Introdução aos Testes Automatizados JUnit

Prof. Otávio Lemos (UNIFESP)
Prof. Fabiano Ferrari (UFSCar)

Every programmer knows
they should write tests for their code.
Few do.
Gamma e Beck

Introdução - JUnit

- JUnit: desenvolvimento iniciou em 1997 por Erich Gamma (padrões) e Kent Beck (XP)
- Infectados por teste (test-infected) – importância de se ganhar confiança no programa por meio de testes automatizados
- JUnit: *Framework* para apoiar teste de unidade de programas Java
- Importante para teste de regressão:
  - Os testes podem ser facilmente reexecutados.

Introdução - JUnit

Por que é um "framework" e não uma Ferramenta de Teste?

Framework

Aplicação semi-completa. Fornece uma estrutura comum que pode ser compartilhada entre aplicações. O desenvolvedor incorpora o framework a sua aplicação e o estende para seus propósitos específicos. Os frameworks são diferentes de bibliotecas, pois oferecem uma estrutura coerente, não somente um conjunto de classes de utilidade.

Introdução - JUnit

Por que é um "framework" e não uma Ferramenta de Teste?

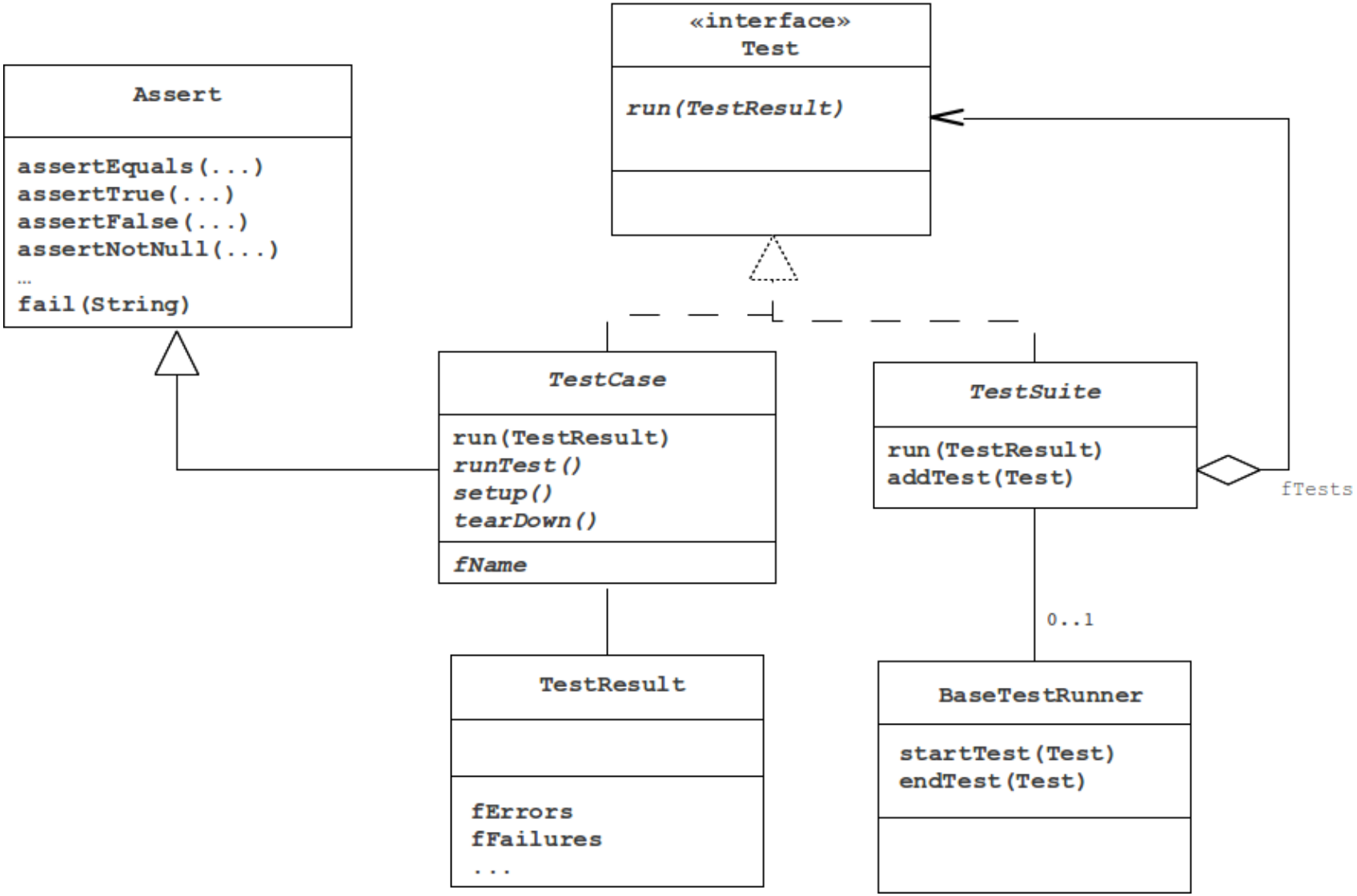
- O JUnit precisa ser estendido para poder ser utilizado.
- Uma ferramenta de teste de software é mais completa que o JUnit.
  - Permite criar sessões (ou projetos) de teste
  - Fornece medidas quantitativas a respeito da qualidade do software em teste e dos casos de teste (quantidade de requisitos de teste exercitados)
  - Cria relatórios sintéticos e analíticos a partir dos resultados do teste

Introdução - JUnit

- JUnit (junit.org) – código aberto, licença da IBM – *Common Public License* – atualmente disponível no GitHub
- Licença – faz com que o software possa ser distribuído sem muitas restrições
- Tornou-se o padrão *de facto* para testes de unidade em Java
- Frameworks xUnit – disponíveis para: ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, REBOL, Smalltalk (entre outros)

Introdução - JUnit

Arquitetura Central



Exemplo

```
public class Calculator {
  public double add(double number1, double number2) {
    return number1 + number2;
  }
}
```

Exemplo

- Preciso testar uma funcionalidade tão simples? Sim, se é uma parte importante do sistema.
- Lembram-se das funções auxiliares defeituosas que causaram desastres?
  - É bom saber que $\mathrm{add}(\mathrm{double},\mathrm{double})$ funciona quando o resto da aplicação que inclui essa operação é entregue.

