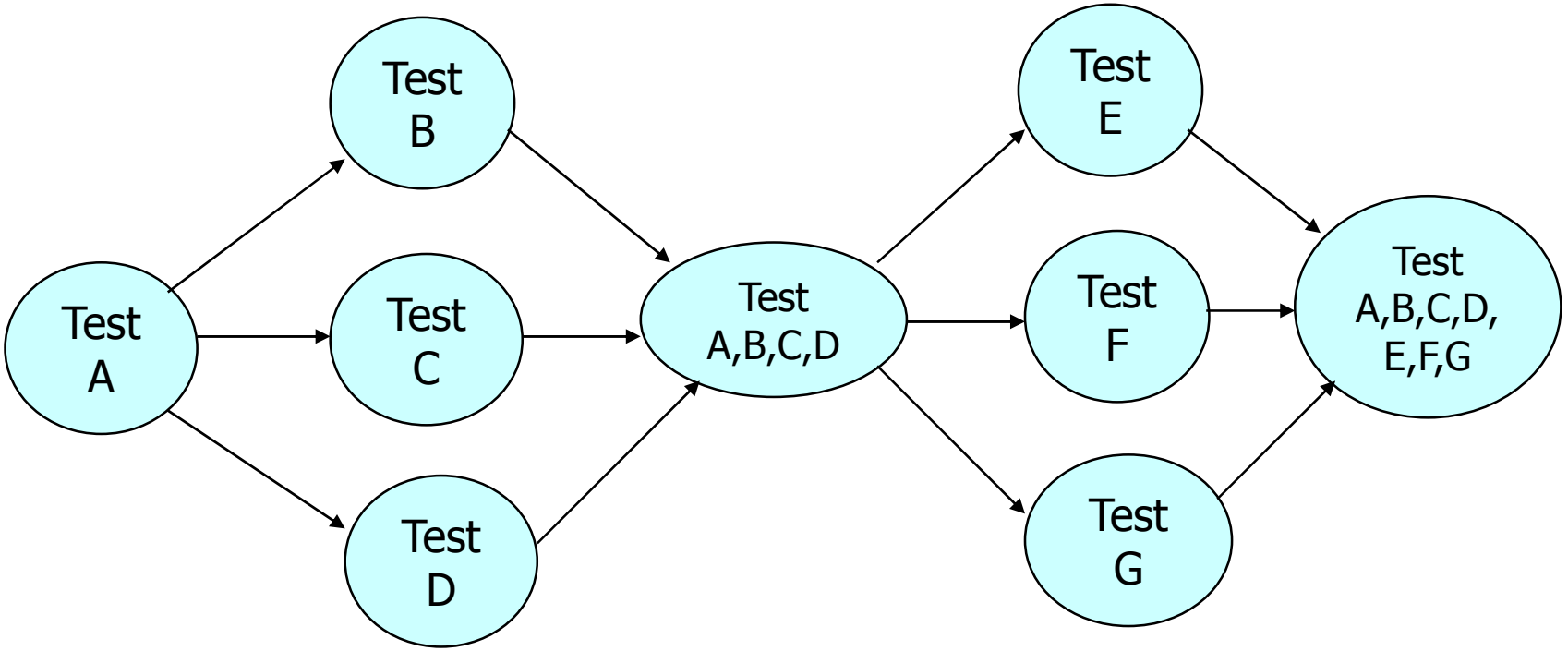
Sandwich集成策略示例





8.4 Integration Testing

Modified Top-down测试示例



Modified top-down测试

- 两种结合策略
  - 深度优先 - 先组装在软件结构的一条主控通路上的所有模块。
  - 宽度优先 - 沿软件结构水平地移动，把处于同一个控制层次上的所有模块组装起来。

