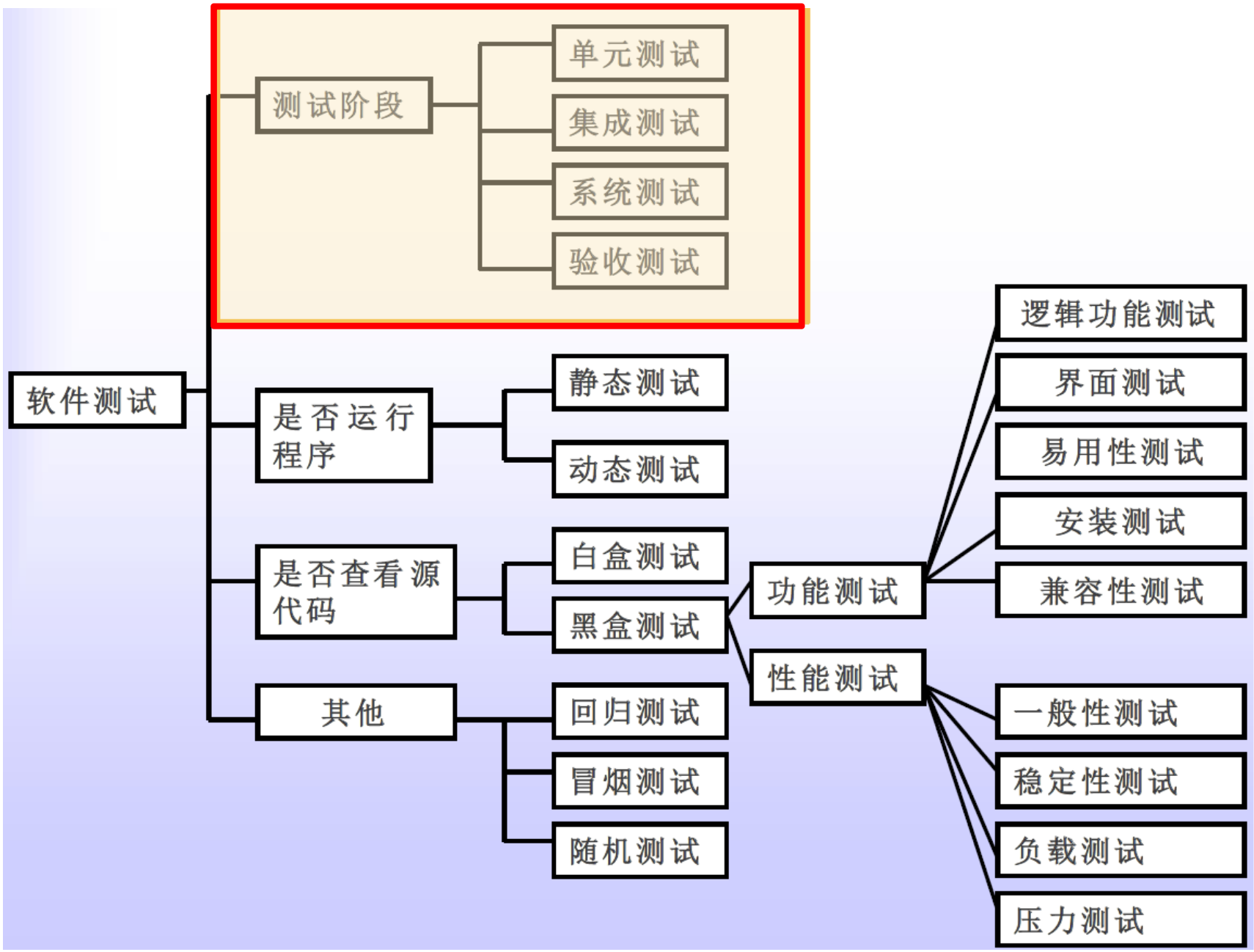
软件质量保证与测试

软件测试级别

软件测试相关名词


软件测试
测试阶段
单元测试
集成测试
系统测试
验收测试
是否运行程序
静态测试
动态测试
是否查看源代码
白盒测试
黑盒测试
其他
回归测试
冒烟测试
随机测试
功能测试
性能测试
逻辑功能测试
界面测试
易用性测试
安装测试
兼容性测试
一般性测试
稳定性测试
负载测试
压力测试

内容提要

- 软件测试级别概述
- 单元测试（Unit Testing）
- 集成测试（Integration Testing ）
- 系统测试（System Testing ）
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

内容提要

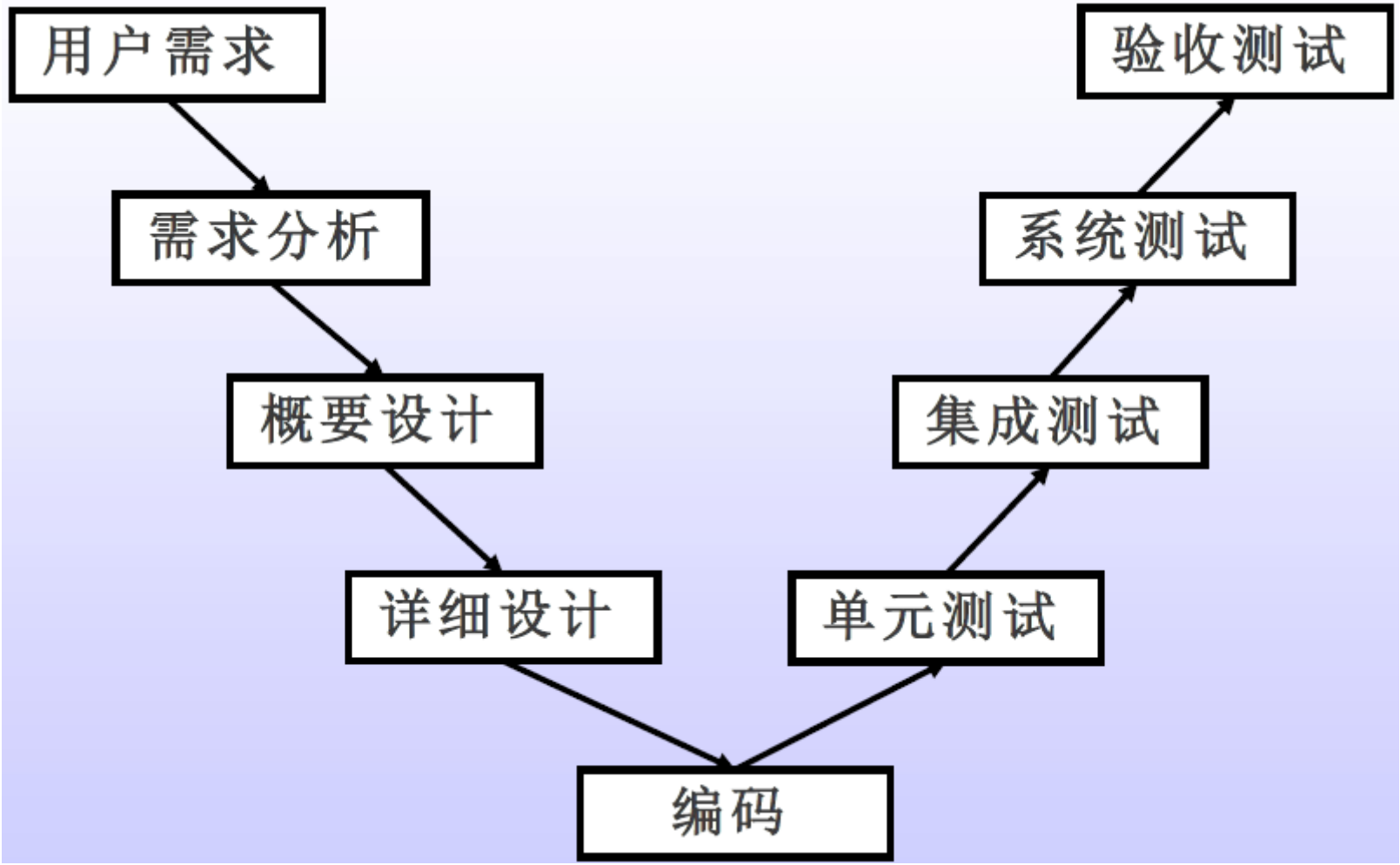
- *软件测试级别概述*
- 单元测试（Unit Testing）
- 集成测试（Integration Testing ）
- 系统测试（System Testing ）
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

软件测试级别概述

- 软件测试要经过4个不同的测试阶段，即单元测试、集成测试、系统测试和验收测试，将这些测试阶段称为软件测试级别.

