FIAP GRADUAÇÃO

Prof. Renato Jardim Parducci

DISCIPLINA: **PROJETO DE SISTEMAS APLICADO AS MELHORES PRÁTICAS EM QUALIDADE DE SOFTWARE E GOVERNANÇA DE TI**

AULA:

15 – TDD - TEST DRIVEN DEVELOPMENT

PROFESSOR:

RENATO JARDIM PARDUCCI

PROFRENATO.PARDUCCI@FIAP.COM.BR

Renato Parducci - YouTube

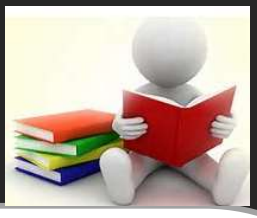
AGENDA DA AULA

- ✓ CMMi nível 3 de maturidade - OPD/OPF/RD
- ✓ MPS.br nível E - DFP/nível D - DRE
- ✓ Desenvolvimento orientado a testes e a comportamento
- ✓ Automação, TDD e BDD

PRÁTICAS E NÍVEL 3 –TS, VER/VAL

Programação orientada por testes TDD

ESTUDO DE CASO SIMULADO



Um membro da equipe GD conversou com um amigo que é desenvolvedor de software em outra empresa e ficou sabendo que lá eles estão usando uma técnica para programar e testar software simultaneamente, fazendo com que os testes Unitários e de Integração aconteçam durante a programação.

Com isso, eles não têm uma etapa ou fase de testes do desenvolvedor. O software, assim que fica com o seu código pronto já pode ser liberado para avaliações Sistêmicas e de Homologação com o PO/usuário, sem a necessidade da equipe de desenvolvedores fazer novos testes de caixa branca, funcionais.

Como C onsuelo conhece bem essa técnica, ele preparou um treinamento do qual você vai participar a partir de agora!

Você vai aprender praticando!
Siga passo a passo as instruções da consultora !

TEST DRIVEN DEVELOPMENT (TDD)

TDD – Test Driven Development é uma técnica de desenvolvimento de software muito difundida entre os desenvolvedores que trabalham com métodos ágeis de produção.

Sua proposta é desenvolver o software a partir dos seus testes, invertendo o processo natural de construir para depois testar.

Essa forma de trabalho proposta no TDD, atende o pedido dos métodos ágeis para que os testes iniciem o quanto antes.

TEST DRIVEN DEVELOPMENT (TDD)

O Que é TDD – Desenvolvimento orientado a testes, afinal?

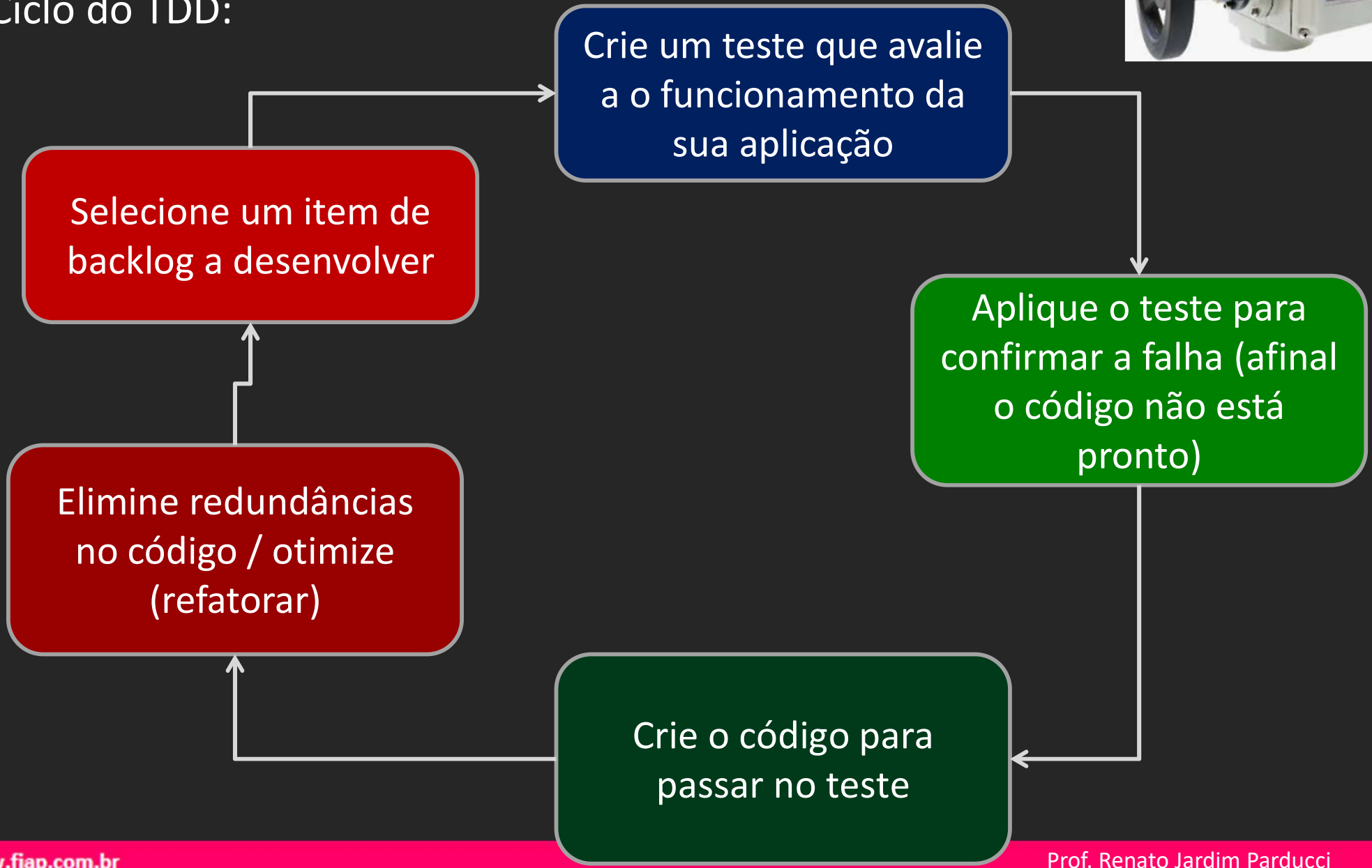
É uma das práticas de desenvolvimento de software sugeridas por diversas metodologias ágeis, como XP e SCRUM. A ideia é fazer com que o desenvolvedor escreva testes automatizados de maneira constante ao longo do desenvolvimento. De forma diferente de como estamos acostumados (primeiro implementar, depois testar), o TDD sugere que o desenvolvedor escreva o teste mesmo antes da implementação.

Por que usar TDD?

- Ajuda a escrever um software com mais qualidade;
- Cria um software mais fácil de ser evoluído e mantido;
- Ajuda a ter um código melhor e mais organizado;
- Testa nossa aplicação de forma rápida e automatizada.

TEST DRIVEN DEVELOPMENT (TDD)

Ciclo do TDD:



TEST DRIVEN DEVELOPMENT (TDD)

O TDD pode ser realizado com qualquer ferramenta de desenvlvimento que possua uma solução de planejamento e execução de teste de software.