8.4 Integration Testing

Modified Sandwich集成策略

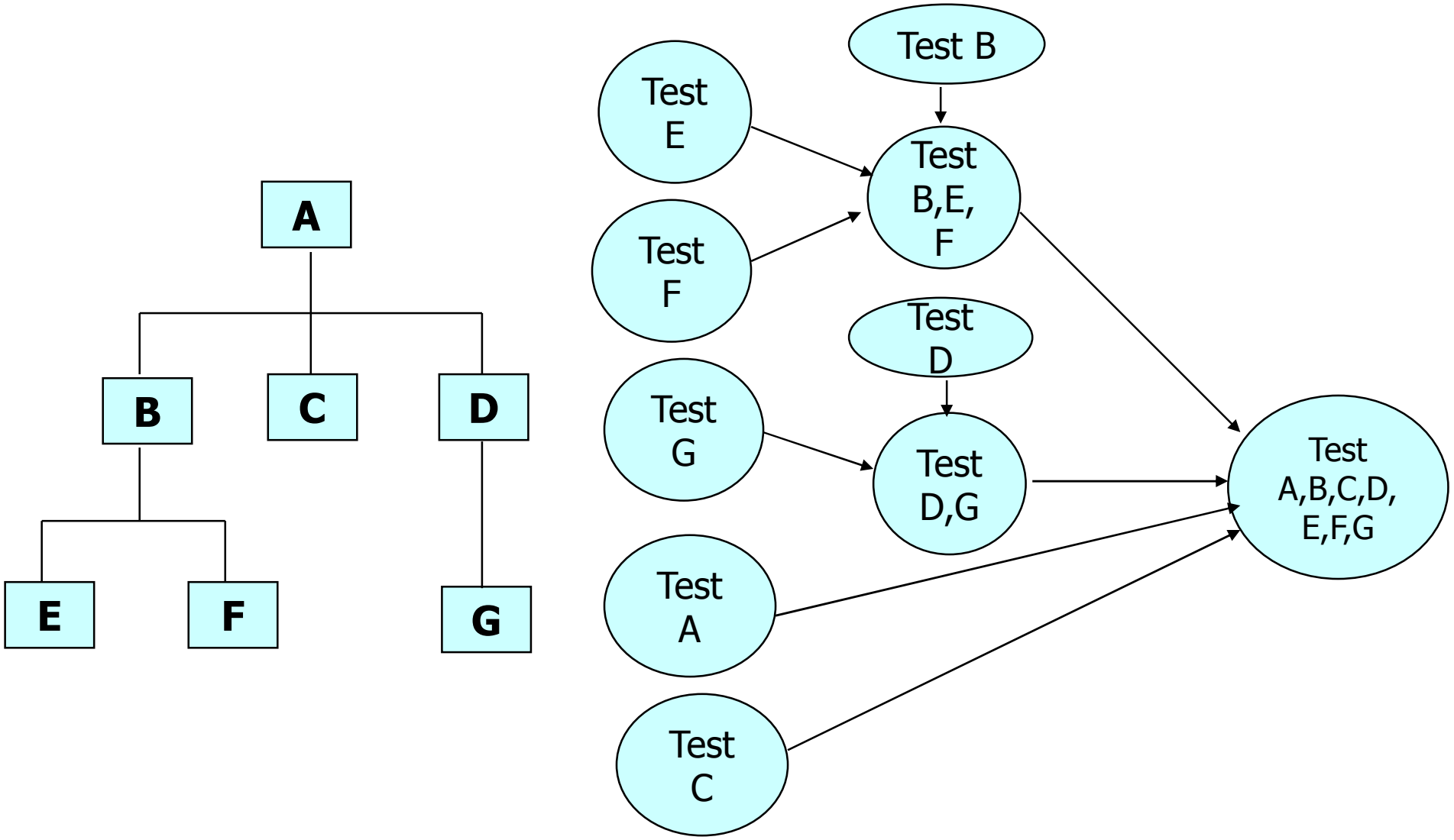


Table 8.7. Comparison of integration strategies

	Bottom-up	*Top-down*	*Modified top-down*	*Big-bang*	*Sandwich*	*Modified sandwich*
Integration	Early	Early	Early	Late	Early	Early
Time to basic working program	Late	Early	Early	Late	Early	Early
Component drivers needed	Yes	No	Yes	Yes	Yes	Yes
Stubs needed	No	Yes	Yes	Yes	Yes	Yes
Work parallelism at beginning	Medium	Low	Medium	High	Medium	High
Ability to test particular paths	Easy	Hard	Easy	Easy	Medium	Easy
Ability to plan and control sequence	Easy	Hard	Hard	Easy	Hard	Hard