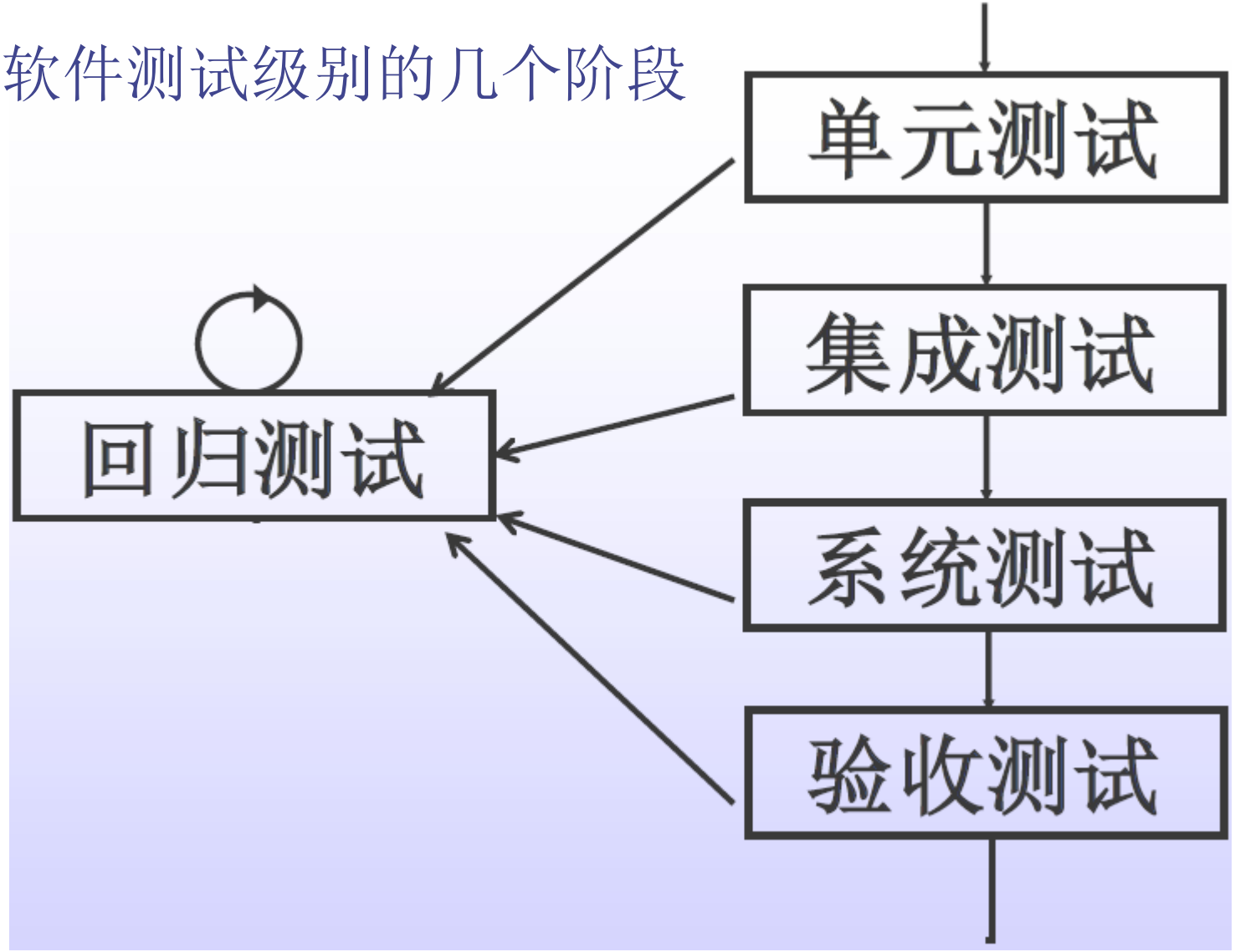
按阶段进行测试是一种基本的测试策略。

软件测试级别与开发的关系

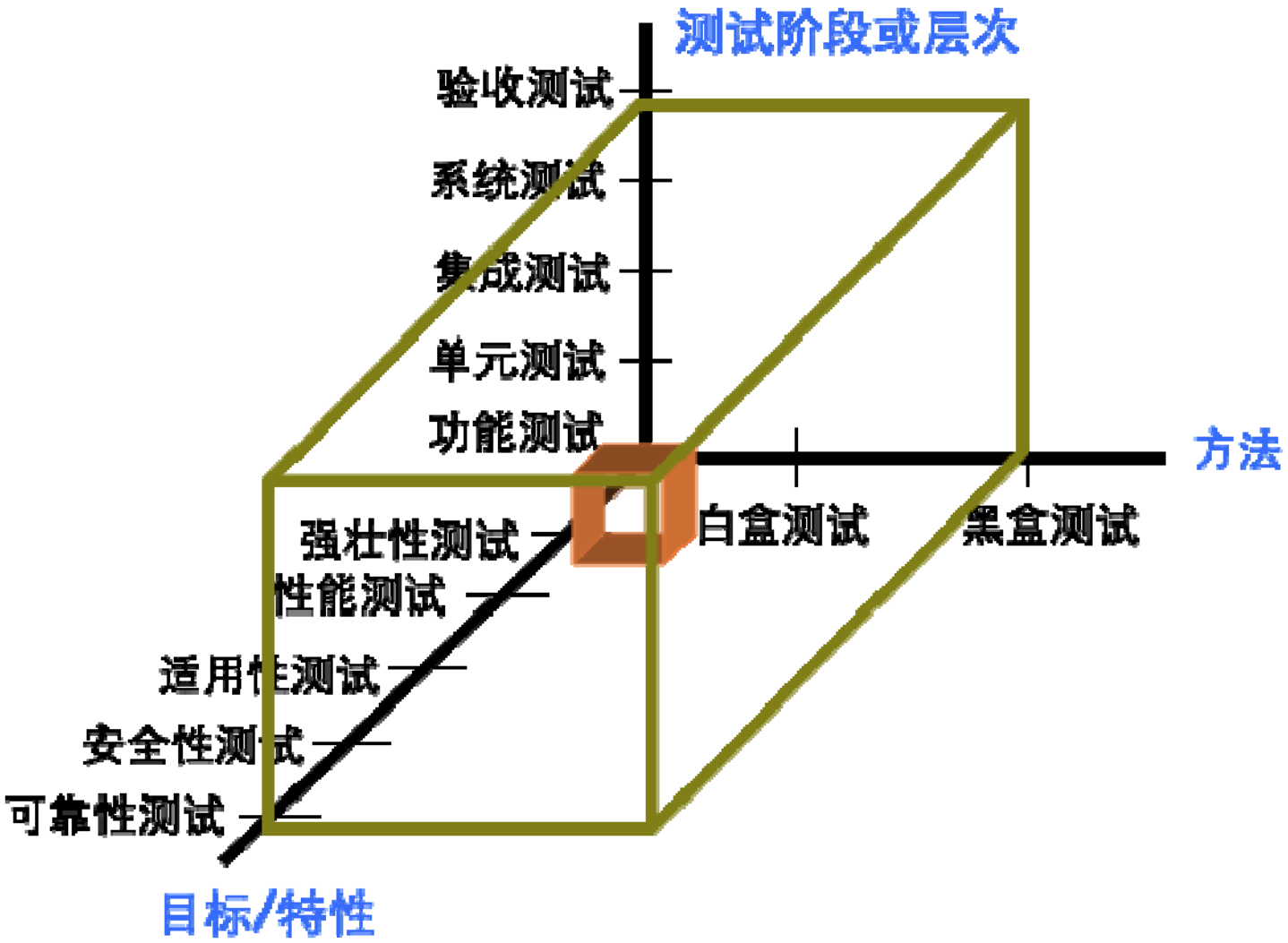


软件测试级别的划分

软件测试级别的几个阶段



软件测试级别和软件测试的关系



内容提要

- 软件测试级别概述
- *单元测试（Unit Testing）*
- 集成测试（Integration Testing ）
- 系统测试（System Testing ）
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

单元测试（Unit Testing）

- 单元测试的概念
- 为什么要引入单元测试
- 什么是单元
- 单元测试的方法
- 单元测试需要的辅助手段
- 单元测试的基本任务
- 单元测试常用工具

单元测试的概念

- 单元测试——是对一个模块的测试，是对软件中的基本组成单位进行的测试，如一个模块、一个过程等等
- 它是软件动态测试的最基本部分，其目的是检验软件基本组成单位的正确性
- 一般在代码完成后由开发人员完成，QA人员辅助。
- 单元测试的依据是“软件详细设计”

为什么要引入单元测试

- 单元测试强调的是基本的质量思想，如果要保证一个系统的质量，首先要保证构成这个系统的所有组成成分的质量
- 单元测试强调的是基础的重要作用

单元测试的重要性

1. 测试效果和时间的影响，认真做好单元测试后，集成测试会变得顺利，整体上更节约时间

2. 测试成本的影响，单元测试容易定位错误，而且通常在早期发现错误

3. 产品质量的影响，单元测试是构筑软件质量的基石

什么是单元

♦ 单元——是开发者创建的最小软件片段，单元是构造系统的基础

1. 不同的编程语言有不同的基本单元：
2. C语言的基本单元是函数；
3. C++和Java的基本单元是类；
4. Basic和COBOL的基本单元可能是整个程序

单元测试的方法

单元测试的基本方法包括：

- 人工静态分析——通过人工阅读代码来查找错误，一般是程序员交叉查看对方的代码
- 自动静态分析——使用工具扫描代码，根据某些预先设定的错误特征，发现并报告代码中的可能错误
- 自动动态测试——使用工具自动生成测试用例并执行被测试程序，来发现并报告错误。白盒或黑盒
- 人工动态测试——人工设定程序的输入和预期的正确输出，执行程序，并判断实际输出是否符合预期，如果不符合预期，则报告错误。白盒或黑盒

单元测试最常使用的是动态白盒测试

单元测试 举例

```
#include <stdio.h>
Void isZero(int m){
    if (m!=0)
      printf("%d",m);
    else
       printf("%d",1);
}
```

```
Void main(){
    int a[5];
    int i;
    printf("请输入5个整数
\n");
    for(int i=0;i<5;i++){
       scanf("%d",&a[i]);
       isZero(a[i]);
    }
}
```

