**实验7基于Junit的测试**

### 实验目的

### 了解Junit框架，掌握如何使用Junit框架来设计和编写可执行的测试用例。

### 2. 知识要点

在团队式的系统开发中，每个人都可能会使用别人所写代码中的类和方法，反之亦然。在系统开发过程中，代码会频繁发生变化，会不断引入新的bug，这对每个开发人员的工作都会产生影响。为了确保整个系统的代码质量，在类和方法层次实施自动化测试不仅能够及时自动发现bug，还能够确保每个开发人员都在编写代码时及时更新或补充相应的测试代码，形成良性的质量保证思维。开源的java单元测试框架JUnit能大大简化进行单元测试所要做的工作。在1997年，由 Erich Gamma 和 Kent Beck 提出。如今Junit已经在几乎所有基于Java的软件项目中得到了应用，为保证代码质量做出了重要贡献。Junit框架有如下特点：

1) 提供的API可以让你写出测试结果明确的可重用单元测试用例

2) 提供了三种方式来显示你的测试结果，而且还可以扩展

3) 提供了单元测试用例成批运行的功能

4) 整个框架设计良好，易扩展

Junit框架包括以下内容：

1）TestCase:测试用例。

2）TestSuite:测试用例的集合，可容纳多个测试用例，也称作测试包或测试集。

3）TestResult:测试结果的描述与记录

4）TestListener:测试过程中的事件监听

5）TestFailure:每一个测试方法所发生的与预期不一致状况的描述，称其为测试失败元素。

6）AssertionFailError:Junit Framework中的出错异常。

对不同性质的被测对象，如Class，Jsp，Servlet，EJB等，Junit有不同的使用技巧。本次实验仅以Class测试为例。感兴趣的同学可以在课下学习其它相关内容。

##### 2.1 Junit安装和设置

Eclipse IDE中已经事先集成了Junit插件。因此在Eclipse中可以直接运行测试程序，不需要进行额外的安装。但是，仍然需要先进行一些必要的设置。最主要的是进行“类路径变量”的设置。

具体步骤：

1）窗口->首选项

2）选中“构建路径->类路径变量”

3）选中设置JUNIT\_HOME

4) 为了DEBUG的需要，还需要添加Junit的源代码路径，JUNIT\_SRC\_HOME

这样，你的系统中就可以使用JUnit编写单元测试代码了

##### 2.2 利用JUnit进行测试

首先新建SampleCalculator类，这是一个能够简单实现加减乘除、平方、开方的计算器类，然后对这些功能进行单元测试。这个类并不是很完美，预留了一些Bug，在注释中都有说明。该类代码如下：

**public** **class** SampleCalculator{

**private** **static** **int** *result*; // 静态变量，用于存储运行结果

**public** **void** add(**int** n) {

*result* = *result* + n;

}

**public** **void** substract(**int** n) {

*result* = *result* - 1; //Bug: 正确的应该是 result =result-n

}

**public** **void** multiply(**int** n) {

} // 此方法尚未写好

**public** **void** divide(**int** n) {

*result* = *result* / n;//Bug

}

**public** **void** square(**int** n) {

*result* = n \* n;

}

**public** **void** squareRoot(**double** n) {

**for**(;;) ; //死循环

}

**public** **void** clear() { // 将结果清零

*result* = 0;

