Análise e Projeto Orientados a Objetos

Testes de unidade

Diretoria Acadêmica de Gestão e Tecnologia da Informação

Introdução

- Por que testar?
  - Para verificar o correto funcionamento do código.
  - Para garantir a qualidade do software.
- Entretanto muitos programadores não testam seus programas como deveriam pois:
  - A execução do teste é tediosa (repetitiva).
  - A elaboração do teste nem sempre é fácil.
  - A interface com o usuário nem sempre colabora para a execução de testes.

Testes Automatizados

- Facilitam a adoção da prática de testes no ambiente de desenvolvimento.
- Dão segurança para alterações de código.
- Essenciais para garantir a qualidade do software.
- Propiciam agilidade na execução dos testes.
- Livra os desenvolvedores do trabalho repetitivo.
- Aumentam a produtividade.

Testes de unidade

- Têm como objetivo testar, de forma isolada, as menores unidades de software.
  - Em programas OO: classes e métodos
- São escritos pelo programadores.
- Facilitam o processo de depuração.
- Documentam o comportamento esperado.
- Levam à criação de classes mais coesas.
- Geralmente utiliza-se uma ferramenta de apoio para a escrita e execução de testes unitários.

Visual Studio 2010 Test Tools

- Conjunto de ferramentas e classes para criação e execução de testes de unidade.

Exemplo – classe Fração

- Vamos escrever e testar uma classe para representar e operar frações matemáticas.
- Abra o VS 2010 e crie um novo projeto do tipo **Class Library**.
- Nomeie o projeto como **FracaoAPI**.

Classe **Fracao**

- Namespace: **FracaoAPI**.
- Atributos inteiros, privados e de somente leitura: **numerador** e **denominador**.
- Propriedades de acesso (get) para os atributos: **Numerador** e **Denominador**.
- Construtor para inicializar os atributos.
- Restrição: denominador não pode ser zero.
  - Construtor deve lançar **ArgumentException**.

Classe **Fracao**

```
class Fracao{
 private readonly int numerador, denominador;
 public int Numerador {
  get { return numerador; }
 }
 public int Denominador {
  get { return denominador; }
 }
 public Fracao(int n, int d){
  if(d == 0) throw new ArgumentException
      ("Denominador não pode ser zero");
  numerador = n;
  denominador = d;
 }
}
```

Escrevendo testes

- Tipicamente criamos uma classe de testes para cada classe a ser testada.
- As classes de teste devem ser criadas em um projeto de testes.
- Adicione um projeto de testes à solução:
  - **File** > **Add** > **new Project**
  - Template: **Visual C#** > **Test** > **Test Project**
- Nome do projeto **FracaoAPITest**.
- Neste novo projeto:
  - Adicione o projeto **FracaoAPI** como referência.
- No arquivo **UnitTest1** do novo projeto:
  - Renomeie o namespace para **FracaoAPI**.
  - Renomeie a classe para **FracaoTest**.

Escrevendo testes

```
namespace FracaoAPI {
 [TestClass]
 public class FracaoTest {

  [TestMethod]
  [ExpectedException(typeof(ArgumentException))]
  public void TestConstrutor_1() {
   new Fracao(2, 0);
  }

  [TestMethod]
  public void TestConstrutor_2() {
   new Fracao(2, 5);
  }
 }
}
```

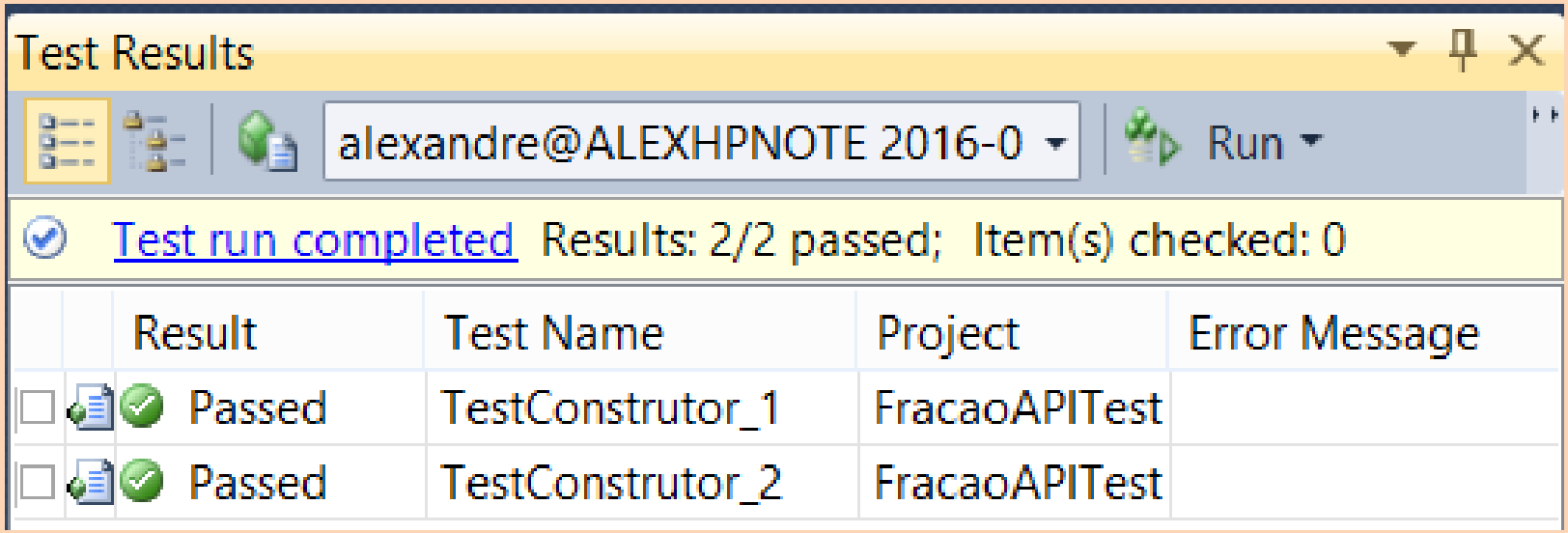
Escrevendo testes

```
namespace FracaoAPI {
 [TestClass]
 public class FracaoTest {

  [TestMethod]
  [ExpectedException(typeof(ArgumentException))]
  public void TestConstrutor_1() {
   new Fracao(2, 0);
  }

  [TestMethod]
  public void Tes
   new Fracao(2,
  }
 }
}
```

