三、单元测试与集成测试

1 单元测试

Unit Testing

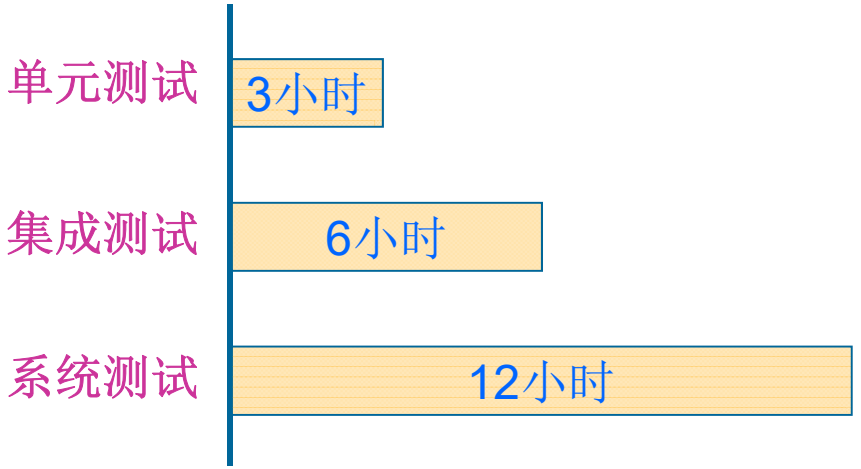
1. 单元测试

- 检验程序最小单位有无错误
  - 一般在编码之后，由开发人员完成。
  - 单元：软件开发中的最小的独立部分，**e.g.** 函数、类、页面

- 为何要进行单元测试?

  ➢检查代码是否符合设计和规范，保证局部代码的质量

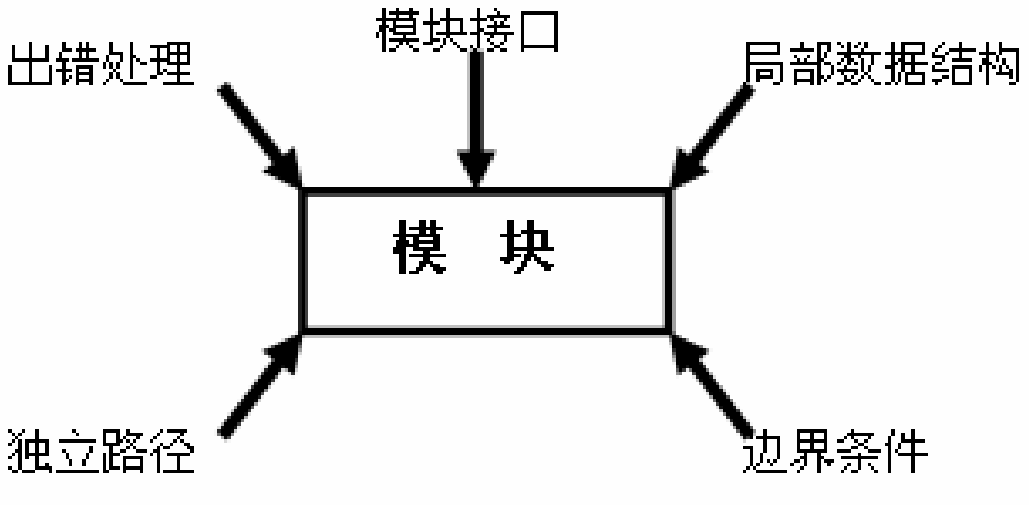
  ➢尽早发现错误，快速编程反馈



1.1 单元测试的主要内容

- 主要关注内部处理逻辑和数据结构
  - 模块接口测试
  - 独立路径测试
  - 局部数据结构测试
  - 错误处理测试
  - 边界测试

关注单元的主要质量风险



模块接口测试

- 检查进出模块的数据是否正确
  - 实参、形参是否匹配
    - 单元自身参数的个数、类型、顺序
    - 单元调用其它子模块时，模块调用的形参、实参是否匹配
  - 模块中对于全局变量是否合理使用
  - 是否由副作用：修改了只做输入用的形式参数
  - 使用其他模块时，是否检查可用性和处理结果
  - 使用外部资源时，是否检查可用性并及时释放资源
    - 内存、文件、硬盘、端口等
  - ……

局部数据结构测试

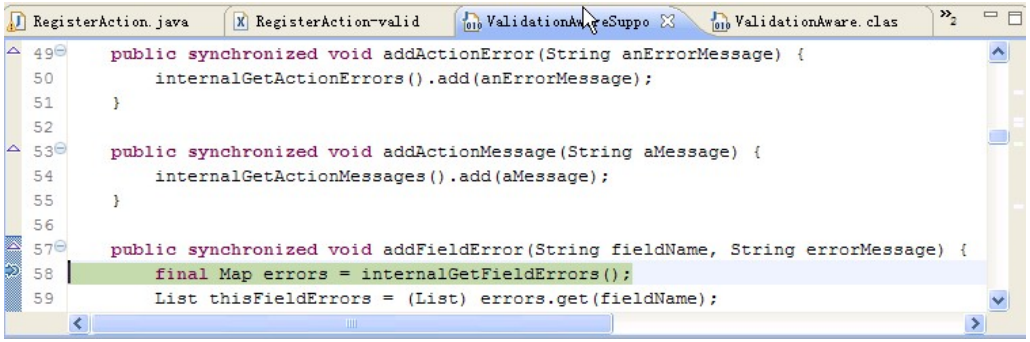
- 检查局部数据结构能否保持完整性和正确性
  - 检查不正确或不一致的数据类型说明
  - 使用尚未赋值或尚未初始化的变量
  - 错误的初始值或错误的默认值
  - 变量或函数名拼写错误： *i* 写成 *j*
  - 错误的类型转换
  - 数组越界
  - 非法指针
  - ……

独立路径测试

- 检查由于计算错误、判定错误、控制流错误导致的问题，保证模块中的每条独立路径(基本路径)都要走一遍
  - 死代码
  - 错误的计算优先级
  - 精度错误：赋值错误、比较运算错误（如浮点比较）
  - 表达式的不正确符号：>、>=、 =、==、!=
  - 循环变量的使用错误
  - 算法错误
  - ……

尽量用调试器单步执行一遍

在所有语句块上下断点，确保每个断点走一遍



错误处理测试

- 检查内部错误处理设施是否有效
  - 是否检查错误出现
    - 资源使用前后
    - 其他模块使用前后
  - 出现错误，是否进行错误处理
    - 抛出错误
    - 通知用户
    - 进行记录
  - 错误处理是否有效
    - 在系统干预前处理
    - 报告和记录的错误真实详细

```
public void creatPlayer(){
    try{
    System.out.println("请输入玩家一的ID：");
    Scanner console =new Scanner(System.in);
    int id=console.nextInt();
    }catch(InputMismatchException e){
        System.out.println("你输入的ID类型不符合");
    }
```

宁可把异常抛给上层，不要把异常闷掉

边界条件测试

- 检查临界数据是否正确处理
  - 普通合法数据是否正确处理
  - 普通非法数据是否正确处理
  - 边界内最接近边界的(合法)数据是否正确处理
  - 边界外最接近边界的(非法)数据是否正确处理
  - 在n次循环的第0次、1次、n次是否有错误；
  - 数据流、控制流中刚好等于、大于、小于确定的比较值时是否出现错误。
  - ……

1.2 单元测试过程

- 简单过程
  1. 实例化被测试对象
  2. 提供测试数据
  3. 调用被测试的方法
  4. 验证测试结果

```
TEST( TestAdd )
{
    Math math;
    int result = math.Add( 11, 12);
    CHECK_EQUAL( 23 , result );
}
```

一个简单的例子，被测对象很独立

单元测试的困难

- 关键困难
  - 如何准备各类被测对象

    那些被测试的对象应该处于何种状态才能通过测试充分暴露问题？这些状态通过何种途径达到？

  - 如何充分测试

    如何构造多样的环境、多样的输入，以充分暴露问题？

    （应用各种白盒、黑盒测试方法）

  - 如何检查测试结果

- 情况1

```
TEST( TestBehit )
{
    CPlayer target;
    target.Init(config);
    target.Behit( );
}
```

对象创建后未进行必要的初始化。可能内存访问异常，无法调用被测试的方法

- 情况2

```
TEST( TestMng )
{
    CMng mng;
    // 如何知道构造函数结果是否正确?
}
```

构造函数没有返回值，不知道如何验证结果

测试结果的分析

- 观察方法
  - 识别**Observer Methods**
    - 如果输出对象是**Stack**类，那么无法直接知道操作结果是否正确，可调用观察方法**toString()**, **size()**等等来从不同角度观察
  - 下断言，用观察方法检查结果
    - 观察方法上往往有一定的**post-condition**来检查输出结果：
    - 如果有个函数 **Tree* transformTree()**，如何进行基本检查，判断结果是正确的？什么是正常的**Tree**？

测试结果的分析

- 录制与比较
  - 直接录制数据结构的二进制内存数据

```
StructA* p = func();
//录制上次测试的结果
dump(p)
//下次测试可以和之前的结果进行比较
memcpy(pLastTime, p, sizeof(StructA);
```