Vamos realizar uma exemplificação do TDD, desenvolvendo e testando um código JAVA, aplicando...

eclipse + JUnit

TEST DRIVEN DEVELOPMENT (TDD)

Exemplificação de TDD

Requisito: Fazer Criptografia de dados, convertendo Letras em Números, seguindo a regra:

A = 19	L = 88	W = 94
B = 11	M = 14	X = 23
C =71	N = 87	Y = 72
D = 42	O = 54	Z = 22
E = 13	P = 55	
F = 99	Q = 46	
G = 12	R = 77	
H = 33	S = 61	
I = 68	T = 18	
J = 21	U = 35	
K = 37	V = 56	

TEST DRIVEN DEVELOPMENT (TDD)

Exemplificação de TDD

1º) Crie um novo projeto no Eclipse (New Java Project) com um nome **Criptografia**

2º) Clique no projeto criado (barra da esquerda do Eclipse)

3º) Crie uma pasta para os arquivos que serão produzidos – Com o nome do projeto selecionado, acione a opção New Source Folder e dê o nome **TDDCriptografia** para a pasta de teste (a pasta aparecerá abaixo do projeto no painel do Eclipse

4º) Selecione a pasta criada e acione a opção New Junit Test Case, dando o nome **testadorCriptografico** para o caso de teste que está sendo criado

TEST DRIVEN DEVELOPMENT (TDD)

Exemplificação de TDD

Você terá nesse momento no editor do Eclipse a seguinte declaração para teste da **ConversaoDados** em JAVA:

```
import static org.junit.Assert.*;

public class testadorCriptografico {

   @Test
   public void test() {
      fail ("sem resposta construída");
   }

}
```

TEST DRIVEN DEVELOPMENT (TDD)

Exemplificação de TDD

Dê um novo nome ao método test, imaginando que vamos começar do básico: a conversão da letra A.

```
import static org.junit.Assert.*;

public class testadorCriptografico {

   @Test
   public void testeCriptografiaA () {
      fail ("sem resposta construída");
   }

}
```

TEST DRIVEN DEVELOPMENT (TDD)

Acrescente sentenças para converter o primeiro dado, a letra A.
Codifique o método que devolva um número em Arábico associado à letra A, imaginando que existirá uma classe chamada “ConversoraDados” que faz a conversão através de um método “converte” que devolve um valor para uma variável “arabico”.
Crie um objeto “conversor” no método ConversaoLetraA, que seja gerado na classe ConversoraDados.

```
import static org.junit.Assert.*;

public class testadorCriptografico {
 @Test
   public void  testeCriptografiaA () {
      Criptografadora conversor = new Criptografadora();
      int  nroConvertido = conversor.criptografa (´A´);
   }
}
```

TEST DRIVEN DEVELOPMENT (TDD)

Como se trata de um teste, vamos incluir uma sentença que verifica se o número retornado é o esperado.
Como estamos testando a conversão do A para 19, vamos incluir essa validação:

```
import static org.junit.Assert.*;

public class testadorCriptografico {
 @Test
   public void  testeCriptografiaA () {
      Criptografadora conversor = new Criptografadora();
      int  nroConvertido = conversor.criptografa (´A´);
      assertEquals(19, nroConvertido);
   }
}
```

TEST DRIVEN DEVELOPMENT (TDD)

SEU TESTE ESTÁ PRONTO mas O CÓDIGO COM A LÓGICA DE CONVERSÃO NÃO EXISTE!
ISSO É TDD!

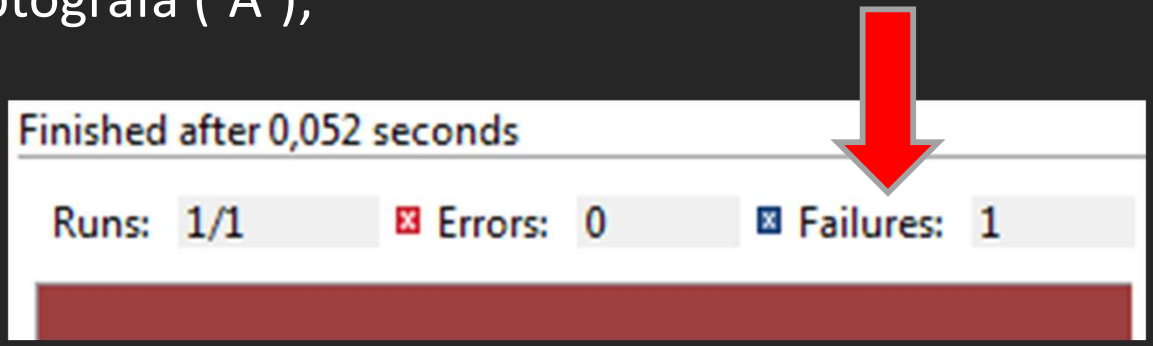
```
import static org.junit.Assert.*;

public class testadorCriptografico {
 @Test
   public void  testeCriptografiaA () {
      Criptografadora conversor = new Criptografadora();
      int  nroConvertido = conversor.criptografa (´A´);
      assertEquals(19, nroConvertido);
   }
}
```

TEST DRIVEN DEVELOPMENT (TDD)

RODE O TESTE!
ELE DEVE FALHAR NESTE MOMENTO POIS A CLASSE NEM EXISTE!
SOMENTE O TESTE FOI CONSTRUÍDO!

EXECUTE O CÓDIGO E VEJA O RESULTADO!

```
import static org.junit.Assert.*;

public class testadorCriptografico {
 @Test
   public void  testeCriptografiaA () {
      Criptografadora conversor = new Criptografadora();
      int  nroConvertido = conversor.criptografa (´A´);
      assertEquals(19, nroConvertido);
   }
}
```



TEST DRIVEN DEVELOPMENT (TDD)

O TDD prega que um problema seja resolvido por vez, partindo da situação mais simples.

Não vamos criar uma lógica de aplicação completa para resolver todos os algarismos!
Vamos passo a passo, iniciando pela conversão de A para 19.

Vamos criar o código que traga o retorno esperado!

TEST DRIVEN DEVELOPMENT (TDD)

Crie a Classe e Método de conversão de A para 19, respeitando a regra funcional.

1º) acesse o menu do projeto novamente, clicando em **SRC, NEW, CLASS**
2º) dê o nome "**Criptografadora" para a classe.**

TEST DRIVEN DEVELOPMENT (TDD)

Não crie a lógica conversão de A para 19. Dê o nome "criptografa" para o método da classe – force o retorno da resposta correta!

```
public class Criptografadora {

  public int criptografar(char Letra) {
    return 19;
  }
}
```

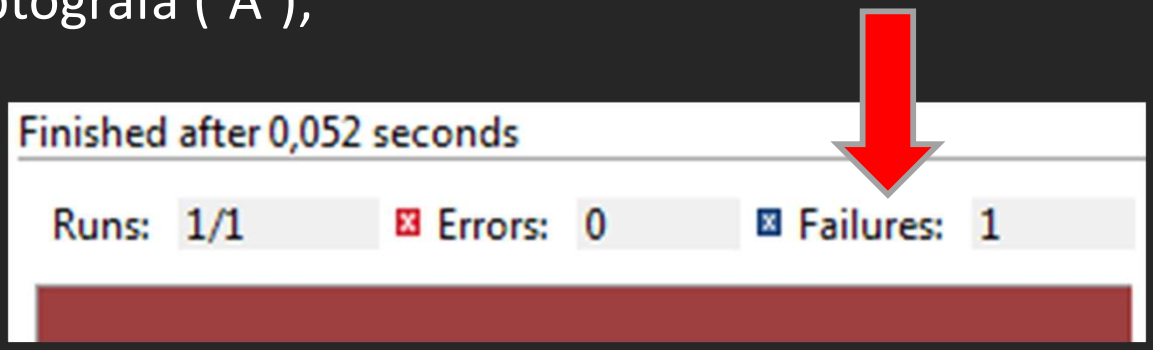
Perceba que forçamos para que tudo dê certo. Criamos um código que teoricamente funciona, da forma mais simples.