单元测试 举例——测试内容

静态测试：

无注释！！！

代码分析，审查

动态测试：

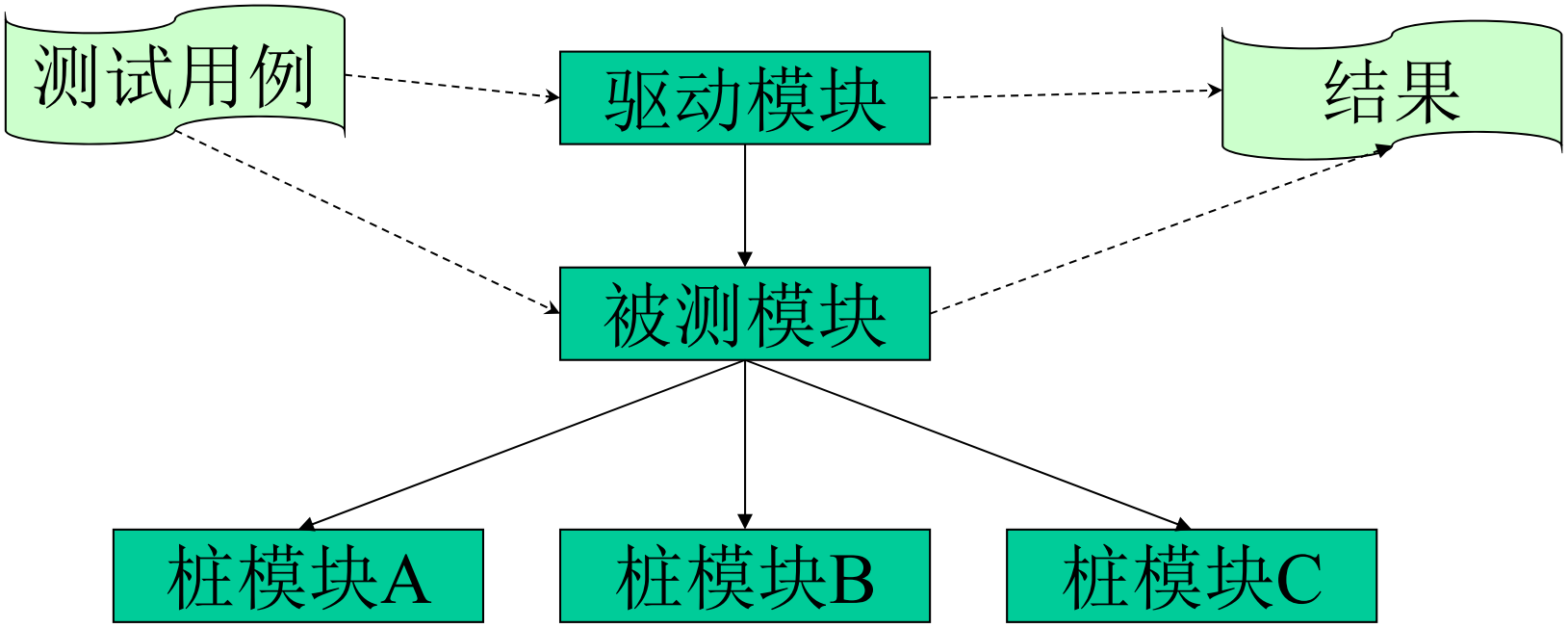
输入边界值问题（多余的输入等）

非法数据的容错性（输入字符）

单元测试需要的辅助手段

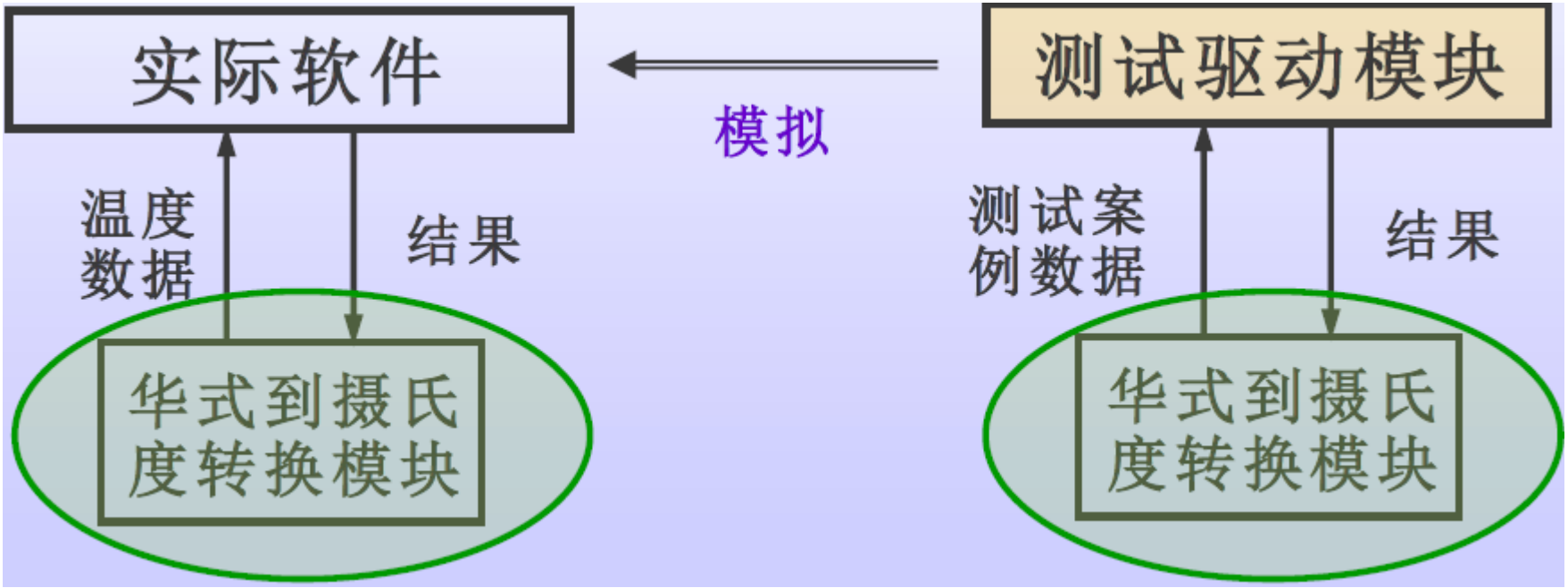
- 进行单元测试时，因为一个被测单元不是一个单独的程序，因此无法单独运行，只有在增加了辅助代码之后才能进行单元测试
- 辅助代码包括：驱动程序/模块**(Driver)**和桩程序/模块（**Stub**）

桩模块和驱动模块



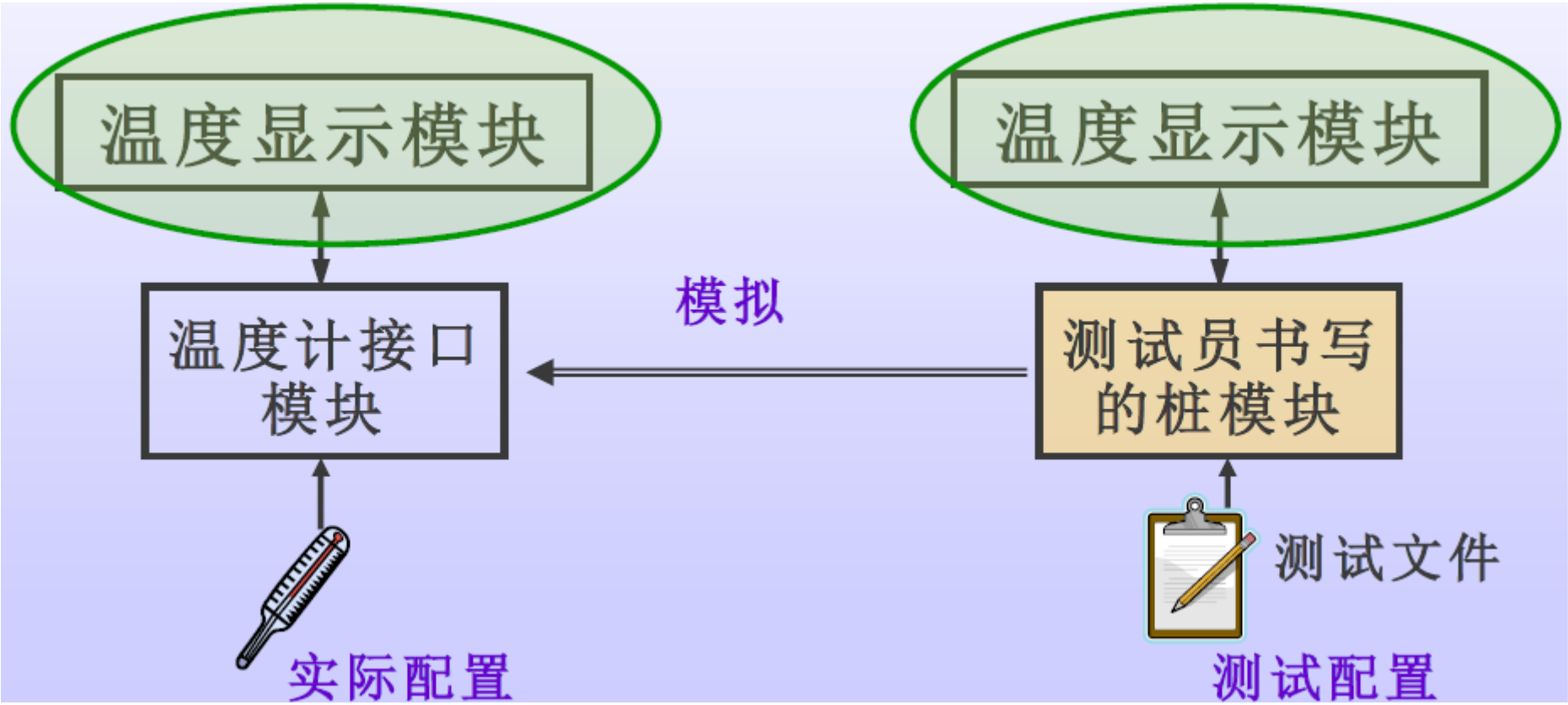
桩模块和驱动模块

驱动模块（**Driver**）——在对底层或子层模块进行单元测试时，所编制的调用被测模块的程序，用于模拟被测模块的上级模块



桩模块和驱动模块

桩模块（**Stub**）——对顶层或上层模块进行单元测试时，所编制的替代下层模块的程序，用于模拟被测模块工作过程中所调用的模块



谁是“桩模块”，谁是“驱动模块”

```
#include <stdio.h>
int fun1(int x,int y){
    return x+y;
}
```

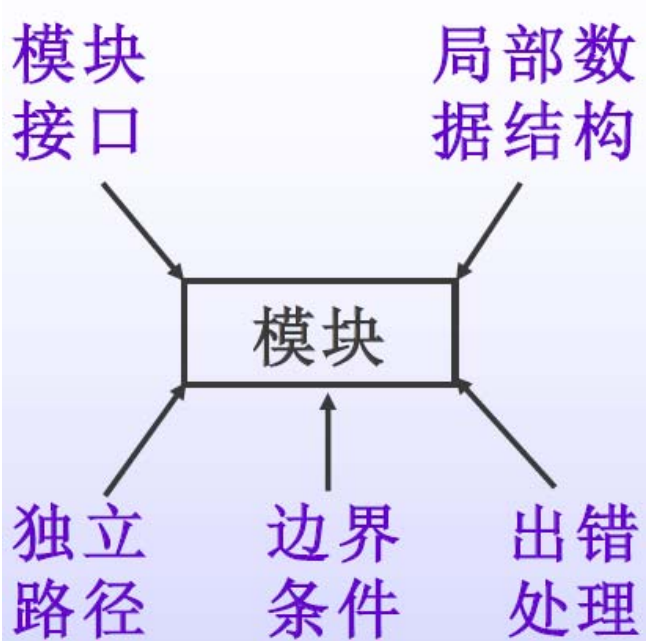
```
#include <stdio.h>
int fun2(int x, int y, int m){
    return fun1(x,y)*m;
}

Void main(){
    int a = 1, b = 2, c;
    c = fun1(a,b);
}
```

单元测试的基本任务

- 单元测试解决5个方面的测试问题：

1. 模块接口测试
2. 局部数据结构测试
3. 独立执行路径测试
4. 各种错误处理测试
5. 模块边界条件测试



模块接口测试

- 模块接口测试检查进出模块的数据流是否正确，这是单元测试的基础

1. 参数个数与类型是否匹配
2. 参数的顺序是否正确
3. 是否会修改只读参数
4. 参数的假设与实际是否一致
5. 全程变量的定义在各模块是否一致

模块局部数据结构测试

测试模块内部数据的完整性、正确性等

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
4. 不相容的数据类型
5. 上溢、下溢或地址错误

模块中所有独立路径测试

测试模块中每一条独立路径，可以发现不正确的控制流错误：

1. 误解的或不正确使用算术优先级
2. 混合类型运算
3. 不正确的判定
4. 不正确的逻辑操作或优先级
5. 不适当地修改循环变量
6. 不正常的或不存在的循环终止

各种错误处理测试

良好的设计应该预先估计到软件运行时可能发生的错误，并给出相应的处理措施

1. 对运行发生的错误描述难以理解
2. 报告的错误与实际遇到的错误不一致
3. 出错后，在错误处理之前引起了系统干预
4. 例外条件的处理不正确
5. 提供的错误信息不足，以致无法定位错误

边界条件测试

检查边界数据处理的正确性，采用边界分析方法设计测试用例

1. 普通合法数据的处理
2. 普通非法数据的处理
3. 边界值内、外边界数据处理
4. 数据流、控制流中等于、大于、小于比较是否出错

单元测试常用工具

类别	工具
C 语言	C++ Test、CppUnit、QA C/C++、CodeWzard、Insure++6.0
Java 语言	Jtest、JUnit、JMock、EasyMock、MockRunner
JUnit 扩展框架	TestNG 、JWebUnit 和 HttpUnit
GUI （功能）	JFCUnit、Marathor
通用的	Rexelint、Splint、McCabe QA、CodeCheck、GateKeeper
.NET	.TEST、Nunit 、Visual Studio Team System的单元测试
Data Object，DAO	DDTUnit，DBUnit
EJB	MockEJB 或者 MockRunner
Servlet，Struts	Cactu，StrutsUnitTest
XML	XMLUnit
内嵌式系统等	Logiscope、JTestCase

内容提要

- 软件测试级别概述
- 单元测试（Unit Testing）
- *集成测试（Integration Testing ）*
- 系统测试（System Testing ）
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

