

第 17 讲 用例学习：JUnit

在6.170中用来测试自己的代码所用的JUnit测试框架以它本身的价值是值得学习的。JUnit测试框架由Kent Beck和Erich Gamma开发。Beck是模式和极限编程(Extreme Programming, XP)的代表，Gamma是一本著名的设计模式方面著作的作者之一。JUnit是开放源代码的，所以你可以自己学习它的源代码。JUnit的发行版本中还有一个不错的说明文档，名字是“烹饪教程”。这个说明文档从设计模式方面说明了JUnit的设计过程，这个教程的大多数资料也是取自此说明文档。

JUnit取得了很大的成功。Martin Fowler，一个有洞察力和注重实际的模式和极限编程的支持者(也是关于对象模型的很棒的《模式分析》一书的作者)，说：

“软件开发行业从没有如此感激从极多到极少的代码行数的转变过。”

JUnit的易用性无疑是它受欢迎的主要原因。也许你会这样想：它做的事情不多，仅仅是做一些测试然后报告结果，JUnit应该是很简单的。事实上，它的代码非常复杂。主要原因是它被设计为一个框架，需要在许多不可预料的情况下进行扩展，所以它被设计为拥有非常复杂的模式和间接结构以允许实现者能够重构框架的一些部分同时保留其它的部分。

另一个复杂的影响因素是使测试容易写出来的要求。有一个聪明的解决方法，做一个映射，将原先的“类”变为“Test”类型的“单个实例”。这里还有一个初看不合理的解决方法。从Assert类继承来的抽象类TestCase包含许多静态声明方法，抽象类TestCase仅简单地允许将静态声明方法的调用写成assert(…)而不是Assert.assert(…)的形式。TestCase绝不是Assert的一个子类型，这显得没什么意义。但这使得TestCase中的代码能够写得更简洁。由于用户写的所有的测试用例都是TestCase类中的方法，使得如此做非常重要。

模式的使用是灵巧和机动的。我们要寻找的关键模式是：

模板方法(Template Method)——框架编程的关键模式；*命令、组合和观察者(Command, Composite and Observer)*。这些模式都由Gamma等人详细地解释，除命令外都已包含在本课程内。

我个人的意见是，JUnit是极限编程王冠上的宝石，它传递着发展方向的基本信息——代码本身足矣。它是某种程序的一个完美例子，这种程序如果没有抽象，没有设计思想的全局表示来解释各个部分怎样结合，就几乎是不可理解的。但它对那种非常依赖注释的代码没有用处，尤其是有注释来描述程序员是坐在哪个瑞士山顶上写这段代码的。大概高海拔和稀薄的空气能够解释这种代码风格。“烹饪教程”是精华。没有它，可能要很长时间来领悟正在发生的事情的微妙。如果有更多设计表示它会更有帮助。“烹饪教程”阐述了简单的观点，而我必须为自己构造一个对象模型来解释，比如：监听器怎样工作。