TEST DRIVEN DEVELOPMENT (TDD)

RODE O TESTE!
ELE DEVE FALHAR NESTE MOMENTO POIS O MÉTODO DA CLASSE ESTÁ COM O CONTRATO ERRADO!

```
public class Criptografadora {

  public int criptografar(charLetra) {
    return 19;
  }
}
```

```
import static org.junit.Assert.*;

public class testadorCriptografico {
 @Test
   public void  testeCriptografiaA () {
     Criptografadora conversor = new Criptografadora();
     int  nroConvertido = conversor.criptografa (´A´);
     assertEquals(19, nroConvertido);
   }
}
```



TEST DRIVEN DEVELOPMENT (TDD)

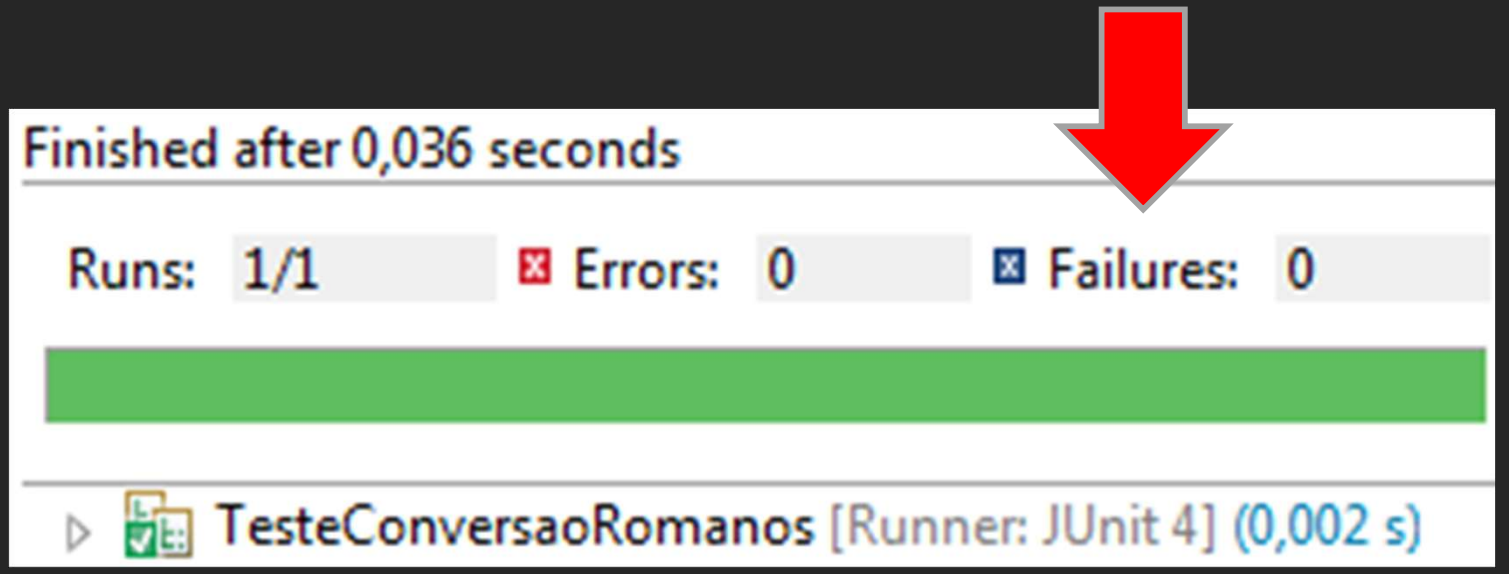
Ajuste o contrato do método!

```
public class Criptografadora {

   public int criptografa(char Letra) {
      return 19;
   }
}
```

Corrija a declaração do parâmetro recebido e de retorno, bem como o nome da classe!

TEST DRIVEN DEVELOPMENT (TDD)

RODE O TESTE NOVAMENTE E VEJA QUE NÃO EXISTIRÃO MAIS ERROS!



TEST DRIVEN DEVELOPMENT (TDD)

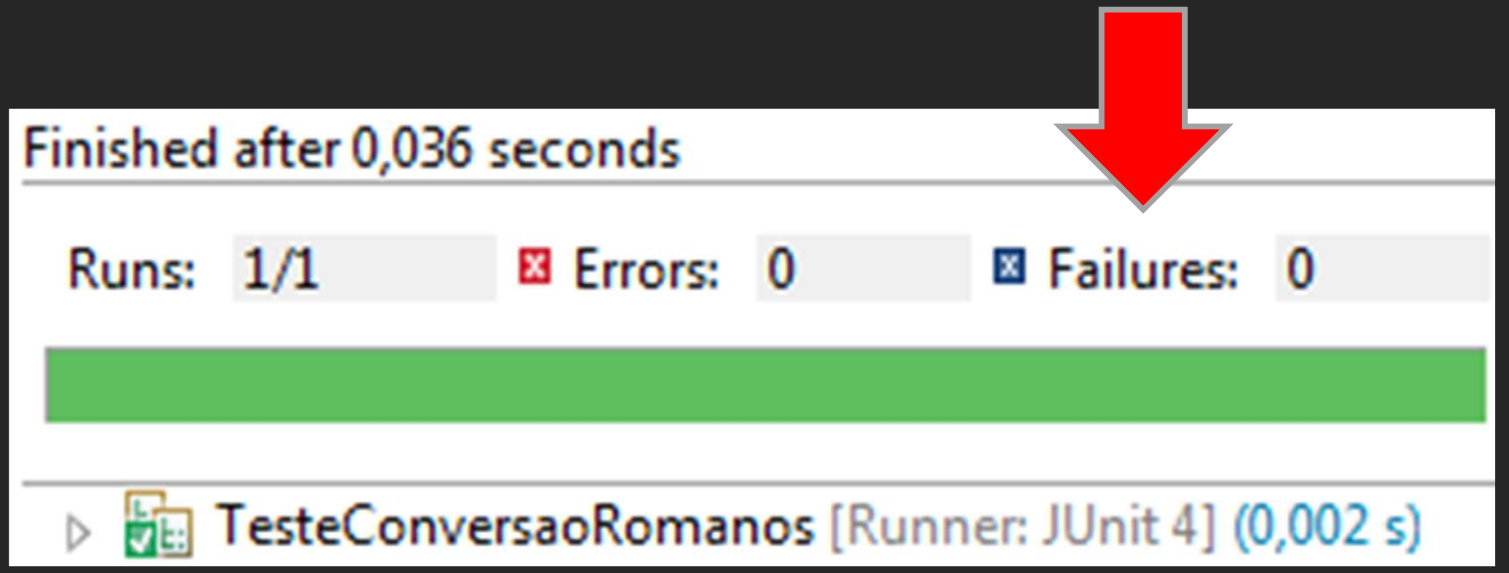
AGORA SIM, REFATORE a aplicação criando a lógica conversão de "A" para 19 de forma a deixar a aplicação flexível para tratar a LETRA que é uma variável e não uma constante.

```
public class Criptografadora {

   public int criptografa(char Letra) {
      if (Letra  ==  ´A´) return 19;
      return 0;
   }
}
```

TEST DRIVEN DEVELOPMENT (TDD)

RODE O TESTE NOVAMENTE E VEJA QUE NÃO EXISTIRÃO ERROS!
SEU CÓDIGO FOI CONSTRUÍDO JÁ FUNCIONANDO!

VOCÊ NÃO FARÁ MAIS FUTUROS TESTES UNITÁRIOS COM ESTE CASO DE TESTE QUE AVALIA A CONVERSÃO DA LETRA COM VALOR "A". SEU CASO DE TESTE FUNCIONAL, UNITÁRIO DE CAIXA BRANCA, JÁ FOI APLICADO!



TEST DRIVEN DEVELOPMENT (TDD)

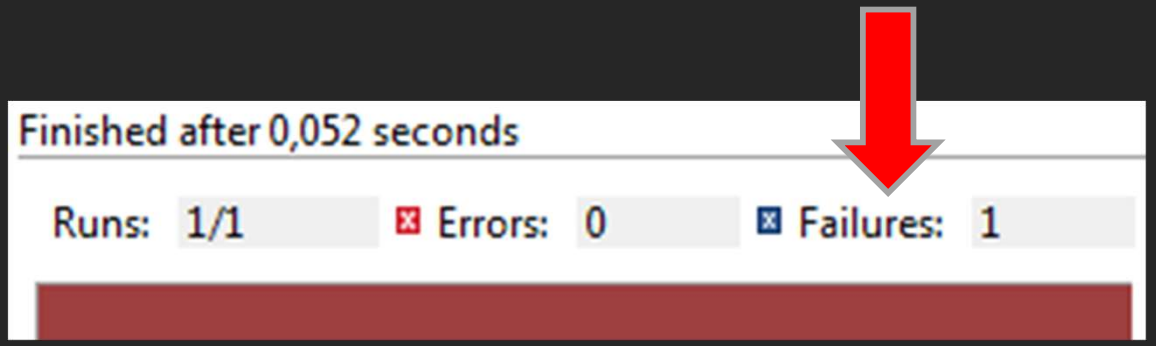
Crie um teste para a letra B que deve ser convertida para 11.

```
import static org.junit.Assert.*;

public class testeCriptografia {
 @Test
   public void  testeCriptografiaA () {
      Criptografadora conversor = new Criptografadora();
      int nroConvertido = conversor.criptografa (´A´);
      assertEquals(19,nroConvertido);
   }
 @Test
 public void testeCriptografiaB () {
      Criptografadora conversor = new Criptografadora();
      int nroConvertido = conversor.criptografa (´B´);
       assertEquals(11, nroConvertido);
   }
}
```

TEST DRIVEN DEVELOPMENT (TDD)

EXECUTE O CASO DE TESTE DO CÓDIGO E VEJA O RESULTADO!