}

**public** **int** getResult() {

**return** *result*;

}

}

将上面的代码编译通过。写一个单元测试用例（在该类的名称上【右键】-》【new】-【Junit Test case】）

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

**import org.junit.After;**

**import org.junit.Before;**

**import org.junit.Ignore;**

**import org.junit.Test;**

**public class SampleTest {//SampleTest是一个测试系统**

**private static SampleCalculator calculator=new SampleCalculator(); //构造一个被测对象**

**@Before**

**public void setUp() throws Exception {**

**calculator.clear();**

**}**

**@After**

**public void tearDown() throws Exception {**

**calculator.getResult();**

**}**

**@Test**

**public void testAdd() {**

**calculator.add(2);**

**calculator.add(3);**

**assertEquals(5,calculator.getResult());**

**}**

**@Test**

**public void testSubstract() {**

**calculator.add(10);**

**calculator.substract(2);**

**assertEquals(8,calculator.getResult());**

**}**

**@Ignore("testMuliply() not completed!")**

**@Test**

**public void testMultiply() {**

**}**

**@Test**

**public void testDivide() {**

**calculator.add(8);**

**calculator.divide(2);**

**assertEquals(4,calculator.getResult());}**

**@Test(expected = ArithmeticException.class)**

**public void divideByZero() {**

**calculator.add(0);**

**calculator.divide(0);**

**}**

**@Test**

**public void testSquare() {**

**calculator.add(2);**

**calculator.square(2);**

**assertEquals(4,calculator.getResult());}**

**@Test(timeout=10)**

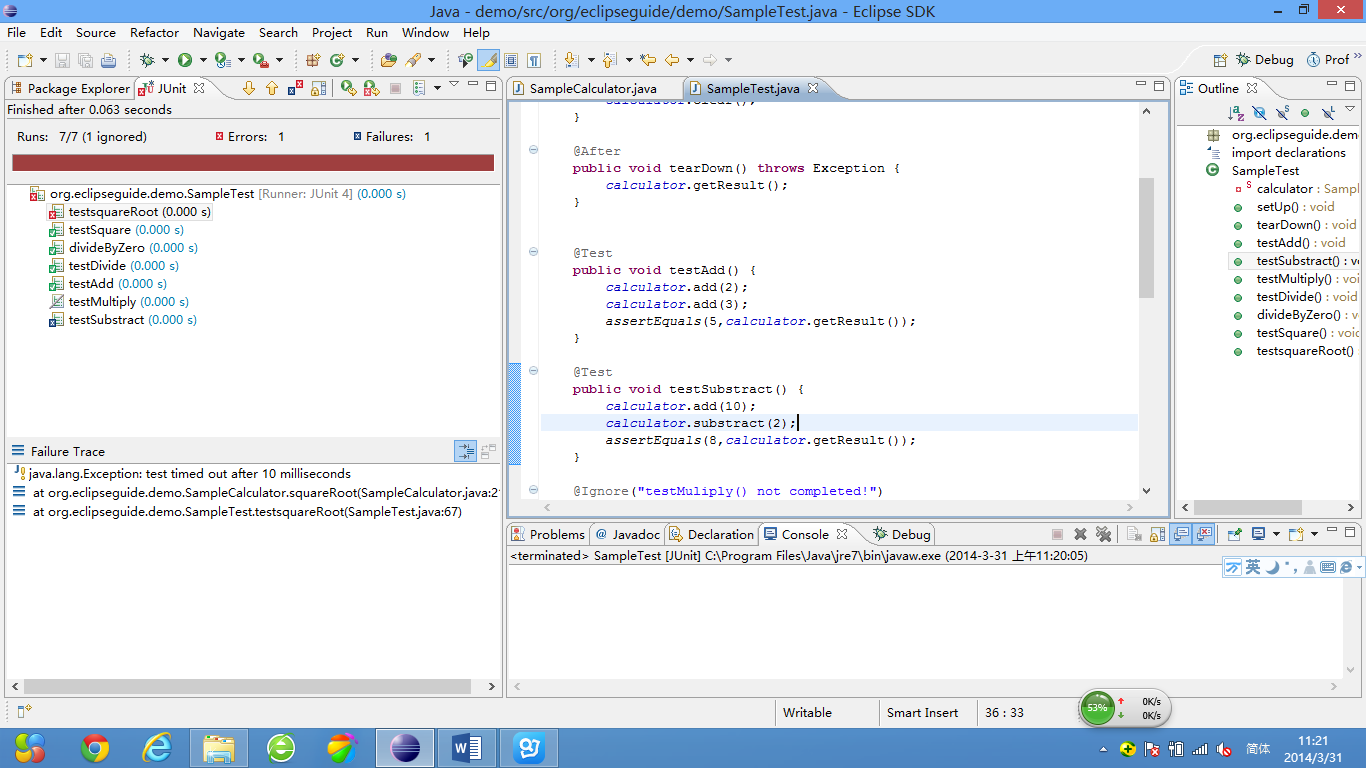
**public void testsquareRoot() {**

**calculator.add(16);**

**calculator.squareRoot(16);**

**assertEquals(4,calculator.getResult());}**

**}**



上图中，左侧的面板给出了各个测试方法的执行情况。进度条是红颜色表示发现错误，具体的测试结果在进度条上面有表示“共进行了7个测试，其中1个测试被忽略，1个测试失败”，存在1个错误。