  - 录制前次读写的文件

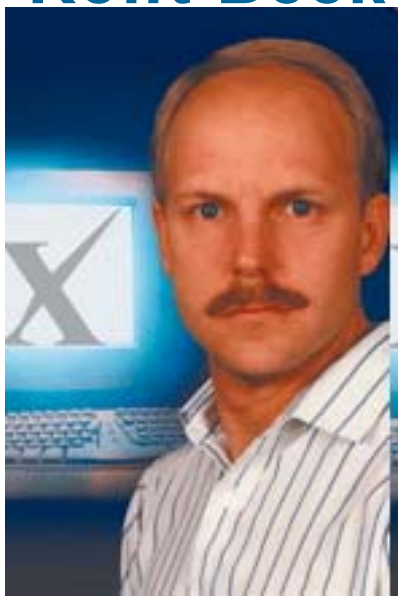
1.3 单元测试工具— JUnit

- Junit的诞生

Erich Gamma

设计模式开创者之一
Eclipse Java总设计师

Kent Beck



软件开发方法学大师
极限编程(eXtream Programming) 的创始人

1.3 单元测试工具— JUnit

- 回归测试框架(Regression testing framework)
- 支持的集成开发环境
  - Eclipse，IntelliJ IDEA, NetBeans, etc.
- 衍生推广
  - CppUnit
  - HttpUnit
  - StrutsTestCase
  - ……

史前单元测试

```
public class Add {
    public int add (int x, int y){
        return x + y;
    }
}

public class TestAdd {

    public static void main(String[] args) {

        Add a = new Add();
        int result = a.add(1, 2)

        if (result == 3)
            System.out.println("Ok!");
        else
            System.out.println("Error!");
    }
}
```

JUnit之前的单元测试

```
public class TestAdd {
    public static void main(String[] args) {
        Add a = new Add();
        int result1 = a.add(1, 2)
        if (result1 == 3)
            System.out.println("Ok!");
        else
            System.out.println("Error!");

        Add b = new Add();
        int result2 = b.add(2, 2)
        if (result2 == 4)
            System.out.println("Ok!");
        else
            System.out.println("Error!");

    }
}
```

不便控制是只执行其中一个测试，还是两个一起执行

一个类中写两个测试

JUnit之前的单元测试

```
public class TestAdd1 {
    public static void main(String[] args) {
        Add a = new Add();
        int result1 = a.add(1, 2)
        if (result1 == 3)
            System.out.println("Ok!");
        else
            System.out.println("Error!");
    }
}

public class TestAdd2 {
    public static void main(String[] args) {
        Add b = new Add();
        int result2 = b.add(2, 2)
        if (result2 == 4)
            System.out.println("Ok!");
        else
            System.out.println("Error!");
    }
}
```

代码很乱，注释屏蔽的代码满天飞

类太多。C++项目通常整个项目才一个main。多个测试要组织成多个项目

史前代码

```
  public static void main(String[] args) {
    try {
      ConstraintGenerator testgen = new ConstraintGenerator();
      Context ctx = testgen.createZ3Context();
      int time;
      long startTime,endTime;
      Collection<Input> inputs = null;

//      String[] Args = {"base64/test000013.smt2","base64/test000015.smt2","base64/test0
//
      String[] Args = {"chcon/test000002.smt2","chcon/test000004.smt2","chcon/test000010
//      int cNum = args.length;createVarValueAssignment
//      BoolExpr[]  exprs = new BoolExpr[cNum-1];
//      for(int i = 0 ; i < cNum - 1; i++){
//        exprs[i] = testgen.ParseSMT2(ctx, args[i]);
//      }
//      int normalPos = Integer.parseInt(args[cNum-1]);

      int cNum = Args.length;
      BoolExpr[]  exprs = new BoolExpr[cNum-1];
      for(int i = 0 ; i < cNum - 1; i++){
        exprs[i] = testgen.ParseSMT2(ctx, Args[i]);
      }

      int normalPos = Integer.parseInt(Args[cNum-1]);
      BoolExpr expr = ctx.mkOr(exprs);
      Map<Expr, Object> normal = testgen.solveFormula(ctx, exprs[normalPos]); //正常解

      Map<Expr, Expr> varassign = testgen.Assign(ctx, normal);
//
      int num = 631;
//      System.out.println("RANDOM");
//      startTime = System.currentTimeMillis();
//      inputs = testgen.RandomGen(ctx, expr, num);
//      endTime = System.currentTimeMillis();
//      System.out.println(endTime - startTime);

      System.out.println("RANDOM RELAX");
      startTime = System.currentTimeMillis();
      inputs = testgen.RandomGenWithRelax(ctx,varassign, expr, 150);
      endTime = System.currentTimeMillis();
      System.out.println("size: " + inputs.size());
      System.out.println(endTime - startTime);


//      for(Input i: inputs) {
//        System.out.println(testgen.evaluate(ctx, expr, i.assignment));
//      }
//      System.out.println(endTime - startTime);

//      String filename = "./test000002.smt2";
//      BoolExpr e = testgen.ParseSMT2(ctx, filename);
//      testgen.getConditions(e);
//      Map<Expr, Set<BoolExpr>> varGroup = testgen.getVarMap(expr);
//      System.out.println("Group size: " + varGroup.size());
//      for(Expr t : varGroup.keySet()) {
//        System.out.println(varGroup.get(t));
//      }
//
/****************** Variation by variable set and relax ******************/
```

```
/****************** Variation by variable set and relax ******************/

//      System.out.println("execute VARSET Relax");
//      startTime = System.currentTimeMillis();
//      inputs = testgen.varSetWithRelax(ctx,expr,varassign);
//      endTime = System.currentTimeMillis();
//      testgen.WriteRes(inputs, Method.VARSETRELAX.getName());
//      testgen.writeTime(endTime-startTime, Method.VARSETRELAX.getName());
//      System.out.println(inputs.size());
//      System.out.println(endTime - startTime);
//
//      /****************** Variation by variable set and relax***************
//
//
//      /******************** Varition by Sign and relax**********************
//
//      System.out.println("execute SIGN Relax");
//      startTime = System.currentTimeMillis();
//      inputs = testgen.signWithRelax(ctx,expr,varassign);
//      endTime = System.currentTimeMillis();
//      testgen.WriteRes(inputs, Method.SIGNRELAX.getName());
//      testgen.writeTime(endTime-startTime, Method.SIGNRELAX.getName());
//      System.out.println(inputs.size());
//      System.out.println(endTime - startTime);
////
//      /******************** Varition by Sign and relax**********************
//
//      /******************** Varition by Combination and relax***************
//
//      System.out.println("execution COMBINE Relax");
//      startTime = System.currentTimeMillis();
//      inputs = testgen.conditionComBineWithRelax(ctx,expr,varassign);
//      endTime = System.currentTimeMillis();
//      testgen.WriteRes(inputs, Method.COMBINERELAX.getName());
//      testgen.writeTime(endTime-startTime, Method.COMBINERELAX.getName());
//      System.out.println(inputs.size());
//      System.out.println(endTime - startTime);


    /******************** Varition by Combination and relax*****************
//      /****************** Variation by variable set ******************/
//
//      System.out.println("execute VARSET");
//      startTime = System.currentTimeMillis();
//      inputs = testgen.variationByVarSet(ctx, expr);
//      endTime = System.currentTimeMillis();
//      testgen.WriteRes(inputs, Method.VARSET.getName());
//      testgen.writeTime(endTime-startTime, Method.VARSET.getName());
//      System.out.println(inputs.size());
//      System.out.println(endTime - startTime);
//
//      /****************** Variation by variable set ******************/
//
//
//      /******************** Varition by Sign ***************************/
//
//      System.out.println("execute SIGN");
//      startTime = System.currentTimeMillis();
//      inputs = testgen.variationSign(ctx, expr);
//      endTime = System.currentTimeMillis();
//      testgen.WriteRes(inputs, Method.SIGN.getName());
//      testgen.writeTime(endTime-startTime, Method.SIGN.getName());
//      System.out.println(inputs.size());
```

JUnit — TestCase

JUnit 3.x版本的测试用例格式

```
import junit.framework.TestCase;

public class Test extends TestCase {
    public Test(String name) {
        super(name);
    }
    public void setUp() {}
    public void tearDown(){}
    public void testHelloWorld() {
        // test Code goes here
        String hello = "Hello";
        String world = "world";
        assertEquals("Hello world", hello+world);
    }
}
```

JUnit — TestCase

JUnit 4.x版本的测试用例格式

```
import static org.junit.Assert.*;

import org.junit.*;

public class CalculatorTest {
    private static Calculator calculator = new Calculator();

    @Before
    public void setUp() throws Exception {
        calculator.clear();
    }

    @Test
    public void testAdd() {
        calculator.add(2);
        calculator.add(3);
        assertEquals(5,calculator.getResult());
    }
}
```