O TESTE DEVE FALHAR PARA A LETRA B!
O MÉTODO DE CRIPTOGRAFIA AINDA NÃO IMPLEMENTOU ESSA LÓGICA!



TEST DRIVEN DEVELOPMENT (TDD)

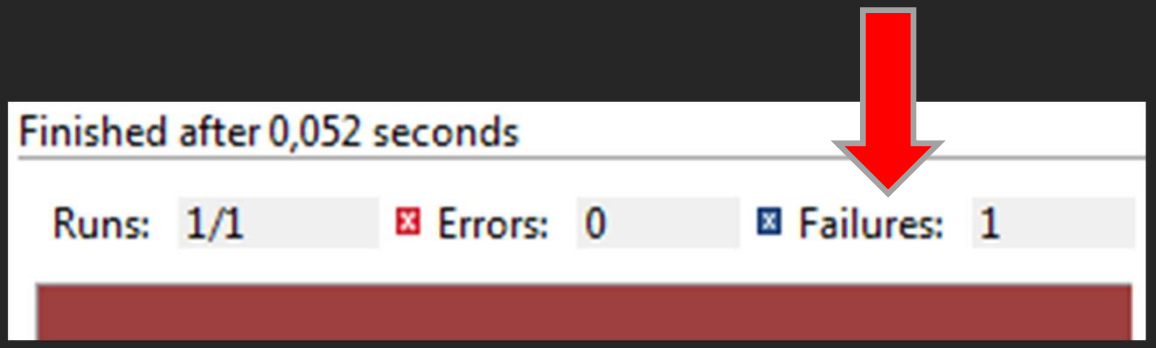
Vamos refatorar a nossa Classe e Método da aplicação para tratar a conversão de B para 11 no método criptografa.

```
public class Criptografadora {

   public int criptografa(char Letra) {
      if (Letra  ==  ´A´) return 19;
      else if (Letra  == ´B´)) return 111;
      return 0;
   }

}
```

TEST DRIVEN DEVELOPMENT (TDD)

EXECUTE O CASO DE TESTE DO CÓDIGO E VEJA O RESULTADO!

O TESTE DEVE FALHAR PARA A LETRA B!
O MÉTODO DE CRIPTOGRAFIA DE B DEVERIA RESPONDER COM 11 E NÃO 111!
PEGAMOS UM POSSÍVEL ERRO DE DIGITAÇÃO DO PROGRAMADOR!
VEJA QUE NÃO TEMOS MAIS PREOCUPAÇÃO COM A CONVERSÃO DE A!



TEST DRIVEN DEVELOPMENT (TDD)

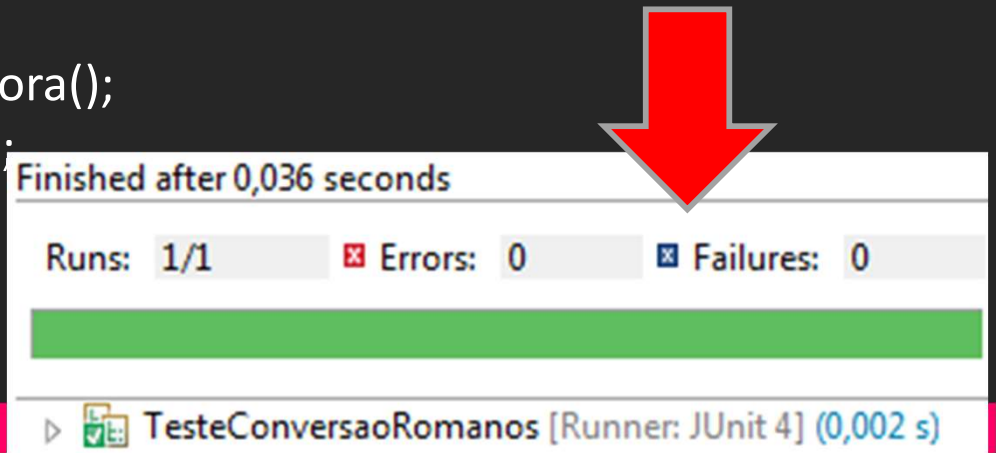
Vamos refatorar a nossa Classe e Método da aplicação para tratar a correção da lógica.

```
public class Criptografadora {

   public int criptografa(char Letra) {
      if (Letra  ==  ´A´) return 19;
      else if (Letra  == ´B´)) return 11;
      return 0;
   }

}
```

TEST DRIVEN DEVELOPMENT (TDD)

EXECUTE O CASO DE TESTE NOVAMENTE E VEJA O RESULTADO!
O TESTE VAI FUNCIONAR!

```
import static org.junit.Assert.*;

public class testeCriptografia {
 @Test
   public void  testeCriptografiaA () {
      Criptografadora conversor = new Criptografadora();
      int nroConvertido = conversor.criptografa (´A´);
      assertEquals(19, nroConvertido);
   }
 @Test
 public void testeCriptografiaB () {
      Criptografadora conversor = new Criptografadora();
      int nroConvertido = conversor.criptografa (´B´);
       assertEquals(11, nroConvertido);
   }
}
```

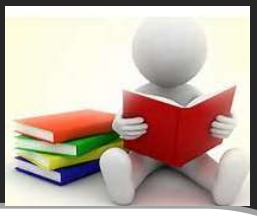


TEST DRIVEN DEVELOPMENT (TDD)

Note que o desenvolvimento do código fonte com a lógica da aplicação acontece para atender uma expectativa de SAÍDA mediante uma ENTRADA de dados.

A construção portanto fica orientada aos testes!

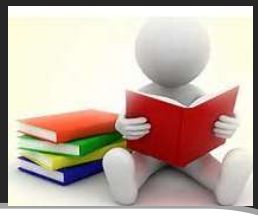
ESTUDO DE CASO SIMULADO



Você está trabalhando em um jogo chamado Cai Não Cai onde, o sistema sorteia um número aleatório de 0 a 10 e se der 0 no sorteio, uma prancha de madeira pneumática vai tombar, derrubando uma pessoa que está sentada nela num tanque de lama!

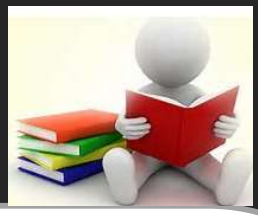
Use TDD para programar e testar a função que sorteia o número e devolve um sinal para derrubar ou não o jogador!

ESTUDO DE CASO SIMULADO



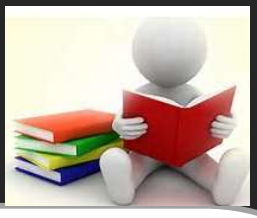
Um programa precisa ser criado para comparar um número com o valor 10000 (dez mil) e se for Menor que isso, o sistema responde “Valor muito baixo para o financiamento!” e caso contrário, o sistema responde “Financiamento enviado para análise!”.

ESTUDO DE CASO SIMULADO



Use TDD para produzir uma função de uma calculadora que some dois números inteiros!

ESTUDO DE CASO SIMULADO



Você teve que refazer a programação de uma Classe de Objetos que tenha métodos para coletar a digitação de três números em sequência e usar esses três números em um método de comparação que apontará o menor e o maior número da sequência digitada.

Construa essa funcionalidade com TDD!

JUNIT E AUTOMAÇÃO

A aplicação da JUNIT permite um tipo de automação de teste unitário chamado AUTOMAÇÃO POR SCRIPT.