如果你是那些怀疑设计表示的学生之一，而且觉得代码非常之重要，你现在就该停止读这

篇文章，躺在椅子上花一晚上来读JUnit的源代码。谁知道呢，这有可能改变你的看法。

你可以从这里下载JUnit的源代码和文档：

<http://www.junit.org/>

这里有一个开放源代码库：

<http://sourceforge.net/projects/junit/>

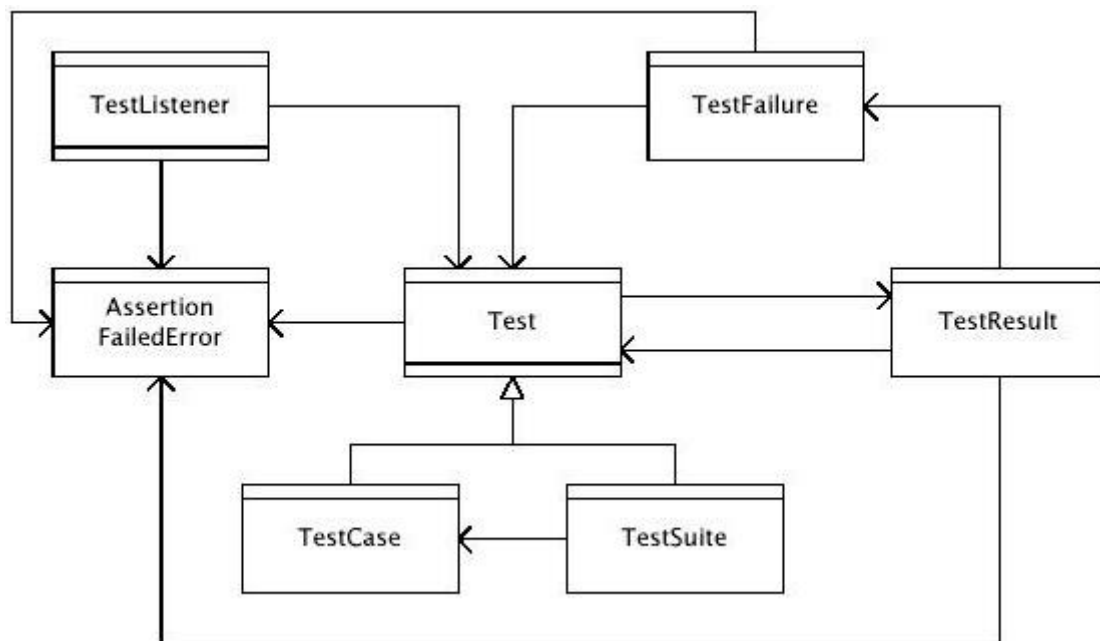
在这里可以查看和报告bug。

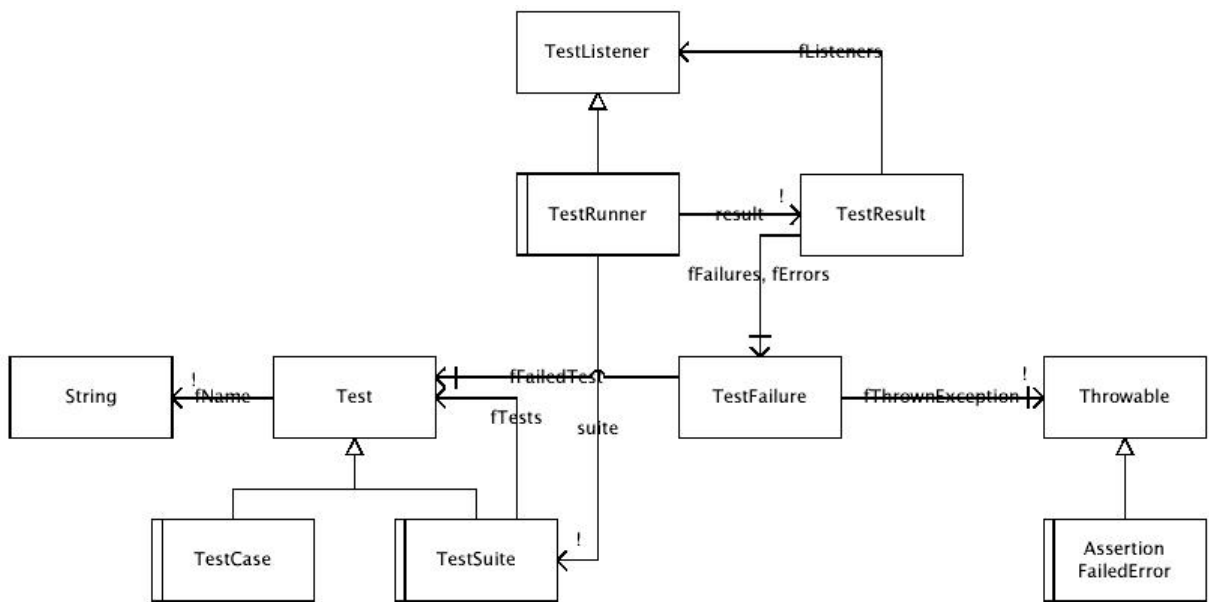
17.1 概述

JUnit有几个包：*framework*用来放置基本框架，*runner*用来放置运行测试程序时的几个抽象类，*textui*和*swingui*放置用户界面，*extensions*用来放一些有用的补充。我们主要研究*framework*包。

