

Curso de C#

Aula 6

Professores

Célio Alencar de Assis
Fábio Duarte Machado

O que será abordado Hoje

- Reflection
- TDD

Reflection

A reflexão é algo interessante que o .Net fornece, com ela podemos escrever código o qual lê as informações do metadado dos objeto em tempo de execução. Essas informações são toda a estrutura existente na classe, portanto métodos, propriedades e até mesmo atributos de classes e métodos são visualizadas.

Como capturar as Informações

Para iniciar a captura de todas as informações da classe, devemos utilizar o namespace **System.Reflection**.

Para visualizar as informações devemos primeiro capturar o **Type** do objeto, ele é responsável por encapsular todas as informações citadas e fornece a possibilidade de manipula-las

Como capturar as Informações

```
private void btnTeste1Reflection_Click(object sender, EventArgs e)
{
    int inteiro = 10; string texto = "Curso C#";
    float Flutuante = 10.2f;
    System.Type Tipo = null;
    Tipo = inteiro.GetType();

    MessageBox.Show(Tipo.Name);
    MessageBox.Show(texto.GetType().Name);
    MessageBox.Show(Flutuante.GetType().Name);
}
```

Listando as propriedades de um objeto

```
private void btnTeste2Reflection_Click(object sender, EventArgs e)
{
    Humano humano = new Humano();

    foreach (var propriedades in humano.GetType().GetProperties())
    {
        MessageBox.Show(propriedades.Name);
    }
}
```


Buscando e atribuindo dados a uma propriedade

```
private void btnTeste3Reflection_Click(object sender, EventArgs e)
{
    Humano humano = new Humano { Idade = 39 };

    PropertyInfo pf = humano.GetType().GetProperty("Idade");
    MessageBox.Show("A idade atual é: " + pf.GetValue(humano).ToString());
    pf.SetValue(humano, 50);
    MessageBox.Show("A idade foi modificada para: " + pf.GetValue(humano).ToString());
}
```

Criando uma instância de uma Classe

```
private void btnTeste4Reflection_Click(object sender, EventArgs e)
{
    var humano = Activator.CreateInstance<Humano>();
    humano.Nome = "Fulano";
    MessageBox.Show("Foi criado uma instância de humano de nome: " + humano.Nome);
}
```


TDD

TDD é uma das práticas de desenvolvimento de software sugeridas por diversas metodologias ágeis, como XP. A ideia é fazer com que o desenvolvedor escreva testes automatizados de maneira constante ao longo do desenvolvimento. Mas, diferentemente do que estamos acostumados, TDD sugere que o desenvolvedor escreva o teste antes mesmo da implementação.

Benefícios

Os benefícios decorrentes do uso de TDD é um software melhor, com mais qualidade, e um código melhor, mais fácil de ser mantido e evoluído.

Por que não testamos?

Não testamos, porque testar sai caro. Imagine o sistema em que você trabalha hoje. Se uma pessoa precisasse testa-lo do começo ao fim, quanto tempo ela levaria?

Semanas? Meses? Pagar um mês de uma pessoa a cada mudança feita no código (sim, os desenvolvedores também sabem que uma mudança em um trecho pode gerar problemas em outro) é simplesmente impossível.

Testes automatizados

Uma maneira para conseguir testar o sistema todo de maneira constante e continua a um preço justo é automatizando os testes. Ou seja, escrevendo um programa que testa o seu programa. Esse programa invocaria os comportamentos do seu sistema e garantiria que a saída é sempre a esperada.

Testes de Unidade

Um teste de unidade testa uma única unidade do nosso sistema. Geralmente, em sistemas orientados a objetos, essa unidade é a classe. Em nosso sistema de exemplo, muito provavelmente existem classes como “CarrinhoDeCompras”, “Pedido”, e assim por diante.

A ideia é termos baterias de testes de unidade separadas para cada uma dessas classes; cada bateria preocupada apenas com a sua classe.

Preciso mesmo escrevê-los?

