Lecture4 Test

1. 介绍

为什么要测试

- 改善代码质量 / 找到 BUG
- 测量质量
    - 证明没有漏洞（有可能的话）
    - 确定软件是否已准备好发布
    - 确定要做什么
    - 看看你是否犯了错误
- 学习软件

一些代价昂贵的错误

NASA 火星太空任务

- 优先级倒置(2004)
- 不同的米制系统(1999)

宝马汽车安全气囊问题

- 召回了 1.5w+ 汽车

Ariane 5 崩溃

- 数值溢出的未捕获异常

UBer 致命崩溃

- 自动驾驶汽车的软件错误

经济影响

The Economic Impact of Inadequate Infrastructure for Software Testing – NIST Report, May 2002

- 因为软件测试基础设施不足导致的年度成本为 595 亿美元
- 通过切实可行的基础设施改进，每年可减少 222 亿美元的成本

什么是测试

- 使用**已知输入（test inputs/data）**运行程序，**检查结果(test oracles)**
- 测试可以记录故障
- 测试可以记录代码

Test Input 和 Test Oracle

例如我们有如下功能的代码

```
1  int calAmount () {
2      int ret = balance * 3;
3      ret = ret + 10;
4      return ret;
5  }
```

现在我们有对其的单元测试

```
1  void testCalAmount() {
2      Account account = new Account();
3      account.setBalance(1); // test input
4      int amount = account.calAmount();
5      assertTrue(amount == 12); // test oracle
6  }
```

测试的术语

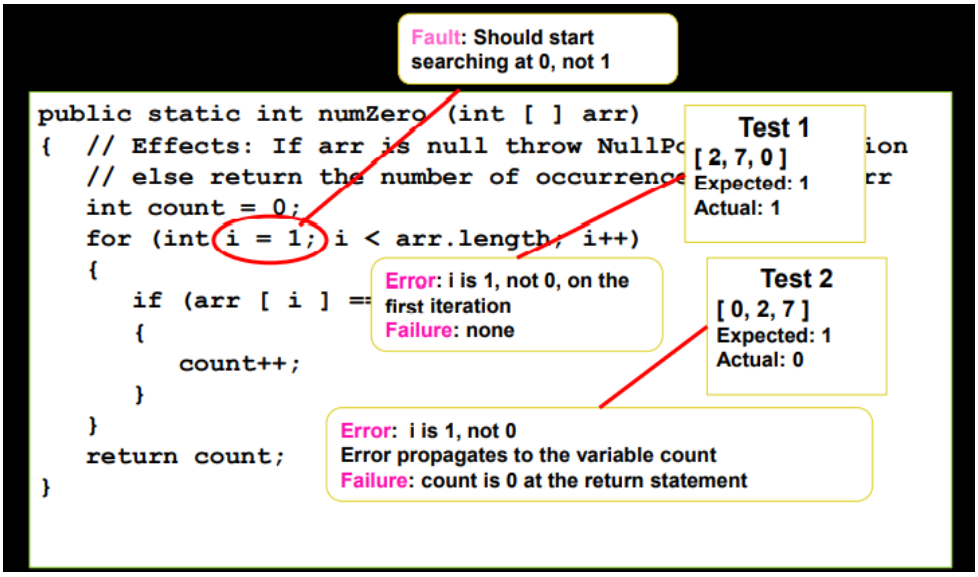
- **Mistake**: Programmer makes a mistake
- **Fault(defect, bug)**: Appears in the program
  - Fault remains undetected during testing (Running the test inputs)
- **Failure**: Program failure occurs during execution (program behaves unexpectedly)
- **Error**: Difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition

具体的区分

- **Failures**：病人给医生一份**症状清单**
  - 在程序中，如果输出结果与预期不符合（外观上），即有 Failures
- **Fault**：医生试图诊断**根本原因**，即疾病
  - 代码中具体哪个位置写错了（问题的根源）
- **Errors**：医生可能会寻找**内部异常情况**(高血压、心律不齐、血流中有细菌)
  - 在程序中，如何由 Fault 导致出现了某些 Error（尽管可能不会导致 Failures）