下图表示对象模型和模块的依赖关系。看了讨论后你会希望遵循这些图来研究。这两个图都只包括框架模块，但我又将*TestRunner*加入对象模块来演示监听器是怎样被连接的；它的连接*suite*和*result*是它的*doRun*方法的局部变量。

注意模块依赖图几乎全部连满了。这对于框架来说不足为奇，模块并不是用来单独使用的。





17.2 命令模式

命令模式将函数封装成对象的形式。这也是面向对象语言中是如何实现闭包的——还记得 6.001 的内容么？command 类典型地只有一个名为 *do*、*run* 或 *perform* 的方法。通过重构这个方法建立子类的一个实例，而且通常封装一些状态（在 6.001 中的术语，闭包的环境）。这样命令就可以作为一个对象被传递，并通过调用来执行。

在 JUnit 中，测试用例被描绘为实现 *Test* 接口的 command 对象：

```
public interface Test {
    public void run();
}
```

实际的测试用例是具体的 *TestCase* 类的子类的实例：

```
public abstract class TestCase implements Test {
    private String fName;
    public TestCase(String name) {
        fName = name;
    }
    public void run() {
        ...
    }
}
```

事实上，实际的代码不太像这样，但从简化的版本入手可以让我们更容易地解释基本的模式。注意构造函数的名称与测试用例的名称相关，这在报告结果时会很有用处。事实上，所有实现 *Test* 的类都有这个性质，所以给 *Test* 接口增加一个方法：

```
public String getName ()
```

会比较好。同样要注意JUnit的作者们约定类的属性中的标识符以小写字母 l 开头（实例变量）。

在我们下周学习 *Tagger* 程序时会见到更详细的命令模式的例子。

17.3 模板方法

用户可能运行一个抽象方法，因此需要所有子类重构它。但多数测试用例有三个阶段：创建上下文（测试环境），执行测试，销毁上下文（测试环境）。我们可以通过运行一个模板方法提取这个公共结构：

```
public void run() {  
    setUp();  
    runTest();  
    tearDown();  
}
```

类中定义的hook方法在缺省时什么也不做：

```
protected void runTest() { }  
protected void setUp() { }  
protected void tearDown() { }
```

这些方法被声明为protected，所以可以由子类访问（当然也可以被重构），但不能从包的外部访问。能够避免来自子类之外的访问固然很好，但Java没有提供这样的模式。一个子类可以选择性地重构这些方法；例如，如果只重构`runTest`方法，将没有专门的`setUp`或`tearDown`行为。

我们在上一章的Java集合API的框架实现中见到了相同的模式。有时候它和*好莱坞原则*有异曲同工之妙。传统的API提供能被客户端调用的方法；相反，框架对它的client的方法进行调用：“不要打电话给我们，我们会给你打电话”。模板的深入使用是框架编程的精髓。它很强大，但也容易将程序写得完全不可理解，因为方法的执行使调用随继承代数成倍增长。

很难知道一个框架中的子类要做什么。类似先决条件和后续条件的技术还没有实现，而且这种技术还处在很粗糙的阶段。你通常必须读框架的源代码才能有效地使用它。Java集合API比大多数框架做得好，因为它在模板方法的规格说明中包含了一些关于这些方法被怎样执行的准确的描述。这当然是对抽象规范说明思想的诅咒，但这在框架的文档中不可避免。