A JUNIT pode ser usada tanto em produção TDD quando na produção tradicional de software e permite:

- Disciplinar o formato dos testes;
- Documentar os casos de teste;
- Aplicar um mesmo teste múltiplas vezes, se necessário (repetir);
- Aproveitar um teste criado por um programador por outras pessoas do time de desenvolvimento (reusar);
- Agilizar a execução dos casos de testes e avalição dos resultados;
- Avaliar se os testes criados cobrem as situações previstas na lógica de um programa de aplicação (análise de cobertura dos testes).

GERENCIAMENTO DO TESTE DE SOFTWARE

Vamos usar JUNIT dentro do Eclipse, aprendendo na prática a criar os testes automatizados, só que agora, no modelo TRADICIONAL de produção onde, primeiro codificamos o software e depois testamos!

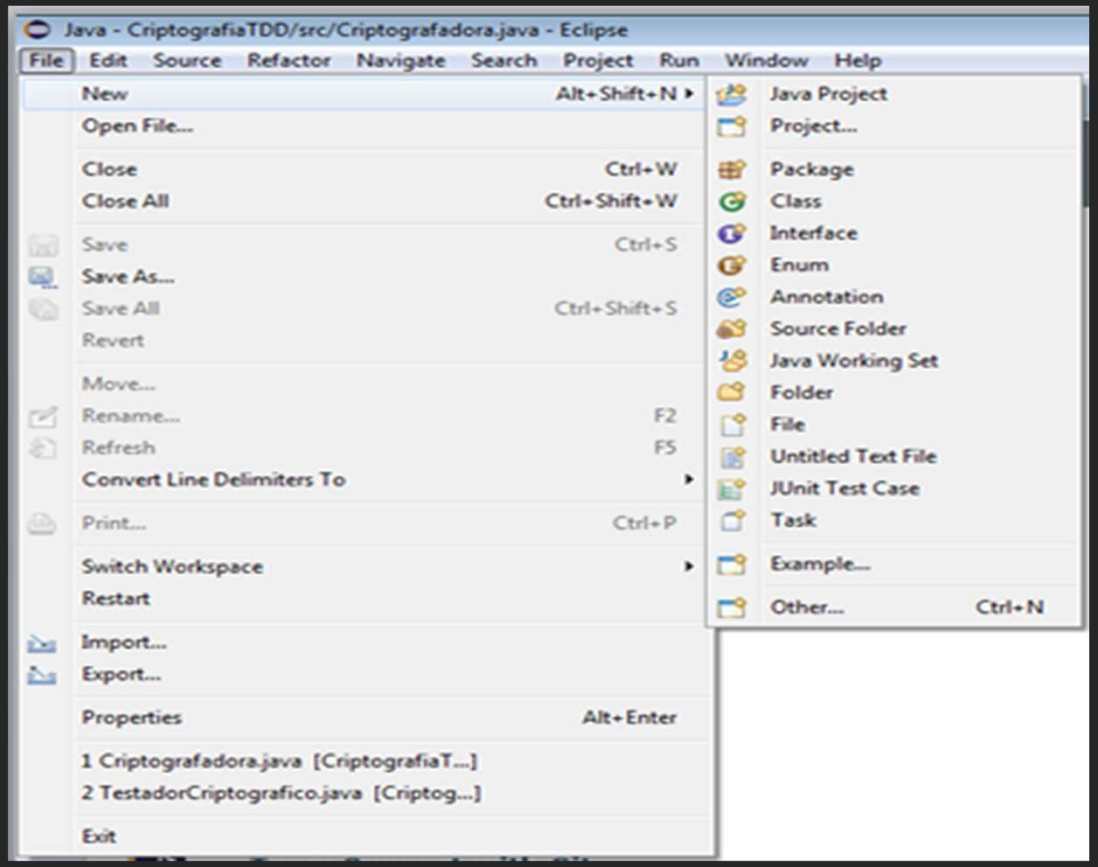
GERENCIAMENTO DO TESTE DE SOFTWARE

Vamos iniciar pela criação de uma Classe JAVA para trabalharmos os testes.

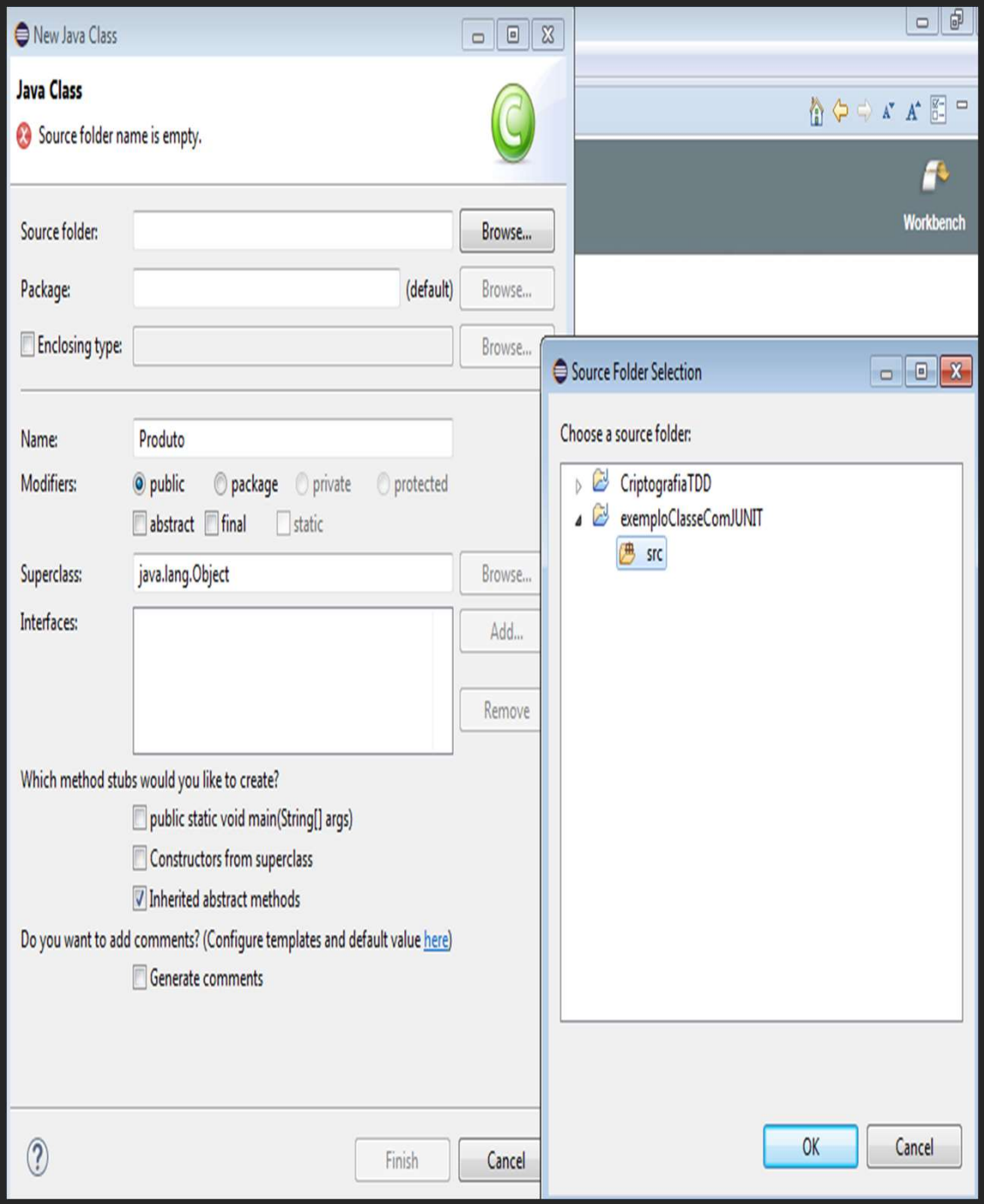
Nos exemplos a seguir, é considerado que a Classe foi escrita exatamente conforme os algoritmos de especificação e modelo UML que foram definidos para seus atributos e métodos.

Queremos confirmar que ela funciona adequadamente, através dos testes.