具体的示例

```java
public static int numZero (int [ ] arr){
    // Effects: If arr is null throw NullPointerException
    // else return the number of occurrences of 0 in arr
    int count = 0;
    for (int i = 1; i < arr.length; i++){
        if (arr [i] == 0){
            count++;
        }
    }
    return count;
}
```



2. JUnit Test

JUnit 介绍

- 开源 Java 测试框架，用于编写和运行可重复的**自动化**测试
- 一个用于编写**测试驱动程序**的结构
- JUnit 的特性包含
  - **断言 Assertions**：测试预期结果的断言
  - **共享测试数据 Sharing common test data**：在不同测试之间共享
  - **测试套件 Test suites**：便于组织和运行测试
  - **测试运行 Test runners**：图形和文本
- JUnit 在工业中得到了广泛的应用
- 可以作为独立的 Java 程序使用
  - 从命令行
  - 从IDE（如 IntelliJ 或 Eclipse）

功能

- JUnit 可以用来测试
  - 一整个对象
  - 部分对象（方法/与之交互的方法）
  - 不同对象之间的交互
- 主要是单元和**集成测试 unit & integration test**，而不是**系统测试 system test**
- 每个测试都嵌入到一个测试方法 test method 中
- **测试类 test class** 包含一个或多个测试方法
- 测试类包含
  - 测试运行test runner：用于运行测试 - main()
  - 一系列测试方法 test method
  - 用于设置每个测试**之前**和之后以及所有测试之前和**之后**的状态的方法

使用 JUnit 编写测试

- 需要使用 `junit.framework.assert` 类的方法
  - Javadoc 对其功能进行了完整的描述
  - 一些有代表性的方法
    - `assertTrue(boolean)`
    - `assertTrue(String, boolean)`
    - `assertEquals(Object, Object)`
    - `assertNull(Object)`
    - `Fail(String)`
- 每个测试方法检查一个条件（断言），并向测试运行程序报告测试是成功还是失败
- 测试运行程序使用结果向用户报告（在命令行模式下）或更新显示（在IDE中）
- 所有的测试方法都**返回 void**

JUnit Test Fixture

- 一个 test fixture 是测试的状态
  - 多个测试使用的对象和变量
  - 初始化（前缀值）
  - 重置值（后缀值）
- 不同的测试可以共同使用**不共享状态**的对象
- 在 fixture 中，对象被称为实例变量 instance variables
- 它们应该在 `@Before` 方法中被声明
  - JUnit 会在执行每一个 `@Test` 方法之前，先执行 `@Before`
- 可以在 `@After` 方法中释放或重置
  - JUnit 会在执行每一个 `@Test` 方法之后，再执行 `@Before`

完整的 JUnit 测试示例

源代码

```
1  public class Stack{
2  ...
3      public String toString(){
4          // EFFECTS: Returns String representation
5          // of this Stack from top to bottom.
6          StringBuffer buf = new StringBuffer("{");
7          for (int i = size-1; i >= 0; i--){
8              if (i < (size-1))
9                  buf.append (", ");
10             buf.append(elements[ i ].toString());
11         }
12         buf.append("}");
13         return buf.toString();
14     }
15 ...
16     public boolean repOk() {
17         if (elements == null) return false;
18         if (size != elements.length) return false;
19         for (int i = 0; i < size; i++) {
20             if (elements[i] == null) return false;
21         }
22         return true;
23     }
24 }
```

测试代码

导入相关的类

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;
```

测试前的设置

```
private Stack stack;
// setUp method using @Before syntax
// @Before methods are run before each test
@Before
public void runBeforeEachTest(){
    stack = new Stack();
}
```

测试后的设置

