TDD simples e prático, Parte III

Fala Desenvolvedores!!!

Neste artigo teremos a continuação da parte II da série sobre TDD. Desta vez codificaremos nosso primeiro pequeno exemplo trivial, um Hello World do TDD! Entre um código e outro, claro que também lembraremos de alguma teoria e alguns pontos interessantes.

Hello World do TDD

Sim. Pra você que está ansioso por algo um pouco mais complexo, realmente neste momento não vamos ter. Assim podemos nivelar o conhecimento com um exemplo trivial, afim de não deixar ninguém perdido.
Vamos lá!

Ambiente

Para o nosso "sistema" vou utilizar **Java+Eclipse+JUnit** . Imagino que você já conheça um pouco de Java e Eclipse, portanto não mostrarei a instalação deles pois fugiremos do escopo do artigo. Caso tenham algum problema, é só entrar em contato.

Alexandre, posso fazer em C#? A vontade! Pode usar C#+NUnit. Posso usar Delphi? A vontade! Pode usar Delphi+DUnit!
Caso alguém queira, posso escrever utilizando um deles também.

Primeiro exemplo

Este será um exemplo bem trivial. No nosso "sistema" precisaremos criar uma Calculadora que calcula as 4 operações básica: Adição, Subtração, Multiplicação e Divisão. Simples assim!
Vamos lá!

Passos para a criação do projeto:
- Crie um novo projeto no Eclipse com o nome de "ArtigoTDD"
- Crie um pacote com o nome "artigotdd.calculadora.teste"

Com a estrutura básica criada, agora vamos criar a nossa primeira classe. A classe Calculadora Alexandre? Não! A classe CalculadoraTeste? Isso mesmo. Vamos fazer um **Teste** em algo que ainda não temos implementado.

Assim, criamos a classe CalculadoraTeste no pacote criado:

```
package artigotdd.calculadora.teste;

public class CalculadoraTeste {

}
```

Tudo certinho por aqui. Agora devemos pensar sobre o nosso problema. Queremos fazer algumas operações nesta calculadora e para começarmos pensaremos em uma soma. Como podemos testar uma soma?
Simples e trivial: Dados dois valores, o resultado deveria ser a soma deles.

Vamos então escrever exatamente este Teste! Vamos criar um método que indique este teste. Para o

JUnit entender que o método é “testável”, temos a anotação “@Test” no método (Não esqueça do import do JUnit).

Assim temos:

```
public class CalculadoraTeste {
	@Test
	public void deveriaSomarDoisValoresPassados() throws Exception {

	}
}
```

Isso mesmo. O nome do nosso método deve mostrar exatamente o que ele está querendo fazer. É comum encontrar métodos de teste começando com “deveria” e inglês você também vai encontrar o “should”.

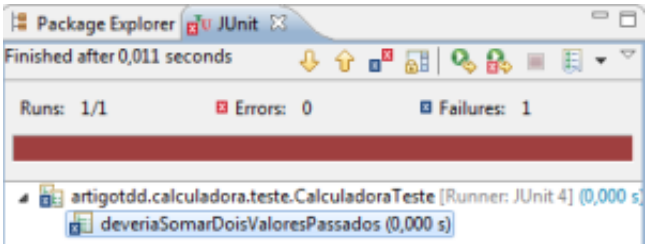
Agora que temos o método de teste, vamos indicar pra ele o que queremos. Vamos agora inserir duas variáveis e usar o método “assertEquals” do próprio JUnit.

Como o próprio nome diz, o “assertEquals” indica que estamos querendo afirmar algo. (Não esqueça do import: “*import static org.junit.Assert.**“)

Vamos ao código:

```
public class CalculadoraTeste {
	@Test
	public void deveriaSomarDoisValoresPassados() throws Exception {
		int valorA = 1;
		int valorB = 2;
		int soma = 0;

		assertEquals(3, soma);
	}
}
```

Pronto! Queremos o resultado 3 para a soma das variáveis *valorA* e *valorB*. Acabamos de escrever o Teste e óbvio que ele não passa. Vamos rodá-lo? Botão direito na classe de teste -> Run As -> JUnit Test.
Barra **vermelha**, era o que temíamos!