集成测试概述

- 单元测试完成之后，将模块连接起来组成软件系统的过程叫做集成。
- 集成的过程就是形成子系统及系统的过程。对这个过程中的子系统进行测试称为集成测试。
- 集成测试的主要依据是《软件概要设计》

集成测试的主要任务

集成测试解决以下5个方面的测试问题

1. 将各模块连接起来，模块相互调用时，数据经过接口是否丢失

2. 将各个子功能组合起来，检查能否达到预期要求的各项功能

3. 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

4. 全局数据结构是否存在问题，会不会被异常修改

5. 单个模块的误差积累起来，是否被放大

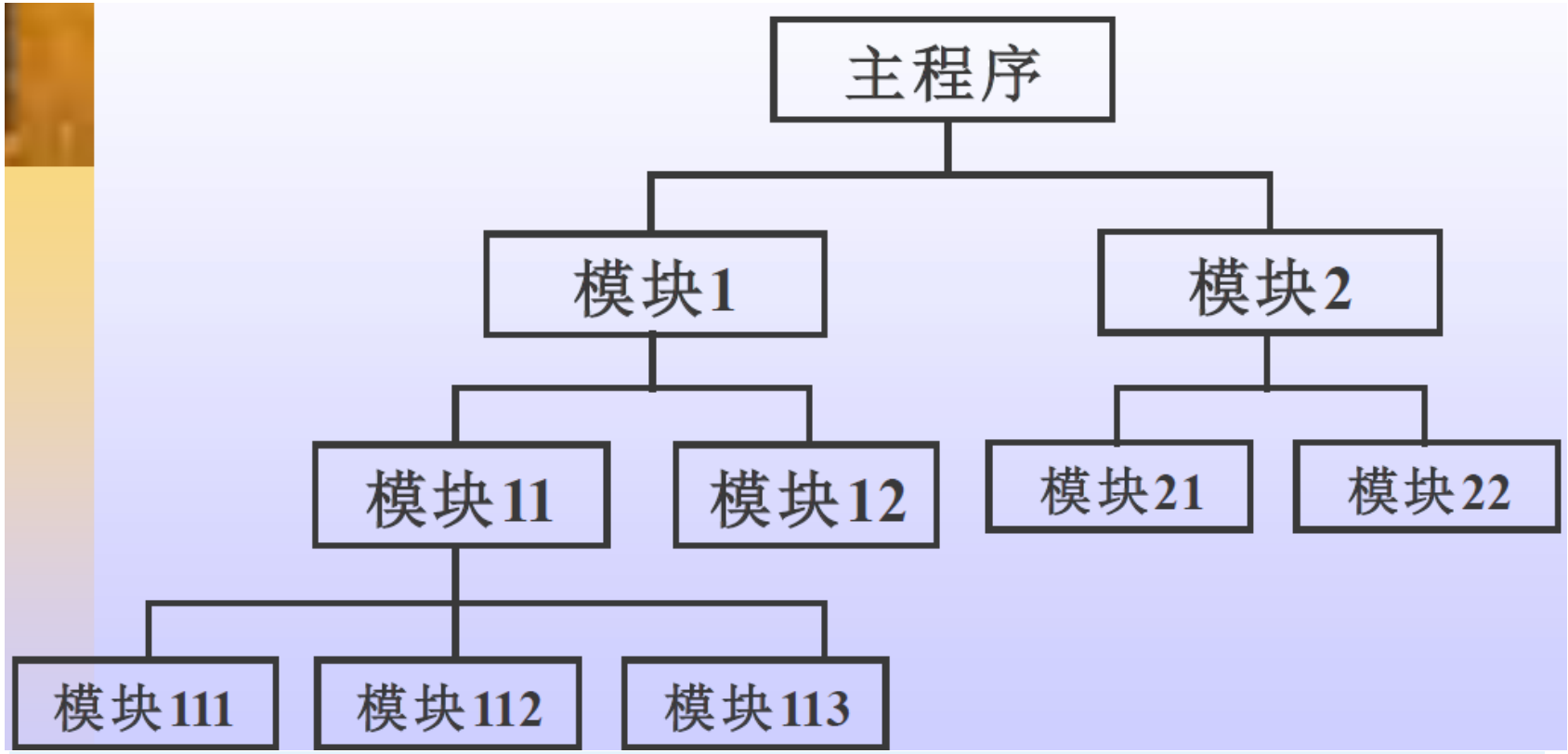
集成测试的实施方案

集成测试的实施方案依赖于模块的集成策略，集成策略包括：

1. 非增量式集成
2. 增量式集成

非增量式集成测试

非增量式集成测试：采用一步到位的方法进行测试，即在完成单元测试后，一次性将所有模块连接起来进行整体测试

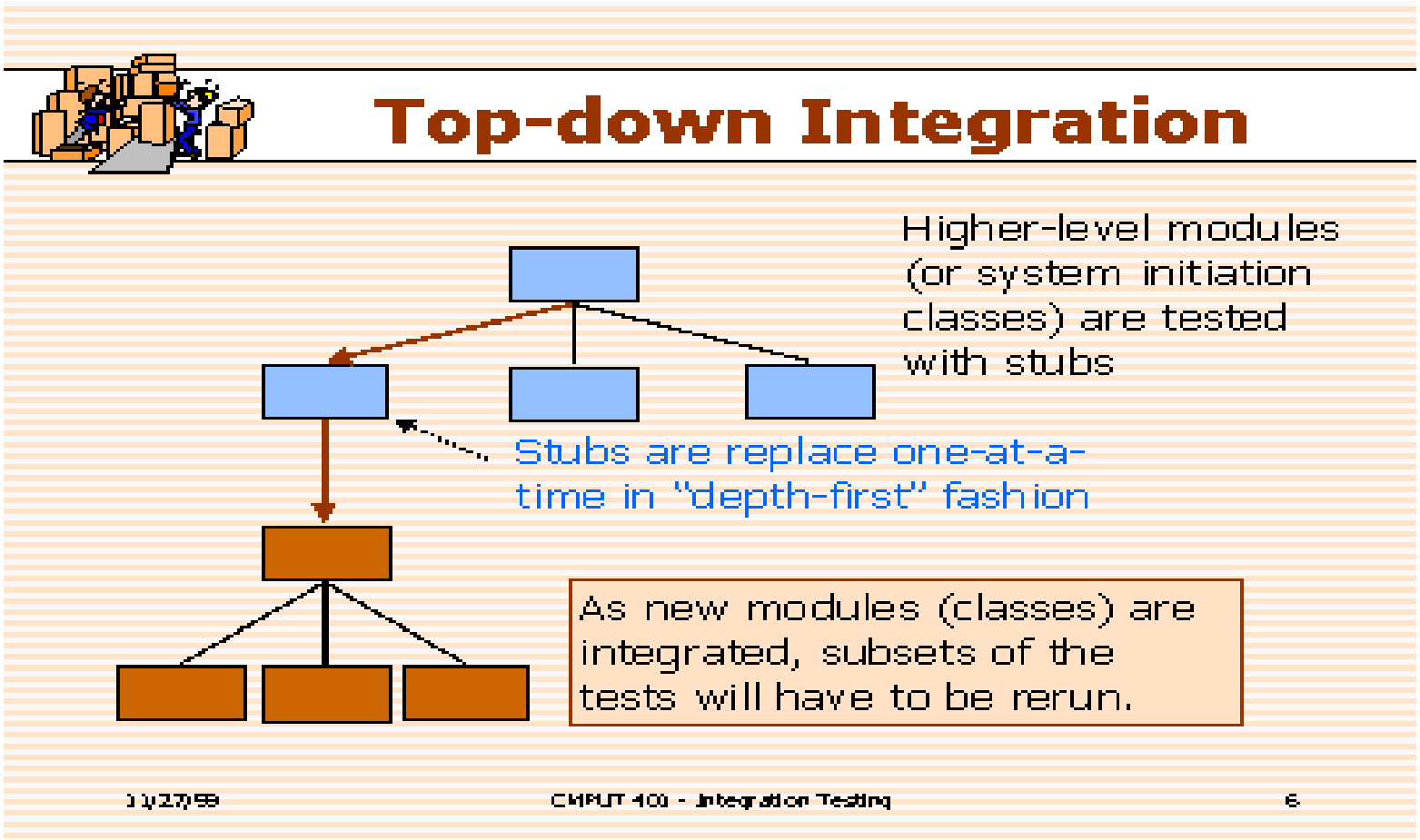


很难确定出错的真正位置。这种方法并不推荐在任何系统中使用，但可以在规模较小的应用系统中使用。

增量式集成测试

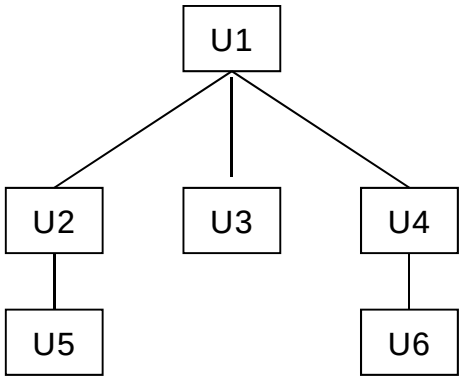
- 增量式测试则是逐步完成测试单元的集成，最后形成整个系统
  - 自顶向下增量式集成测试
  - 自底向上增量式集成测试
  - 三明治增量式测试（混合增量式测试）

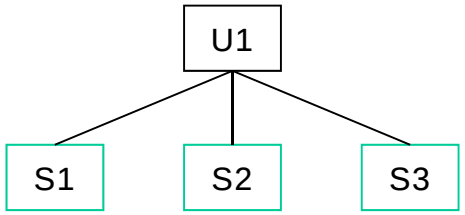
自顶向下集成



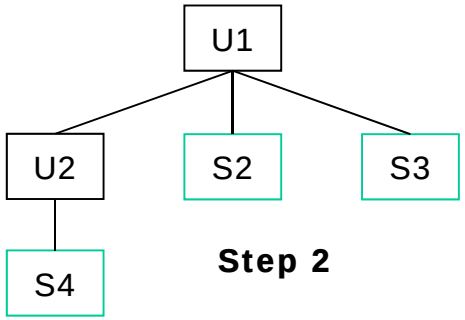
自顶向下增量式测试模块集成的顺序是首先集成主控模块（主程序），然后依照控制层次结构向下进行集成。从属于主控模块的按深度优先方式（纵向）或者广度优先方式（横向）集成到结构中去。