**1)解释Junit4中的各元数据：**

**@Before：**使用了该元数据的方法在每个测试方法执行之前都要执行一次。（初始化方法）

**@After：**使用了该元数据的方法在每个测试方法执行之后要执行一次。（释放资源）

注意：@Before和@After标示的方法只能各有一个。这个相当于取代了JUnit4.0以前版本中的setUp和tearDown

**@BeforeClass：**针对所有测试，只执行一次，且必须为public static void;

**@AfterClass：**针对所有测试，将会在所有测试方法执行结束后执行一次，且必须为public static void;

**@Test(expected=\*.class)**

通过@Test元数据中的expected属性。expected属性的值是一个异常的类型，表示只接受这种类型的异常作为正确的结果。

以上面程序为例，相当于以下代码：

public void testDivideByZero() {

try {

calculator.divide(0);

fail();

catch (ArithmeticException e) { }

}

**@Test(timeout=xxx):**

该元数据传入了一个时间（毫秒）给测试方法，

如果测试方法在制定的时间之内没有运行完，则测试也失败。

**@Ignore：**

该元数据标记的测试方法在测试中会被忽略。当测试的方法还没有实现，或者测试的方法已经过时，或者在某种条件下才能测试该方法（比如需要一个数据库联接，而在本地测试的时候，数据库并没有连接），那么使用该标签来标示这个方法。同时，你可以为该标签传递一个String的参数，来表明为什么会忽略这个测试方法。比如：@lgnore(“该方法还没有实现”)，在执行的时候，仅会报告该方法没有实现，而不会运行测试方法。

总结：一个JUnit 的单元测试用例执行顺序为：

@BeforeClass -->

@Before --> @Test --> @After --> 【第一个测试方法】

@Before --> @Test --> @After --> 【第二个测试方法】

@Before --> @Test --> @After --> 【第三个测试方法】

@AfterClass

每一个测试方法的调用顺序为：

@Before --> @Test --> @After

**2）断言**

JUnit提供以Assert为关键字的一系列函数作为断言，在测试单元第一行都要引入Assert包（**import static org.junit.Assert.\*;）**用到该包中的以下几个函数：

* 用于结果状态验证：验证结果是否与预期的一样。

Assert.assertEquals 验证是否相等

* 防卫断言：验证调用功能前所作的各项假设。

Assert.assertTrue(list.isEmpty()) /assertFalse 假设结果true/false

* 交互断言：验证代码与其协作者对象的交互行为是否正确

Assert.assertTrue(list.contains(aaa));

**3）Errors和Failures的区别**

Failure表示测试实际返回结果与预期结果不一致，其原因可能是测试代码有误（如写错了预期结果）和代码有bug。Error表示测试无法正常执行，发生了错误。如果测试中发生了Error，应按照下面的顺序来检查：

第一步，检查测试所需的环境，如：数据库连接

第二步，检查单元测试代码

第三步，检查被测代码（比如说上例中出现的超时Timeout错误）。

**4）参数化测试**

编写测试用例的思路一定是需要考虑包括边界情况在内的各种输入对于输出结果的影响。你可能遇到过这样的函数，它的参数有许多特殊值，或者说他的参数分为很多个区域。比如，一个对考试分数进行评价的函数，返回值分别为“优秀，良好，一般，及格，不及格”，因此你在编写测试的时候，至少要写5个测试，把这5中情况都包含了，这确实是一件很麻烦的事情。我们还使用我们先前的例子，测试一下“计算一个整数的平方”这个函数，应该分三类：正数、0、负数。直观地想象一下测试代码应该包括如下几类：

