Auditoria e Qualidade de Sistemas

Prof. Edgard Davidson C. Cardoso

Qualidade de Sistemas

TESTE DE UNIDADE - JUNIT

una

Motivação para o uso

- Melhorar a qualidade do Software
- Encontrar erros assim que são criados
- Simplicidade de uso
- Pensar nos testes de unidade ajuda a entender o problema que se está modelando

Testes de Unidade

- Ou testes de Caixa Branca
- Preocupa-se com a forma como o programa é confeccionado.
- O testador deve entender o código para gerar os casos de teste, caso estes sejam feitos após a codificação
- A granularidade fina.

O que testar

- As bordas, quando disponíveis são sempre bons candidatos.
  - Máximos e mínimos
  - Acima dos máximos e abaixo dos mínimos
- Valores válidos e inválidos
- Esperados e não esperados

O que testar

- Se um programa, função, método etc... apresenta um intervalo então valores das bordas deste intervalo são excelentes candidatos a serem testados
  - Ex.: Um programa gera tabuadas do 0 ao 64.

Entrada	Saída	Motivação
0	Tab. do 0	Primeiro valor válido para o programa
64	Tab. do 64	Ultimo valor de entrada válido
-1	Msg. de erro	Primeiro valor abaixo da faixa de val. válidos
65	Msg. de erro	Primeiro valor acima da faixa de val. válidos

O que testar

- Se um programa fixa número máximo de elementos, então estes limites devem ser testados tanto quanto valores intermediários.
  - Ex.: Um dado programa que cria registros é limitado a faixa de 0 a 255 registros. Então uma massa de dados deve ser criada para confirmar e testar estes limites.

Entrada	Saída	Motivação
0 regs	Criação Ok.	Primeiro número de registros válidos
1 reg	Criação Ok.	Número de registros intermediário válido
255 regs	Criação Ok.	Maior volume válido de registros
256 regs	Msg. de erro	Primeiro volume inválido de registros

O que testar

- Considerando orientação a objeto
  - Se um método recebe um objeto bons candidatos a teste são: null, o objeto vazio, ou construído com seu construtor default, e na forma esperada pelo método.

JUnit

- Testes de unidade em Java
- Simples e de fácil utilização
  - Programas de teste
- Pode ser utilizado para testes de sistema e regressão
- Classe a ser testada
- Chamador dos testes
  - Chama os casos de teste
- Casos de teste
  - A classe, ou método, que realmente testa a classe a ser testada

Teste de Unidade

- Verificação da menor unidade de um software
  - Componente ou módulo de software
- Voltado para caixa-branca
- Quando e por quem deve ser feito?
  - Testes de unidade devem idealmente ser criados antes da codificação pelo desenvolvedor

Teste de Unidade

- O que deve ser testado ?
  - Dados de I/O
  - Estrutura de dados
  - Condições limites
  - Caminhos básicos (independentes) da estrutura de controle
  - Iterações
  - Exceções
  - Regras de negócio

- Como verificar exceções?
  - A descrição do erro deve ser clara e correta
  - A mensagem deve estar coerente com o erro encontrado
  - O erro não pode gerar saída inesperada do sistema
  - A descrição do erro deve facilitar a identificação da sua causa

O que testar

- As bordas, quando disponíveis são sempre bons candidatos.
  - Máximos e mínimos
  - Acima dos máximos e abaixo dos mínimos
- Valores válidos e inválidos
- Esperados e não esperados

Exemplo de teste de condições limites

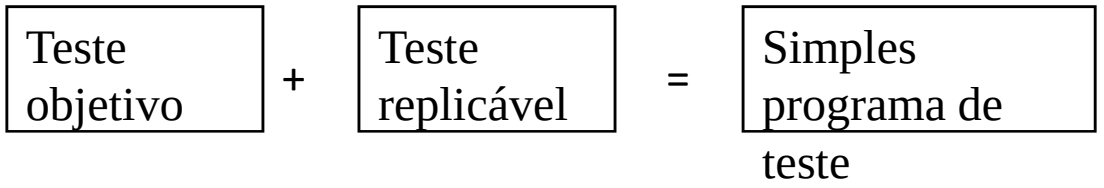
- `for(int i=0; ((i+k)<h) && (i<50)); i++);`

  - Duas condições básicas
    - **i** é menor que 50?
    - **i+k** é menor que h?