```
// tear-down method using @After
// @After methods are run after each test
@After
public void runAfterEachTest(){
    stack = null;
}

```

测试

```
@Test
public void testToString(){
    stack = stack.push (new Integer (1));
    stack = stack.push (new Integer (2));
    assertEquals ("{2, 1}", stack.toString());
}

```

- 不要将几个独立的测试结合到一个测试方法里去
- 在没有自动化的情况下，大型测试具有降低运行许多测试的成本的优势
- 在自动化的情况下，小型测试允许我们更容易地识别故障

```
@Test
public void testRepOkA(){
    boolean result = stack.repOk();
    assertEquals (true, result);
}
@Test
public void testRepOkB(){
    stack = stack.push (new Integer(1));
    boolean result = stack.repOk();
    assertEquals (true, result);
}
@Test
public void testRepOkC(){
    stack = stack.push (new Integer (1));
    stack = stack.pop();
    boolean result = stack.repOk();
    assertEquals (true, result);
}
@Test
public void testRepOkD(){
    stack = stack.push (new Integer (1));
    stack.top();
    boolean result = stack.repOk();
    assertEquals (true, result);
}
```

运行测试

在 IDE 中，我们可以运行所有的 JUnit 测试

我们也可以使用 `main()` 方法在命令行中执行

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import junit.framework.JUnit4TestAdapter;
// This section declares all of the test classes in the program.
@RunWith (Suite.class)
@Suite.SuiteClasses ({ StackTest.class }) // Add test classes here.
public class AllTests{
    // Execution begins at main(). In this test class, we will execute
    // a text test runner that will tell you if any of your tests fail.
    public static void main (String[] args){
        junit.textui.TestRunner.run (suite());
    }
    // The suite() method is helpful when using JUnit 3 Test Runners or Ant.
    public static junit.framework.Test suite(){
        return new JUnit4TestAdapter (AllTests.class);
```

```
16      }
17  }
```

- JUnit 提供测试驱动
  - 基于字符的测试驱动程序，从命令行运行
  - 基于 GUI 的测试驱动程序 `junit.swingui.TestRunner`
    - 允许程序员指定测试类来运行
    - 创建一个 Run 的按钮
- 如果测试失败，JUnit 会给出失败的位置和抛出的任何异常

其它话题

断言模式 Assertion Patterns

如何判断测试是否通过

- **状态测试 State Testing Patterns**：验证的是被测代码是否能返回正确的结果，不关心使用哪个方法和代码解决，只关心返回结果的正确性
  - Final State Assertion
    - 最常见的模式：Arrange，Act，Assert
  - Guard Assertion
    - 动作前后都断言（前置条件测试）
  - Delta Assertion
    - 验证状态的相对变更
  - Custom Assertion
    - 对复杂验证规则进行编码
- **交互测试 Interaction Assertions Patterns**：验证被测代码正确的调用了某些方法
  - 验证预期的交互
  - 在 Mocking 工具中大量使用
  - 与状态测试相比，分析非常不同

参数化测试

测试具有相似值的函数，如何避免测试代码膨胀？

- 简单的例子：两个数相加
  - 给给定的一对数相加就像给其他任何一对数相加一样
  - 您真的只想编写一个测试
- 参数化的单元测试为每个逻辑数据值集调用构造函数
  - 然后对每组数据值运行相同的测试
  - 使用 `@Parameters` 注释标识的数据值列表

```
1  import org.junit.*;
2  import org.junit.runner.RunWith;
3  import org.junit.runners.Parameterized;
4  import org.junit.runners.Parameterized.Parameters;
```

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

@RunWith(Parameterized.class)
public class ParamTest {
    public int sum, a, b;
    public ParamTest (int sum, int a, int b) {
        this.sum = sum; this.a = a; this.b = b;
    }

    @Parameters
    public static Collection<Object[]> parameters() {
        return Arrays.asList (new Object [][] {{0, 0, 0}, {2, 1, 1}});
    }