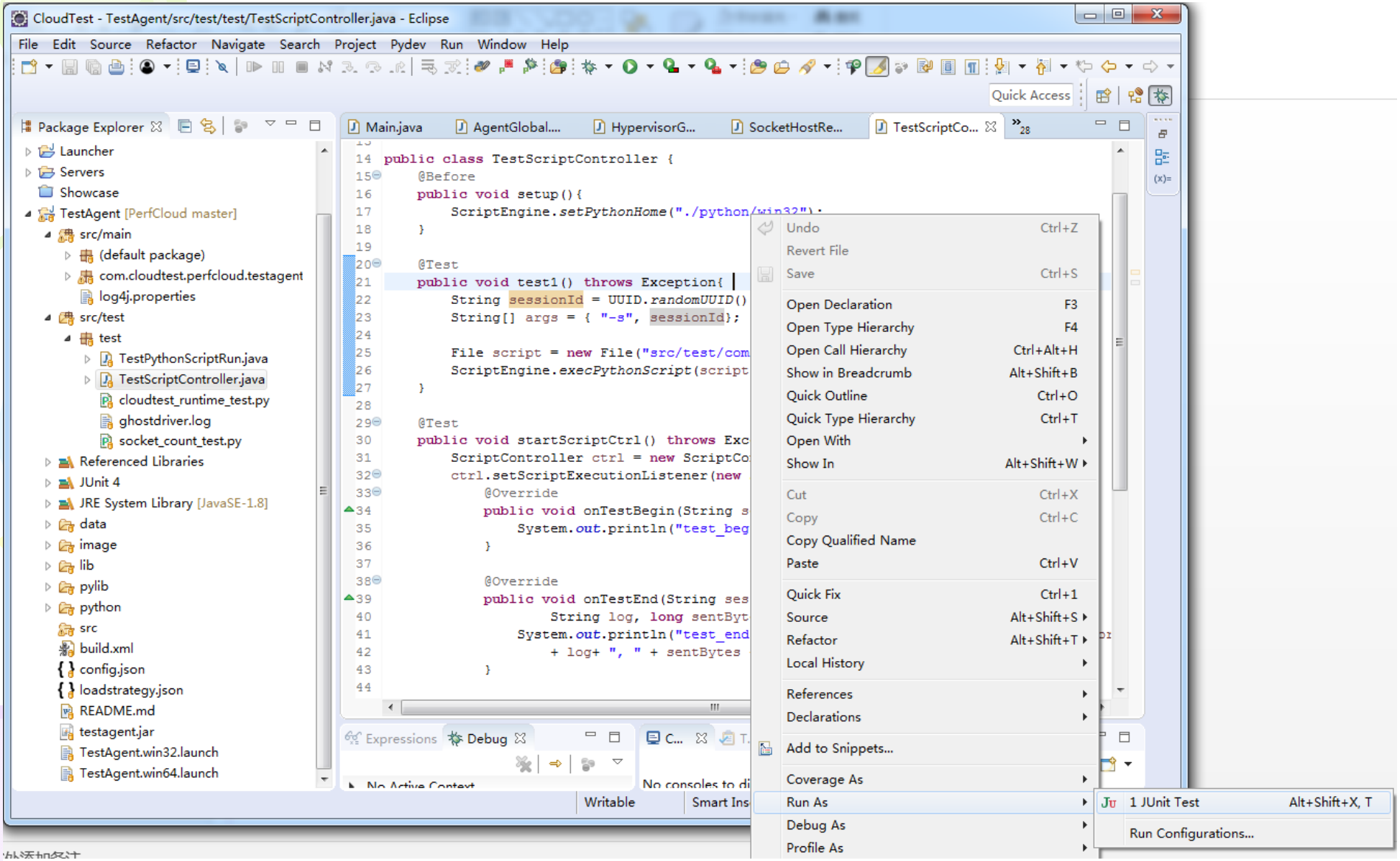
JUnit — TestCase

```
public class CalculatorTest {
    private static Calculator calculator = new Calculator();

    @Test
    public void testAdd1() {
        calculator.add(2);
        calculator.add(3);
        assertEquals(5,calculator.getResult());
    }
    @Test
    public void testAdd2() {
        calculator.add(3);
        calculator.add(4);
        assertEquals(7,calculator.getResult());
    }
}
```

现在一个类中可以写上任意多个测试了！

每个测试都可以单独像**main()**方法那样运行

没事不要再写**main()**方法，一个项目一般只有极少的**main()**


CloudTest - TestAgent/src/test/test/TestScriptController.java - Eclipse
File Edit Source Refactor Navigate Search Project Pydev Run Window Help
Quick Access
Package Explorer
Launcher
Servers
Showcase
TestAgent [PerfCloud master]
src/main
(default package)
com.cloudtest.perfcloud.testagent
log4j.properties
src/test
test
TestPythonScriptRun.java
TestScriptController.java
cloudtest_runtime_test.py
ghostdriver.log
socket_count_test.py
Referenced Libraries
JUnit 4
JRE System Library [JavaSE-1.8]
data
image
lib
pylib
python
src
build.xml
config.json
loadstrategy.json
README.md
testagent.jar
TestAgent.win32.launch
TestAgent.win64.launch
Main.java
AgentGlobal....
HypervisorG...
SocketHostRe...
TestScriptCo...
public class TestScriptController {
@Before
public void setup(){
ScriptEngine.setPythonHome("./python/win32");
}
@Test
public void test1() throws Exception{
String sessionId = UUID.randomUUID()
String[] args = { "-s", sessionId};
File script = new File("src/test/com
ScriptEngine.execPythonScript(script
}
@Test
public void startScriptCtrl() throws Exc
ScriptController ctrl = new ScriptCo
ctrl.setScriptExecutionListener(new
@Override
public void onTestBegin(String s
System.out.println("test_beg
}
@Override
public void onTestEnd(String ses
String log, long sentByt
System.out.println("test_end
+ log+ ", " + sentBytes
}
Expressions
Debug
No Active Context
No consoles to di
Writable
Smart Ins
Undo Ctrl+Z
Revert File
Save Ctrl+S
Open Declaration F3
Open Type Hierarchy F4
Open Call Hierarchy Ctrl+Alt+H
Show in Breadcrumb Alt+Shift+B
Quick Outline Ctrl+O
Quick Type Hierarchy Ctrl+T
Open With
Show In Alt+Shift+W
Cut Ctrl+X
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Quick Fix Ctrl+1
Source Alt+Shift+S
Refactor Alt+Shift+T
Local History
References
Declarations
Add to Snippets...
Coverage As
Run As
Debug As
Profile As
1 JUnit Test Alt+Shift+X, T
Run Configurations...

JUnit — TestSuite

```
import junit.framework.*;

public class AllTests {

    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }
    public static Test suite ( ) {
        TestSuite suite= new TestSuite("All JUnit Tests");
        suite.addTest(VectorTest.suite());
        suite.addTest(new Test.class);
        return suite;
    }
}
```

