Teste Estrutural

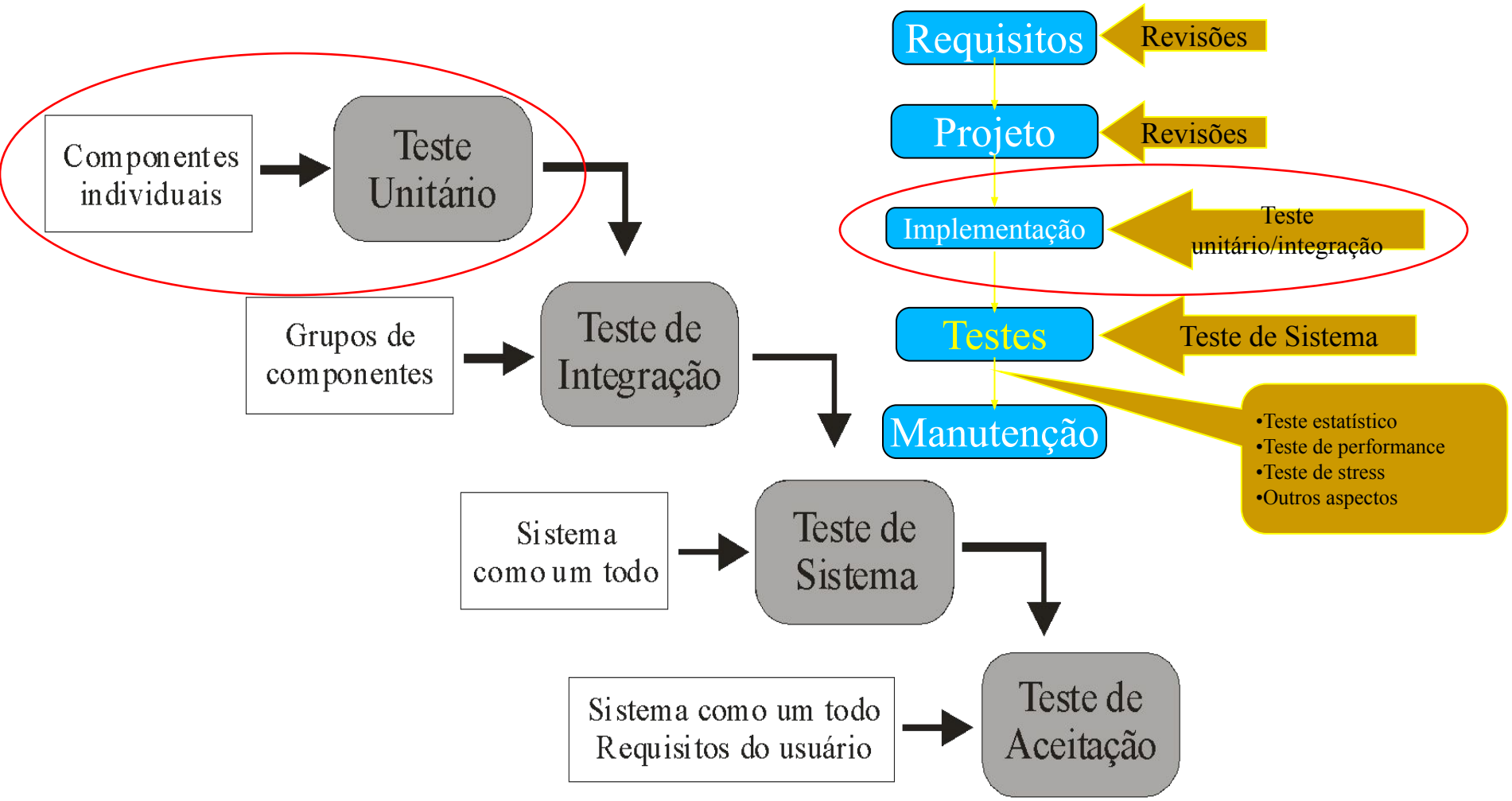
Introdução ao teste unitário

Prof. Iara Carnevale de Almeida
Engenharia de Software II

Teste Unitário: objetivos

- Assegurar que cada unidade está funcionando de acordo com sua especificação funcional
- Projetam-se testes para revelar defeitos relativos:
  - A descrição das funcionalidades
  - Aos algoritmos
  - Aos dados
  - A lógica de controle
- Casos de teste são projetados usando-se técnicas de teste funcional e técnicas de teste estrutural