O atributo **[TestClass]** indica que a classe contém casos de teste.
O atributo **[TestMethod]** indica que o método é um caso de teste.
O atributo **[ExpectedException]** indica que durante a execução do caso de teste, uma exceção é esperada. Isso significa que o caso de teste só é considerado bem-sucedido se a exceção ocorrer.

Executando os testes

- Clique com o botão direito sobre a classe **FracaoTest** e selecione a opção **Run Tests**.



Assertivas

- São utilizadas para verificar se uma dada condição é verdadeira. Quando a assertiva é válida, o teste passa. Em caso contrário, o teste falha.
- As assertivas são métodos da classe **Assert** (namespace **Microsoft.VisualStudio.TestTools.UnitTesting**).

Assertivas

Assertiva/método	Descrição
AreEqual(Double esperado, Double atual, Double delta)	Verifica se a diferença entre **esperado** e **atual** está dentro de **delta**.
AreEqual(Object esperado, Object atual)	Verifica se **esperado** e **atual** são objetos iguais segundo o método **equals**.
IsFalse(Boolean condição)	Verifica se **condição** é falsa.
IsTrue(Boolean condição)	Verifica se **condição** é verdadeira.

Assertivas

Assertiva/método	Descrição
IsNotNull(Object objeto)	Verifica se **objeto** não é nulo.
IsNull(Object objeto)	Verifica se **objeto** é nulo.
AreNotSame(Object esperado, Object atual)	Verifica se **esperado** e **atual** não são referências para objetos distintos.
AreSame(Object esperado, Object atual)	Verifica se **esperado** e **atual** são referências para o mesmo objeto.
Fail(String mensagem)	Faz com que o teste falhe utilizando uma mensagem informativa.

Método multiplicar

- Deve multiplicar a fração em questão com outra fração e retornar o resultado como uma nova instância.

```
public Fracao Multiplicar(Fracao f)
{
 int n = numerador * f.Numerador;
 int d = denominador * f.Denominador;
 return new Fracao(n, d);
}
```

Método multiplicar: teste

```
[TestMethod]
public void TestMultiplicar() {
 Fracao f1 = new Fracao(1, 2);
 Fracao f2 = new Fracao(3, 5);
 Fracao resultado = f1.Multiplicar(f2);
 Assert.AreEqual(3, resultado.Numerador);
 Assert.AreEqual(10, resultado.Denominador);
}
```

Facilitando as comparações

- Implementar método **Equals** em **Fracao**:

```
public override bool Equals(Object o) {
 if (o is Fracao) {
  Fracao f = (Fracao)o;
  return (numerador == f.Numerador &&
          denominador == f.Denominador);
 }
 return false;
}
```

Facilitando as comparações

- Testar método **Equals**:

```
[TestMethod]
public void TestEquals_1() {
 Fracao f1 = new Fracao(2, 3);
 Fracao f2 = new Fracao(2, 3);
 Assert.IsTrue(f1.Equals(f2));
}

[TestMethod] public void TestEquals_2() {
 Fracao f1 = new Fracao(2, 3);
 Fracao f2 = new Fracao(3, 5);
 Assert.IsFalse(f1.Equals(f2));
}
```

Facilitando as comparações

- Atualizar método **TestMultiplicar**:

```
[TestMethod]
public void TestMultiplicar() {
 Fracao f1 = new Fracao(1, 2);
 Fracao f2 = new Fracao(3, 5);
 Fracao resultado = f1.Multiplicar(f2);
 Fracao esperado = new Fracao(3, 10);
 Assert.AreEqual(esperado, resultado);
}
```

Observações sobre métodos de teste

- Devem ser independentes. Um caso de teste não deve depender dos efeitos ou resultados de outros casos de testes.
- Não há como determinar a ordem de execução dos casos de teste.
- Ocorrências de exceções não previstas acarretam na falha do teste.

Atributos de apoio

- **[TestInitialize]**: marca um método a ser executado antes de cada método de teste.
- **[TestCleanUp]**: marca um método a ser executado após cada método de teste.
- **[ClassInitialize]**: marca um método a ser executado antes do início de todos os métodos de teste.
- **[ClassCleanUp]**: marca um método a ser executado após o término de todos os métodos de teste.

Exercícios

- Implementar e testar métodos para as operações de divisão, adição e subtração de frações.
- Modificar o construtor de forma que sempre sejam criadas frações reduzidas. Escrever testes para a nova implementação.

Referências

- Gayda, Karen. **Unit testing using visual studio 2010**. Disponível em <http://pt.slideshare.net/kgayda/unit-testing-in-visual-studio-2010>.
- Microsoft Developer Network. **Classe Assert**. Disponível em <https://msdn.microsoft.com/pt-br/library/microsoft.visualstudio.testtools.unittesting.assert(v=vs.100).aspx>.
- Richardson, Chris. **POJOS em ação**, Ciência Moderna, Rio de Janeiro: 2007.

Informações bibliográficas

- Autor: Alexandre G. de Lima
- Data: janeiro de 2016.
- Local: Natal-RN