17.4 组合模式

如我们在11章中讨论的，测试用例组合为包。但你对测试包做什么在本质上是和你对单个

测试做什么都是相同的：运行和报告结果。这暗示了该使用组合模式，在组合模式中一个复合对象与它的元素构件共享接口。

这里，接口是 *Test*，复合是 *TestSuite*，构成元素是 *TestCase* 的成员。*TestSuite* 是实现 *Test* 的一个具体类，但不像 *TestCase* 的 *run* 方法，它的 *run* 方法调用测试包中每个测试用例的 *run* 方法。*TestCase* 的实例通过 *addTest* 方法被添加到 *TestSuite* 的实例中；以后我们会看到，也有构造器可以将多个测试用例复合成 *TestSuite*。

在 Gamma 的书中，组合模式的例子中有包含所有复合操作的接口。据此，*Test* 应该包含 *addTest* 这样只应用于 *TestSuite* 对象的方法。模式描述的实现部分提出了“透明”和“安全”之间的一个权衡。“透明”使复合和叶子对象看起来相同，“安全”避免不适当的操作被调用。根据我们在子类型化一章中的讨论，问题在于接口是否该是一个真的子类型。我认为应该是，因为“安全”带来的好处超过透明带来的好处，而且对接口的复合操作的结果进行推论非常容易出错。据此，JUnit 在 *Test* 接口中不包含 *addTest*。

17.5 收集参数

Test 的 *run* 方法如下：

```
public void run(TestResult result);
```

它使用一个变异的参数来记录运行测试的结果。Beck 把它称为“收集参数”，并把它看为一个设计模式。

一个测试可能因为两种原因失败：产生错误的结果（不抛出预期的异常），或者抛出一个非预期的异常（比如 *IndexOutOfBoundsException*）。JUnit 将前者称为“故障”（Failure），将后者称为“错误”（Error）。*TestResult* 的一个实例包括一系列故障和一系列错误，每个故障或错误被表现为 *TestFailure* 类的一个实例，*TestFailure* 类包含与 *Test* 之间的联系和与故障或错误产生的异常对象之间的联系。（故障经常生成异常，因为即使一个非预期的结果没有生成异常，*test* 中用的 *assert* 方法也能将错配（mismatch）转换为异常）。

TestSuite 中的 *run* 方法本质上没有改变；只是调用每个测试的 *run* 方法时将 *TestResult* 跳过。*TestCase* 中的 *run* 方法是这样的：

```
public void run (TestResult result) {
    setUp ();
    try {
        runTest ();
    }
    catch
    (AssertionFailedError e) {
        result.addFailure (test, e);
    }
}
```

```

    }
    (Throwable e) {
        result.addError (test, e);
    }
    tearDown ();
}

```

事实上，模板方法`run`的控制流比我们写出的复杂得多。这里使用一些伪代码段来展示。它忽略了`setUp`和`tearDown`的活动，并考虑到了文本用户界面中`TestSuite`的使用：

```

junit.textui.TestRunner.doRun (TestSuite suite) {
    result = new TestResult ();
    result.addListener (this);
    suite.run (result);
    print (result);
}
junit.framework.TestSuite.run (TestResult result) {
    forall test: suite.tests
        test.run (result);
}
junit.framework.TestCase.run (TestResult result) {
    result.run (this);
}
junit.framework.TestResult.run (Test test) {
    try {
        test.runBare ();
    }
    catch (AssertionFailedError e) {
        addFailure (test, e);
    }
    catch (Throwable e) {
        addError (test, e);
    }
}
junit.framework.TestCase.runBare (TestResult result) {
    setUp();
    try {
        runTest();
    }
    finally {
        tearDown();
    }
}
}

```

*TestRunner*是一个用户界面类，它调用框架并显示结果。有一个GUI版本的 *junit.swingui* 和一个简单的控制台版本的 *junit.textui*，我们已经在这里摘录了一部份，以后还会提到，现在先忽略它。