Crie um projeto no Eclipse e depois, ... a Classe JAVA Calculadora, descrita ao lado *(new JAVA Class EJB)*

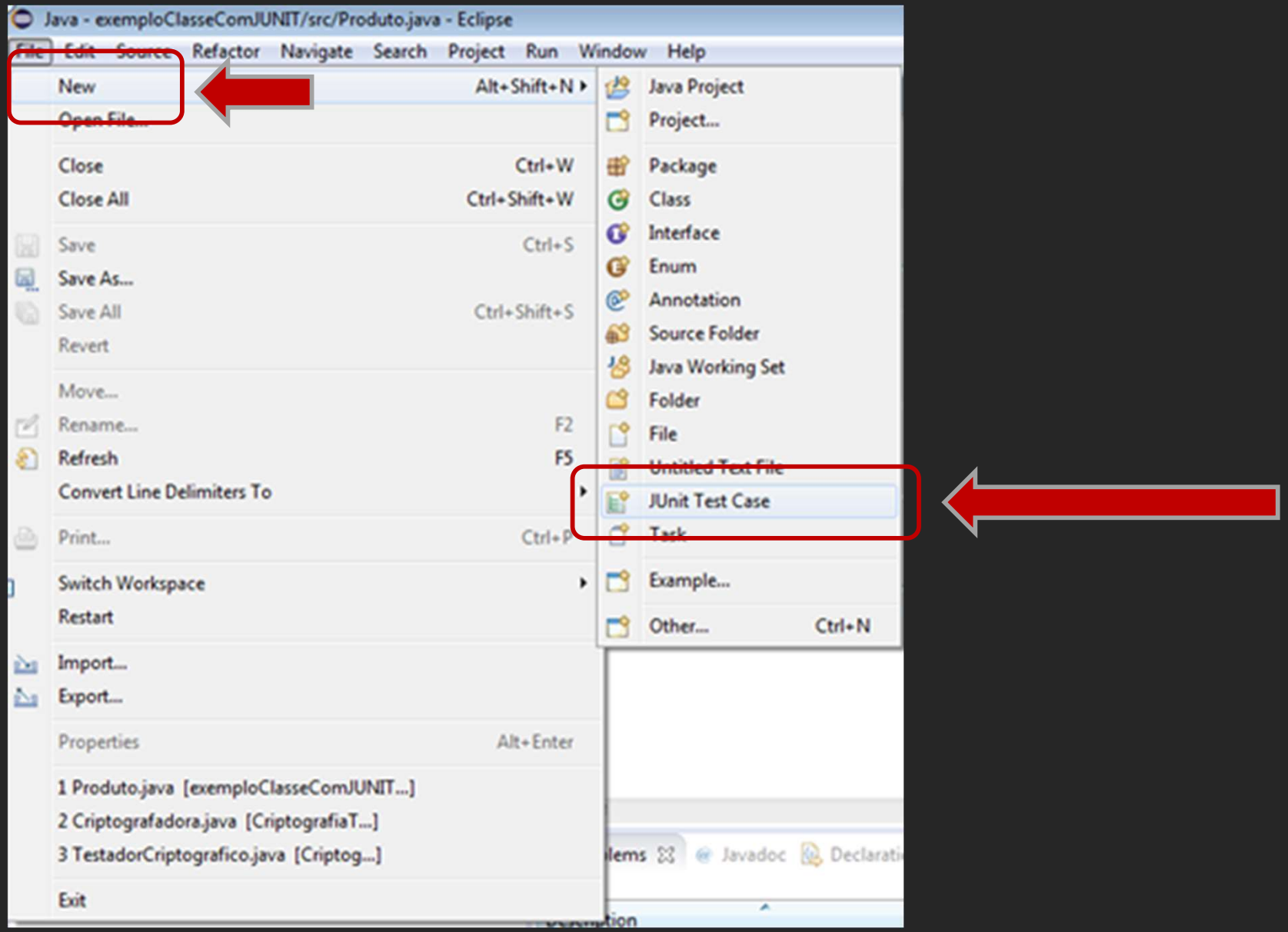



```
public class Calculadora{

    // atributo
    private int resultado = 0;

    // método somar
    public int somar( int n1, int n2 ){

       resultado = n1 + n2;
       return resultado;
    }

    // método subtrair
    public int subtrair( int n1, int n2 ){

       resultado = n1 - n2;
       return resultado;
    }

    // método multiplicar
    public int multiplicar( int n1, int n2 ){

       resultado = n1 * n2;
       return resultado;
    }

    // método dividir
    public int dividir( int n1, int n2 ){

       resultado = n1 / n2;
       return resultado;
    }
}
```

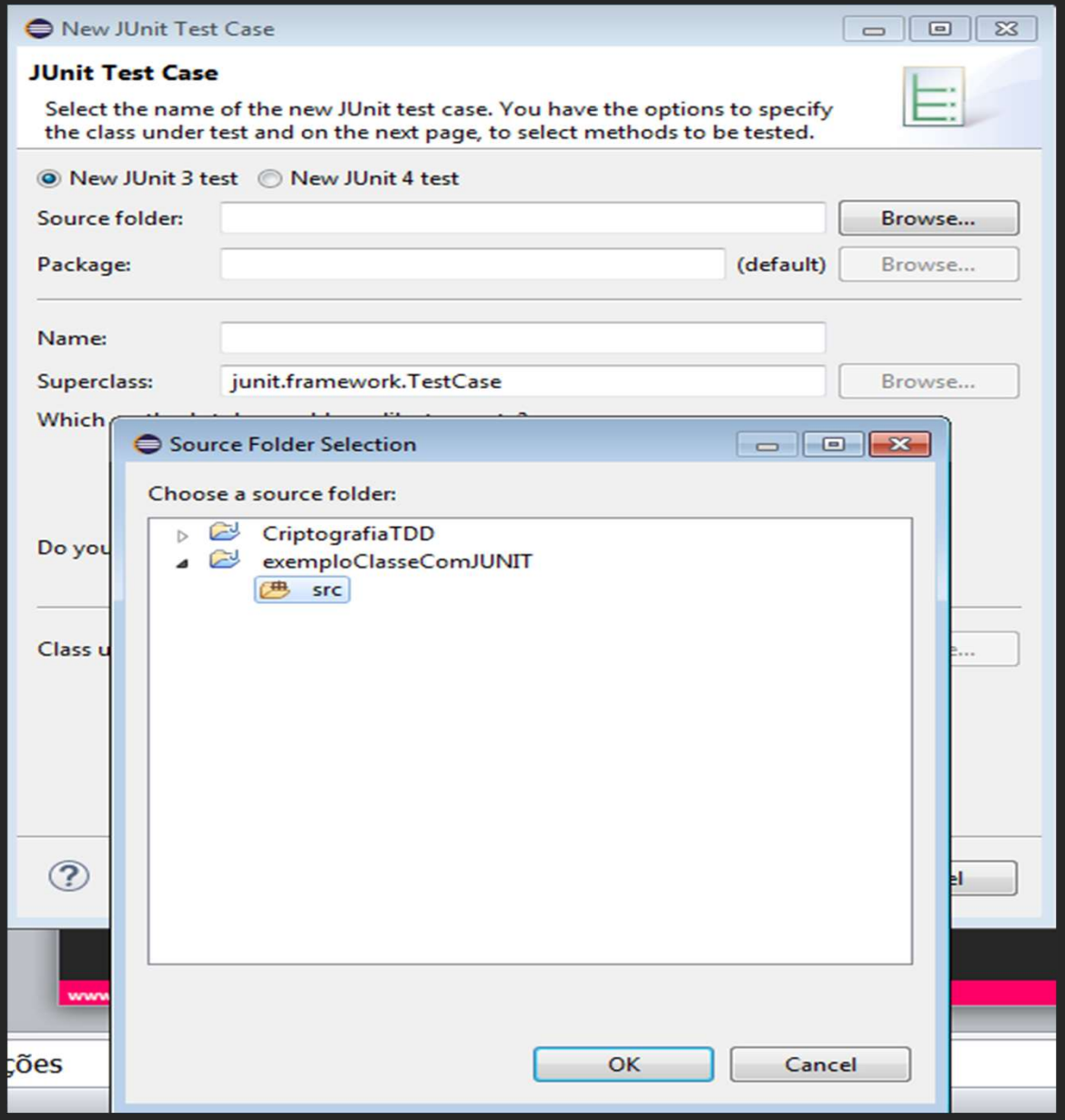



```
public class Calculadora{

    // atributo
    private int resultado = 0;

    // método somar
    public int somar( int n1, int n2 ){

       resultado = n1 + n2;
       return resultado;
    }

    // método subtrair
    public int subtrair( int n1, int n2 ){

       resultado = n1 - n2;
       return resultado;
    }

    // método multiplicar
    public int multiplicar( int n1, int n2 ){

       resultado = n1 * n2;
       return resultado;
    }

    // método dividir
    public int dividir( int n1, int n2 ){

       resultado = n1 / n2;
       return resultado;
    }
}
```

Agora, vamos aos testes...
Crie uma JUNIT Test Case (new JUNIT)



Dê o nome de TesteCalculadora para a Classe de teste que será criada.