Localização no Ciclo de vida



Teste Unitário: Definição de Unidade

- Uma unidade é o menor componente de software que se pode testar
  - Em um sistema procedural
    - Função ou procedure
  - Em um sistema orientado a objetos
    - Uma classe
  - Em qualquer um dos casos
    - Um componente comprado de um terceiro e que está sob avaliação (COTS)
    - Um componente que será reusado a partir de uma biblioteca desenvolvida pela própria organização

Teste Unitário: papéis x processo

- Quem testa?
  - Desenvolvedor?
  - Equipe de testes?
- Importante:
  - Projetar os testes **antes** do desenvolvimento da unidade.
  - Os bugs encontrados devem ser registrados como parte da história do módulo.
- A informalidade na etapa de teste unitário faz com que um número maior de bugs seja detectado nas etapas de teste de integração e teste de sistema, onde o custo de localização e correção é maior

Entradas para o teste unitário

- Especificação do módulo antes da implementação do mesmo
  - Desenvolvimento de casos de teste usando técnicas funcionais
  - Fundamental como oráculo
- Código fonte do módulo
  - Desenvolvimento de casos de teste complementares após a implementação do módulo usando técnicas estruturais
  - Não pode ser usado como oráculo

Artefatos gerados pelo teste unitário

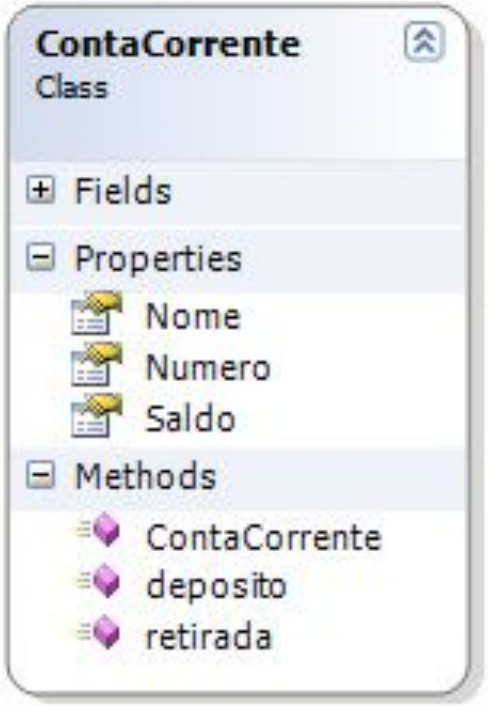
- Dependem do tipo de linguagem de programação
  - Linguagens imperativas
    - Trechos de código que exercitam as funções ou procedures que se deseja testar
  - Linguagens orientadas a objetos
    - Classes *drivers*
  - Em ambos pode ocorrer a necessidade do desenvolvimento de *dubles*

Artefatos gerados pelo teste unitário

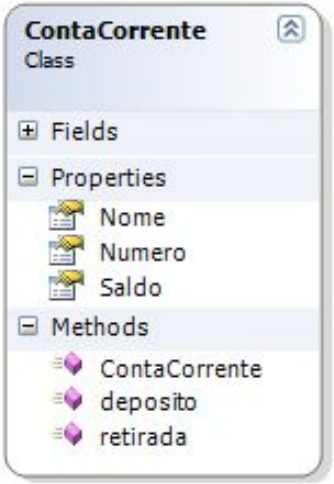
- Classes *drivers*
  - São as classes que contém os casos de teste.
  - Procuram exercitar os métodos da classe "alvo" buscando detectar falhas.
  - Normalmente: uma classe *driver* para cada classe do sistema.
- *Dubles*
  - Simulam o comportamento de classes necessárias ao funcionamento da classe "alvo" e que ainda não foram desenvolvidas.
  - Quando a classe correspondente ao duble estiver pronta será necessário re-executar o *driver* que foi executado usando-se o *duble*.

Exemplo de criação de classe “driver”

- A partir do projeto de uma classe pode-se especificar os casos de teste;
- Deve-se criar um conjunto de casos de teste capaz de cobrir as funcionalidades básicas da classe.