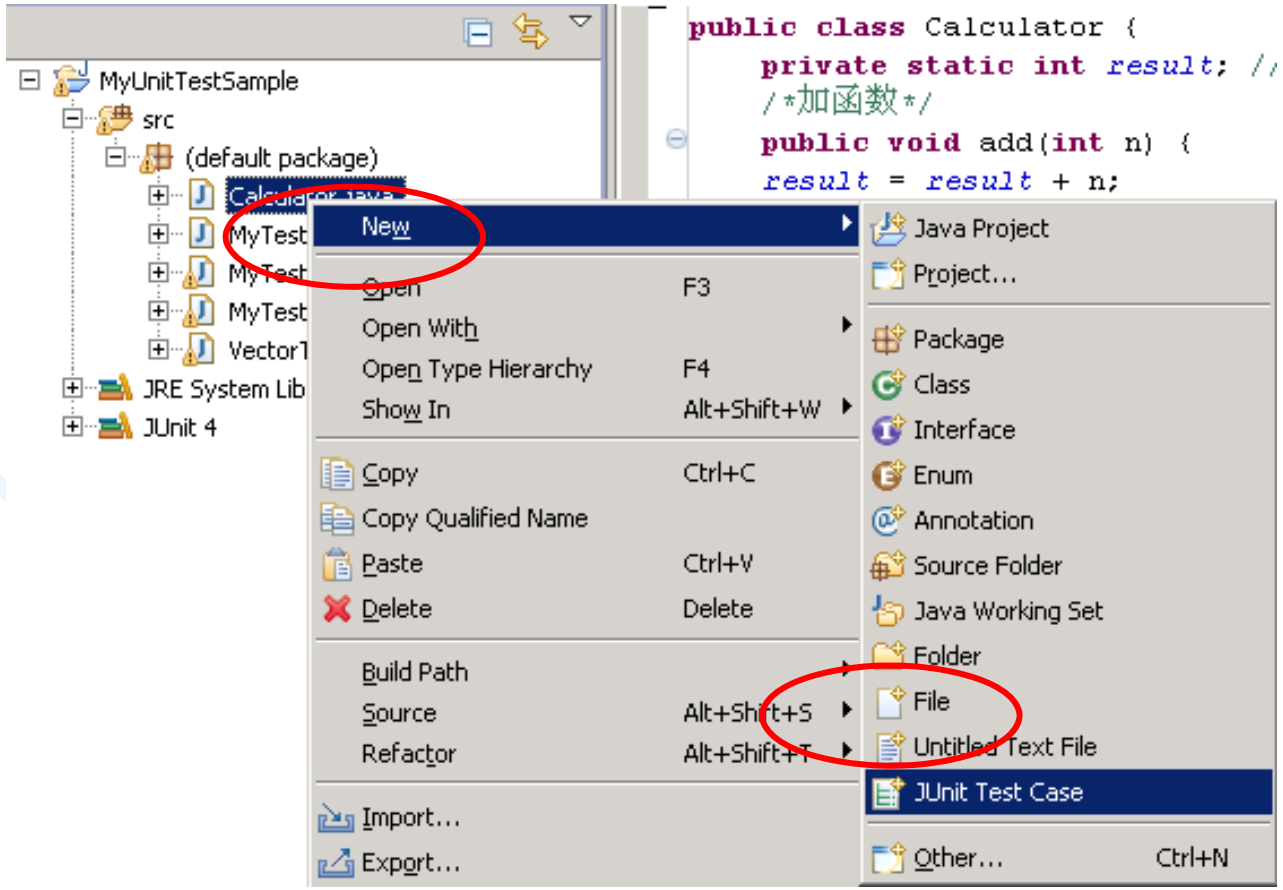
还可以灵活组装成测试集批量运行

用Junit重构史前代码

```
@Test
public void testNegateCondition() throws Exception {
  String baseDir = "./test/chcon";
  String[] smt2Files = CHCON_SMT2;

  test("chcon", baseDir, smt2Files, Method.NC.toString());
}

@Test
public void testModifiedNegateCondition() throws Exception {
  String baseDir = "./test/chcon";
  String[] smt2Files = CHCON_SMT2;

  test("chcon", baseDir, smt2Files, Method.MNC.toString());
}

@Test
public void testModifiedNegateConditionMin() throws Exception {
  String baseDir = "./test/chcon";
  String[] smt2Files = CHCON_SMT2;

  test("chcon", baseDir, smt2Files, Method.MNC.toString() + "_MIN");
}

@Test
public void testCNF() throws Exception {
  String baseDir = "./test/chcon";
  String[] smt2Files = CHCON_SMT2;

  test("chcon", baseDir, smt2Files, Method.NCNF.toString());
}

@Test
public void testCNFMin() throws Exception {
  String baseDir = "./test/chcon";
  String[] smt2Files = CHCON_SMT2;

  test("chcon", baseDir, smt2Files, Method.NCNF.toString() + "_MIN");
}
```