GERENCIAMENTO DO TESTE DE SOFTWARE

O processo que vamos seguir para realizar a criação dos testes será:

1º) Criar uma Classe de Teste para Classe de Implementação;

2º) Criar um Método de Teste diferente dentro da Classe de Teste para cada simulação de comportamento que for necessária (um Método para cada Caso de Teste).

3º) Criar e aplicar um Método de Teste por vez, isolando problemas (keep it simple – faça as coisas de forma simples).

```
import static org.junit.Assert.assertEquals;

import junit.framework.TestCase;

public class TesteCalculadoraTest extends TestCase {

    /**
     * Teste de somar na Calculadora.
     */
    @Test
    public void testeSomar() {
        int nro1 = 5;
        int nro2 = 5;
        Calculadora calc= new Calculadora();
        int resultadoEsperado = 10;
        int resultadoReal= calc.somar(nro1, nro2);
        assertEquals(resultadoEsperado, resultadoReal);
    }

}
```

Implementação do Método de teste SOMA, dentro da Classe de Teste de um Objeto Calculadora

```
import static org.junit.Assert.assertEquals;
import junit.framework.TestCase;

public class TesteCalculadoraTest extends TestCase {

    ...

    /**
     * Teste de subtrair na  Calculadora.
     */
    @Test
    public void testeSubtrair() {
        int nro1 = 5;
        int nro2 = 3;
        Calculadora calc = new Calculadora();
        int resultadoEsperado= 2;
        int resultadoReal= calc.subtrair(nro1, nro2);
        assertEquals(resultadoEsperado resultadoReal);
    }

}
```

Acrescente esse Método de teste da SUBTRAÇÃO logo em seguida do Método de teste de SOMA que você fez anteriormente

```
import static org.junit.Assert.assertEquals;
import junit.framework.TestCase;

public class TesteCalculadoraTest extends TestCase {

   . . .

   /**
    * Teste de multiplicar na Calculadora.
    */
   @Test
   public void testeMultiplicar() {
      int nro1 = 3;
      int nro2 = 3;
      Calculadora calc = new Calculadora();
      int resultadoEsperado = 9;
      int resultadoReal = calc.multiplicar(nro1, nro2);
      assertEquals(resultadoEsperado, resultadoReal);
   }
}
```

Acrescente esse Método de teste da MULTIPLICAÇÃO logo em seguida do Método de teste de SUBTRAÇÃO que você fez anteriormente

```
import static org.junit.Assert.assertEquals;
import junit.framework.TestCase;

public class TesteCalculadoraTest extends TestCase {

   . . .

    /**
     * Teste de dividir na Calculadora.
     */
    @Test
    public void testeDividir() {
        int nro1 = 6;
        int nro2 = 2;
        Calculadora calc = new Calculadora();
        int resultadoEsperado= 3;
        int resultadoReal = calc.dividir(nro1, nro2);
        assertEquals(resultadoEsperado, resultadoReal);
    }

}
```

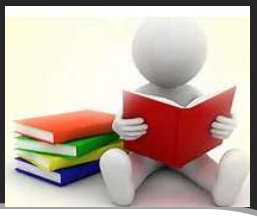
Acrescente esse Método de teste da DIVISÃO logo em seguida do Método de teste de MULTIPLICAÇÃO que você fez anteriormente

Agora, crie a Classe descrita ao lado e a JUNIT para testar todos os métodos da Classe.

Faça os testes para criar um objeto e instanciá-lo e depois testar a recuperação de dados.

```
public class  Produto{

    private double peso;
    private double altura;

    public double getPeso() {
      return peso;
    }

    public void setPeso(double peso) {
      this.peso = peso;
    }

    public double getAltura() {
      return altura;
    }

    public void setAltura(double altura) {
      this.altura = altura;
    }

}
```

ESTUDO DE CASO SIMULADO



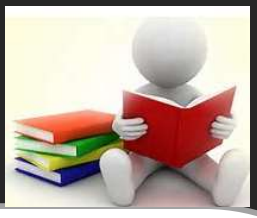
Após o treinamento, um dos participantes falou que muitas vezes, uma aplicação que está desenvolvendo envolve validar várias classes, uma a uma.

Ele quer saber se é possível usar a JUNIT para fazer vários testes simultâneos, de forma organizada.

Uma preocupação adicional é poder ajustar o ambiente de teste (situação de tabelas, conexão de banco, iniciação de variáveis, antes e depois de cada caso de teste aplicado, de forma a evitar que “lixo” (instâncias) de um teste feito, contaminem um teste seguinte.

Consuelo apontou que esses recursos existem e vai usar um programa que está em desenvolvimento pela equipe para explicar como funcionam esses recursos.

ESTUDO DE CASO SIMULADO

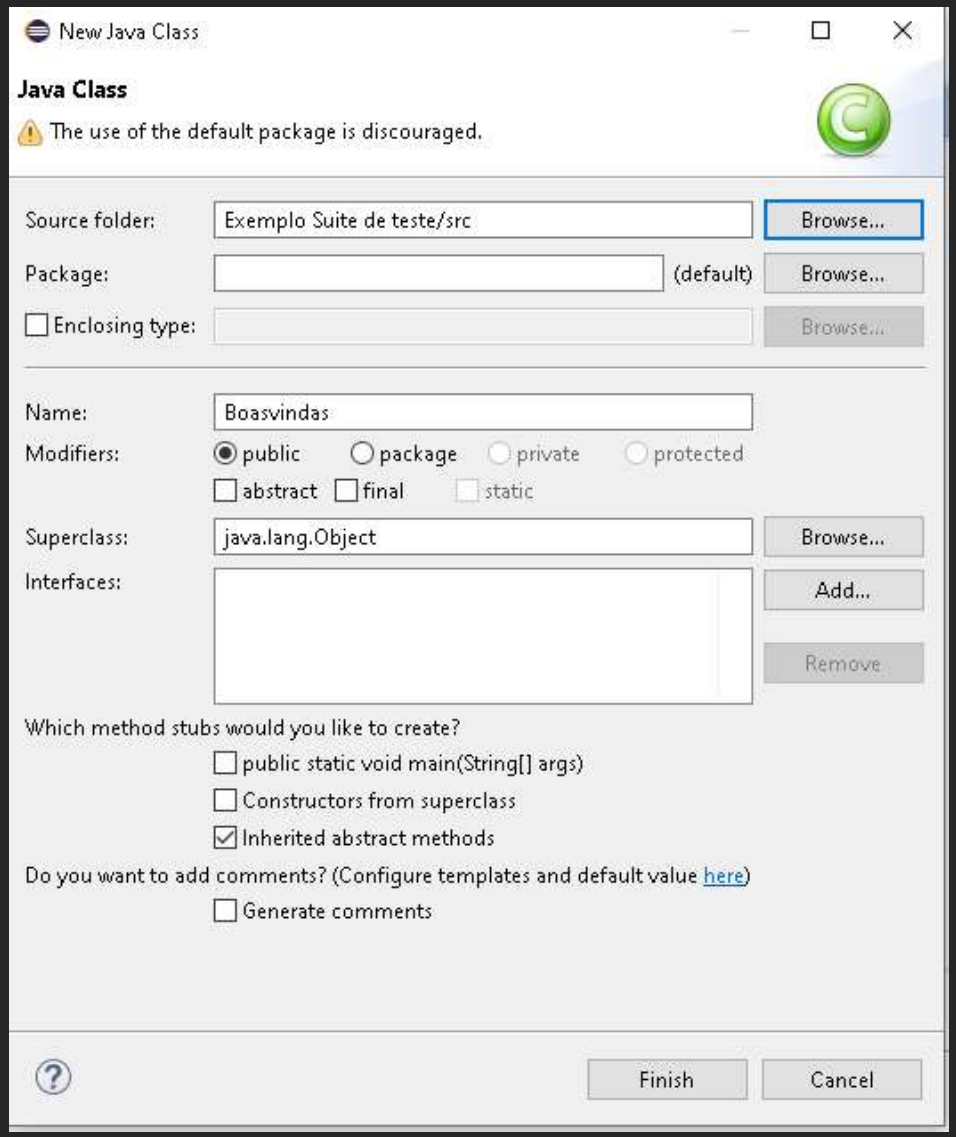


1º) Crie os testes para a Classe a seguir, a qual vai ser validada junto com a Calculadora que você acabou de desenvolver!

