JUnit单元测试

第一章：JUnit的框架原理分析

1． JUnit主要用于单元测试，所谓的单元测试就是常常说的白盒测试。它是一个开源的由JAVA开发的一个用于测试的框架。

2． 下面我们主要是来看一下JUnit的设计原理

2.1. 首先我们来看一下JUnit的框架图。

2.2. JUnit的几个基本的概念：TestCase，TestSuite，TestFixtrue

TestCase: 代表一个测试用例，每一个TestCase实例都对应一个测试，

这个测试通过这个TestCase实例的名字标志，以便在测试结果中指明哪

个测试出现了问题.TestCase继承自Assert,因此可以实现各种断言。

TestSuite：代表需要测试的一组测试用例，也就是测试用例的集合，

TestFixtrue：代表一个测试环境。它用于组合一组测试用例，这组测试

用例需要共同的测试运行环境。

2.3. junit的设计

2.3.1. Test接口: 代表一个测试。它是框架的主接口有两个方法：

int countTestCases();//返回所有测试用例的个数。

void run(TestResult result);//运行一个测试，并且收集运行结果

到TestResult.

2.3.2. TestCase类: TestCase实现了Test接口，是框架提供的供我们继承的类，我们的所有的测试方法都需要在TestCase的子类中定义，并且符合特定的设计协议。

一个TestCase实例代表一个具体的测试实例，对应一个对某一方法或概念的测试。每个TestCase实例都有一个名字。

一个TestCase类却定义了一个TestFixture。具体的说就是我们自己定义的TestCase子类中可以定义很多的public 没有参数的 testxxx方法。运行时，每个testxxx都在自己的fixture中运行。每个运行的TestCase都有一个名字，如果不指定，一般是TestCase中定义的test方法的名字。

2.3.3. TestSuite类: 和TestCase一样TestSuite也实现了Test接口。一个TestSuite可以包含一系列的TestCase。把testCase组装入TestSuite有几种方式：

A，通过将TestCase的Class参数传入TestSuite的构造函数，TestSuite会自动收集TestCase中所有的public的没有参数的testxxx方法加入TestSuite中。

B，构造空的TestSuite后通过void addTest(Test test)方法添加测试。

C：构造空的TestSuite后通过void addTestSuite(Class testClass) 方法添加测试集。

2.3.4. TestResult类: 主要通过runProtected方法运行测试并收集所有运行结果

2.3.5. TestRunner类: 启动测试的主类，我们可以通过直接调用它运行测试用例，IDE和其他一些工具一般也通过这个接口集成JUnit.

2.3.6. Assert类: 用于断言，TestCase继承自该类，我们的测试方法通过这些断言判断程序功能是否通过测试

2.3.7. TestListener接口: 测试运行监听器，通过事件机制处理测试中产生的事件，主要用于测试结果的收集。

以上是框架的核心接口和类的介绍，通过上面的介绍我们很容易看出来Test，

TestCase和TestSuite的设计采用了Composite模式。这样JUnit可以一次运行

一个测试用例，也可以一次运行多个测试用例，TestRunner只关心Test接口，而

对运行的是单个的TestCase还是同时运行多个TestCase并不在意

3. JUnit同时使用了Command模式，对于典型的Command模式一般有5

种角色 :

3.1命令角色（Command）：声明执行操作的接口。有java接口或者抽象

类来实现

3.2. 具体命令角色（Concrete Command）：将一个接收者对象绑定于一

个动作；调用接收者相应的操作，以实现命令角色声明的执行操作的接

口.

3.3. 客户角色（Client）：创建一个具体命令对象（并可以设定它的接收者）。

3.4. 请求者角色（Invoker）：调用命令对象执行这个请求.

3.5. 接收者角色（Receiver）：知道如何实施与执行一个请求相关的操作。

任何类都可能作为一个接收者。

Test接口可以认为是命令模式中的命令角色Command接口，void run(TestRes

ult result)接口方法定义了需要执行的操作；TestCase可以看作是具体命令角色，

但又不全是，因为我们还需要自己通过继承TestCase类定义测试方法，这样的每一

个测试方法都回被包装在一个TestCase实例中。TestResult可以看作请求者角色

（Invoker），它会通过protected void run(final TestCase test) 运行测试并

收集结果。我们自己写的Test方法可以认为是接收者角色（Receiver），因为我们

的方法才具体执行这个命令。TestRunner就是客户角色（Client），它通过TestRe

sult result= createTestResult()构造TestResult，并通过suite.run(result)运

行测试用例（suite是一个Test接口的具体实例，可以是TestCase也可以是Test

Suite，但客户端不关心它是什么，这就是组合模式的好处。同时，suite.run（res

ult）又调用result.run（test），如果不仔细分析，就会被这种设计搞迷惑）.