@Test

public void square1() {

calculator.square(2);

assertEquals(4, calculator.getResult());

}

@Test

public void square2(){

calculator.Square(0);

assertEquals(0, calculator.getResult());

}

@Test

public void square3(){

calculator.square(-3);

assertEquals(9, calculator.getResult());

}

为了简化类似的测试，JUnit4提出了“参数化测试”的概念，只写一个测试函数，把这若干种情况作为参数传递进去，一次性的完成测试。

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

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Parameterized;

**import** org.junit.runners.Parameterized.Parameters;

**import** java.util.Arrays;

**import** java.util.Collection;

@RunWith(Parameterized.**class**)

**public** **class** SquareTest {

**private** **static** SampleCalculator *calculator*=**new** SampleCalculator();

**private** **int** param;

**private** **int** result;

@Parameters

**public** **static** Collection data(){

**return** Arrays.*asList*(**new** Object[][]{

{2, 4},

{0, 0},

{-3, 9},

});

}

//构造函数，对变量进行初始化

**public** SquareTest(**int** param, **int** result){

**this**.param = param;

**this**.result = result;

}

}

##### 2.3 成批运行test case

这是前面提到的JUnit框架的元素之一。它方便于系统单元测试的成批运行。使用起来也是非常简单，代码示例：

**import** org.junit.runner.RunWith;

**import** org.junit.runners.Suite;

@RunWith(Suite.**class**)

@Suite.SuiteClasses({SampleTest.**class**, SquareTest.**class**})