自顶向下集成--深度优先组装方式

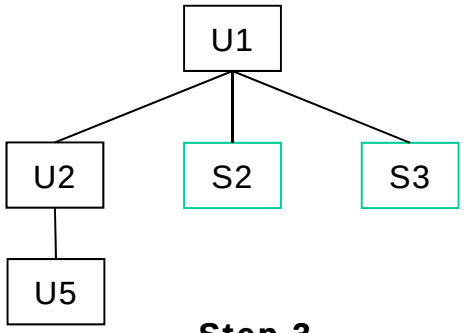




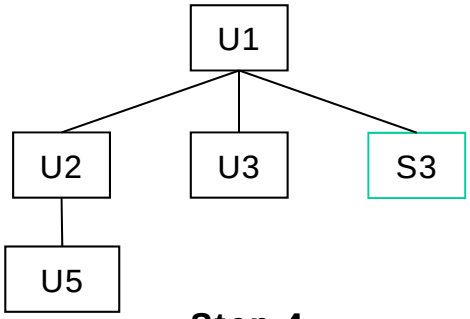
Step 1



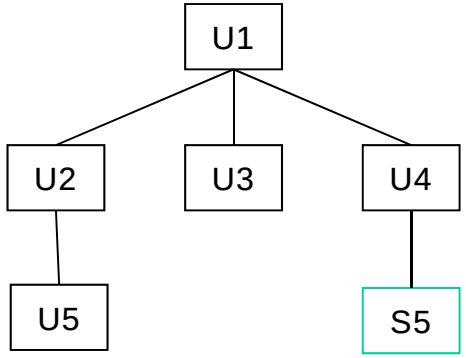
Step 2



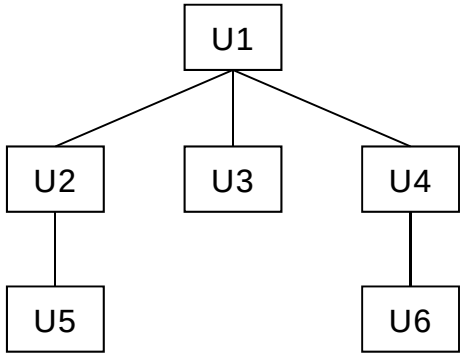
Step 3



Step 4

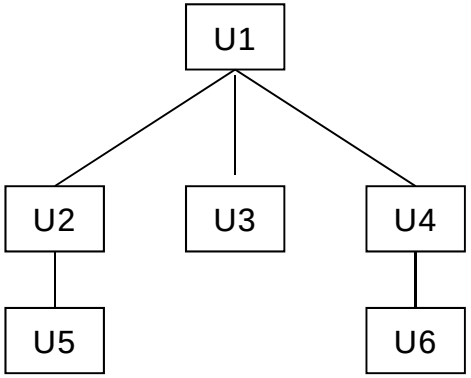


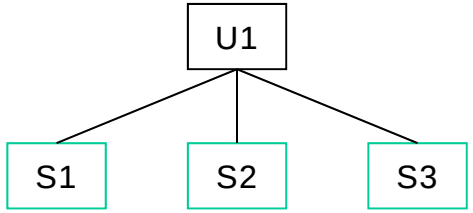
Step 5



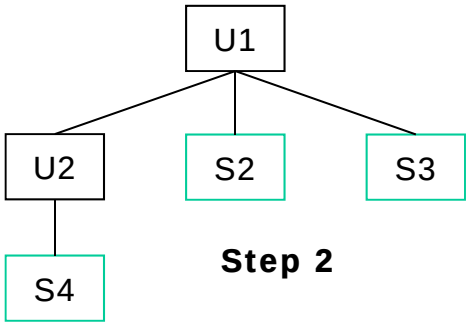
Step 6

自顶向下集成—宽度优先组装方式

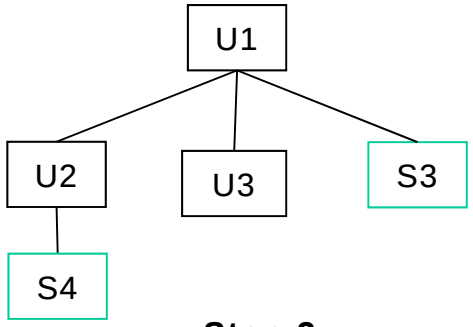




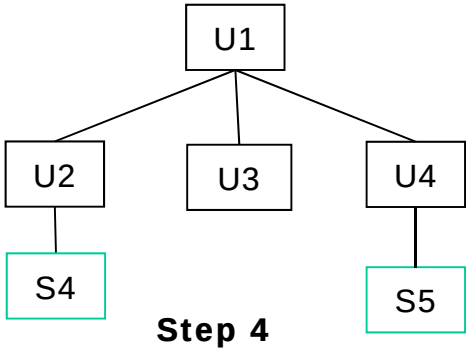
Step 1



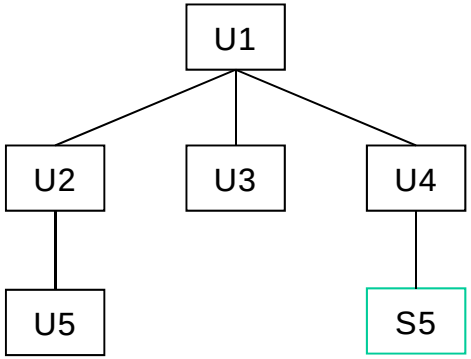
Step 2



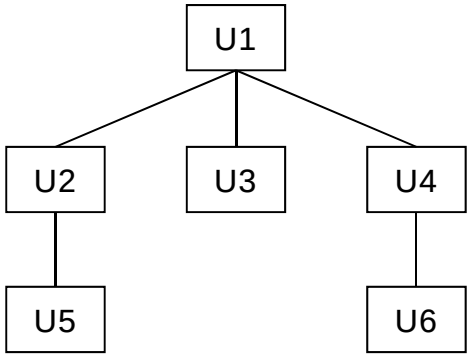
Step 3



Step 4



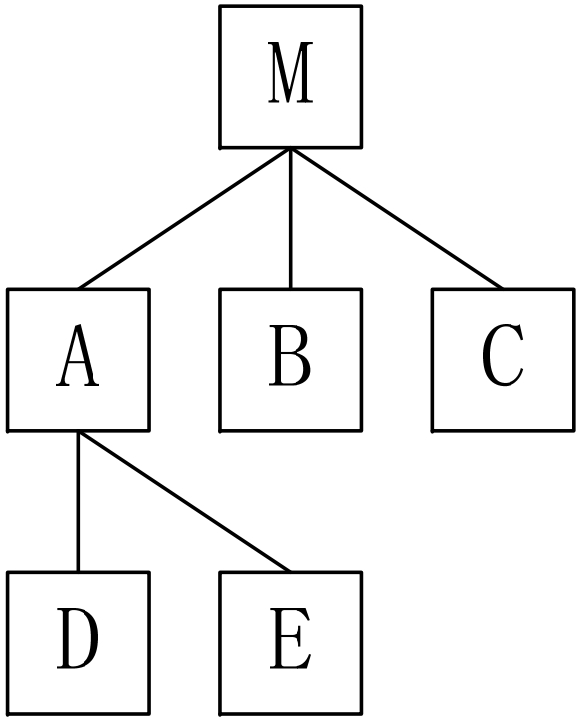
Step 5



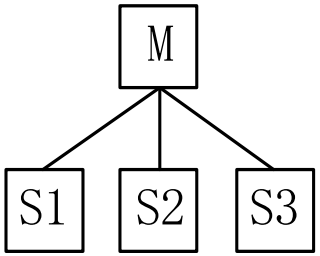
Step 6

练习

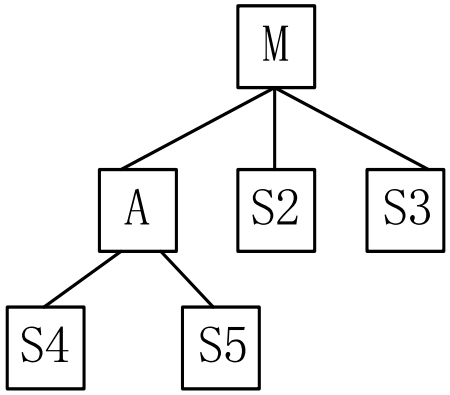
对如下结构采用自顶向下深度优先策略进行测试



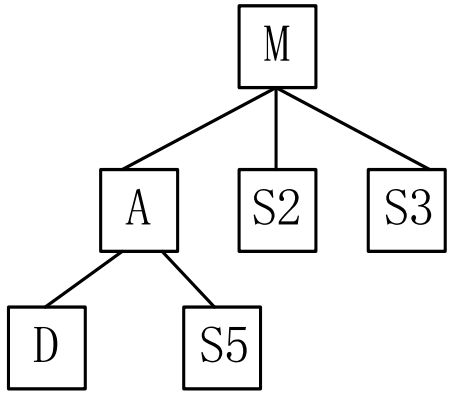
练习



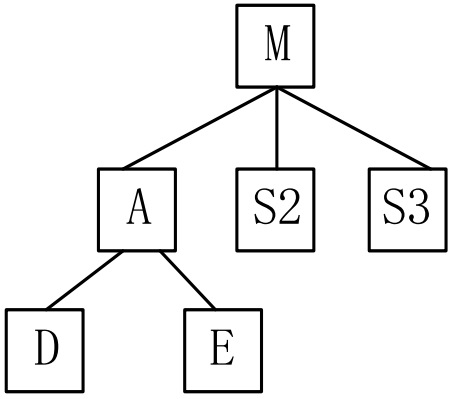
测试M



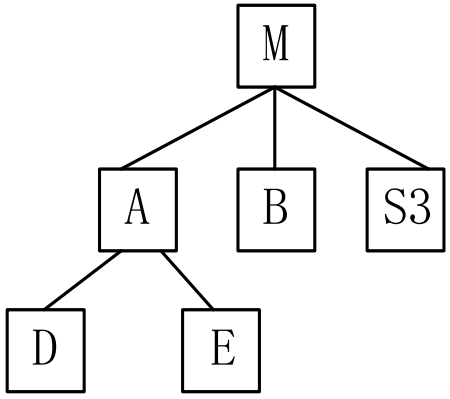
加入A



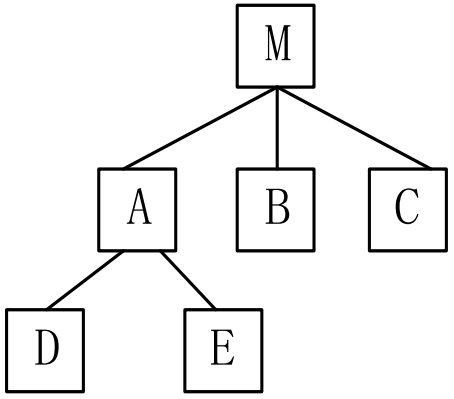
加入D



加入E