它是这样工作的。*TestRunner*对象建立一个新的 *TestResult*来存放测试结果，它执行测试包，并输出结果。*TestSuite*的 *run*方法调用每个元素测试的 *run*方法；每个元素测试自身也可以是 *TestSuite*对象，这样方法就会递归调用。这是复合带来的简易性的很好的例子。最后，由于有规定 *TestSuite*未经特地指定不能包含它自己的不变式，也不能由 *TestSuite*中的代码执行，所以调用 *TestCase*类型的对象的 *run*方法可以使递归方法进行到底。

现在 *TestCase*的 *run*方法将 *TestResult*对象换为 *TestCase*对象作为接受者，并将 *TestCase*作为参数调用 *TestResult*的 *run*方法（为什么？）。然后 *TestResult*的 *run*方法调用 *TestCase*的 *runBare*方法，*runBare*方法才是执行测试的真正的模板方法。如果测试失败，则抛出一个异常，由 *TestResult*的 *run*方法接收，并将此测试和异常打包为 *TestResult*的一个故障或错误。

17.6 观察者模式

对于一个交互的用户界面，我们会希望当测试增加时能够显示测试结果。为了达到这个目的，JUnit使用了观察者模式。

*TestRunner*类实现一个有 *addFailure*方法和 *addError*方法的接口 *TestListener*。它扮演观察者（*Observer*）的角色。*TestResult*类扮演主体的角色；它提供一个方法：

```
public void addListener(TestListener listener)
```

此方法创建一个观察者。当 *TestResult*的 *addFailure*方法被调用，除了更新故障列表外，它还对每个观察者调用 *addFailure*方法：

```
public synchronized void addFailure(Test test, AssertionError e) {
    fFailures.addElement(new TestFailure(test, e));
    for (Enumeration e = cloneListeners().elements(); e.hasMoreElements(); ) {
        ((TestListener)e.nextElement()).addFailure(test, e);
    }
}
```

在文本用户界面中，*TestRunner*的 *addFailure*方法只简单地输出一个字母F。在图形用户界面中，它将故障添加到列表并将进度条改为红色。

17.7 反射处理

我们回顾一下，一个测试用例是 *TestCase*类的一个实例。在一般的Java环境下创建一个测试包，用户必须为每一个测试用例创建一个全新的 *TestCase*的子类，并对其进行实例化。一个

好方法是使用匿名内部类，将测试用例创建为一个没有名字的子类的实例。但这样做仍然很繁琐，所以JUnit提供了一个聪明的处理办法。

用户为每个测试包——名为 *MySuite*——提供一个类，视其为 *TestCase* 的一个子类，*TestCase* 包含许多测试方法，每个测试方法的名字都以字符串“test”开头。它们将作为单独的测试用例。

```
public class MySuite extends TestCase {
    void testFoo () {
        int x = MyClass.add (1, 2);
        assertEquals (x, 3);
    }
    void testBar () {
        ...
    }
}
```

类对象 *MySuite* 自身被传递给 *TestSuite* 的构造器。使用反射，*TestSuite* 中的代码将 *MySuite* 实例化给以“test”开头的每个方法，将方法的名字作为参数传递给构造函数。结果，对于每个测试方法，一个全新的 *TestCase* 对象被创建，并将其名字与测试方法的名字捆绑。*TestCase* 的 *runTest* 方法使用反射，调用那些名字符合 *TestCase* 对象自身名字的方法，基本是这样：

```
void runTest () {
    Method m = getMethod (fName);
    m.invoke ();
}
```

这个方案是模糊和危险的，你不该在自己的代码中模仿。在这里是可行的，因为它只局限于JUnit代码的很小一部分，而且给JUnit的用户带来了极大的便利。

17.8 自学中的问题

我为JUnit构造对象模型时遇到了这些问题。它们都没有准确答案。

- 为什么监听器被附在 *TestResult* 中？*TestResult* 自身不已经是一种监听器了吗？
- 一个 *TestSuite* 能够不包含任何测试么？它可以包含自己吗？
- 测试的名字必须是唯一的吗？
- *TestFailure* 中的 *fFailedTest* 属性都是指向 *TestCase* 的吗？