**public** **class** TestAll {

｝

##### 2.4 JUNIT单元测试时统计代码的覆盖率工具

单元测试是对软件基本组成的单元进行测试，如对函数、类等的测试。

覆盖率包括以下几个方面：

1.语句覆盖：指的是代码中所有语句都被至少执行一遍，常用工具：TrueCoverage等；

2.判断覆盖：指每个判断分别取真和假各一次来进行测试；

3.条件覆盖：指程序中每个判断中的每个条件的所有可能取值的全部组合情况至少被执行一次；

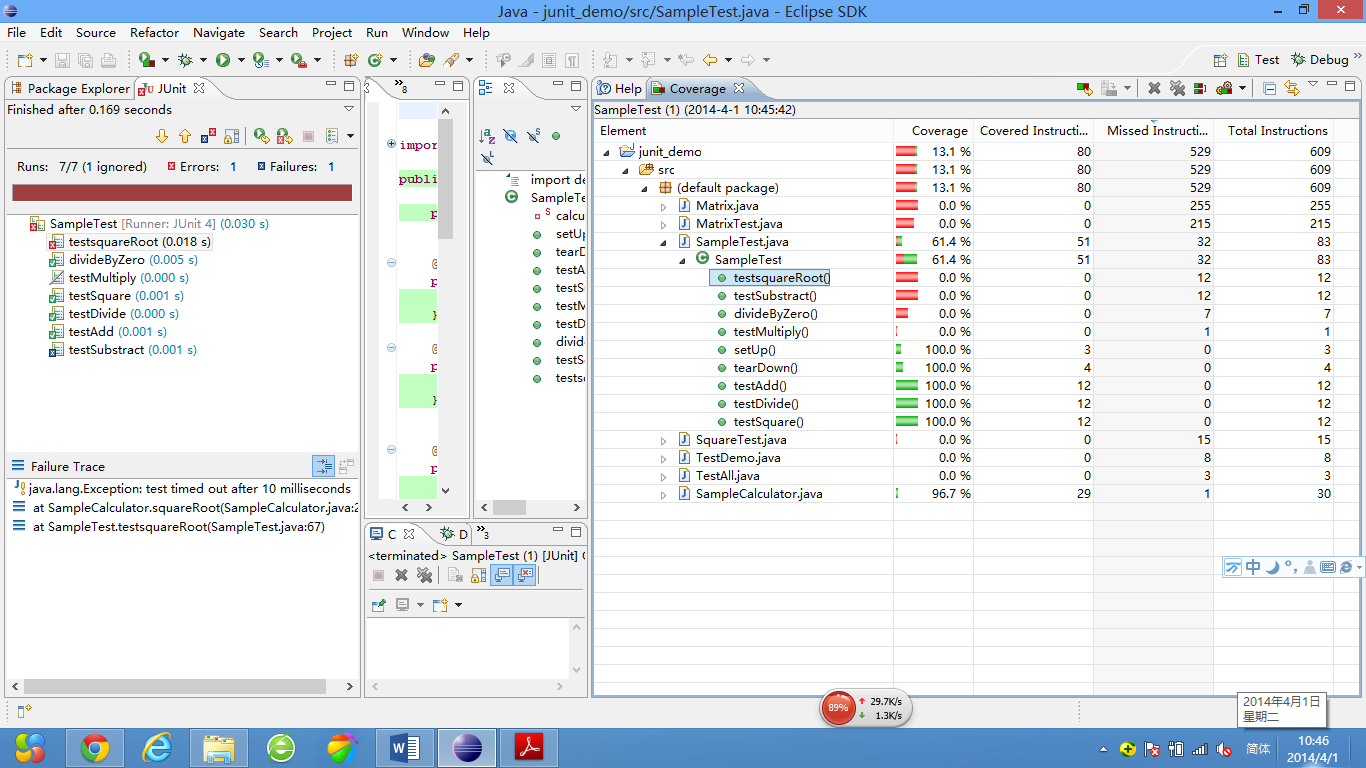
4.路径覆盖：程序中执行到路径数量和所有可能的执行路径的比率；

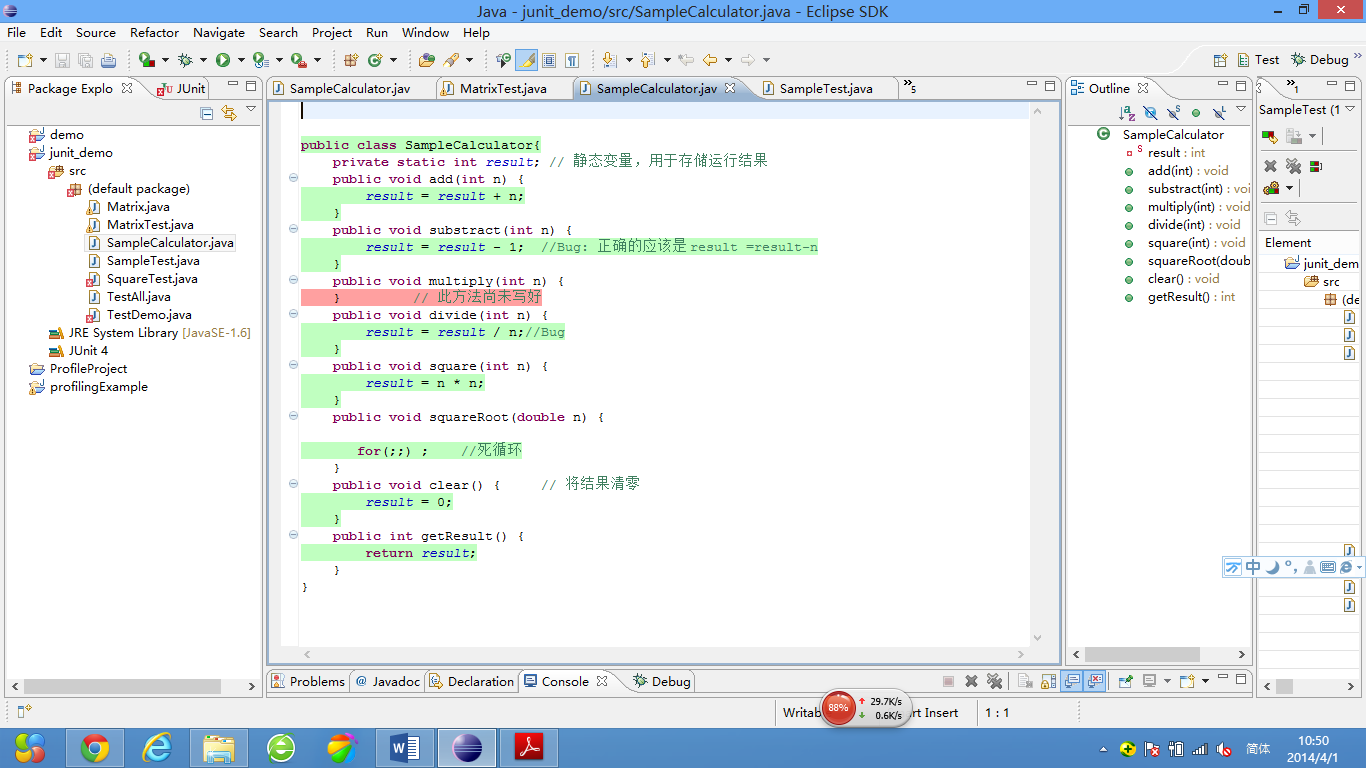
Eclipse中单元测试覆盖率统计的支持工具主要有EclEmma、Clover与Cobertura插件。安装相关插件的步骤为：Help->Install New Sofoware->Add->Add Repository，输入要安装的插件的网址，实现自动下载安装插件。