第二章：JUnit的好处和单元测试的编写原则

现在世面上有很多的测试工具，比如说NUNIT，PHPUNIT等。但是在JAVA的世界里面JUnit是最适合我们的单元测试工具。

A：可以使测试代码与产品代码分开

B：针对某一个类的测试代码通过较少的改动便可以应用于另一个类的测试

C：易于集成到测试人员的构建过程中，JUnit和Ant的结合可以实施增量开发

D：JUnit是公开源代码的，可以进行二次开发

E：可以方便地对JUnit进行扩展

编写原则:

A: 是简化测试的编写，这种简化包括测试框架的学习和实际测试单元的编写

B: 是使测试单元保持持久性

C: 是可以利用既有的测试来编写相关的测试

第三章：JUnit的应用以及扩展

下面是关于JUNIT的扩展方面的,主要说的是junit.extensions包

1. ExceptionTestCase是TestCase的子类，用于对预见引发异常时的测试。生成该对象的时候要指明应该引发的异常的类型。runTest的时候会catch并吸收该异常。如果未发现该异常，则 测试失败

2. ActiveTestSuite是TestSuite的子类，用独立的线程来运行每个测试实例。runTest(Test, TestResult)开启新线程来运行Test。run(TestResult)对所有的Test运行runTest(Test, TestResult)，并等待所有线程结束。

3. TestDecorator是Assert的子类，并实现了Test接口。它封装Test并把其Test.run(TestResult)作为一个basicRun(TestResult)。TestDecorator.run(TestResult)直接调用basicRun。子类可以重载run(TestResult)，从而在原Test.run(TestResult)前后加入新的修饰代码

4. RepeatedTest是TestDecorator的子类。RepeatedTest.run(TestResult)重复指定次数运行TestDecorator.run(TestResult)。因此countTestCases()也要在TestDecorator.countTestCases()基础上乘以重复次数。

5. TestSetup是TestDecorator的子类。增加新的fixture。在basicRun(TestResult)前运行setUp()，在之后运行tearDown()。 BUG没有调用TestResult.startTest/endTest，即不产生开始/结束测试事件。

下面是个简单的例子：

Mytest.java类：

/\*

\* �������� 2006-7-31

\*

\* TODO Ҫ��Ĵ���ɵ��ļ���ģ�壬��ת��

\* ���� �� ��ѡ�� �� Java �� ������ʽ �� ����ģ��

\*/

/\*\*

\* @author mahb

\*

\* TODO Ҫ��Ĵ���ɵ�����ע�͵�ģ�壬��ת��

\* ���� �� ��ѡ�� �� Java �� ������ʽ �� ����ģ��

\*/

public class Mytest {

String name;

String birthday;

int age;

/\*\*

\* @return ���� age��

\*/

public int getAge() {

return age;

}

/\*\*

\* @param age Ҫ���õ� age��

\*/

public void setAge(int age) {

this.age = age;

}

/\*\*

\* @return ���� birthday��

\*/

public String getBirthday() {

return birthday;

}

/\*\*

\* @param birthday Ҫ���õ� birthday��

\*/

public void setBirthday(String birthday) {

this.birthday = birthday;

}

/\*\*

\* @return ���� name��

\*/

public String getName() {

return name;

}

/\*\*

\* @param name Ҫ���õ� name��

\*/

public void setName(String name) {

this.name = name;

}

}

MytestTest .java类

import junit.framework.Test;

import junit.framework.TestCase;

import junit.framework.TestSuite;

/\*

\* �������� 2006-7-31

\*

\* TODO Ҫ��Ĵ���ɵ��ļ���ģ�壬��ת��

\* ���� �� ��ѡ�� �� Java �� ������ʽ �� ����ģ��

\*/

/\*\*

\* @author mahb

\*

�� ����ģ��

\*/

public class MytestTest extends TestCase {

/\*

\* @see TestCase#setUp()

\*/

protected void setUp() throws Exception {

super.setUp();

}

/\*

\* @see TestCase#tearDown()

\*/

protected void tearDown() throws Exception {

super.tearDown();

}

/\*\*

\* Constructor for MytestTest.

\* @param arg0

\*/

public MytestTest(String arg0) {

super(arg0);

}

public static Test suite(){

TestSuite ts = new TestSuite();

//TestSuite ts = new TestSuite(MytestTest.class);

ts.addTest(new MytestTest("testgetName"));

return ts;

}

public void testgetName(){

Mytest mt = new Mytest();

mt.setName("mahb");

// assertEquals("gaofei",mt.getName());

System.out.println("12345455---" + mt.getName());

}

public void testgetBirthday(){

Mytest mt = new Mytest();

mt.setBirthday("1983.05.26");

// assertNull(mt.getBirthday());

assertNotNull(mt.getBirthday());

assertTrue(mt.getBirthday().equals("1982.1.20"));

System.out.println("-------"+mt.getBirthday());

}

}