Exemplo de Casos de Teste



Configuração para os casos de testes	Conta Nome: “ ” Número: 1 Saldo inicial: R$ 0,00
Casos de teste	(1) Efetuar depósito de R$ 1000,00; Conferir saldo.
	(2) Efetuar depósito negativo; Verificar lançamento de exceção.
	(3) *Efetuar depósito de R$ 1000,00;* Efetuar uma retirada de R$ 1000,00; Conferir saldo.
	(4) Efetuar retirada de R$ 6000,00; Verificar lançamento de exceção.
	(5) Efetuar retirada negativa; Verificar lançamento de exceção.

Exemplo de classe driver

```
public class ContaCorrenteTest{

    public void Depositar( ){
        ContaCorrente target = new ContaCorrente( );
        target.setNome("Fulano");
        target.depositar(1000.0M);
        if (target.getSaldo == 1000.0M)
                Console.WriteLine("CasoTeste1-depositar: Pass");
        else
                Console.WriteLine("CasoTeste1-depositar: Fail");
    }
...
```

Exemplo de classe driver (cont)

```
...
public void Retirar( ){
        ContaCorrente target = new ContaCorrente( );
        target.setNome("Fulano");
        target.depositar(1000.0M);
        target.retirar(1000.0M);
        if (target.getSaldo == 0.0M)
                Console.WriteLine("CasoTeste3-retirar: Pass");
        else
                Console.WriteLine("CasoTeste3-retirar: Fail");

    }
...
```

Vantagens no uso de classes drivers

- Exige que se reflita sobre as funcionalidades da classe e sua implementação **antes** de seu desenvolvimento
- Permite a identificação rápida de bugs mais simples
- Permite garantir que a classe cumpre um conjunto de requisitos mínimos (os garantidos pelos testes)
- Facilita a detecção de efeitos colaterais no caso de manutenção ou *refactoring*

Dificuldades no uso das classes drivers

- Necessidade de construção do “cenário” em cada método
- Necessidade de construir um programa para executar os casos de teste
- Dificuldade em se trabalhar com grandes conjuntos de dados de teste
- Dificuldade para coletar os resultados
- Dificuldade para automatizar a execução dos testes

Exemplo de classe driver (cont)

Finalizar os casos de teste (2), (4) e (5):

Configuração para os casos de testes	Conta Nome: “ ” Número: 1 Saldo inicial: R$ 0,00
Casos de teste	(1) Efetuar depósito de R$ 1000,00; Conferir saldo.
	(2) Efetuar depósito negativo; Verificar lançamento de exceção.
	(3) *Efetuar depósito de R$ 1000,00;* Efetuar uma retirada de R$ 1000,00; Conferir saldo.
	(4) Efetuar retirada de R$ 6000,00; Verificar lançamento de exceção.
	(5) Efetuar retirada negativa; Verificar lançamento de exceção.

Exercício: Projeto Verifica Triângulo

```
public int identificaTriangulo2(int a, int b, int c) {
   if ((a < b + c) && (b < a + c) && (c < b + a)) {
      if ((a == b) && (b == c))
         return (int)tipos.EQUILATERO;
      else
      if ((a != b) && (a != c) && (b != c))
         return (int)tipos.ESCALENO;
      else return (int)tipos.ISOSCELES;
   }
   return (int) tipos.INVALIDO;
}
```

Exercício: Projeto Verifica Triângulo

- Elaborar casos de testes
- Construir classe driver para os casos de testes

RECOMENDAÇÕES

Recomendações

- Projete casos de teste independentes uns dos outros;
- Não teste apenas o "positivo". Garanta que seu código responde adequadamente em todos os cenários;
- Crie um driver para cada classe;
- Inclua o nome do método em cada teste. Exemplo: *Load:*
  - *PositiveLoadTest*
  - *NegativeLoadTest*
  - *PositiveScalarLoadtest*
- Depure os testes quando for o caso. Não se esqueça de que os testes também são código !!

Limites

- O teste unitário não deve cruzar certos limites !!!
- Um teste não é um teste unitário se:
  - "Conversa" com o banco de dados
  - "Comunica-se" através da rede
  - "Interage" com o sistema de arquivos
  - Não pode ser executado ao mesmo tempo que os demais testes unitários
  - Necessita de ajustes na configuração do ambiente (edição de arquivos de configuração) para poder ser executado.