  - Casos de teste
    - Teste em que **i+k** seja maior que **h** ao chegar no for
    - Teste em que **i+k** seja o responsável pela saída do for
    - Teste em que **i < 50** seja responsável pela saída do for
    - Teste em que as duas condições ocorrem juntas

JUnit

- *Framework* simples e efetiva para testes de unidade em Java
  - É uma aplicação semi-completa
  - Fornece estrutura comum, reusável, que pode ser compartilhada entre aplicações
  - Desenvolvedores incorporam a *framework* dentro da sua aplicação, extendendo necessidades específicas
- JUnit é open source



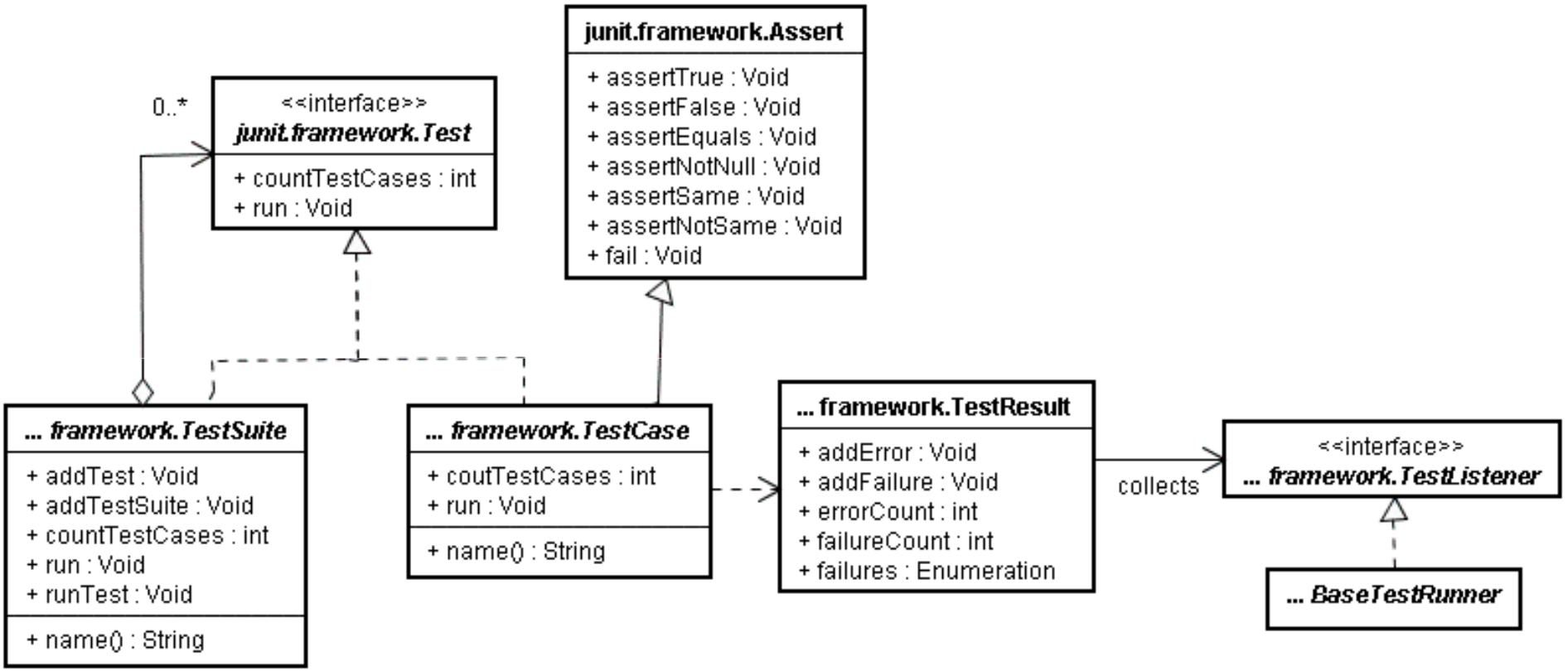
Dois requisitos básicos juntos na idéia de um simples programa de teste

JUnit

- Boas práticas que uma *framework* de testes de unidade
  - Cada teste de unidade deve rodar independente de todos os outros testes de unidade
  - Erros devem ser detectados e reportados teste por teste
  - Deve ser fácil para definir quais testes de unidade irão executar