Programa de teste

```
public class TestCalculator {
  public static void main(String[] args) {
    Calculator calculator = new Calculator();
    double result = calculator.add(10,50);
    if (result != 60) {
      System.out.println("Bad result: " + result);
    }
  }
}
```

Exemplo

- Problemas: e se o código falhar? Ficar olhando a tela com cuidado – mensagem de erro
- Não muito desejável... utilizar exceções é melhor
- Outro problema: e se quisermos mais testes? Mover teste para o seu próprio método...

Programa de teste

```
public class TestCalculator {

  private int nbErrors = 0;
  public void testAdd() {
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    if (result != 60) {
      throw new RuntimeException("Bad result: " + result);
    }
  }

  public static void main(String[] args) {
    TestCalculator test = new TestCalculator();
    try { test.testAdd(); }
    catch (Throwable e) {
      test.nbErrors++;
      e.printStackTrace();
    }
    if (test.nbErrors > 0) {
      throw new RuntimeException("There were " +
        test.nbErrors + " error(s)");
    }
  }
```

Regras em Frameworks de Teste

- Melhorias no programa de teste enfatizam três regras do uso de frameworks de teste:
  1. Cada caso de teste de unidade deve executar independentemente de outros
  2. Erros devem ser detectados e relatados pelo teste
  3. Deve ser fácil definir quais casos de teste vão executar

Como ficaria no JUnit

```
import org.junit.Test;
import static org.junit.Assert.*;
public class TestCalculator {
  @Test
  public void testAdd() {
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    assertEquals(60, result, 0);
  }
}
```

Criação de Casos de Teste

- Quando se quer testar algo utilizando JUnit, é necessário:
  1. Criar uma classe de teste;
  2. Anotar os métodos de teste com org.junit.Test;
  3. Checar saídas obtidas com saídas esperadas utilizando métodos assert* importando-os estaticamente de org.junit.Assert.*. Por exemplo:
     - assertEquals(objeto ou literal com valor esperado, objeto ou variável com valor obtido)
     - assertTrue(variável booleana)
     - assertFalse(variável booleana)
     - assertNotNull(objeto)

Criação de Casos de Teste

- o comando $\mathrm{fail}()$ pode ser utilizado para garantir que o programa não execute determinada parte do teste, em geral lançando uma exceção antes
- $@\mathrm{Test}(\mathrm{expected} = \mathrm{ClasseDeExceção})$ também pode ser utilizado

Exemplo

```
package math;

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

public class BasicMathTest {

        @Test
        public void testSum() {
                int result = BasicMath.sum(1,1);
                assertEquals(2, result);
        }

        @Test
        public void testIsDecimal() {
                assertTrue(BasicMath.isDecimal(2.1));
        }

        @Test
        public void testDivisionByZero() {
                try {
                        BasicMath.divide(10, 0);
                        fail();
                } catch (ArithmeticException e) {}
        }

        @Test(expected = ArithmeticException.class)
        public void testDivisionByZero2() {
                BasicMath.divide(10, 0);
        }
}
```

Configurações de teste (*fixtures*)

- Muitas vezes – necessárias configurações utilizadas em vários testes
- Por exemplo: objetos em determinado estado; conexão com um banco de dados
- Para isso: anotar método com `@Before`, para executá-lo em cada teste, ou `@BeforeClass`, para executá-lo uma só vez para todos os testes
- O mesmo pode ser necessário depois da execução dos testes
- Nesse caso: anotar método com `@After` ou `@AfterClass`

Exemplo

```
package finance;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class MoneyTest {

        private Money f12CHF;
        private Money f14CHF;
        private Money f28USD;

        @Before
        public void setUp() {
                f12CHF= new Money(12, "CHF");
                f14CHF= new Money(14, "CHF");
                f28USD= new Money(28, "USD");
        }

        // ...

}
```

Execução de Casos de Teste

- Os casos de teste podem ser executados por meio de uma classe que invoca um executor de testes.
- Exemplo: `org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...)`
- O Eclipse já possui um plugin nativo para execução de testes JUnit – Recomendado!

Exercícios

1. Implemente os casos de teste funcionais para testar o programa "*Identificador Silly Pascal*" (aula anterior) em JUnit.
2. Implemente os casos de teste funcionais para testar o programa "*Cadeia de Caracteres*" (aula anterior) em JUnit.

- Envie sua resolução via moodle ({programa + casos de teste} compactados)

Exercícios

- Crie classes de equivalência para o seguinte programa:

Considere um módulo de software cujo objetivo é validar (1) o nome de um item de quitanda; e (2) uma lista das diferentes quantidades em gramas nas quais o item pode vir. O nome do item deve conter somente letras e ter de 2 a 15 caracteres. Cada quantidade deve ser um valor de 30 a 2000, números inteiros apenas. As quantidades devem ser cadastradas em ordem ascendente (menores quantidades primeiro). Para cada item, um máximo de cinco quantidades pode ser cadastrada. O nome do item deve ser colocado primeiro, seguido de uma vírgula, seguido pela lista de quantidades. Uma vírgula é usada para separar cada quantidade. Os espaços devem ser ignorados na entrada.