    @Test
    public void additionTest() {
        assertEquals(sum, a+b);
    }
}
```

JUnit 理论

- 这些是有着**实际参数**的单元测试
  - 到目前为止，我们只看到了没有参数的测试方法
- 契约模型（AAA模型）：假设 Assume、行动 Act、断言 Assert
  - 假设 Assumptions（前提条件）适当限制数值
  - 行动在审查下执行活动
  - 断言（后置条件）检查结果

```java
@Theory
public void removeThenAddDoesNotChangeSet(Set<String> set, String string) { //
Parameters!
    assumeTrue(set.contains(string)) ; // Assume
    Set<String> copy = new HashSet<String>(set); // Act
    copy.remove(string);
    copy.add(string);
    assertTrue (set.equals(copy)); // Assert
    // System.out.println(“Instantiated test: “ + set + “, “ + string);
}
```

- 问题来了，带参数的话，这些参数的数据从哪里来呢?
- 从 `@DataPoint` 注释中假设子句为真的所有值组合
- 在这种特殊情况下，有四种（九种）组合
- 注意 `@DataPoint` 格式是一个数组

```
@DataPoints
public static String[] string = {"ant", "bat", "cat"};

@DataPoints
public static Set[] sets = {
    new HashSet(Arrays.asList("ant", "bat")),
    new HashSet(Arrays.asList("bat", "cat", "dog", "elk")),
    new HashSet(Arrays.asList("Snap", "Crackle", "Pop"))
};

```

JUnit 相关资源

一些 JUnit 教程

- http://open.ncsu.edu/se/tutorials/junit/
- http://www.laliluna.de/eclipse-junit-testing-tutorial.html
- http://www.diasparsoftware.com/template.php?content=jUnitStarterGuide
- http://www.clarkware.com/articles/JUnitPrimer.html

JUnit 文档

- http://www.junit.org/

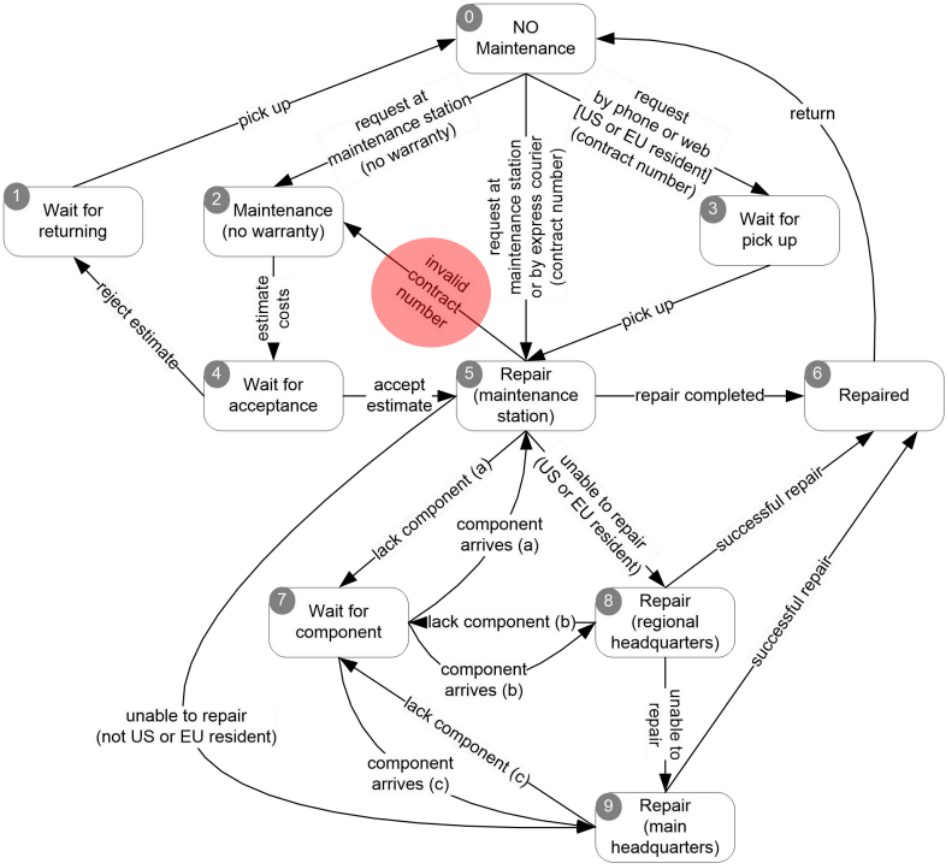
3. 测试驱动开发 TDD

Kent Beck 规则

- Beck 的测试驱动开发概念以两个基本规则为中心