- Como utilizar ?
  - O JUnit é um arquivo jar (`junit.jar`)
  - Deve ser adicionado ao seu projeto
  - Possui integração com ferramentas como: Eclipse, JBuilder e IntelliJ

JUnit – Estrutura



JUnit – Estrutura

- Test
  - Interface que todos os tipos de classes de teste devem implementar
  - Na framework existe somente duas classes como essa: *TestCase* e *TestSuite*
- TestCase
  - Classe principal que deve ser extendida ao criar testes
  - É uma subclasse de *Test*
  - Possui métodos que implementam testes específicos como *setUp* e *tearDown*
- TestSuite
  - É outra subclasse de *Test*
  - Implementa uma coleção de testes: *TestCase* e outras *TestSuite*

JUnit – Estrutura

- Assert
  - É uma superclasse de *TestCase* que fornece todos os métodos *assert* implementados nos testes
- TestFailure
  - Encapsula um erro ou falha quando ocorre durante a execução dos testes
  - Identifica o teste que falhou e a exceção responsável pelo erro
- TestResult
  - Acumula os resultados dos testes executados
  - Informa quando um teste inicia e termina, bem como erros

JUnit – Estrutura

- TestListener
  - Interface que é implementada por qualquer classe que deseja verificar o progresso de um teste executado
  - Métodos são declarados para notificar o início e o fim de cada teste, bem como a corrência de erros
- TestRunner
  - Executa os testes de um *TestCase* e exibe os resultados
  - Duas versões: gráfica e textual

JUnit - Casos de teste

- Devem estender “junit.framework.TestCase”
- É recomendado que o nome dos casos de teste iniciem com “test”
  - Ex.: testeUm()
- Os casos de teste não recebem parâmetros

JUnit – TestCase

- *TestCase* é a classe mais usada no JUnit
- O nome da classe é importante, deve iniciar ou terminar com a palavra *Test*
  - Exemplo: *TestMinhaClasse* ou *MinhaClasseTest*

```
import junit.framework.TestCase;

public class TestMinhaClasse extends
TestCase
{
}
```

  - O nome permite que os coletores automaticamente encontrem as classes de teste
    - O mesmo se aplica aos *TestSuites*

SetUp

- Inicializa o ambiente para a realização dos testes

```
protected void setUp ()
{
    servidor = “qualquerCoisa.una.br”;
    valorAnterior = ClasseEmTeste.obterValor();
} // set Up
```

TearDown

- Usado para “limpar” o ambiente depois da realização dos testes

```
protected void tearDown ()
{
  ClasseEmTeste.atribuirValor(valorAnterior);
 } // tearDown
```

JUnit – Assertions

- Duas versões:
  - Parâmetro *String* que contem mensagem a ser exibida em caso de falha
  - Sem parâmetro
  - `void fail()`
  - `void fail(String message)`

- *fail*
  - Método simples `fail()`
  - Verificações de exceções são feitas dentro do bloco *try-catch*
    - `Fail` é acionado, caso a exceção esperada não seja levantada

JUnit – Assertions

- **Exemplo** (cont)

```
public void testDivisaoPorZero() {
    try {
        int x = 3 / 0;
        fail (“Divisão por zero não foi identificada”);
    }
    catch (ArithmeticException ex) {
    }
}
```

Exemplos de testes

- Testando se operações retornam exceções indevidas

```
try
    {
      Conexao.conectarBancoOracle(BancoValido);
    } catch (ExcecaoPersistencia e)
    {
      fail(" Excecao percistencia: " + e.getMessage());
    }
```

Exemplos de testes

- Testando se operações não retornam exceções

```
try
    {
     Conexao.conectarBancoOracle(BancoInexistente);
     fail("Deveria ter lancado excecao, Banco invalido");
    } catch (ExcecaoPersistencia e)
    {
     // Ok a exceca foi lancada
    } catch (AssertionFailedError e)
     {
            throw(e); // sobe com o erro encontrado
     } catch (Throable e)
     {
            TestRunner.printTrace(e);
            fail(" Excecao inexperada: " + e.getMessage());
     }
```

JUnit – Assertions

- assertXXX - comandos utilizados para verificar:
  - Valor obtido equivale ao valor esperado
  - Dado nulo ou não
  - Valor verdadeiro ou falso
  - Se é o objeto esperado
  - Se o conteúdo de dois objetos é o mesmo
- Métodos
  - *assertTrue* e *assertFalse*
  - *assertNull* e *assertNotNull*
  - *assertSame* e *assertNotSame*
  - *assertEquals*

JUnit - Casos de teste

- assert – Garante que uma determinada condição seja verdadeira. Caso não seja uma exceção é levantada(**AssertionFailedError**).