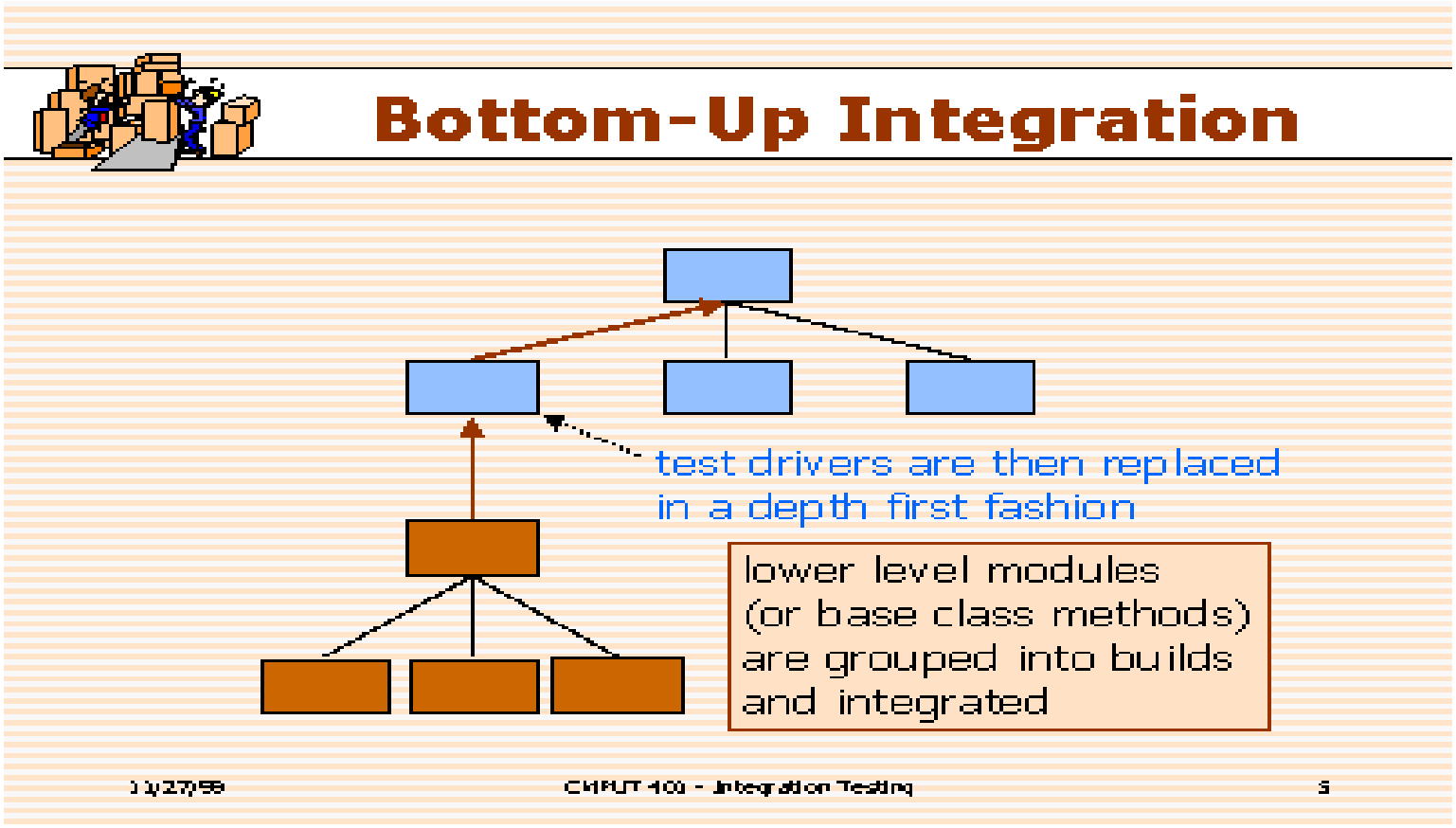


加入B



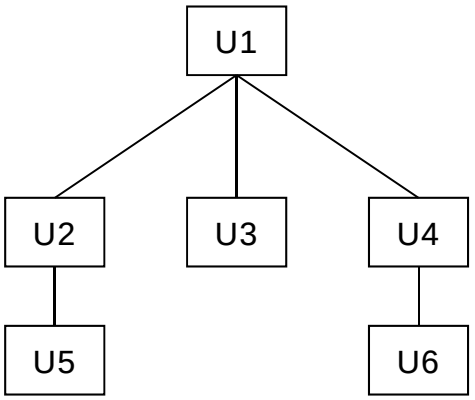
加入C

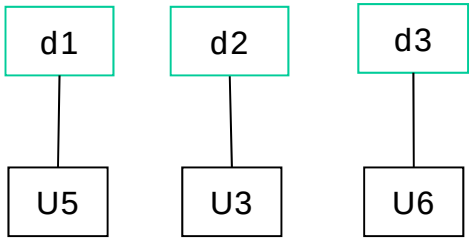
自底向上集成



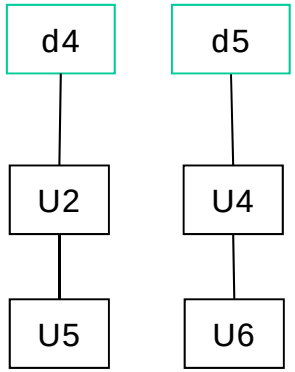
从具有最小依赖性的底层组件开始，按照依赖关系树的结构，逐层向上集成，以检验系统的稳定性。
最常用的集成策略，其他方法都或多或少应用此种方法。

自底向上集成

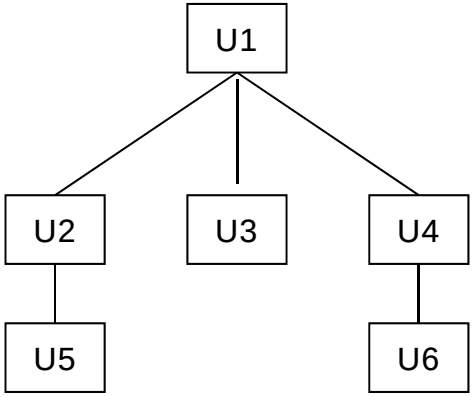




Step 1



Step 2



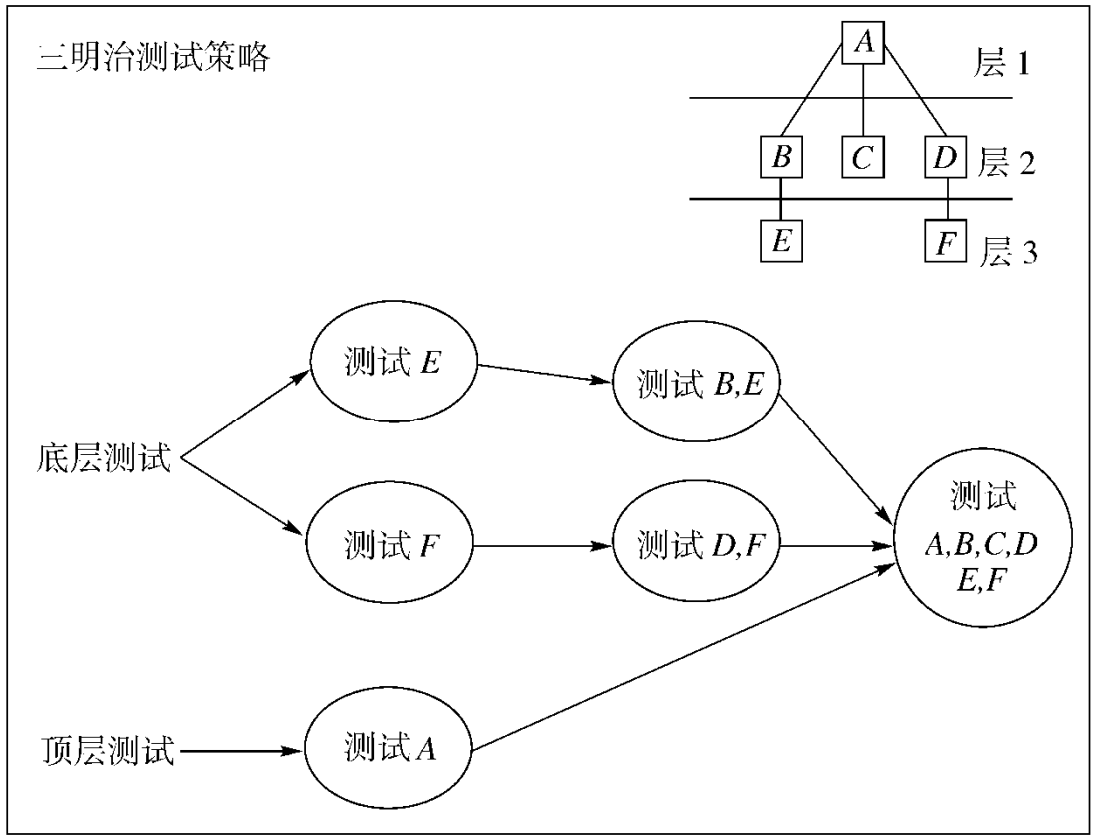
Step 3

需要为所测模块或子系统编制相应的驱动模块

两种集成策略的比较

测试方法	优点	缺点	适用范围
自顶向下	1. 如果程序错误趋向于发生在程序的顶端时，有利于查出错误。 2. 可以较早出现程序的轮廓。 3. 加进输人 / 输出模块后，较方便描述测试用例。	1. *桩*模块开发量大，较难设计。 2. 模块介入使结果较难观察。 3.底层验证被推迟；底层组件测试不充分。	1.产品控制结构比较清晰和稳定； 2.高层接口变化较小； 3.底层接口未定义或经常可能被修改； 4.产口控制组件具有较大的技术风险，需要尽早被验证； 5.希望尽早能看到产品的系统功能行为。
自底向上	1. 如果程序错误趋向于发生在程序的底端时，有利于查出错误。 2. 容易产生测试条件和观察测试结果。 3. 容易编写驱动模块。	1. 在加入最后一个模块之前，程序不能作为一个整体存在。 2. *驱动*开发量大。 3.对高层的验证被推迟，设计上的错误不能被及时发现。	1.适应于底层接口比较稳定； 2.高层接口变化比较频繁； 3.底层组件较早被完成。

混合策略--三明治集成方法(Sandwich Integration)



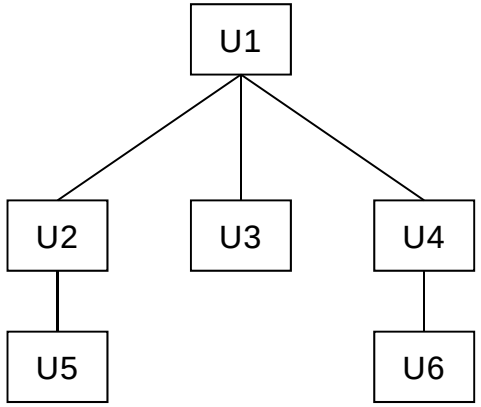
混合法：对软件结构中较上层，使用的是“自顶向下”法；对软件结构中较下层，使用的是“自底向上”法，两者相结合