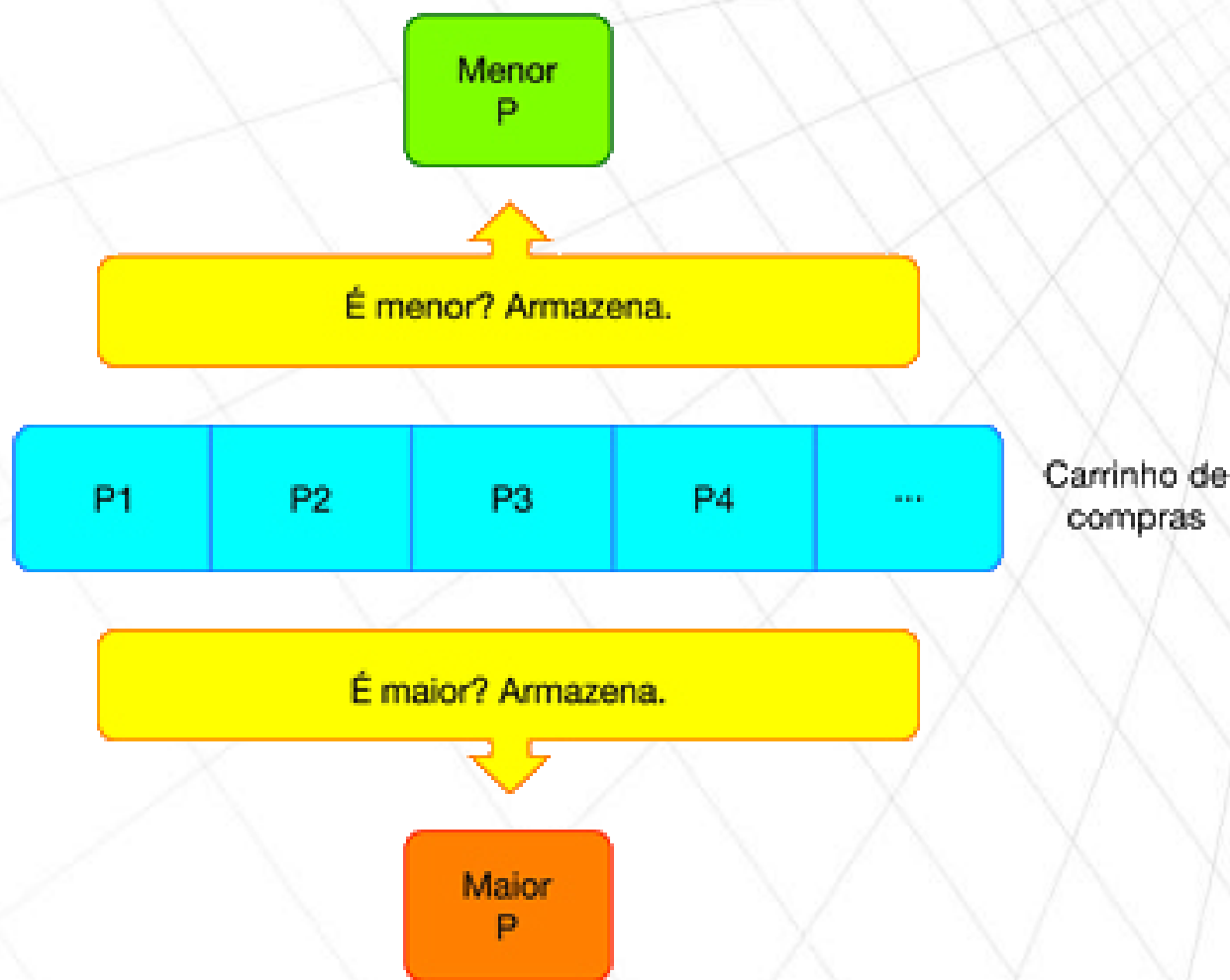
Imagine uma loja virtual que precisa encontrar em seu carrinho de compras, os produtos de maior e menor valor. Um possível algoritmo para esse problema seria percorrer a lista de produtos no carrinho, comparar um a um e guardar sempre a referência para o menor e maior produto encontrado até então.

Possível implementação

```
public class MaiorEMenor
{
    public Produto Menor { get; private set; }
    public Produto Maior { get; private set; }
    public void encontra(CarrinhoDeCompras carrinho)
    {
        foreach (Produto produto in carrinho.Produtos)
        {
            if (Menor == null || produto.Valor < Menor.Valor)
            {
                Menor = produto;
            }
            else if (Maior == null ||
                produto.Valor > Maior.Valor)
            {
                Maior = produto;
            }
        }
    }
}
```

Veja o método *encontra()*. Ele recebe um *CarrinhoDeCompras* e percorre a lista de produtos, comparando sempre o produto corrente com o “menor e maior de todos”. Ao final, temos na propriedade *maior* e *menor* os produtos desejados.

A figura abaixo mostra como o algoritmo funciona:



Para exemplificar o uso dessa classe, veja o código abaixo:

```
private void btnTestarMaiorEMenor_Click(object sender, EventArgs e)
{
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Produto("Liquidificador", 250.0));
    carrinho.adiciona(new Produto("Geladeira", 450.0));
    carrinho.adiciona(new Produto("Jogo de pratos", 70.0));
    MaiorEMenor algoritmo = new MaiorEMenor();
    algoritmo.encontra(carrinho);
    MessageBox.Show("O menor produto: " + algoritmo.Menor.Nome);
    MessageBox.Show("O maior produto: " + algoritmo.Maior.Nome);
}
```

O carrinho contem três produtos: liquidificador, geladeira e jogo de pratos.