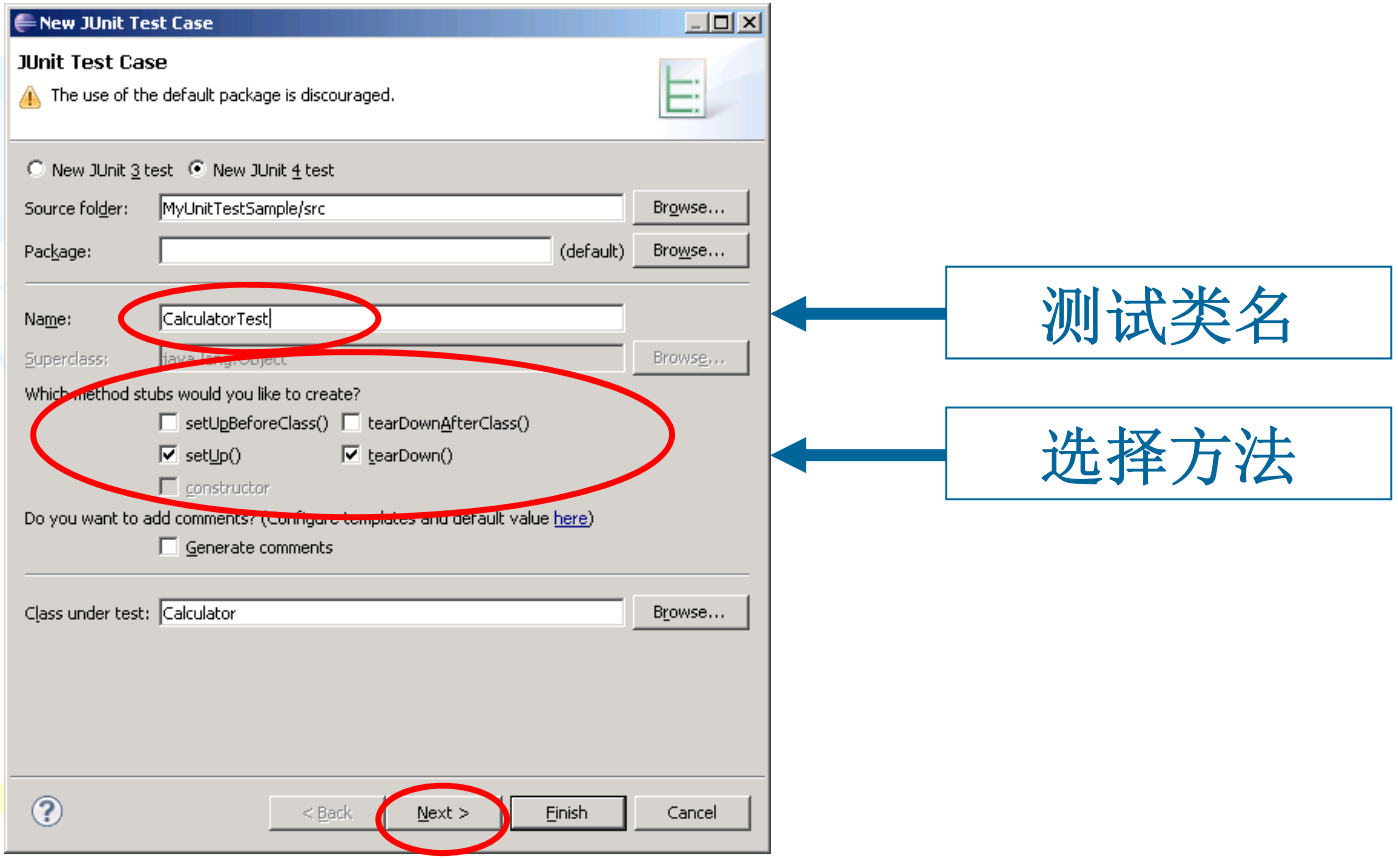
Junit的使用

- 自动生成测试框架（批量生成更方便）
  - **Calculator.java→右键菜单→New→JUnit Test Case**



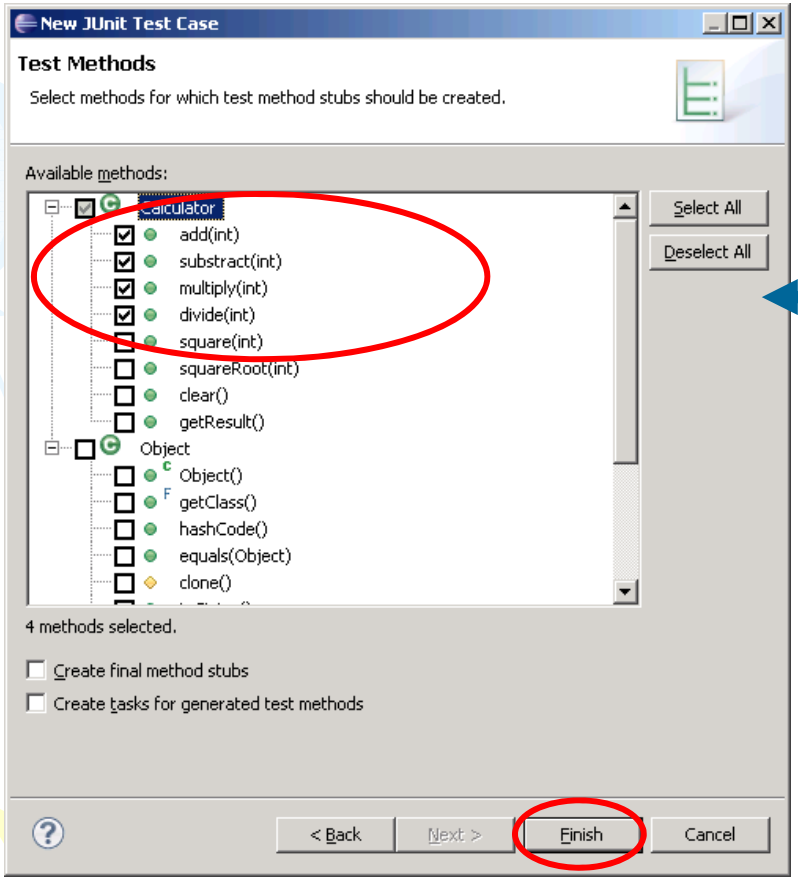
Junit的使用

- 自动生成测试框架



Junit的使用

- 自动生成测试框架

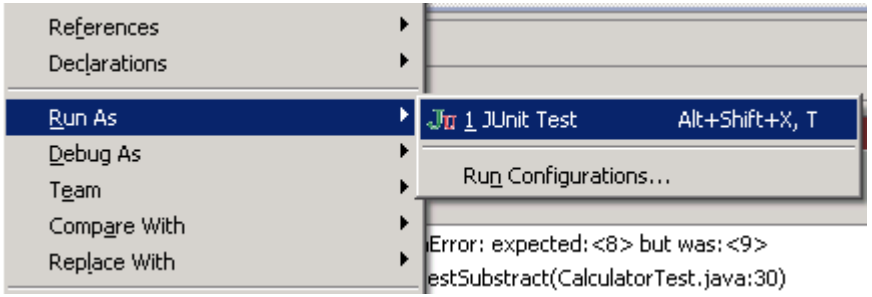


选择待测方法

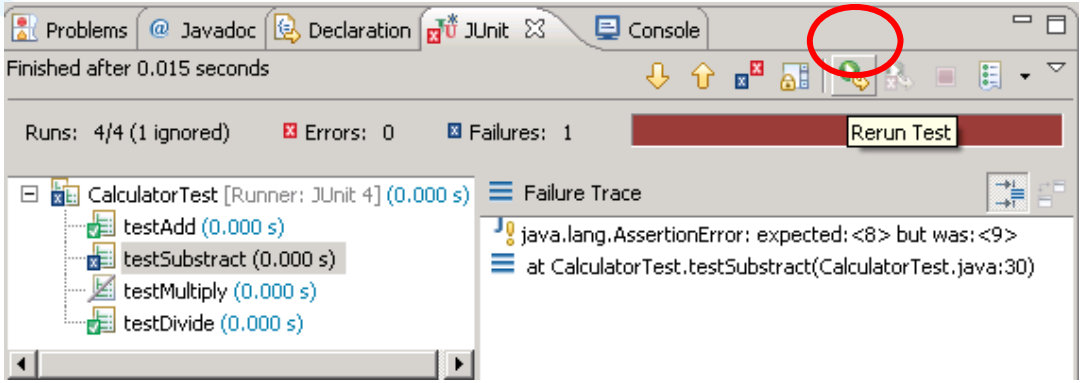
Junit的使用

- 执行CalculatorTest测试

方法1：右键菜单→Run As→JUnit Test

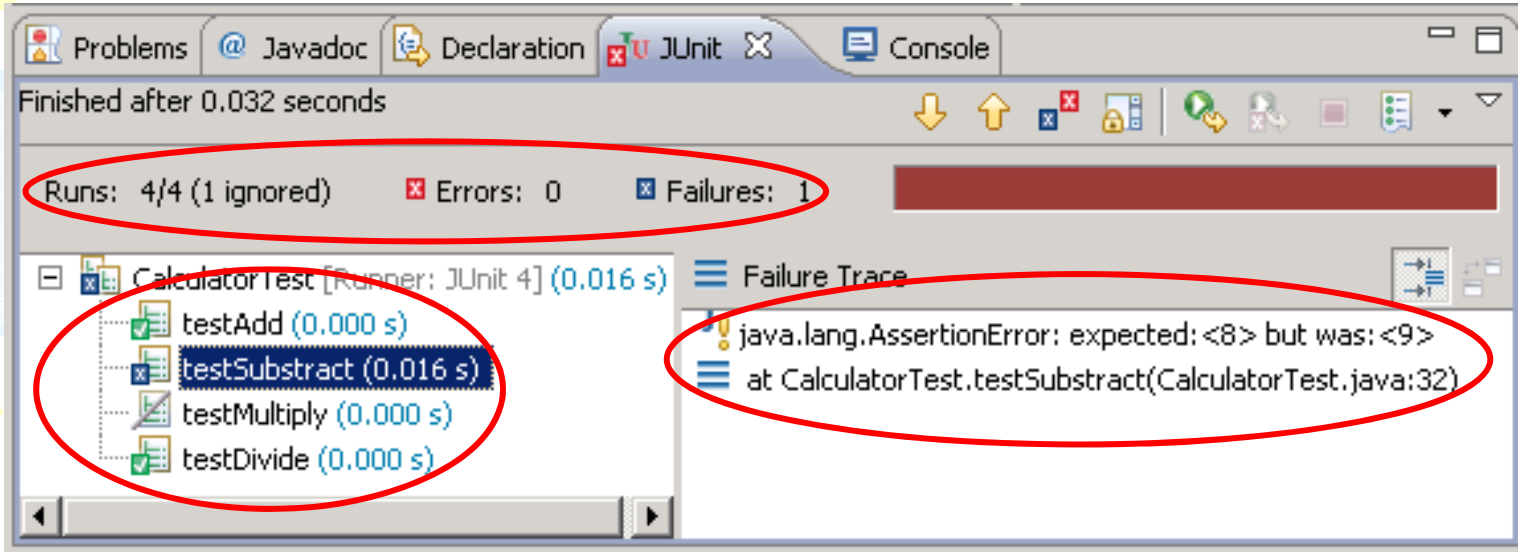


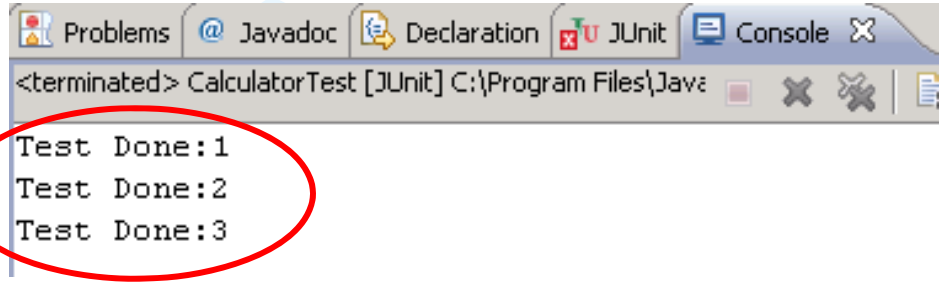
方法2：JUnit插件



Junit的使用

- CalculatorTest测试结果





JUnit的价值

- 为程序增加了多个入口
  - 除了**main()**，每个单元测试也可独立执行
- 一种清晰、标准的单元测试框架
- 测试用例可重用
- 多种方式展示测试结果，而且还可以扩展
- 提供了单元测试用例成批运行的功能
- 超轻量级，简单易用

1.4 单元测试的优点

- 它是一种确认(Validation)行为
- 它是一种设计(Design)行为
  - 先写单元测试明确接口
- 它是一种编写文档(Documentation)的行为
  - 单元测试告诉我们模块该怎么调用
- 它具有回归性
  - 一次写完后面可以反复执行

单元测试可以使开发过程自然而然地变得“敏捷”

- 很方便的检查修改后代码的质量
- 尽早发现设计问题

1.5 单元测试的局限性

1. 只验证单元自身的功能，不能捕获系统范围的错误
   - 系统错误：集成错误、性能问题等

2. 被测模块现实中可能接收的复杂或少见输入情况难以预料

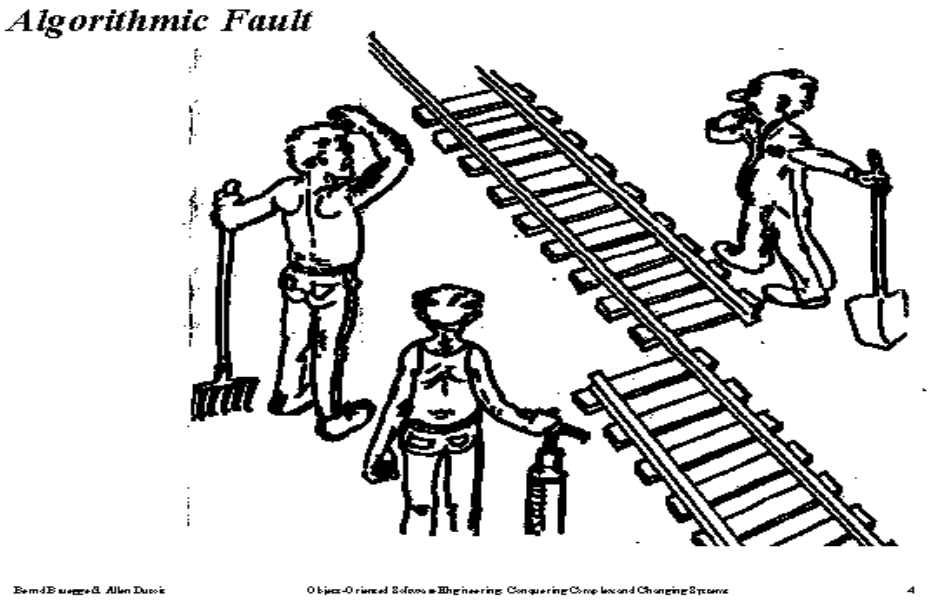
2 集成测试

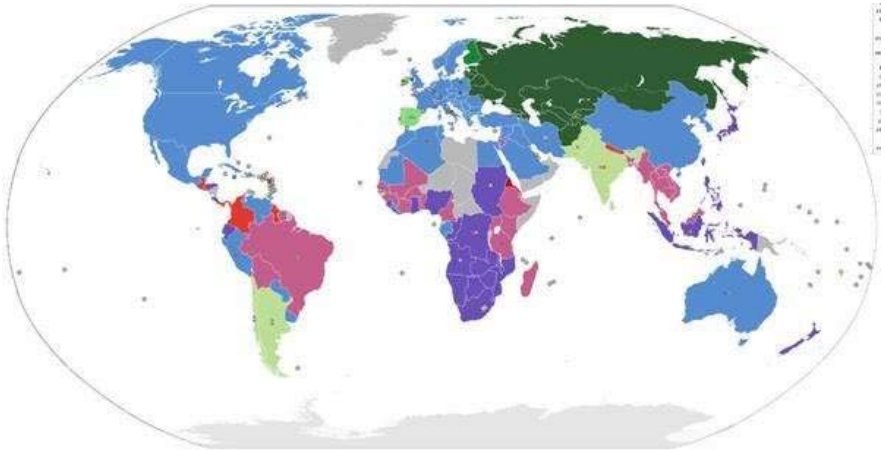
Integration Testing

2.1 为什么要集成测试

- **1999,火星气象轨道卫星消失**
  - 洛克希德.马丁　　英制加速度
  - 喷气推进实验室　公制加速度

  各自合格，但接口不一致





车轨不同的地区

2.2 集成测试的概念

- **集成测试的概念**

  又称组装测试、联合测试、子系统测试或部件测试。

  在单元测试的基础上，将所有模块按照设计方案（如结构图）集成为子系统或系统，进行集成测试。

- **与系统测试的差别**
  - 层次更低
  - 输入、输出边界不同
    - 单元测试的输入由程序给入
    - 集成测试的输入由程序+设备给出
    - 系统测试的输入由设备给入（键盘、鼠标等）