8.4 Integration Testing

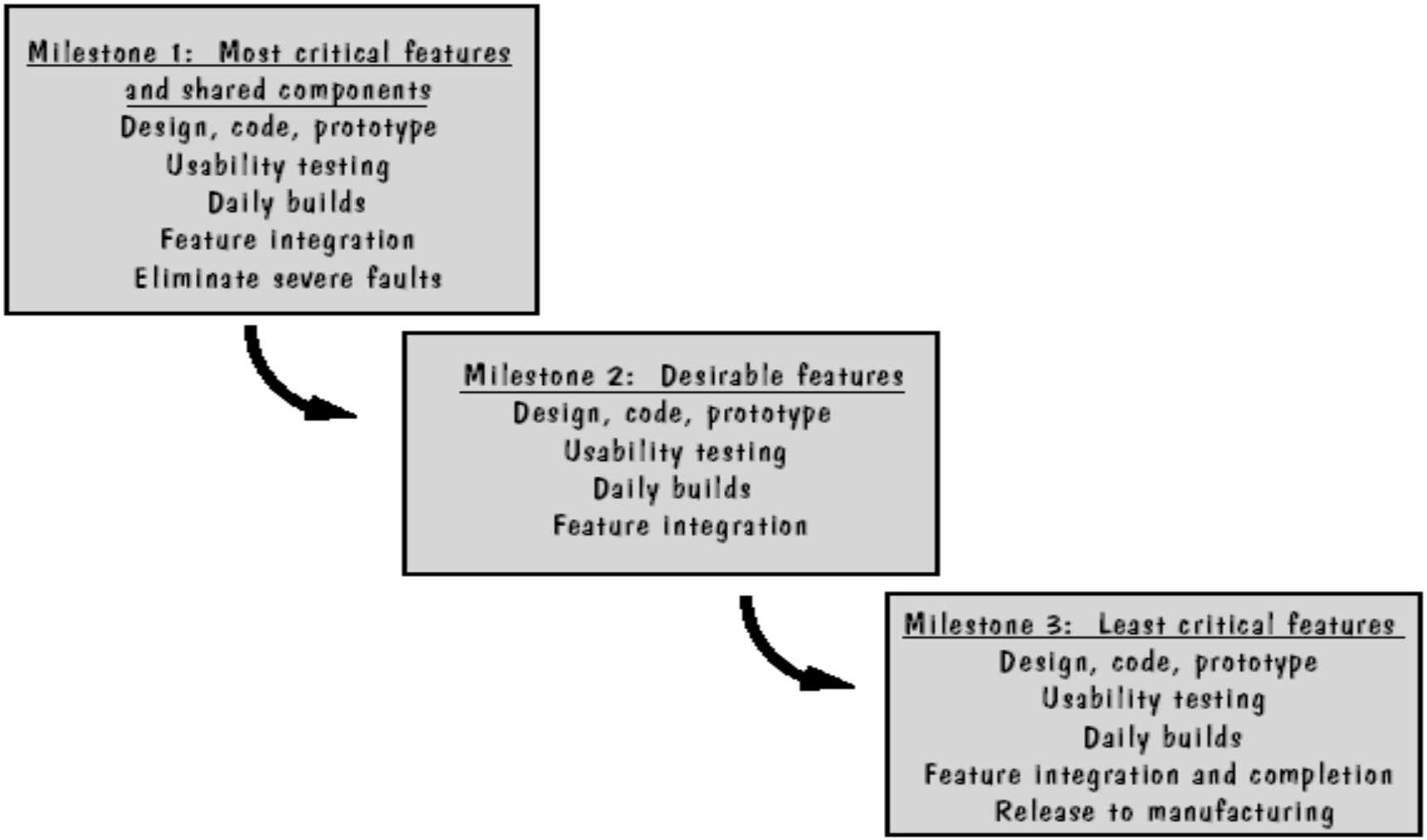
Builds At Microsoft微软的集成方法

Market-driven integration strategy市场驱动集成策略

- Work in teams
  - Team size – 3 to 8 developers3到8人一组
  - Different teams are responsible for different features不同的小组负责不同的特征
  - Each team is allowed to change the specification of features每个小组都可以改动说明的特征
- The process iterates among designing, building, testing components while involving customers in the testing process该过程在测试过程中包含顾客的同时，在设计、构建、测试组件之间迭代