    - 除非自动测试失败，否则不要写一行代码
    - 减少复制

非正式的需求

如果客户提供的维修合同号不合法，则按非保修项目处理



非正式需求中的歧义

如果客户提供的维护合同号不合法

- 合同编号不能包含字母或特殊字符?
- 合同号必须是 5 位数字?
- 合同号不能以 0 开头?

基于测试的需求

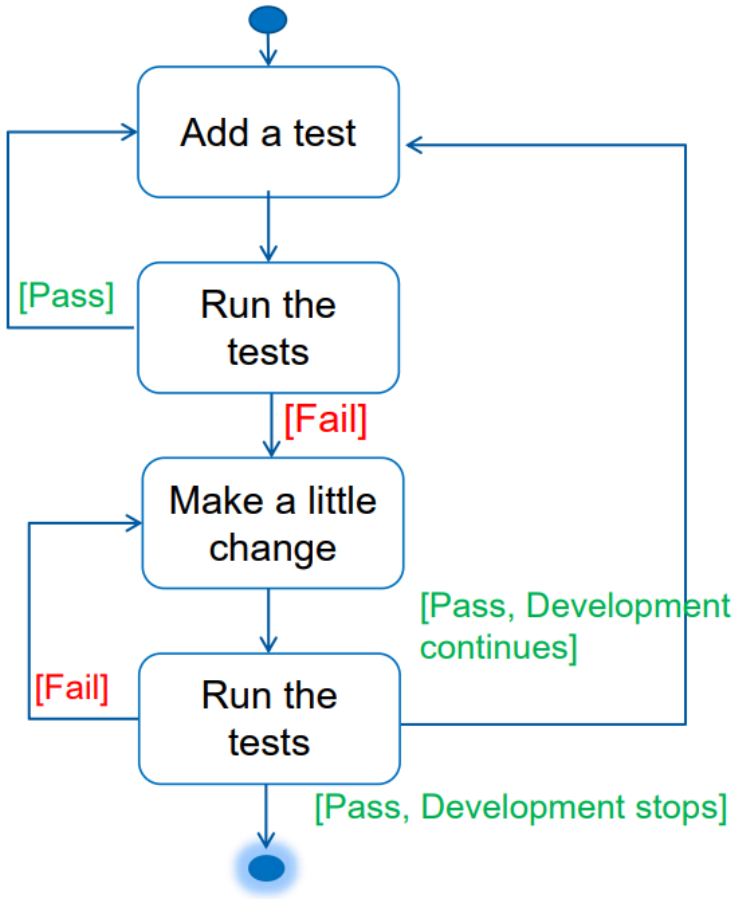
```
@Test
public void testContractNumberCorrectLength() {
    assertTrue(contract.isValidContractNumber(“12345”));
}
@Test
public void testContractNumberTooLong() {
    assertFalse(contract.isValidContractNumber(“53434434343”));
}
@Test
public void testContractNumberNoSpecialCharacter() {
    assertTrue(contract.isValidContractNumber(“08067”));
}
@Test
public void testContractNumberWithSpecialCharacter() {
    assertFalse(contract.isValidContractNumber(“98&67”));
}

```

非正式需求与测试用例

- 测试用例比需求更具体