É fácil perceber que o jogo de pratos é o produto mais barato (R\$ 70,00), enquanto que a geladeira é o mais caro (R\$ 450,00). A saída do programa é exatamente igual a esperada:

O menor produto: Jogo de pratos

O maior produto: Geladeira

Apesar de aparentemente funcionar, se esse código for para produção, a loja virtual terá problemas.

O Primeiro Teste de Unidade

A classe MaiorEMenor respondeu corretamente ao teste feito acima, mas ainda não é possível dizer se ela realmente funciona para outros cenários. Observe o código abaixo:

```
private void btnTestarMaiorEMenorOrdemTrocada_Click(object sender, EventArgs e)
{
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Produto("Geladeira", 450.0));
    carrinho.adiciona(new Produto("Liquidificador", 250.0));
    carrinho.adiciona(new Produto("Jogo de pratos", 70.0));
    MaiorEMenor algoritmo = new MaiorEMenor();
    algoritmo.encontra(carrinho);
    MessageBox.Show("O menor produto: " + algoritmo.Menor.Nome);
    MessageBox.Show("O maior produto: " + algoritmo.Maior.Nome);
}
```


O Primeiro Teste de Unidade

O código do slide anterior não é tão diferente do anterior. Os produtos, bem como os valores, são os mesmos; apenas a ordem em que eles são inseridos no carrinho foi trocada.

Espera-se então que o programa produza a mesma saída. Mas, ao executa-lo, a seguinte saída é gerada:

O menor produto: Jogo de pratos

System.NullReferenceException

-

Message=Referência de objeto não definida para uma instância de um objeto.

O Primeiro Teste de Unidade

Escrever um teste automatizado não é tarefa tão árdua. Ele, na verdade, se parece muito com um teste manual.

Imagine um desenvolvedor que deva testar o comportamento do carrinho de compras da loja virtual quando existem dois produtos cadastrados:

1. *“clikaria em comprar em dois produtos”,*
2. *“iria para o carrinho de compras”,*
3. *verificaria “a quantidade de itens no carrinho (deve ser 2)” e o “o valor total do carrinho (que deve ser a soma dos dois produtos adicionados anteriormente)”.*

De forma generalizada, o desenvolvedor primeiro pensa em um **cenário**, depois executa uma **ação** e por fim, **valida** a saída.



O Primeiro Teste de Unidade

```
private void btnTestarMaiorEMenorOrdemTrocada_Click(object sender, EventArgs e)
{
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Produto("Geladeira", 450.0));
    carrinho.adiciona(new Produto("Liquidificador", 250.0));
    carrinho.adiciona(new Produto("Jogo de pratos", 70.0));
    MaiorEMenor algoritmo = new MaiorEMenor();
    algoritmo.encontra(carrinho);
    MessageBox.Show("O menor produto: " + algoritmo.Menor.Nome);
    MessageBox.Show("O maior produto: " + algoritmo.Maior.Nome);
}
```

É preciso que o próprio teste faça a validação e informe o desenvolvedor caso o resultado não seja o esperado. Para melhorar o código acima, agora só introduzindo um framework de teste automatizado.

O Primeiro Teste de Unidade

```
private void btnTestarMaiorEMenorOrdemTrocada_Click(object sender, EventArgs e)
{
    CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
    carrinho.adiciona(new Produto("Geladeira", 450.0));
    carrinho.adiciona(new Produto("Liquidificador", 250.0));
    carrinho.adiciona(new Produto("Jogo de pratos", 70.0));
    MaiorEMenor algoritmo = new MaiorEMenor();
    algoritmo.encontra(carrinho);
    MessageBox.Show("O menor produto: " + algoritmo.Menor.Nome);
    MessageBox.Show("O maior produto: " + algoritmo.Maior.Nome);
}
```

É preciso que o próprio teste faça a validação e informe o desenvolvedor caso o resultado não seja o esperado. Para melhorar o código acima, agora só introduzindo um framework de teste automatizado.

Framework de Testes Automatizados

Existem diversos Framework de Teste automatizados, isso varia de acordo com a linguagem de programação utilizada:

- **C#**

1. Nunit
2. UnitTesting do Próprio Visual Studio o qual iremos utilizar.

- **Java**

1. JUnit

- **SmallTalk**

1. SUnit

O Primeiro Teste de Unidade

Indicação de que esta é uma classe de Testes.