8.4 Integration Testing

Microsoft Synch-and-stabilize approach

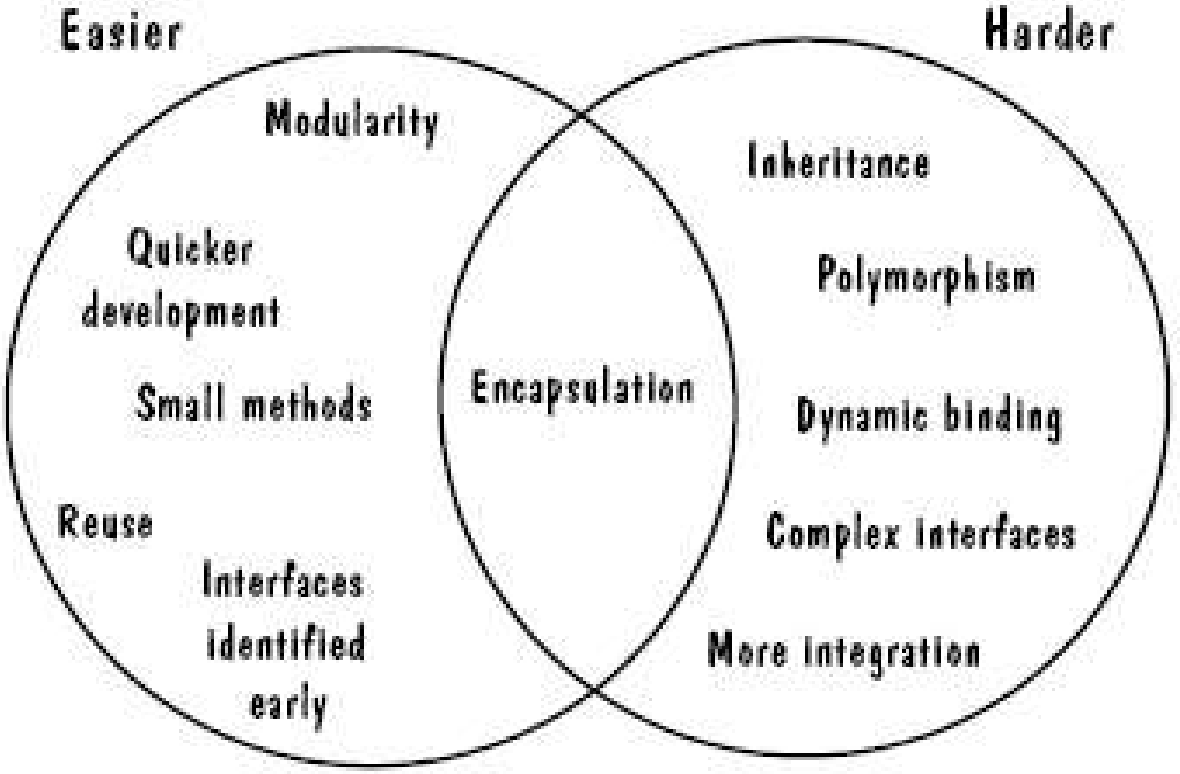


8.5 Guidelines to Test OO Code

- Objects might be missing if
  - You find asymmetric associations or generalization发现有不对称的关系或类
  - You find disparate attributes and operations on a class发现一个类中有完全无关的属性和操作
  - One class is playing two or more roles一个类承担两个或更多的职责
  - An operation has no good target class一个操作没有好的目标类
  - You find two associations with the same name and purpose发现两个关系有同样的名字和目的