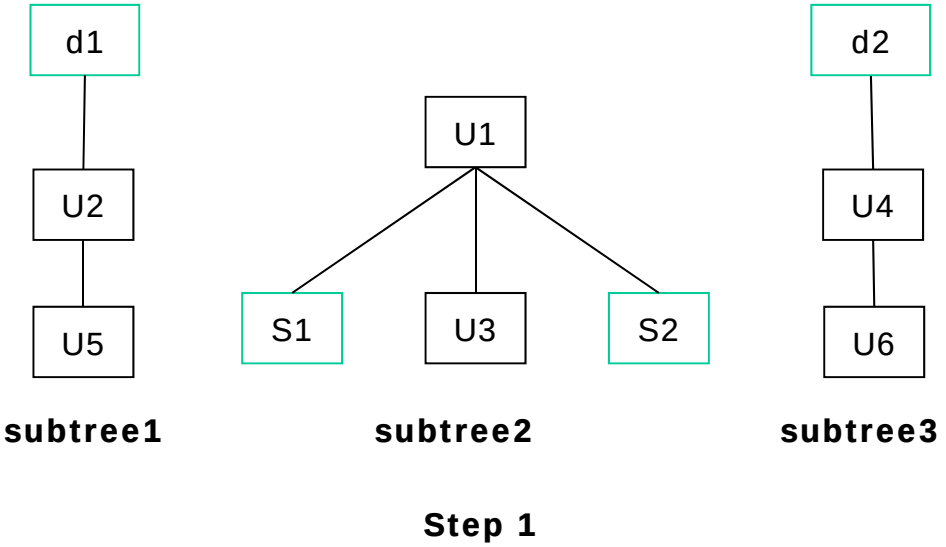
采用三明治方法的优点是：它将自顶向下和自底向上的集成方法有机地结合起来。不需要写桩程序，因为在测试初自底向上集成已经验证了底层模块的正确性。采用这种方法的主要缺点是：在真正集成之前每一个独立的模块没有完全测试过。

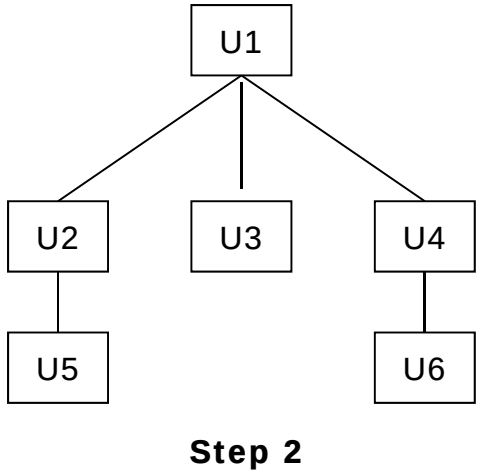
三明治集成方法

为减少桩模块和驱动模块，

在分解树的子树上进行集成







- 优点：

集合了自顶向下和自底向上两种策略的优点

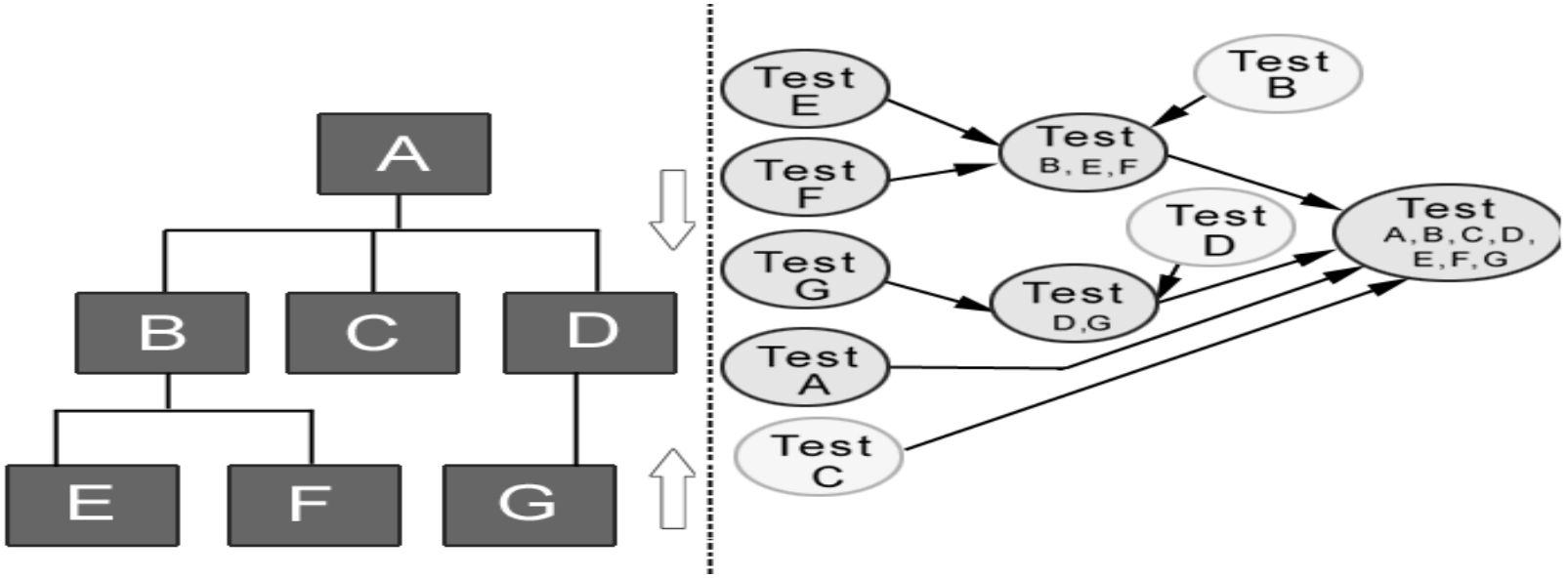
- 缺点：

中间层测试不充分

- 适用范围：

适应于大部分软件开发项目

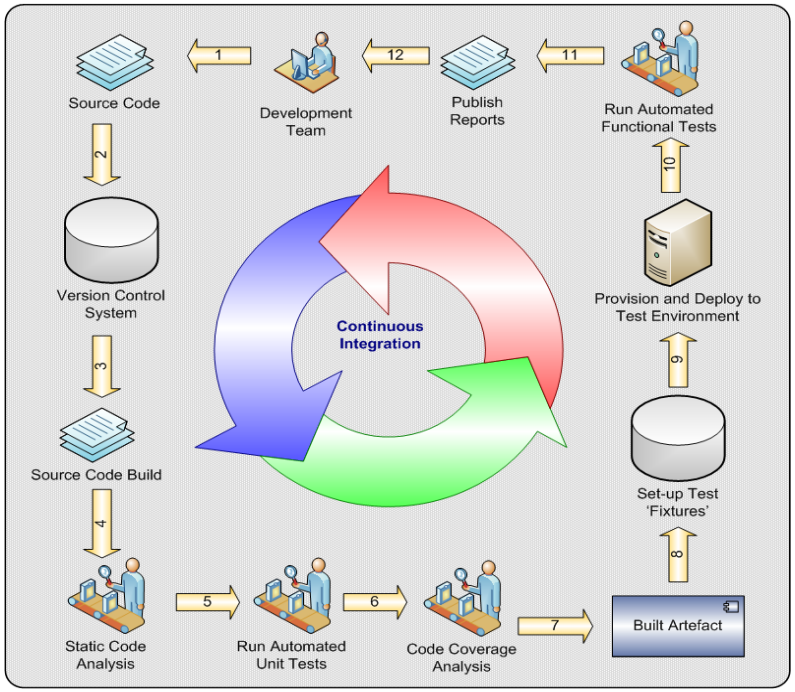
改善的三明治集成方法

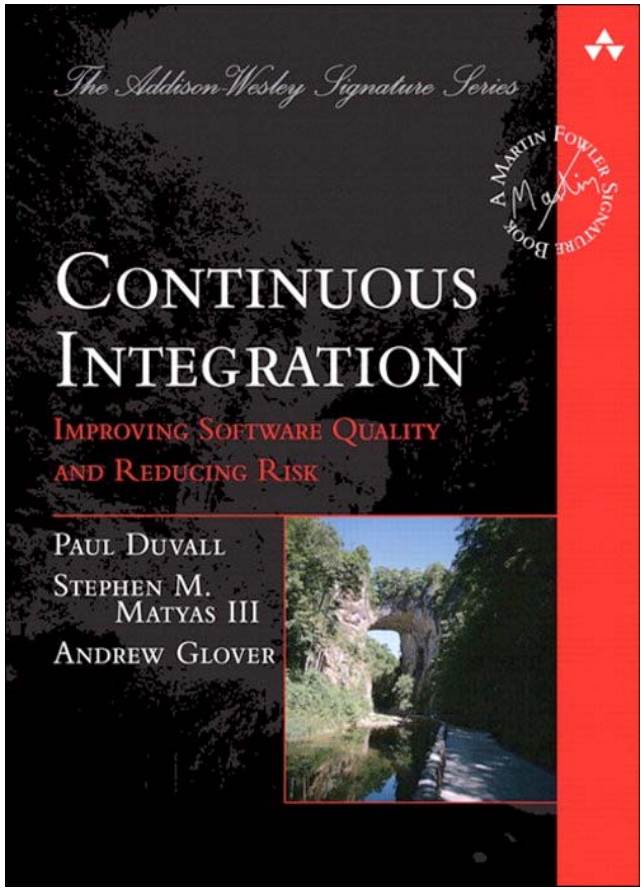


改进的三明治集成方法，不仅自两头向中间集成，而且保证每个模块得到单独的测试，使测试进行得比较彻底 。

持续集成（CI-Continuous integration ）

持续集成是软件开发越来越普遍的一种优秀实践，即团队开发成员经常集成他们的工作，通常每天新完成的代码至少集成一次，也就意味着每天可能会发生多次集成 ——Martin Fowler



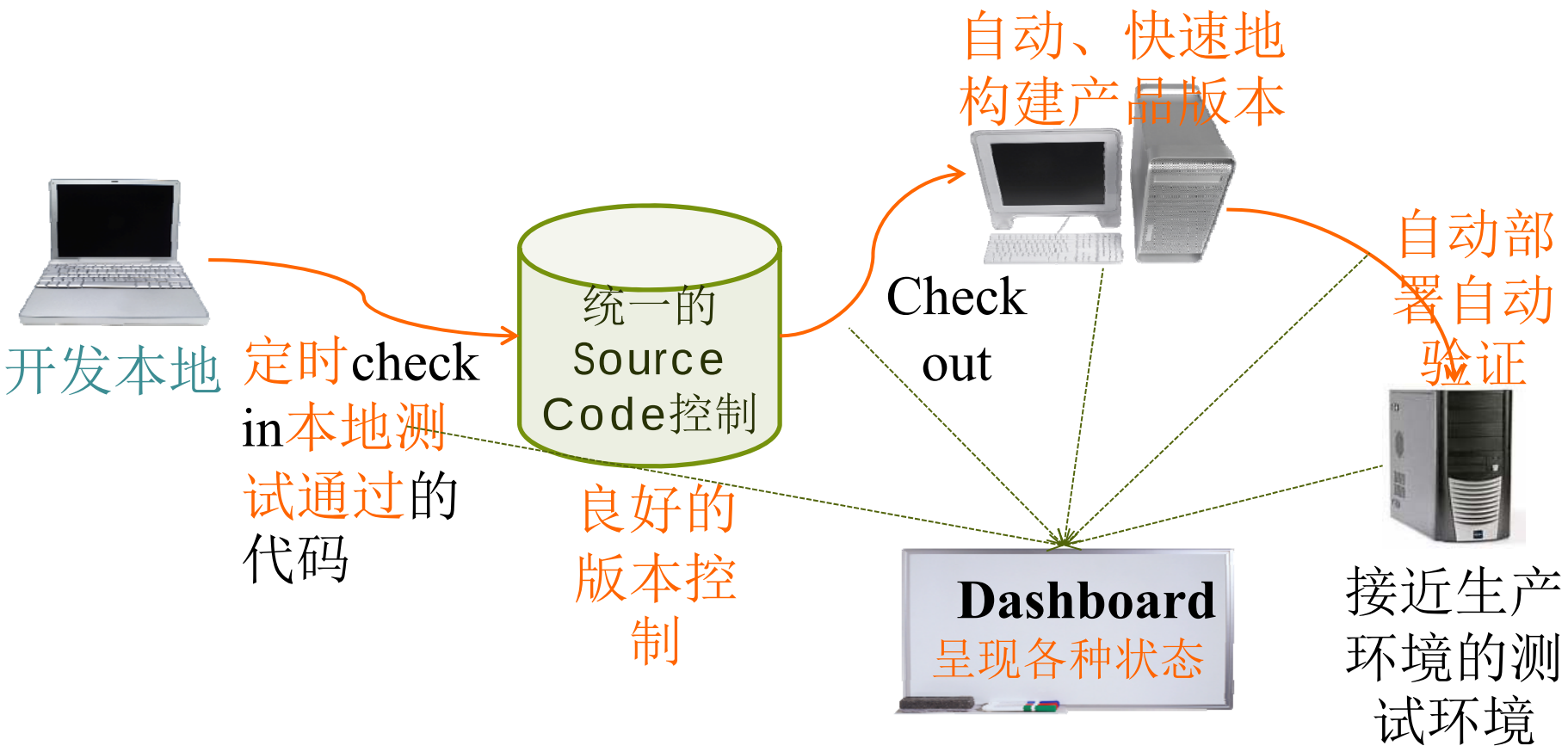


如何做好持续集成?