EclEmma： <http://update.eclemma.org>

Cobertura：<http://ecobertura.johoop.de/update/>

首选选择你要测试的Test目录，右击，右键菜单里面多了一个“Coverage as”的选项，先选择Open Coverage Diag，在Coverage那个tab选择你需要覆盖的代码。然后直接点击Coverage按钮，这时测试程序会在JUNIT中执行一遍，执行结束后会生成覆盖率统计结果。





源代码中，标志为绿色的部分表示全部执行，红色的部分表示未执行，黄色表示部分执行。

### 3．实验作业

码头仓库是划分为n×m个格子的矩形阵列。有公共边的格子是相邻格子。当前仓库中有的格子是空闲的；有的格子则已经堆放了沉重的货物（障碍物）。仓库管理员有一项任务，要将一个小箱子推到指定的格子上去。

* 管理员可以在仓库中移动，但不能跨过已经堆放了货物的格子。
* 管理员站在与箱子相对的空闲格子上时，可以做一次推动，把箱子推到另一相邻的空闲格子
* 推箱时只能向管理员的对面正前方向推。

对于给定的仓库布局，以及仓库管理员在仓库中的位置和箱子的开始位置和目标位置，给出的程序采用广度优先搜索，能够计算出把一个把箱子从起始位置推到目标位置的最短路径长度并打印箱子的运动轨迹。

**数据输入：**

由控制台提供输入数据。输入数据的第1 行有2个正整数n和m（1<=n,m<=20），

表示仓库是n×m个格子的矩形阵列。接下来有n行，每行有m个整数，表示格子的状态。

0 表示格子空闲；

1 表示格子上放了不可移动的沉重货物（障碍物）；

2 表示箱子的初始位置；

3 表示箱子的目标位置；

4 表示仓库管理员的初始位置；

**结果输出：**

将计算出的箱子的运动轨迹和推动次数输出。如果仓库管理员无法将箱子从开始位置推到目标位置则输出“No solution！”。

**输入示例：**

5 5

0 3 0 0 0

1 0 1 4 0

0 0 1 0 0

1 0 2 0 0

0 0 0 0 0

**输出示例：**

the total steps is:

4

the trace of box is:

(3,2)

(3,1)

(2,1)

(1,1)

(0,1)

程序包中分别给出了规格说明、输入样例、程序编译后文件，请根据前面示例的Junit框架，对程序进行单元测试，测试用例围绕readmap()，check(int x, int y)， bfs\_man(int sx, int sy, int ex, int ey, int bx, int by)，bfs()四个方法进行。Junit程序打包上传提交。