与单元和系统测试的区别

测试类型	对象	目的	测试依据	测试方法
单元测试	局部模块	消除局部模块内的逻辑和功能上的错误和缺陷	模块逻辑设计，外部说明	白盒为主
集成测试	模块间的集成和调用关系	找出与设计相关的程序结构、模块调用关系、模块间接口方面的问题	结构设计	白盒+黑盒
系统测试	整个系统(包括硬件等)	对整个系统进行一系列的整体、有效性测试	系统结构设计、需求规格说明等	黑盒

2.3 集成测试流程

集成测试计划	集成测试分析与设计	集成测试实现	集成测试执行	集成测试评估
软件体系结构初步分析	集成测试对象分析	集成测试工具开发	建立集成测试环境	集成测试数据分析
关键特性分析	集成策略选择	集成测试代码开发	执行集成测试	集成测试评估
工作量估计	集成测试工具选择和设计	集成测试用例开发	测试结果记录	
资源安排	集成测试代码设计			
进度安排	集成测试用例设计			

2.3.1 制定集成测试计划

- 集成测试计划应在概要设计阶段完成

- 集成测试计划的制定依据：
  - ✧需求规格说明书；
  - ✧概要设计说明书；
  - ✧产品开发计划书

集成测试计划的内容有：

- 确定集成测试对象和测试范围；
- 确定集成测试阶段性时间进度；
- 确定测试角色和分工；
- 考虑外部技术支援的力度和深度，以及相关培训安排；
- 初步考虑测试环境和所需资源；
- 集成测试活动风险分析和应对；
- 定义测试完成标准；

哪些特性比较关键？包含缺陷的风险大？缺陷影响大？——需要集成测试

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

2.3.2 集成测试分析和设计

- 主要任务是制定集成测试大纲（测试方案）

  集成测试大纲规定了今后的集成测试内容和测试方法。

  所有集成测试均在该大纲的框架下进行。

- 具体工作内容：
  - ✧ 确定测试需求：在哪个层面进行测试，测试哪些内容？
  - ✧ 确定集成策略
  - ✧ 评估测试风险
  - ✧ 确定测试方法、优先级
  - ✧ 确定集成测试代码设计
  - ✧ 集成测试用例设计
  - ✧ 集成测试工具和资源

集成测试计划	集成测试分析与设计	集成测试实现	集成测试执行	集成测试评估

确定测试需求

1. 分析体系结构，确定集成测试层次

   常见集成层次：类 / 函数级、组件级、进程级、...

2. 确定集成测试的参与模块划分：对哪些模块进行集成测试？

   - 重要：被集成的几个模块关系紧密，能完成某种重要功能
   - 可测、易测：
     - 有明显的输入输出。
     - 外围模块与集成模块之间没有太多、太频繁的调用关系。
     - 模拟外围模块发往被集成模块的消息容易构造、修改
     - 外围模块发往被测模块的消息能够模拟大部分实际情况

3. 集成测试接口的确定（分析模块间的交互途径）

   - 子系统内模块间接口；
   - 子系统（进程）间接口；
   - 系统与操作系统的接口；
   - 系统与硬件的接口；
   - 系统与第三方软件的接口

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

确定测试需求

- 常见接口类型
  - √ 函数接口
  - √ 类接口：类接口的测试一般可以通过继承、参数类、不同类方法调用等策略来实现。
  - √ 组件接口
    - √ **CORBA接口**
    - √ **Java RMI和RMI-IIOP**
    - √ **Microsoft COM/DCOM/COM+**
  - √ **Web Service接口: SOAP服务, RESTful API，非标准HTTP接口**
  - √ 进程间接口：中间文件、数据库、**socket** 、消息队列 、管道、信号、信号量、共享内存、远程过程调用等。

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

集成测试需求简单示例

模块	集成目标	耦合点	.....
A组件	B组件	XXX函数调用	

确定集成策略

- 对于具体要测试的目标，怎么来实施测试？
  - 集成测试涉及集成顺序、编制辅助代码等工作，需要用适当的方式来开展测试

- 一个好的集成策略应该具有以下特点：
  - ✧ 使被测对象能够得到比较充分的测试，尤其是其包含的关键特性
  - ✧ 操作难度恰当，需要做的辅助工作量恰当，缺陷易诊断
  - ✧ 整体工作量对于投入测试的资源来说大致相当，参加测试的人力、环境、时间等资源能够得到充分利用

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

集成测试代码与用例设计

■ 集成测试代码设计

常见集成测试代码包括：

- ✧ 为测试方便添加的代码，如**print**打印、断言等；
- ✧ 驱动模块、桩模块；
- ✧ 测试执行框架代码。

■ 集成测试用例设计

- – 输入数据设计
- – 输出结果分析：明确正确的测试结果内容

（集成测试中，测试用例常常由代码构成）

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

2.3.3 集成测试实现

- 该阶段主要工作是依据测试大纲和详细设计分析开发测试用例、测试脚本及相应驱动程序、桩程序，如果有必要，则要开发测试工具。

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

2.3.4 集成测试执行

- 搭建测试环境
- 执行测试用例
- 测试异常记录
- 编写测试报告

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

2.3.5 集成测试评估

- 集成测试数据分析
  - ✧缺陷趋势、缺陷密度、缺陷分布、工作量分布
- 设计与代码质量评估
  - ✧规模复杂度、结构复杂度、设计复杂度

集成测试 计划	集成测试 分析与设计	集成测试 实现	集成测试 执行	集成测试 评估

2.4 集成策略

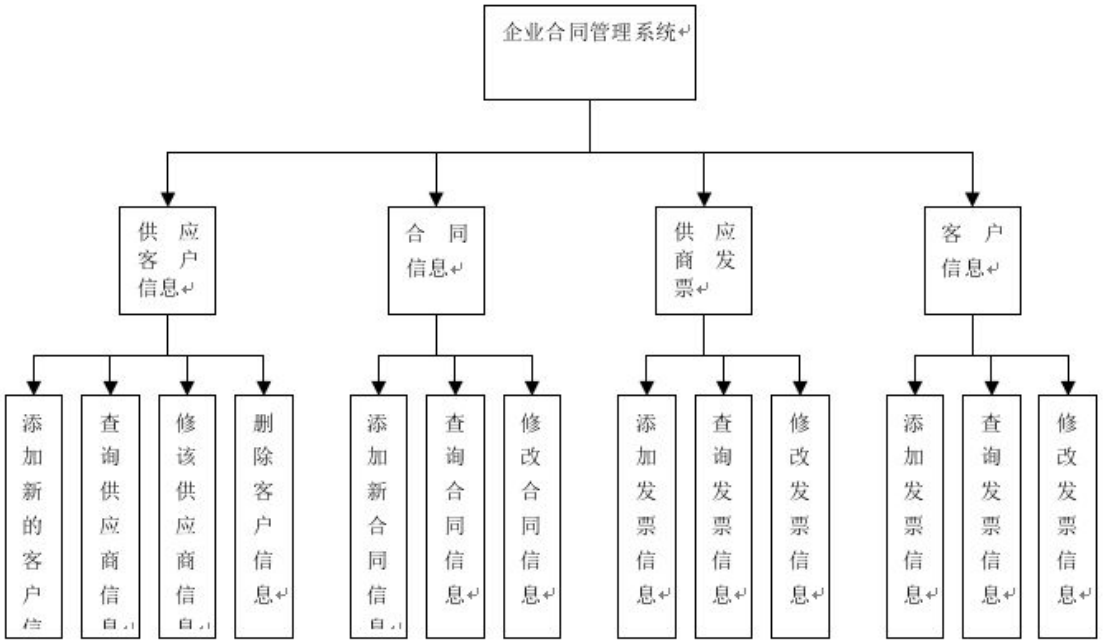
- 即软件模块集成（组装）的方式、方法
  - 非渐增式测试策略：先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序。
  - 渐增式测试策略：把下一个要测试的模块同已经测试好的模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合进来测试。

集成的依据

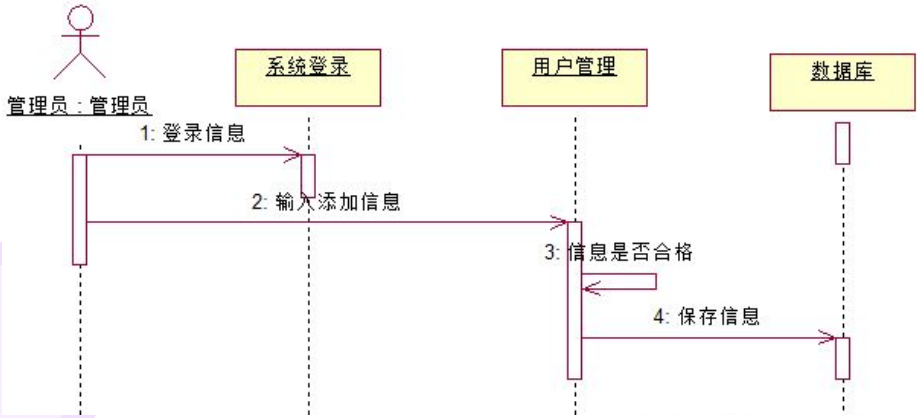
- 功能分解层次树
- 函数调用图
- 类关系图
- 进程调用图
- 分层模型：e.g. OSI网络协议七层
- 网络服务调用图
- UML状态图
- UML协作图
- UML顺序图