- 但是，如何基于测试用例开发代码?
    - 遵循**测试驱动开发 Test Driven Development (TDD)** 中的步骤

测试驱动开发的步骤



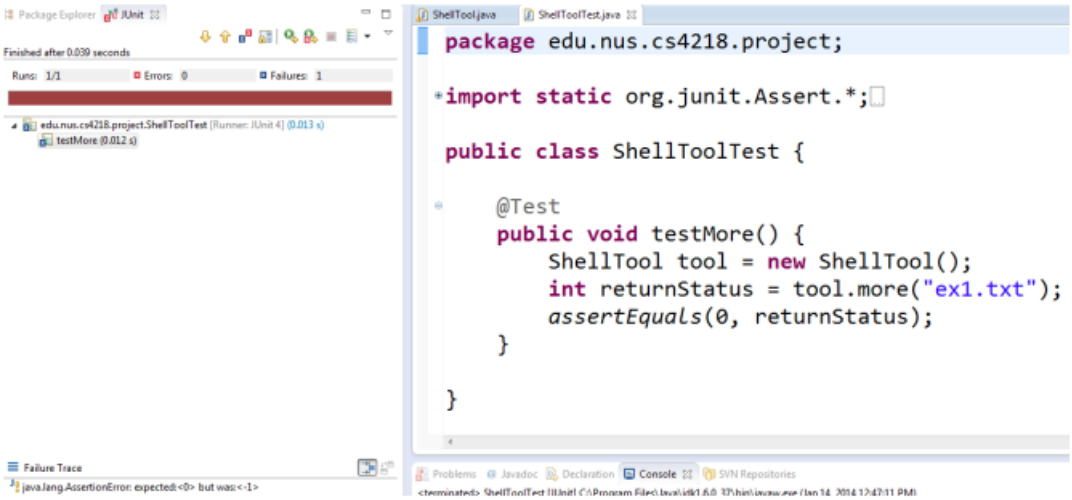
迭代过程

- 快速添加一个测试
- 运行所有的测试然后看失败的新的测试
- 对代码进行改变
- 运行所有的测试然后看它们成功
- 重构以删除重复

测试优先的场景

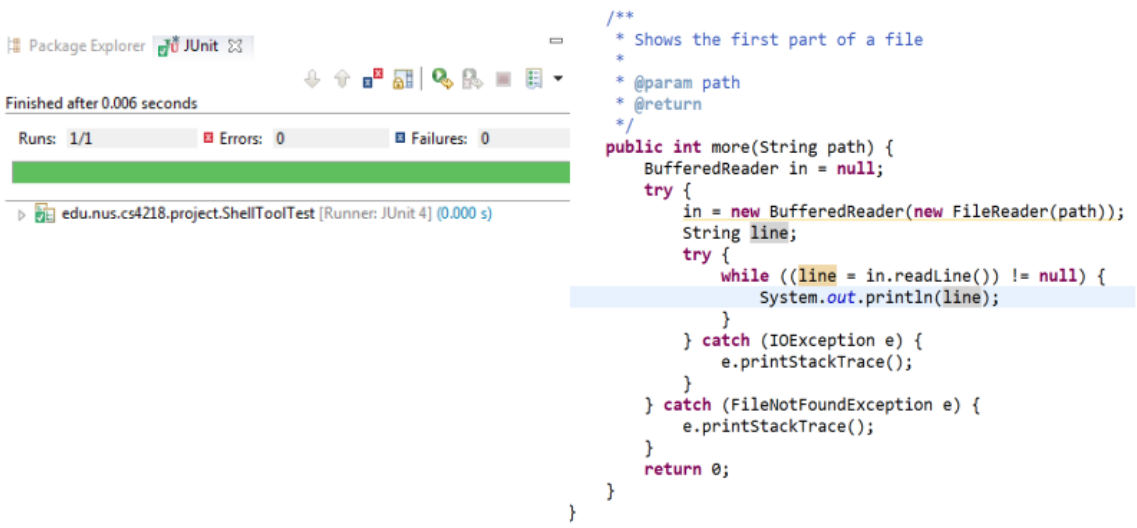
- 新添加的功能编写测试
    - 这些测试用例将作为实现的规范
    - 这些测试用例现在应该失败了，因为相应的方法没有实现
    - 编写最少的代码使测试通过
    - 添加更多测试

Run The Tests



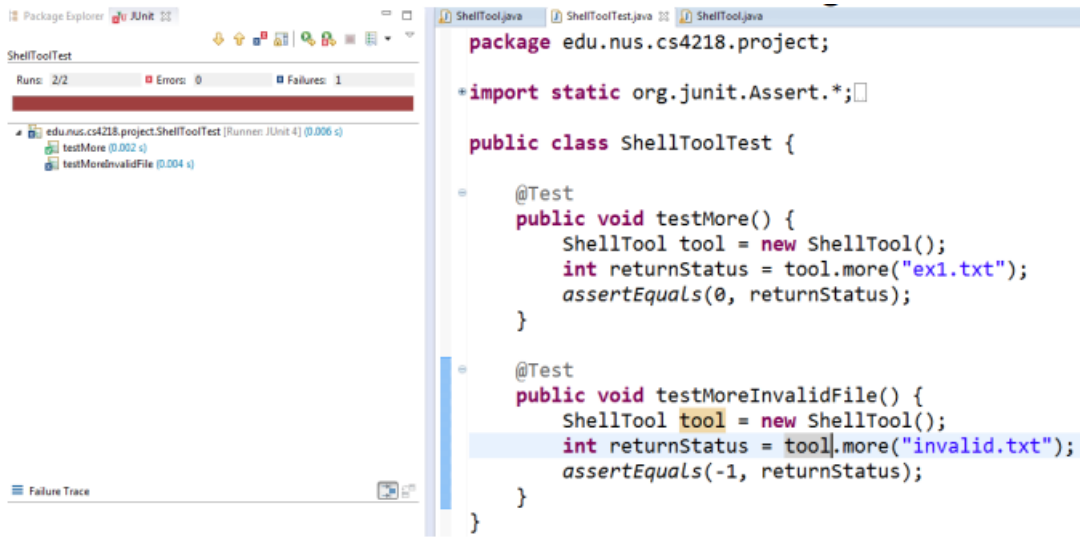
- 运行您的团队可以看到失败的测试
    - 失败的测试用例表明缺少功能

Make Them Pass



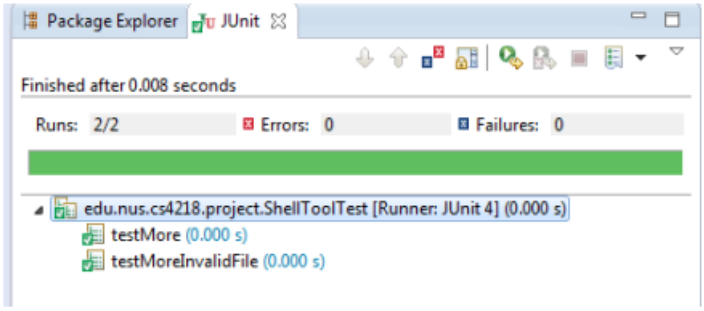
- 添加代码以通过失败的测试
    - 在实现所有缺失的功能之后，所有失败的测试用例现在应该通过了

Add More Tests



- 添加更多测试
  - 新增的组件或帮助方法
  - 检查角落情况
- 运行新的测试集，查看失败的测试

Make Them Pass



- 添加更多代码以使添加的测试通过
  - 在实现了所有的辅助方法并检查了角落用例之后，所有新的失败的测试用例现在都应该通过了