- 维护单源存储库
- 实现自动构建、自动部署、自动Build验证
- 每个人每天都有贡献
- 每一份贡献都应用于构建主线上
- 加快构建过程
- 在一个相同的生产环境中（clone）执行测试
- 使任何人都可以简单获取最新的可执行文件
- 每个人都可以看到当前状况
- 自动部署

FROM：http://martinfowler.com/articles/continuousIntegration.html

持续集成原理


自动、快速地
构建产品版本
自动部
署自动
验证
开发本地
定时check
in本地测
试通过的
代码
统一的
Source
Code控制
Check
out
良好的
版本控
制
Dashboard
呈现各种状态
接近生产
环境的测
试环境

示例：Maven 覆盖整个CI生命周期

mvn [options] [<goal(s)>] [<phase(s)>]

- validate: 验证项目是否正确以及相关信息是否可用
- compile: 编译
- test: 通过JUnit等单元测试
- package: 根据事先指定的格式（比如jar）进行打包
- integration-test: 部署到运行环境中，准备集成测试
- verify: 对包进行有效性和质量检查（BVT）
- install: 安装到本地代码库
- deploy: 在集成或发布环境，将包发布到远程代码库

```
mvn install:install-file -
DgroupId=<your_group_id> -
DartifactId=<your_artifact_id> -
Dversion=<version> -
Dfile=<path_of_jar_file>
-Dpackaging=jar
-DgeneratePom=true
```

clean: 清除以前的构建物

site: 生成项目文档

内容提要

- 软件测试级别概述
- 单元测试（Unit Testing）
- 集成测试（Integration Testing ）
- *系统测试（System Testing ）*
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

系统测试概述

- 集成测试通过后，各个模块形成了一个统一的系统
- 系统测试即是对形成的整个系统进行测试，系统测试通常是消耗测试资源最多的地方
- 系统测试要测试系统的完整性及有效性

系统测试的内容

- 系统测试——软件系统是指交付给用户的完整软件产品，它可能由软件、硬件、用户手册、培训材料等组成。系统测试关注在最高集成条件下产生的软件缺陷
- 典型的系统测试包括很多类型的测试，例如：功能测试，易用性测试，安全性测试，可靠性和可用性测试，性能测试，备份和恢复测试，便携式测试以及其他测试等

系统测试的准备工作

1. 收集需求说明书、参考手册等
2. 做好测试计划
3. 设计好测试案例
4. 组织好测试过程
5. 处理好测试结果

系统测试的任务

1. 功能测试（黑盒测试）
2. 用户界面测试
3. 性能测试
4. 可靠性、稳定性测试
5. 恢复测试
6. 安全性测试
7. 强度测试
8. 易用性测试

性能测试

性能测试——Flat负载测试

一次加载所有的用户，然后在预定的时间段内持续运行。

Ramp-up负载测试

用户数量是不断上升的。

安全性测试

- 安全性测试是检查系统对非法侵入的防范能力。测试人员假扮非法入侵者，采用各种办法试图突破防线。

例如：

  - 想方设法截取或破译口令；
  - 专门开发软件来破坏系统的保护机制；
  - 故意导致系统失败，企图趁恢复之机非法进入；
  - 试图通过浏览非保密数据，推导所需信息等等。

- 理论上讲，只要有足够的时间和资源，没有不可进入的系统。因此系统安全设计的准则是，使非法侵入的代价超过被保护信息的价值，此时非法侵入者已无利可图。

可靠性测试

- 可靠性（Reliability）是产品在规定的条件下和规定的时间内完成规定功能的能力，它的概率度量称为可靠性。
- 软件可靠性是软件系统的固有特性之一，它表明了一个软件系统按照用户的要求和设计的目标，执行其功能的可靠程度。
- 软件可靠性与软件缺陷有关，也与系统输入和系统使用有关。

内容提要

- 软件测试级别概述
- 单元测试（Unit Testing）
- 集成测试（Integration Testing ）
- 系统测试（System Testing ）
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

验收测试

- 验收测试概述
- 验收测试的主要内容
- α 测试
- β 测试

验收测试概述

也叫确认测试

- 验收测试在软件开发结束后，在产品发布之前进行
- 验收测试面向客户，从客户使用和业务场景的角度出发，目的是得到用户的理解和认同

需求规格说明书

验收测试的主要内容

1. 配置复审
2. 易用性测试
3. 安装测试，可恢复性测试
4. 兼容性测试
5. 软件文档（用户手册、操作手册）检查
6. 用户界面测试
7. 软件功能和性能测试与测试结果的评审

用户界面测试

用户界面的7个要素:

- 符合标准和规范。
- 直观性。
- 一致性。
- 灵活性。
- 舒适性。
- 正确性。
- 实用性。

用户界面测试没有具体量化的指标，主观性较强。

可安装性测试

- 系统软件安装
- 应用软件安装
- 服务器的安装
- 客户端的安装
- 产品升级安装
- 等等

可恢复性测试

- 主要检查系统的容错能力。当系统出错时，能否在指定时间间隔内修正错误或重新启动系统。
- 可恢复测试首先要通过各种手段，让软件强制性地发生故障，然后验证系统是否能尽快恢复。
  - 对于自动恢复需验证重新初始化、检查点、数据恢复和重新启动等机制的正确性；
  - 对于人工干预的恢复系统，还需估测平均修复时间，确定其是否在可接受的范围内。

α测试

- α 测试——是由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 测试是在受控制的环境下进行的测试
- α 测试的目的是评价软件产品的**FURPS**（**Function**、**Usability**、**Reliability**、**Performance**、**Security**，即功能、易用性、可靠性、性能和安全性）。

α测试的特点

1. 它是在开发环境下进行的（环境是可控的）

2. 它不需要测试用例评价软件使用质量

3. 用户往往没有相关经验，可以由开发人员或测试人员协助用户进行测试

- α 测试的关键在于尽可能逼真地模拟实际运行环境和操作，并尽最大努力涵盖所有可能的用户操作方式。

β测试

- β 测试——是指软件开发公司组织各方面的典型用户在日常工作中实际使用软件系统，并要求用户报告异常情况、提出批评意见，然后软件开发公司再对软件进行改错和完善
- β 测试通常采用黑盒测试方法

β测试的特点

1. β 测试是由软件用户在一个或多个用户的实际使用环境下进行的测试
2. β 测试时开发人员通常不在测试现场
3. β 测试是免费的
4. β 测试

课堂练习

1. 判断对错

（1）为了快速完成集成测试，采用一次性集成方式是适宜的。

（2）单元测试通常由开发人员进行。

（3）性能测试通常需要辅助工具的支持。

2. 单元测试中设计测试用例的依据是（ ）。

A）概要设计规格说明书

B）用户需求规格说明书

C）项目计划说明书

D）详细设计规格说明书

3. 根据软件需求规格说明书，在开发环境下对已经集成的软件系统进行的测试是（ ）。

A）系统测试　　B） 单元测试

C）集成测试　　D） 验收测试

课堂练习

4. 集成测试对系统内部的交互以及集成后系统功能检验了何种质量特性（ ）

A）正确性　　B）可靠性

C）可使用性　　D）可维护性

5. 单元测试主要针对模块的几个基本特征进行测试，该阶段不能完成的测试是（ ）。

A）系统功能　　B）局部数据结构

C）重要的执行路径　　D）错误处理

6. 软件测试过程中的集成测试主要是为了发现（ ）阶段的错误。

A）需求分析　　B）概要设计

C）详细设计　　D）编码

课堂练习

7. 集成测试时，能较早发现高层模块接口错误的测试方法为（ ）。
A）自顶向下增量式测试
B）自底向上增量式测试
C）非增量式测试
D）系统测试

8. 与验收测试阶段有关的文档是( )。
A）需求规格说明书　　B）概要设计说明书
C）详细设计说明书　　D）源程序

内容提要

- 软件测试级别概述
- 单元测试（Unit Testing）
- 集成测试（Integration Testing ）
- 系统测试（System Testing ）
- 验收测试（Acceptance Testing ）
- 回归测试（Regression Testing）

回归测试

- 回归测试概述
- 回归测试的策略

回归测试概述

回归测试是指软件被修改或扩充后重新进行软件测试

- 回归测试不是一个测试阶段，而是一种可以用于单元测试、集成测试、系统测试及验收测试各个测试过程的测试技术
- 通常而言，程序的每次变动都将引起回归测试

回归测试的策略

1. 完全重复测试
2. 选择性重复测试

完全重复测试

- 完全重复测试把所有的测试用例全部重新执行一遍，以确认缺陷修改的正确性和修改后周边是否受到影响
- 优点：是一种比较安全的策略，再测试全部用例具有最低的遗漏错误的风险
- 缺点：测试成本很高

选择性重复测试

1. 基于风险选择测试

- 基于一定的风险标准，从测试用例库中选择回归测试集

2. 基于操作剖面选择测试

- 优先选择那些针对最重要或最频繁使用功能的测试用例，释放和缓解最高级别的风险，有助于发现那些对可靠性有最大影响的故障

3. 再测试修改的部分

- 将回归测试局限于被改变的模块和它的接口上

回归测试的基本过程

1. 识别软件中被修改的部分

2. 从原测试用例库**T**中，选择那些对新版本依然有效的测试用例，其结果是建立一个新的测试用例库**T0**

3. 依据策略从**T0**中选择测试用例进行测试

4. 如果必要，生成新的测试用例集（或修改已有测试用例集）**T1**，用于测试**T0**无法充分测试的软件部分

5. 用**T1**进行测试

第**2**、**3**步是验证修改是否破坏了现有的功能，第4、5步是验证修改工作本身