[TestClass]

```
public class MaiorEMenorTest
{
```

Indicação de que este é um método de Testes.

[TestMethod]

```
public void OrdemDecrescente()
{
```

//Cenário

```
CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
carrinho.adiciona(new Produto("Geladeira", 450.0));
carrinho.adiciona(new Produto("Liquidificador", 250.0));
carrinho.adiciona(new Produto("Jogo de pratos", 70.0));
MaiorEMenor algoritmo = new MaiorEMenor();
```

//Ação

```
algoritmo.encontra(carrinho);
```

//Validação

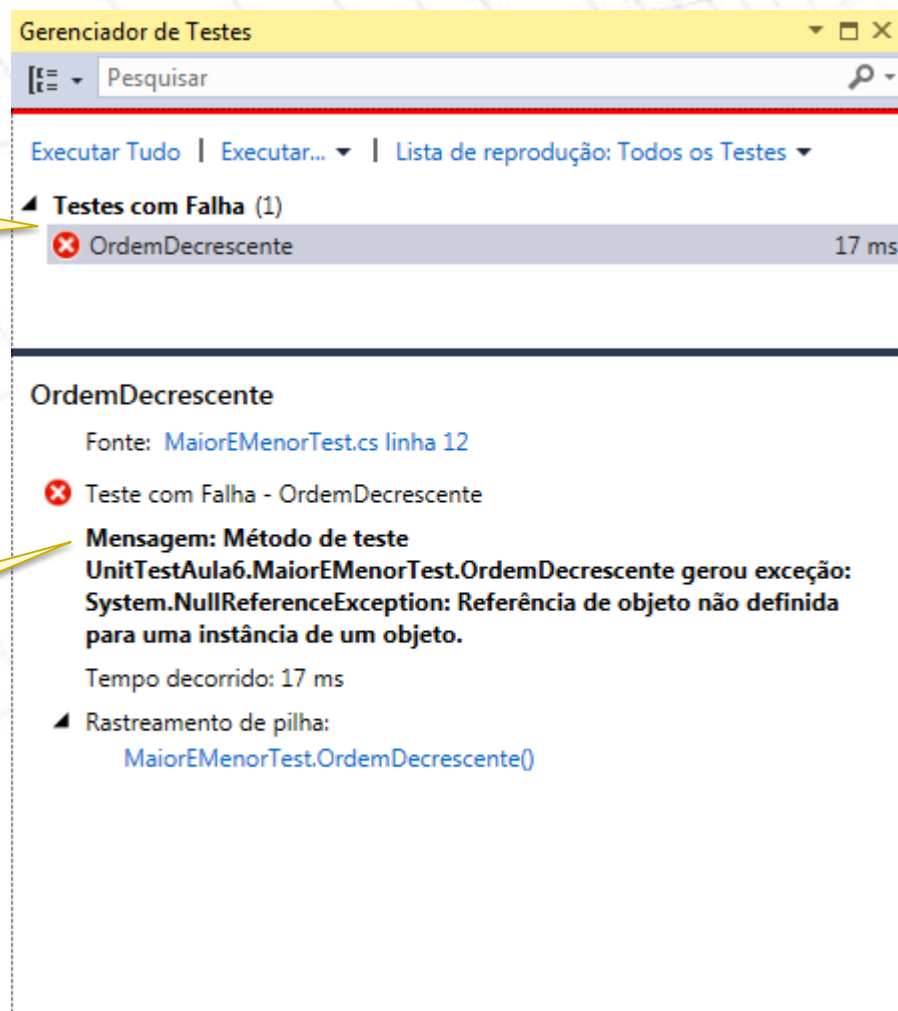
```
Assert.AreEqual("Jogo de pratos", algoritmo.Menor.Nome);
Assert.AreEqual("Geladeira", algoritmo.Maior.Nome);
```

```
    }
}
```


O Primeiro Teste de Unidade

Método de Teste
que falhou

Descrição da Falha



The screenshot shows the 'Gerenciador de Testes' (Test Explorer) window in Visual Studio. At the top, there is a search bar with the text 'Pesquisar'. Below it, a toolbar contains 'Executar Tudo' (Run All), 'Executar...' (Run), and 'Lista de reprodução: Todos os Testes' (Test List: All Tests). The main list shows 'Testes com Falha (1)' (Failed Tests (1)) with a red 'X' icon and the test name 'OrdemDecrescente' with a duration of '17 ms'. Below this, the details for the failed test 'OrdemDecrescente' are shown. It indicates the source as 'MaiorEMenorTest.cs linha 12' and the error message: 'Teste com Falha - OrdemDecrescente'. The message states: 'Mensagem: Método de teste UnitTestAula6.MaiorEMenorTest.OrdemDecrescente gerou exceção: System.NullReferenceException: Referência de objeto não definida para uma instância de um objeto.' (Message: Test method UnitTestAula6.MaiorEMenorTest.OrdemDecrescente threw exception: System.NullReferenceException: Object reference not set to an instance of an object.). It also shows the execution time as 'Tempo decorrido: 17 ms' and the stack trace starting with 'Rastreamento de pilha: MaiorEMenorTest.OrdemDecrescente()'.