Limites

- Testes que não respeitam os limites não são "de todo maus" ...
- Respeitando os limites teremos:
  - Testes que executam rapidamente
  - Testes com alto grau de acoplamento apenas com as classes que testam
  - Testes sem acoplamento com a camada de persistência ou de interface
  - Testes que "resistem" melhor a manutenção do código !!

Automatização dos Testes Unitários

O Framework xUnit

- Foi criado no contexto do surgimento do eXtreme Programming em 1998;
- Permite a criação de testes unitários:
  - Estruturados
  - Eficientes
  - Automatizados
- Sua concepção adapta-se facilmente aos IDEs de desenvolvimento

xUnit e o Visual Studio

- Desde a versão 2005 o VS possui sua implementação do xUnit

Conceito no xUnit	Conceito no VS	Descrição
Test	TestMethod	Os testes propriamente ditos
Test Fixture	TestClass	Grupos lógicos de testes
Test Suite	Test List	Conjuntos de TestClass (uma biblioteca de testes)
Test Runner	O próprio VS	Máquina de execução dos testes e geração de relatórios

Os recursos do VS: exemplo de driver

```
using Microsoft.VisualStudio.QualityTools.UnitTesting.Framework;

namespace TesteDoTerminalBancario{
    [TestClass()]
    public class ContaCorrenteTest{

        [TestMethod()]
        public void retirarTest(){
            ContaCorrente target = new ContaCorrente(100, "Fulano");
            target.depositar(5000.0M);
            target.retirar(1000);
            Assert.AreEqual(4000.0M,target.Saldo);
        }

        [TestMethod()]
        public void depositarTest(){
            ContaCorrente target = new ContaCorrente(100, "Fulano");
            target.depositar(5000.0M);
            target.depositar(1000.0M);
            Assert.AreEqual(6000.0M, target.Saldo);
        }
    }
}
```

Atributos

Asserções

Asserções

- Testes unitários automatizados são baseados em asserções.
- Asserções são declarações do que acreditamos ser correto. Quando elas falham geram exceções que são capturadas pelo gerador de relatórios de teste unitário do VS.
- O VS disponibiliza 3 classes de asserções
  - Assert
  - StringAssert
  - CollectionAssert

Conjunto de asserções disponíveis

Classe Assert	Classe StringAssert	Classe CollectionAssert
AreEqual	Contains	AllItemsAreInstancesOfType
AreNotEqual	DosNotMatch	AllItemsAreNotNull
AreNotSame	EndsWith	AllItemsAreUnique
EqualsTeste	Matches	AreEqual
Fail	StartsWith	AreEquivalent
GetHashCodeTests		AreNotEqual
Inconclusive		AreNotEquivalent
IsFalse		Contains
IsInstanceOfType		DosNotContain
IsNotInstanceOfType		IsNotSubsetOf
IsNotNull		IsSubsetOf
IsNull		
IsTrue		

Atributos mais comuns

Atributo	Descrição
TestClass()	Indica um conjunto de testes (classe driver)
TestMethod()	Indica um caso de testes
AssemblyInitialize()	O método é executado antes que se execute o primeiro método de teste do primeiro driver selecionado para execução
ClassInitialize()	O método é chamado antes da execução do primeiro teste do driver
TestInitialize()	O método é chamado antes da execução de cada método de teste
TestCleanUp()	O método é chamado após a execução de cada método de teste
ClassCleanUp()	O método é chamado depois de serem executados todos os testes do driver
AssemblyCleanUp	O método é chamado depois de serem executados todos testes do assembly
Description()	Fornece uma descrição para o método
Ignore()	Ignora o driver ou método por qualquer razão
ExpectedException()	Quando se espera que um teste retorne uma exceção

O Processo no VS

- Processo de desenvolvimento com teste unitário (versão 1):
  1. Definir a interface (esqueleto) da classe alvo
  2. Definir o conjunto de casos de teste
  3. Implementar a classe driver
  4. Completar a codificação da classe alvo
  5. Executar os testes
  6. Corrigir os bugs, se houverem

Atributos

- O executor do VS necessita identificar as classes e métodos de teste.
- Atributos são usados para esta identificação.
- As classes e métodos de teste devem ser corretamente anotadas usando estes atributos.