概要设计中通常会给出系统的构架和集成方法，集成测试以此为参照

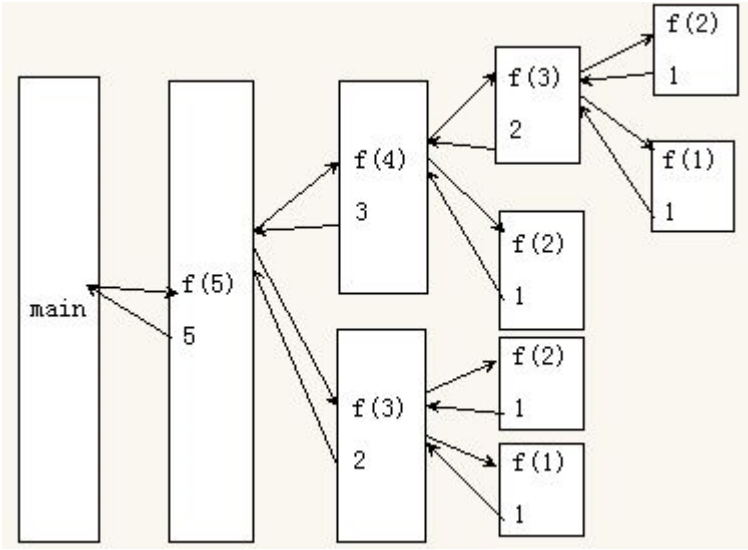
不同模型对应不同层次的集成



功能分解树(按功能单元集成)

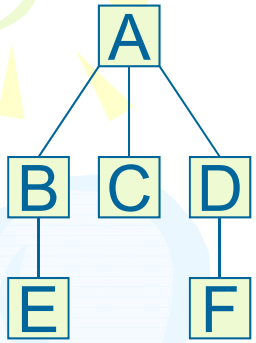


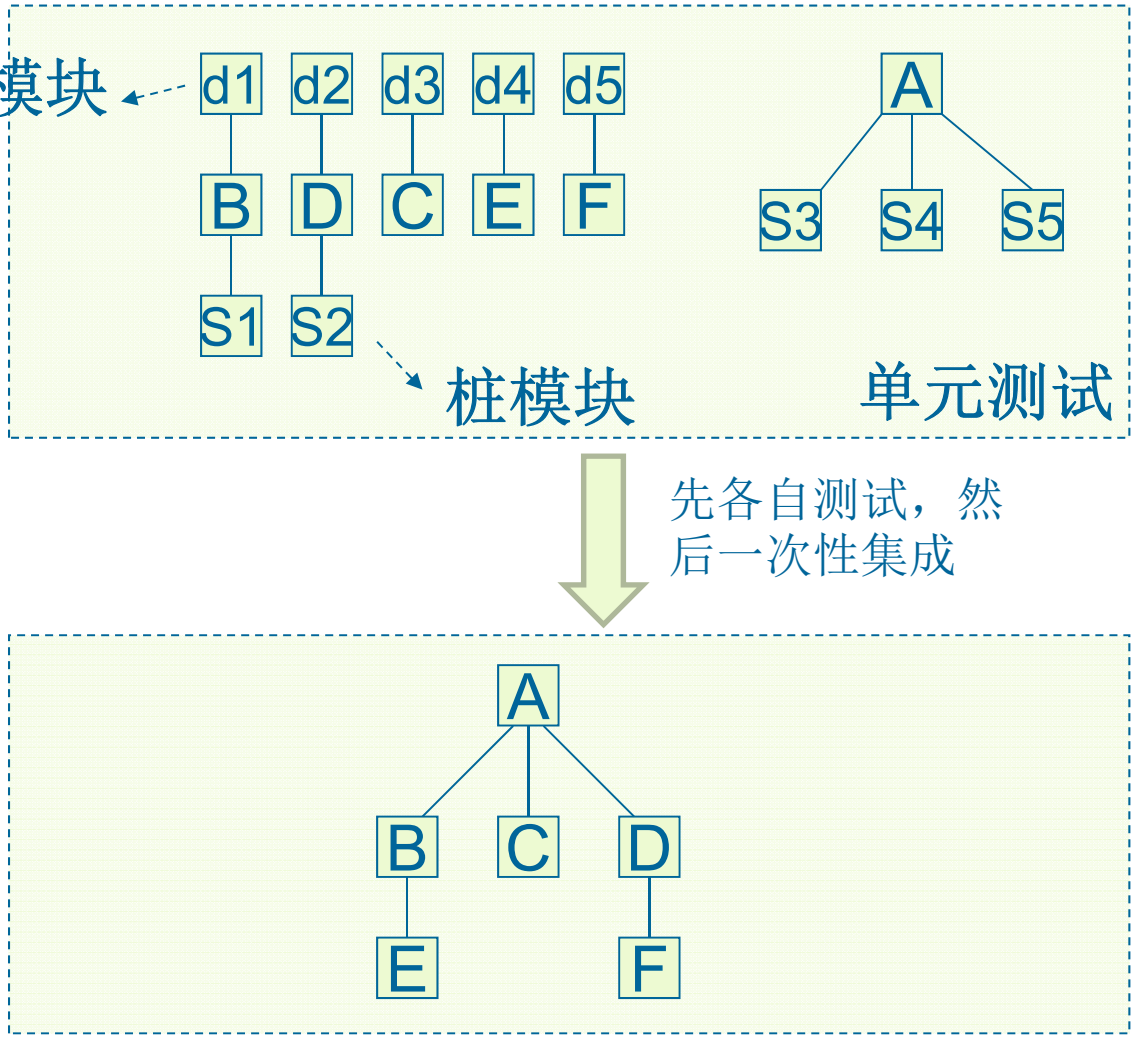
UML顺序图(按对象间关系集成）



函数调用图

2.3.1 非渐增式测试策略—大棒集成方法 *(Big-bang Integration)*





因为所有模块是一次集成的，所以很难确定出错的真正位置、错误的原因。

适合在规模较小的应用系统中使用。

驱动模块与桩模块

- 测试某一模块时，调用它的模块或它所依赖的模块可能还未实现好，需要有其它模块来模拟：
  - 驱动模块（driver）
    - 用以模拟被测模块的上级模块。

      把相关的数据传送给被测模块，启动被测模块，并获得相应的结果。
  - 桩模块（stub）
    - 用以模拟被测模块工作过程中所调用的模块。

      由被测模块调用，一般只进行很少的数据处理，例如打印信息等，以便检验被测模块与其下级模块的接口联通性
    - 面向对象中有时也称Mock Object，其它领域有称模拟器、仿真器

2.3.2 渐增式测试策略

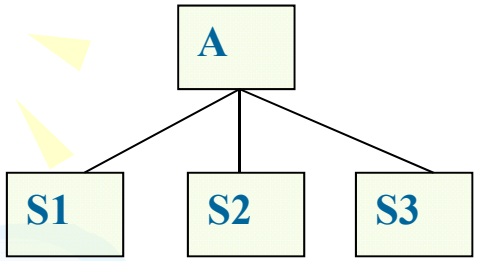
- 逐步组装成为要求的软件系统，在组装的过程中边连接边测试，以发现连接过程中产生的问题。
- 优点：相对非渐增式策略，可较早发现模块间的接口错误；发现问题也易于定位。
- 缺点：测试周期比较长，可以同时投入的人力物力受限。
- 三种方式：
  - ✧自顶向下（**Top Down**）
  - ✧自底向上（**Bottom Up**）
  - ✧混合式集成

A.自顶向下

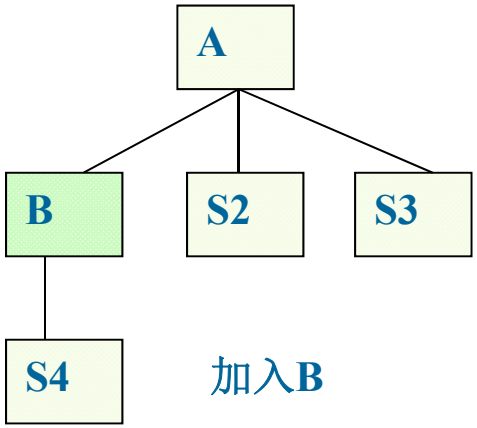
将模块按系统的结构，沿控制层次自顶向下进行集成

- 步骤：
  - ✧对主控模块进行测试，用桩模块代替所有直接附属于主控模块的模块
  - ✧根据选定的结合策略（深度/宽度优先），每次用一个实际模块代替一个桩模块（新结合进来的模块往往又需要新的桩模块）
  - ✧在结合下一个模块的同时进行测试
  - ✧为保证加入模块不引进新错误，必要时进行回归测试
  - ✧从第2步开始重复进行上述过程，直至所有的模块都已集成到系统中。