Gerenciador de Testes

[Test Icon] Pesquisar

Executar Tudo | Executar... | Lista de reprodução: Todos os Testes

Testes com Falha (1)

✖ OrdemDecrescente 17 ms

OrdemDecrescente

Fonte: [MaiorEMenorTest.cs](#) linha 12

✖ Teste com Falha - OrdemDecrescente

Mensagem: Método de teste
UnitTestAula6.MaiorEMenorTest.OrdemDecrescente gerou exceção:
System.NullReferenceException: Referência de objeto não definida
para uma instância de um objeto.

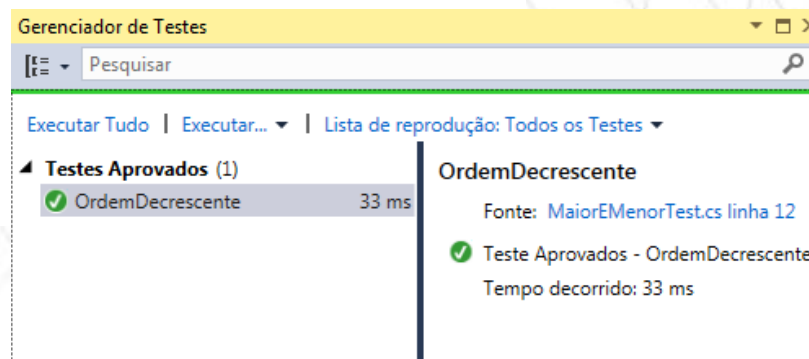
Tempo decorrido: 17 ms

Rastreamento de pilha:
[MaiorEMenorTest.OrdemDecrescente\(\)](#)

O Primeiro Teste de Unidade

```
public class MaiorEMenor
{
    public Produto Menor { get; private set; }
    public Produto Maior { get; private set; }
    public void encontra(CarrinhoDeCompras carrinho)
    {
        foreach (Produto produto in carrinho.Produtos)
        {
            if (Menor == null || produto.Valor < Menor.Valor)
            {
                Menor = produto;
            }
            //else
            if (Maior == null || produto.Valor > Maior.Valor)
            {
                Maior = produto;
            }
        }
    }
}
```

A Causa
da Falha



Test-Driven Development

Desenvolvedores (ou qualquer outro papel que é executado dentro de uma equipe de software) estão muito acostumados com o “*processo tradicional*” de desenvolvimento:

Primeiro a implementação e depois o teste. Entretanto, uma pergunta interessante é:

Será que é possível inverter, ou seja, testar primeiro e depois implementar?

E mais importante, faz algum sentido?



O problema dos números romanos

Numerais romanos foram criados na Roma Antiga e eles foram utilizados em todo o seu império.

Os números eram representados por sete diferentes símbolos, listados

na tabela a seguir.

- I, unus, 1, (um)
- V, quinque, 5 (cinco)
- X, decem, 10 (dez)
- L, quinquaginta, 50 (cinquenta)
- C, centum, 100 (cem)
- D, quingenti, 500 (quinhentos)
- M, mille, 1.000 (mil)

O problema dos números romanos

Para representar outros números, os romanos combinavam estes símbolos, começando do algarismo de maior valor e seguindo a regra:

- Algarismos de menor ou igual valor a direita são somados ao algarismo de maior valor;
- Algarismos de menor valor a esquerda são subtraídos do algarismo de maior valor.

Por exemplo, **XV** representa 15 ($10 + 5$) e o numero **XXVIII** representa 28 ($10 + 10 + 5 + 1 + 1 + 1$). Ha ainda uma outra regra: nenhum símbolo pode ser repetido lado a lado por mais de 3 vezes.

Por exemplo, o número 4 é representado pelo número **IV** ($5 - 1$) e não pelo número **IIII**.

Dado um numeral romano, o programa deve converte-lo para o numero inteiro correspondente.

Cenários

- Um símbolo
- Dois símbolos iguais
- Quatro símbolos dois a dois
- Dois símbolos diferentes do menor para maior

Cenário - Um símbolo - I

```
[TestMethod]
public void DeveEntenderOSimboloI()
{
    //Cenário
    string numeroRomano = "I";
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    //Ação
    int numero = romano.Converte(numeroRomano);
    //Validação
    Assert.AreEqual(1, numero);
}
```

Cenário - Um símbolo - I

Ao executa-lo, o teste falha. Mas não ha problema; isso já era esperado.