Mas já esperávamos pois este é o ciclo: Test->Red->Green->Refactor

E o que o Trace nos diz? Você já ouviu essa frase do seu cliente? “Eu queria isso, mas está fazendo aquilo”. É exatamente o que temos aqui:

No nosso Trace, o JUnit indica que esperava o valor *3* porém foi encontrado o valor *0*.

E agora o nosso objetivo é fazer o Teste passar! Colocamos agora a classe responsável pela

implementação da funcionalidade:

```
public class CalculadoraTeste {
	@Test
	public void deveriaSomarDoisValoresPassados() throws Exception {
		int valorA = 1;
		int valorB = 2;
		Calculadora calculadora = new Calculadora();
		int soma = calculadora.soma(valorA, valorB);

		assertEquals(3, soma);
	}
}
```

Este código nem mesmo compila! Sendo bem ortodoxo em relação ao TDD, realmente este é o nosso próximo passo. Não criamos a classe pra depois usá-la e sim usamos a classe pra depois criá-la.

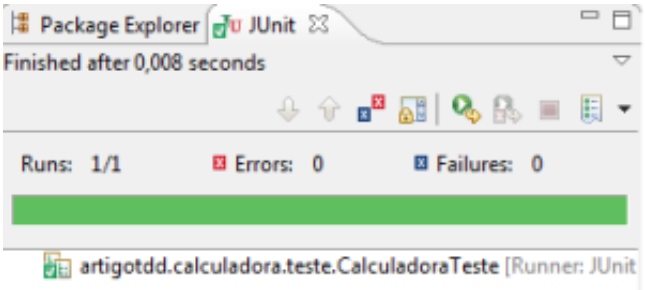
Agora que o compilador gentilmente com uma linha vermelha e um “x” vermelho nos avisou do erro, basta criarmos a classe Calculadora.
Vamos dar um passo um pouquinho maior agora e também criar o método “soma” na classe Calculadora, recebendo dois inteiros:

```
public class Calculadora {

	public int soma(int valorA, int valorB) {
		return 0;
	}
}
```

Ótimo! Se rodarmos o nosso Teste, vamos ver a barra vermelha novamente. Isso por que criamos o método mas ele não implementa o que precisamos.
Vamos agora implementar:

```
public class Calculadora {

	public int soma(int valorA, int valorB) {
		return valorA + valorB;
	}
}
```

Agora sim! Rodando o nosso Teste temos a sonhada barra **verde**:



Agora temos a última etapa do ciclo: **Refatoração**! Como é um exemplo muito trivial não temos aqui alguma refatoração interessante, vamos deixar a refatoração para quando o nosso sistema crescer ou quando as funcionalidades forem modificadas e a refatoração aparecer naturalmente.

Seguindo os mesmos passos anteriores, vamos criar agora o teste para a subtração. Desta vez não faremos passo a passo novamente, pois será idêntico ao método soma:

Teste da subtração:

```
@Test
public void deveriaSubtrairDoisValoresPassados() throws Exception {
        Calculadora calculadora = new Calculadora();
        int valorA = 1;
        int valorB = 2;
        int soma = calculadora.subtrai(valorA, valorB);

        assertEquals(3, soma);
}
```

Método na classe Calculadora:

```
public int subtrai(int valorA, int valorB) {
        return valorA - valorB;
}
```

Finalizando

Neste artigo fizemos uma pequena introdução, escrevendo uma pequena funcionalidade de um sistema já com TDD, utilizando Eclipse e JUnit. Deixarei pra vocês se divertirem implementando a Divisão e Multiplicação.