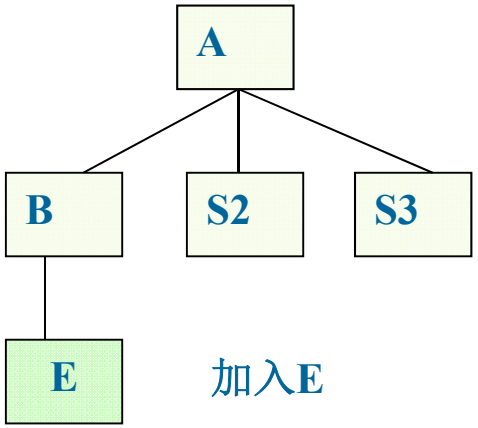
自顶向下—深度优先



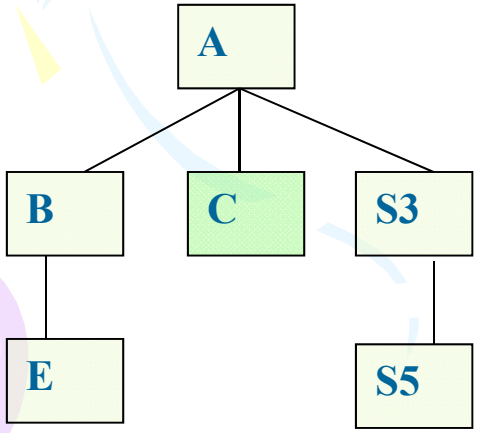
测试A



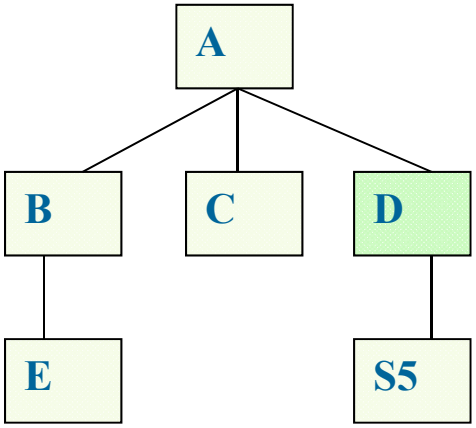
加入B



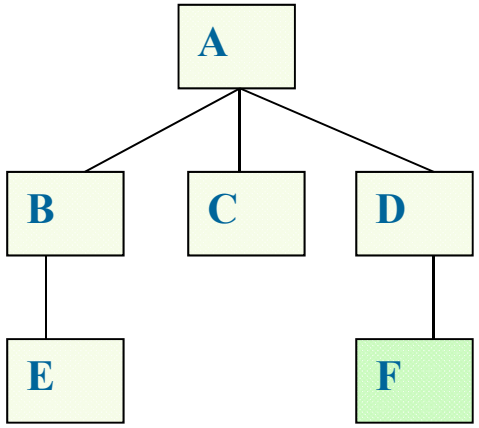
加入E



加入C



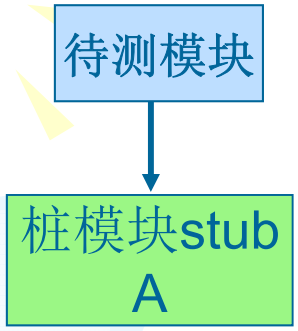
加入D



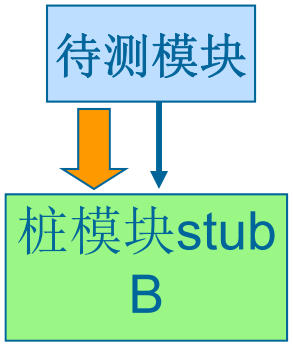
加入F

自顶向下—桩模块

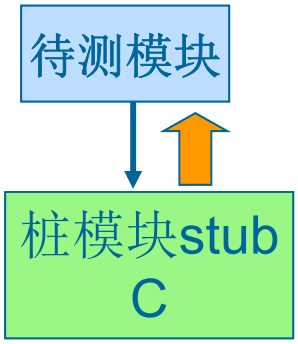
- 桩模块的编写有如下几种常见选择：



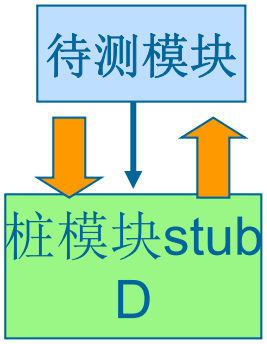
显示跟踪信息
(表明已联通)



显示传递
的参数信息



从一个表或
外部文件等
返回一个值



进行一项表查询
以根据输入参数
返回输出参数


表示传递的数据消息

为准确地实施测试，应当让桩模块正确而有效地模拟子模块的功能和合理的接口，不能是只包含返回语句或只显示该模块已调用信息、不执行任何功能的哑模块。

自顶向下—桩模块

- 如果不能使桩模块正确地向上传递有用的信息，可以采用以下解决办法：

（1）将一些测试推迟到桩模块用实际模块替代之后进行

（2）进一步开发能模拟实际模块功能的桩模块；

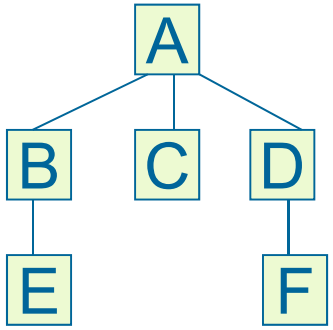
（3）自底向上集成和测试软件

B. 自底向上

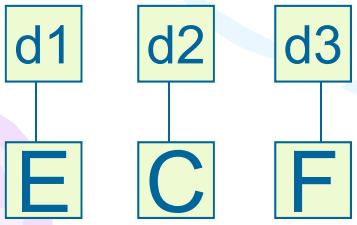
- 从模块结构的最底层开始集成和测试
- 具体策略：
  1. 由驱动模块控制最底层模块的并行测试，也可以把最底层模块组合成实现某一特定软件功能的簇，由驱动模块控制它进行测试。
  2. 用实际模块代替驱动模块，与它已测试的直属子模块集成为子系统。
  3. 为子系统配备驱动模块，进行新的测试。
  4. 判断是否已集成到达主模块，否则执行（2）

自底向上

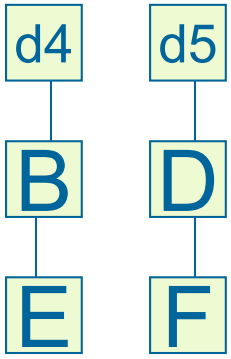
从模块结构的最底层的模块开始组装和测试



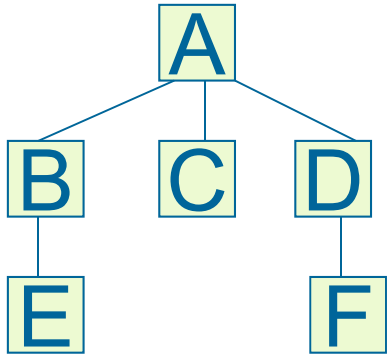
如何测试？



第一步



第二步



第三步

自底向上

- 自底向上进行集成和测试时，需要为所测模块或子系统编制相应的驱动模块。常见的几种类型的驱动模块如下：

调用从属模块

从外部表（文件）中传递参数

显示参数

兼有驱动程序B、C的功能

⇩⇧ 表示传送的参数信息

随着集成层次的向上移动，驱动模块将大为减少。

自顶向下和自底向上法的比较

	自顶向下法	自底向上法
优点	✧不需要测试驱动程序 ✧程序主体形成较早，**较早实现并验证系统的主要功能** ✧能在早期发现上层模块的接口错误	✧不需要桩模块，建立驱动模块一般比建立桩模块容易 ✧可以把最容易出问题的部分在早期解决（底层易出问题）。 ✧可以实施多个模块的并行测试，提高测试效率
缺点	✧**需要桩模块**：要使桩模块能够模拟实际子模块的功能可能十分困难 ✧**复杂、容易出问题的模块一般在底层**，到集成后期才遇到这些模块，一旦发现问题将导致过多的回归测试 ✧这种方法在**早期不能充分展开人力**	✧ 主体较晚才形成，不能较早验证程序主要功能

几种集成方法性能的比较

	自底向上	自顶向下	大棒
集成	早	早	晚
基本程序 能工作时间	晚	早	晚
需要驱动程序	是	否	是
需要桩程序	否	是	是
工作并行性	中	低	高
特殊路径测试	容易	难	容易
计划与控制	容易	难	容易

也可以根据实际情况采用混合的策略

集成策略的选择

- 在集成测试时，应当确定关键模块，对这些关键模块及早进行测试

- 关键模块一般具有以下几种特征之一
  1.满足某些特殊软件需求
  2.在程序的模块结构中位于较高的层次
  (高层控制模块)
  3.较复杂、较易发生错误
  4.有明显定义的性能要求

集成测试核心步骤总结

- 分析软件体系结构，确定集成测试的层次和集成测试的依据，如函数调用图、功能分解图等
- 确定关键模块：时间有限，重点是对关键模块进行集成测试
- 确定集成策略：自顶向下、自底向上还是混合方式
- 确定具体集成测试需求：都有哪些接口要测，逐个列成表格
- 设计集成测试的相关代码和工具
- 执行测试并分析结果

持续集成
(Continuous Integration)

- 在代码演进过程中，不断进行build，构造出完整可运行的软件，随时发现编译和链接失败问题
- 在build的同时，执行一组伴随build之后的冒烟测试（版本验证测试）

随时可能会有新版本的获取需求。
且错误不及时发现越往后越难修复