Para fazê-lo passar, introduziremos ainda uma segunda regra:

o código escrito deve ser sempre o mais simples possível.

Com essa regra em mente, o código mais simples que fará o teste passar é fazer simplesmente o método converte() retornar o numero 1:

```
public class ConversorDeNumeroRomano
{
    public int Converte(string numeroRomano)
    {
        return 1;
    }
}
```

Cenário - Um símbolo - V

```
[TestMethod]
public void DeveEntenderOSimboloV()
{
    //Cenário
    string numeroRomano = "V";
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    //Ação
    int numero = romano.Converte(numeroRomano);
    //Validação
    Assert.AreEqual(5, numero);
}
```

Cenário - Um símbolo - V

Esse teste também falha.

Novamente faremos a implementação mais simples que resolverá o problema. Podemos, por exemplo, fazer com que o método `Converte()` verifique o conteúdo do número a ser convertido: Se o valor for “I”, o método retorna 1; se o valor for “V”, o método retorna 5:

```
public class ConversorDeNumeroRomano
{
    public int Converte(string numeroRomano)
    {
        if (numeroRomano.Equals("I")) return 1;
        else if (numeroRomano.Equals("V")) return 5;
        return 0;
    }
}
```

Solução Sem IF

Ao invés de escrever um monte de ifs para cada símbolo, é possível usar um switch, que corresponde melhor ao nosso cenário. Melhorando ainda mais, ao invés do switch, é possível guardar os símbolos em uma coleção chaveada do tipo Dictionary que guardaria o algoritmo como chave e o inteiro como valor. Sabendo disso, vamos nesse momento alterar o código para refletir a solução:

Solução Sem IF

```
public class ConversorDeNumeroRomano
{
    Dictionary<string, int> tabela = new Dictionary<string, int>();

    public ConversorDeNumeroRomano()
    {
        tabela.Add("I", 1);
        tabela.Add("V", 5);
        tabela.Add("X", 10);
        tabela.Add("L", 50);
        tabela.Add("C", 100);
        tabela.Add("D", 500);
        tabela.Add("M", 1000);
    }

    public int Converte(string numeroRomano)
    {
        return tabela[numeroRomano];
    }
}
```

Cenário - Dois símbolos iguais - II

```
[TestMethod]
public void DeveEntenderDoisSimbolosComoII()
{
    //Cenário
    string numeroRomano = "II";
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    //Ação
    int numero = romano.Converte(numeroRomano);
    //Validação
    Assert.AreEqual(2, numero);
}
```

Cenário - Dois símbolos iguais - II

Para fazer o teste passar de maneira simples, e possível simplesmente adicionar os símbolos “II” na tabela.

O teste passa. Mas, apesar de simples, essa não parece uma boa ideia de implementação.

É hora de refatorar esse código novamente. Uma possível solução seria iterar em cada um dos símbolos no numeral romano e acumular seu valor; ao final, retorna o valor acumulado.

Cenário - Dois símbolos iguais - II

```
public class ConversorDeNumeroRomano
{
    Dictionary<char, int> tabela = new Dictionary<char, int>();

    public ConversorDeNumeroRomano()
    {
        tabela.Add('I', 1);
        tabela.Add('V', 5);
        tabela.Add('X', 10);
        tabela.Add('L', 50);
        tabela.Add('C', 100);
        tabela.Add('D', 500);
        tabela.Add('M', 1000);
    }

    public int Converte(string numeroRomano)
    {
        int acumulador = 0;
        for (int i = 0; i < numeroRomano.Length; i++)
            acumulador += tabela[numeroRomano[i]];

        return acumulador;
    }
}
```

Cenário - Quatro símbolos dois a dois

XXII

```
[TestMethod]
public void DeveEntenderQuatroSimbolosDoisADoisComoXXII()
{
    //Cenário
    string numeroRomano = "XXII";
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    //Ação
    int numero = romano.Converte(numeroRomano);
    //Validação
    Assert.AreEqual(22, numero);
}
```


Cenário - Dois símbolos diferentes do menor para maior IX

```
[TestMethod]
public void DeveEntenderNumerosComoIX()
{
    //Cenário
    string numeroRomano = "IX";
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    //Ação
    int numero = romano.Converte(numeroRomano);
    //Validação
    Assert.AreEqual(9, numero);
}
```

Cenário - Dois símbolos diferentes do menor para maior IX

Para fazer esse teste passar, é necessário pensar um pouco melhor sobre o problema.