8.5 Guidelines to Test OO Code

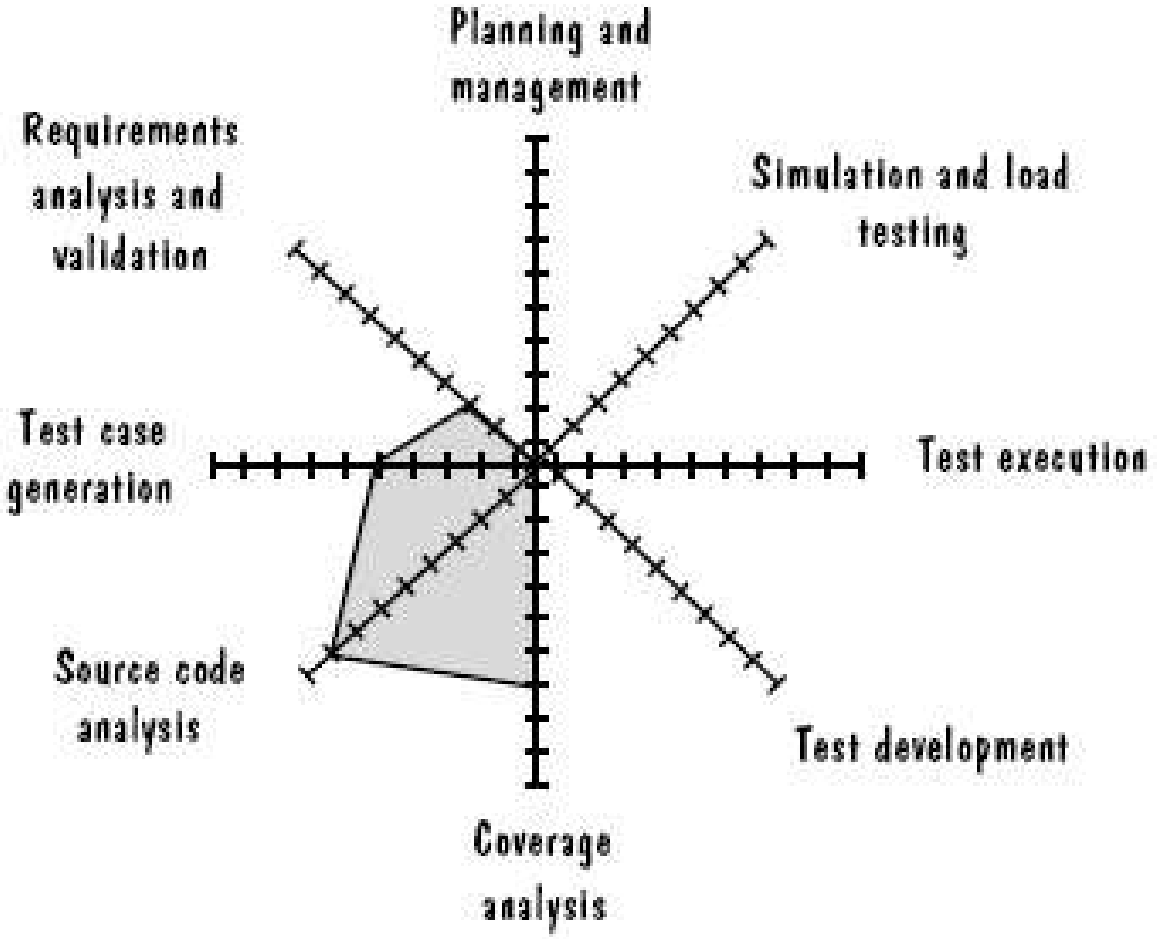
Differences between object-oriented and traditional testing