  Ex:. *assert("mensagem de erro", var1 == var2);*

- assertEquals – Verifica se dois valores passados como parâmetro são iguais

  Ex:. *assert("mensagem de erro", var1, var2);*

- fail – Retorna uma exceção caso execute esta linha.

  Ex:. *fail("não deveria passar por esta linha");*

JUnit – TestCase

- Corpo da classe
  - Configurações iniciais (pré-condições)
  - Testes das funcionalidades
  - Verificação das pós-condições
- Ordem de desenvolvimento
  - Escrever testes pontuais – identificação dos erros mais precisa
    - Cada teste deve verificar uma parte específica da funcionalidade (ex. um método)
  - Para cada teste pontual do *TestCase*
    - Escrever os *asserts*
    - Fazer o que for necessário para gerar os resultados (ex. dependências existentes em relação a outros cadastros)
  - Configurar pré-condições (gerais para todos os testes do *TestCase*)
- Sugestão: comece pelas funcionalidades mais simples e incremente gradualmente os testes !

JUnit – TestCase

- Exemplo teste pontual

```
public void testeListaVazia ()
{
   ListaFilmes listaVazia = new ListaFilmes();

   assertEquals (“Tamanho da lista vazia deveria
                ser zero.”,0, listaVazia.tamanho());
   assertTrue (“Uma lista deve informar que está
                 vazia.”,listaVazia.eVazia());
}
```

- Todos os métodos devem ter a assinatura da forma
  abaixo para serem identificadas pela *framework* do JUnit

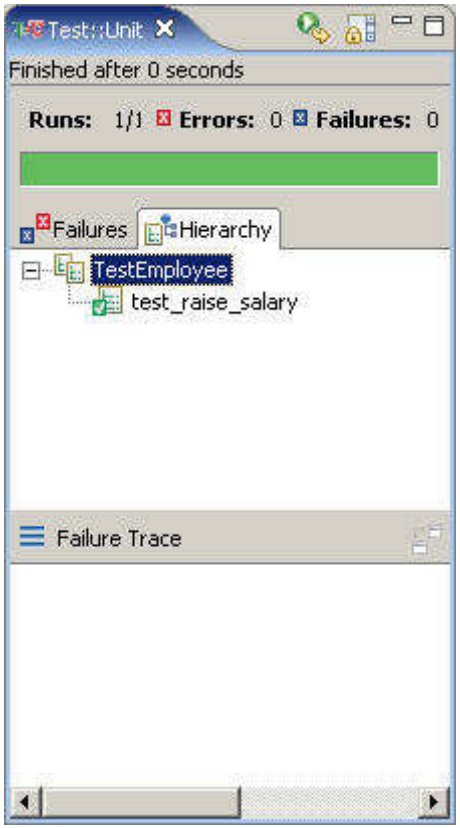
```
public void testQualquerCoisa ()
```

JUnit – Execução dos Testes

- Utiliza-se o *TestRunner* ou plugins para IDEs (ex. Eclipse)
- *TestRunner*
  - Gráfico
    - Cada teste aprovado recebe uma marca de *check* √ e o reprovado um **X**
  - Textual
    - `junit.textui.TestRunner junit.samples.AllTests`
  - Erros e falhas
    - Resposta de um teste: passou (barra verde) ou falhou (barra vermelha)
    - Se não passou, identifica se foi falha ou erro
    - Erro - foi lançada alguma exceção não esperada
    - Falha – resposta de algum *assert* diferente da esperada

JUnit – Execução dos Testes

- Componente swing *TestRunner* do Eclipse



una

Mais informações

- http://www.junit.org
- Kent Beck, Erich Gamma, JUnit Cookbook - http://www.junit.org/junit/doc/cookbook/cookbook.htm
- Kent Beck, Erich Gamma, JUnit Test Infected: Programmers Love Writing Tests - http://www.junit.org/junit/doc/testinfected/testing.htm