Reparem que os símbolos em um numeral romano, da direita para a esquerda, sempre crescem.

Quando um numero a esquerda é menor do que seu vizinho a direita, esse numero deve então ser subtraído ao invés de somado no acumulador.

Cenário - Dois símbolos diferentes do menor para maior

IX

```
public int Converte(string numeroRomano)
{
    int acumulador = 0;
    int ultimoVizinhoDaDireita = 0;

    for (int i = numeroRomano.Length - 1; i >= 0; i--)
    {
        // pega o inteiro referente ao simbolo atual
        int atual = tabela[numeroRomano[i]];

        // se o da direita for menor, o multiplicaremos
        // por -1 para torná-lo negativo
        int multiplicador = 1;
        if (atual < ultimoVizinhoDaDireita) multiplicador = -1;

        acumulador += atual * multiplicador;

        // atualiza o vizinho da direita
        ultimoVizinhoDaDireita = atual;
    }

    return acumulador;
}
```

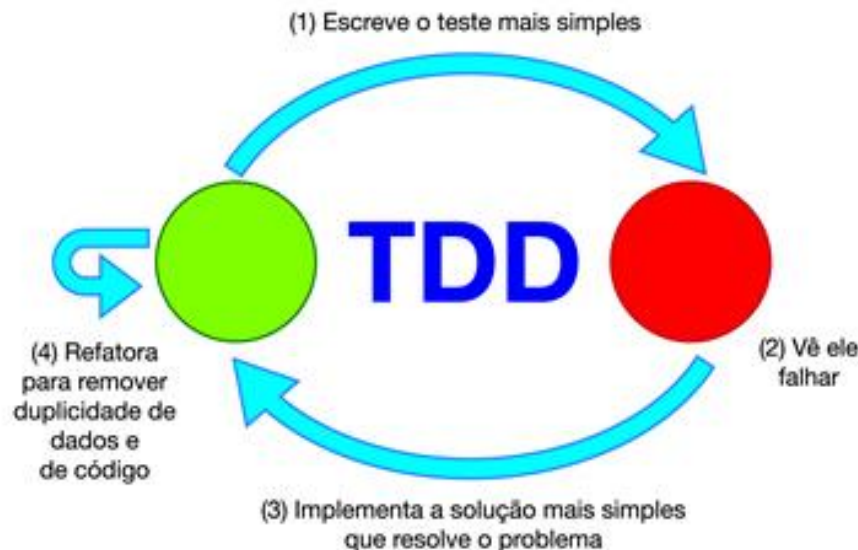
Cenário – Símbolos Complexos como XXIV

```
[TestMethod]
public void DeveEntenderNumerosComplexosComoXXIV()
{
    //Cenário
    string numeroRomano = "XXIV";
    ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
    //Ação
    int numero = romano.Converte(numeroRomano);
    //Validação
    Assert.AreEqual(24, numero);
}
```