00测试中更容易和更困难的部分

8.5 Guidelines to Test OO Code

Differences between object-oriented and traditional testing



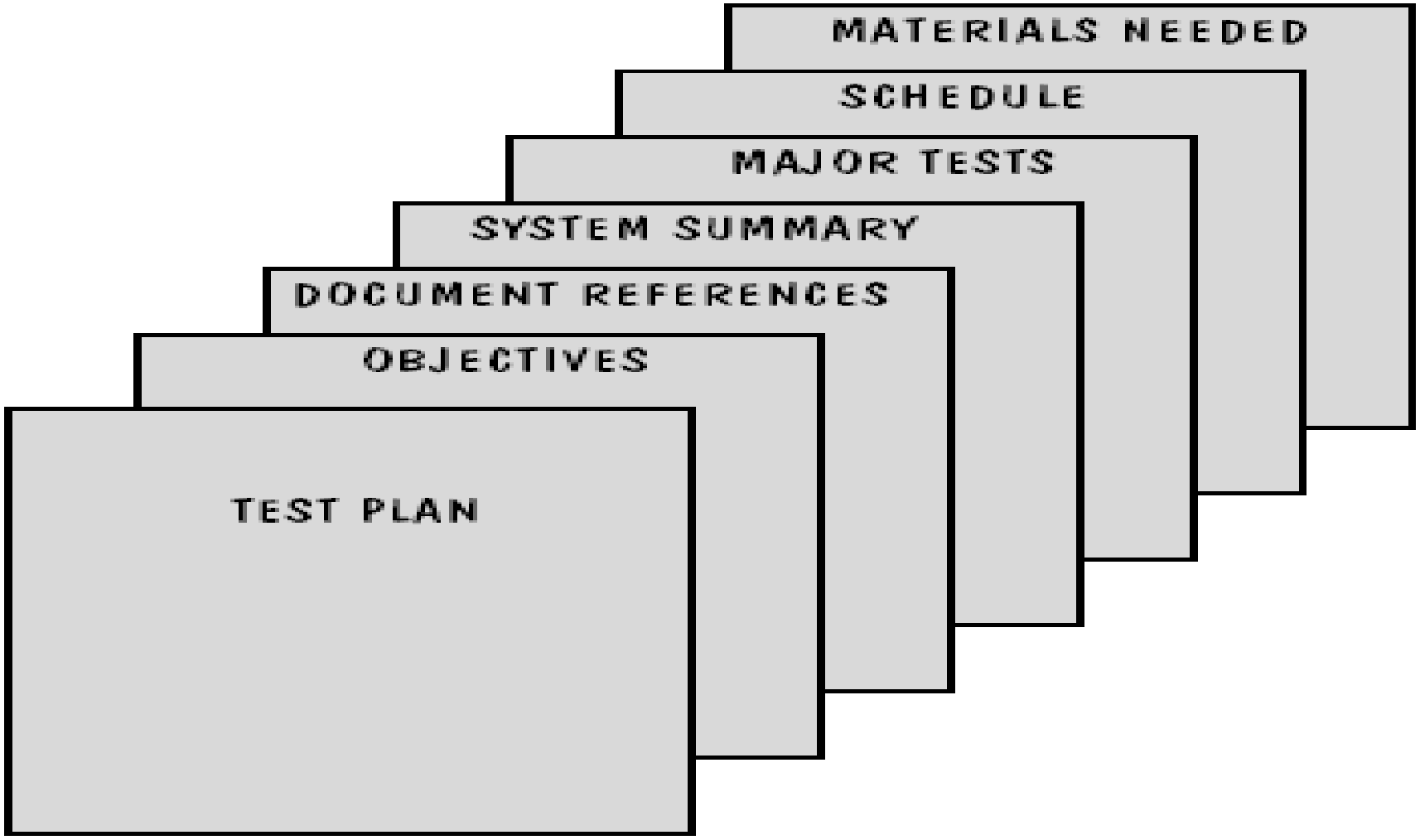
00测试不同于传统测试的主要方面

8.6 Test planning

- Establish test objectives制定测试目标
- Design test cases设计测试实例
- Write test cases编写测试实例
- Test test cases测试测试实例
- Execute tests执行测试
- Evaluate test results评价测试结果

8.5 Guidelines to Test OO Code

Parts of a test plan



8.6 Automated testing tools自动测试工具

- Code analysis代码分析
  - Static analysis静态分析
    - code analyzer代码分析器
    - structure checker结构检查器
    - data analyzer数据分析器
    - sequence checker序列检查器
  - Dynamic analysis动态分析
    - program monitor程序监视器
- Test execution测试执行
  - Capture and replay 获取和重放
  - Stubs and drivers存根和驱动
  - Automated testing environments自动测试环境
- Test case generators测试实例生成器

8.7 When to stop testing何时停止测试

- Coverage criteria覆盖标准（覆盖>85%的路径等）
- Fault seeding错误播种

$$\frac{\text{detected seeded faults}}{\text{total seeded faults}} = \frac{\text{detected non-seeded faults}}{\text{total non-seeded faults}}$$

- Confidence in the software软件中的置信度

$$C = \begin{cases} 1 & \text{if } n > N \\ S/(S\text{-}N\text{+}1) & \text{if } n \leq N \end{cases}$$