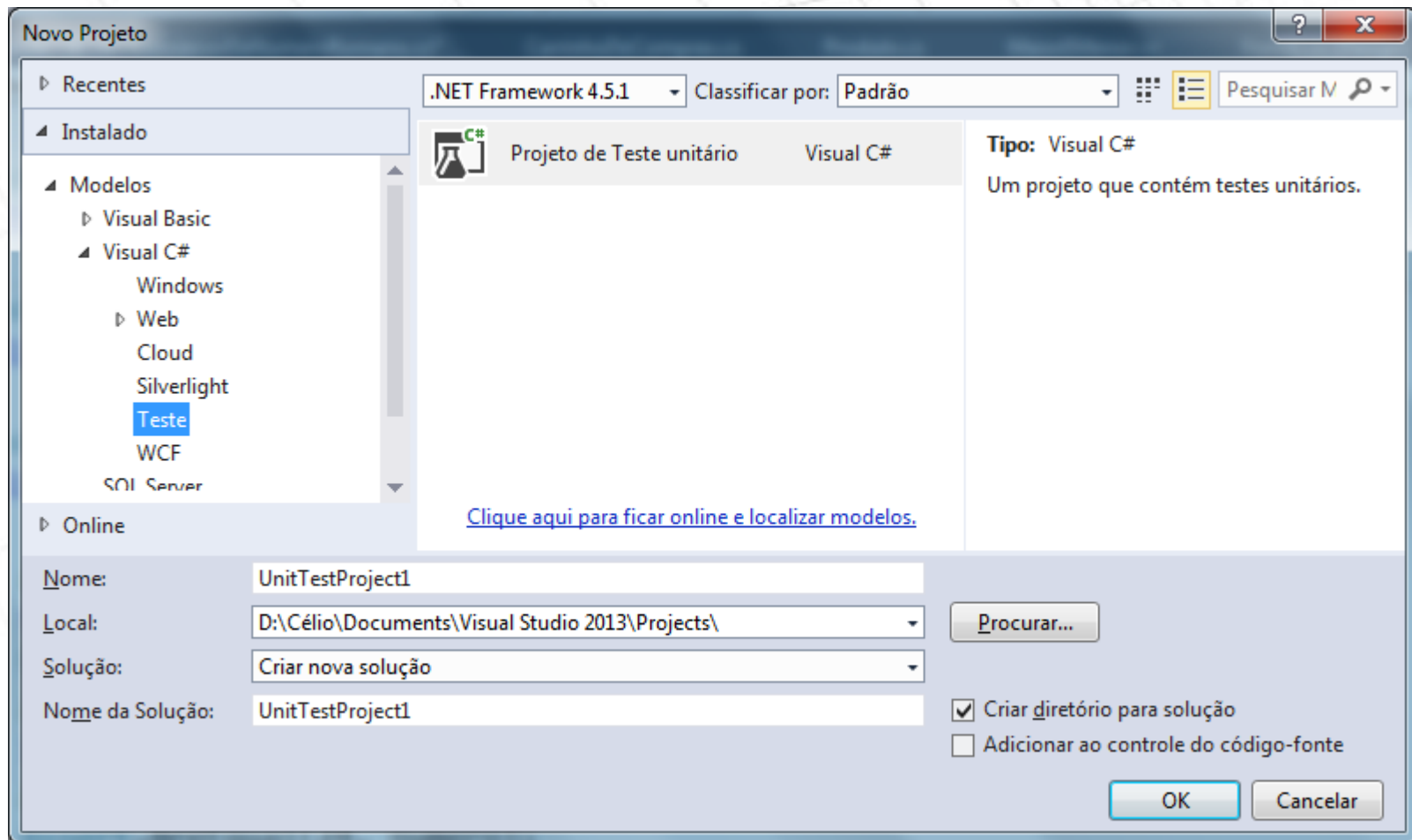
Refletindo sobre o assunto

De maneira mais abstrata, o ciclo que foi repetido ao longo do processo de desenvolvimento da classe acima foi:

- Escrevemos um teste de unidade para uma nova funcionalidade;
- Vimos o teste falhar;
- Implementamos o código mais simples para resolver o problema;
- Vimos o teste passar;
- Melhoramos (refatoramos) nosso código quando necessário.



Novo Projeto de Testes Unitários



Resolver o seguinte problema:

Suponha que uma empresa precise calcular o salário do funcionário e seus descontos.

Para calcular esse desconto, a empresa leva em consideração o salário atual e o cargo do funcionário.

1

As regras de negócio são as seguintes:

- Desenvolvedores possuem 20% de desconto caso seu salário seja maior do que R\$ 3000,0. Caso contrario, o desconto é de 10%.
- DBAs e testadores possuem desconto de 25% se seus salários forem maiores do que R\$ 2500,0. 15%, em caso contrario.

A Classe que ira calcular o salário será chamada de CalculadoraDeSalario, e terá um método CalculaSalario que receberá como parâmetro um funcionário.

```
public enum Cargo
{
    DESENVOLVEDOR,
    DBA,
    TESTADOR
}
```

```
public class Funcionario
{
    public String Nome { get; set; }
    public double Salario { get; set; }
    public Cargo Cargo { get; set; }
}
```

Como lojista Para fidelizar um cliente Quero saber quanto de desconto posso conceder na compra.

②

Critério de Aceite:

Acima de R\$ 100 e abaixo de R\$ 200 => 10%

Acima de R\$ 200 e abaixo de R\$ 300 => 20%

Acima de R\$ 300 e abaixo de R\$ 400 => 25%

Acima de R\$ 400 => 30%

3

Crie uma Classe para criptografar e descriptografar usando a Cifra de César.

Esta é uma forma simples de criptografia, que consistem em mover as letras do alfabeto. Dado um número n , vamos mover as letras no alfabeto, por exemplo:

Dado $n = 3$ e o texto “Rafael Guimaraes Sakurai”, vamos aumentar em 3 cada caractere, assim a versão criptografada ficará :

“Udidho Jzlpdudhv Vdnzudl”, e para descriptografar basta diminuir em 3 cada caractere.

- 4 Cria a Classe DAO de DB Totalmente Genérica usando Reflection.

Referencias Bibliográficas

- Test-Driven Development – Teste e Design no Mundo Real – Maurício Aniche
- Trabalhando com Reflection em C#
<http://www.devmedia.com.br/trabalhando-com-reflection-em-c/26871#ixzz3IR4nOJKC>