Essa Classe, exibe uma mensagem de boas vindas ao usuário da Calculadora digital.

Crie essa Classe dentro do mesmo projeto da Calculadora!

Crie um projeto no Eclipse e depois, ...
a Classe JAVA para exibir mensagem de boas vindas customizada ao usuário de um sistema




```
public class Boasvindas{

  private String menssagem;

  //Construtor de Objeto na Classe
  public Boasvindas(String mens){
    this.menssagem = mens;
  }

  // Exibição da mensagem
  public String exibirMenssagem(){
    System.out.println(this.menssagem);
    return this.menssagem;
  }

  // Exibição da parte fixa da mensagem
  public String completarMenssagem(){
      String compmens;
      compmens = "Ola! Seja bem vindo a sua calculadora pessoal"
      System.out.println(compmens);
      return compmens;
  }
}
```

Agora, vamos criar os testes para essa classe!

```
import static org.junit.Assert.*;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TesteMensagem {

  @Test
  public void testeCriaMesnsagemPadrao() {
                String mensx;
                mensx = "Pedro Bo";
                Boasvindas mensagemUsuario = new Boasvindas(mensx);
                String mensretorno = mensagemUsuario.completarMenssagem();
                assertEquals("Ola! Seja bem vindo a sua calculadora pessoal", mensretorno);
  }

  @Test
  public void testeExibeMesnsagem() {
                String mensx;
                mensx = "Pedro Bo";
                Boasvindas mensagemUsuario = new Boasvindas(mensx);
                String mensRetorno;
                mensRetorno = mensagemUsuario.exibirMenssagem();
                assertEquals(mensx, mensRetorno);
  }
}
```

Você agora tem duas Classe e duas Classes de testes no mesmo projeto!
Para rodá-las todas juntas, precisará criar uma Suite de Teste!

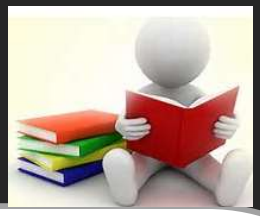
TesteMensagem.java

```
1  import static org.junit.Assert.*;
4
5  public class TesteMensagem {
6      /**
7      * Teste de boas vindas a Calculadora.
8      */
9      @Test
10         public void testeCriaMesnsagemPadrao() {
11             String mensx;
12             mensx = "Pedro Bo";
13             Boasvindas mensagemUsuario = new Boasvindas(mensx);
```

Boasvindas.java | Calculadora.java | TesteCalculadoraTest.java

```
8      * Teste de somar na Calculadora.
9      */
10     @Test
11     public void testeSomar() {
12         int nro1 = 5;
13         int nro2 = 5;
14         Calculadora calc= new Calculadora();
15         int resultadoEsperado = 10;
16         int resultadoReal= calc.somar(nro1, nro2);
17         assertEquals(resultadoEsperado, resultadoReal);
18     }
19
```

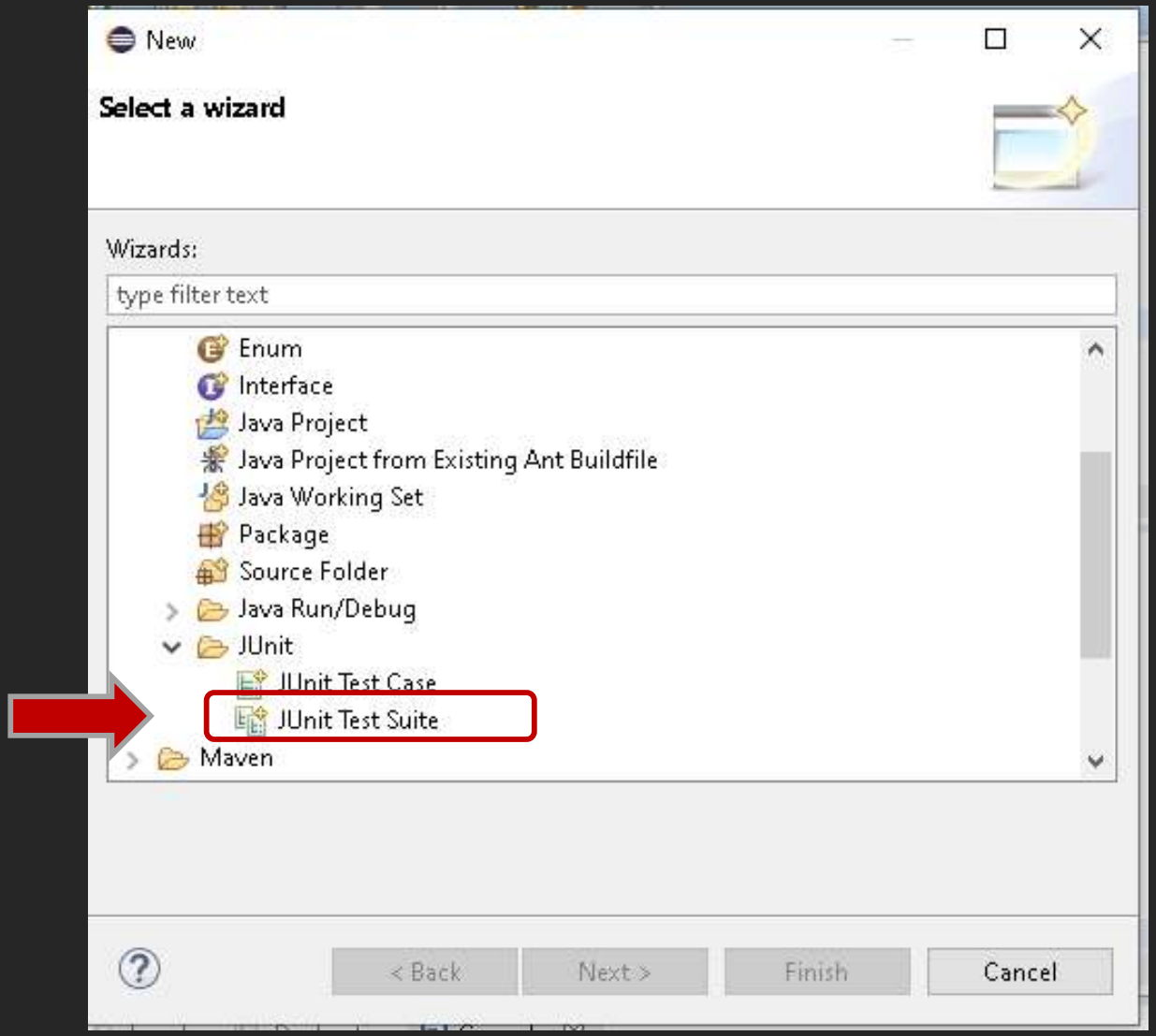
ESTUDO DE CASO SIMULADO



2º) Agora, vamos mixar esses testes com os que você criou anteriormente!

Siga os passos...

Crie a suíte de teste, selecionando o Folder do seu projeto*!*

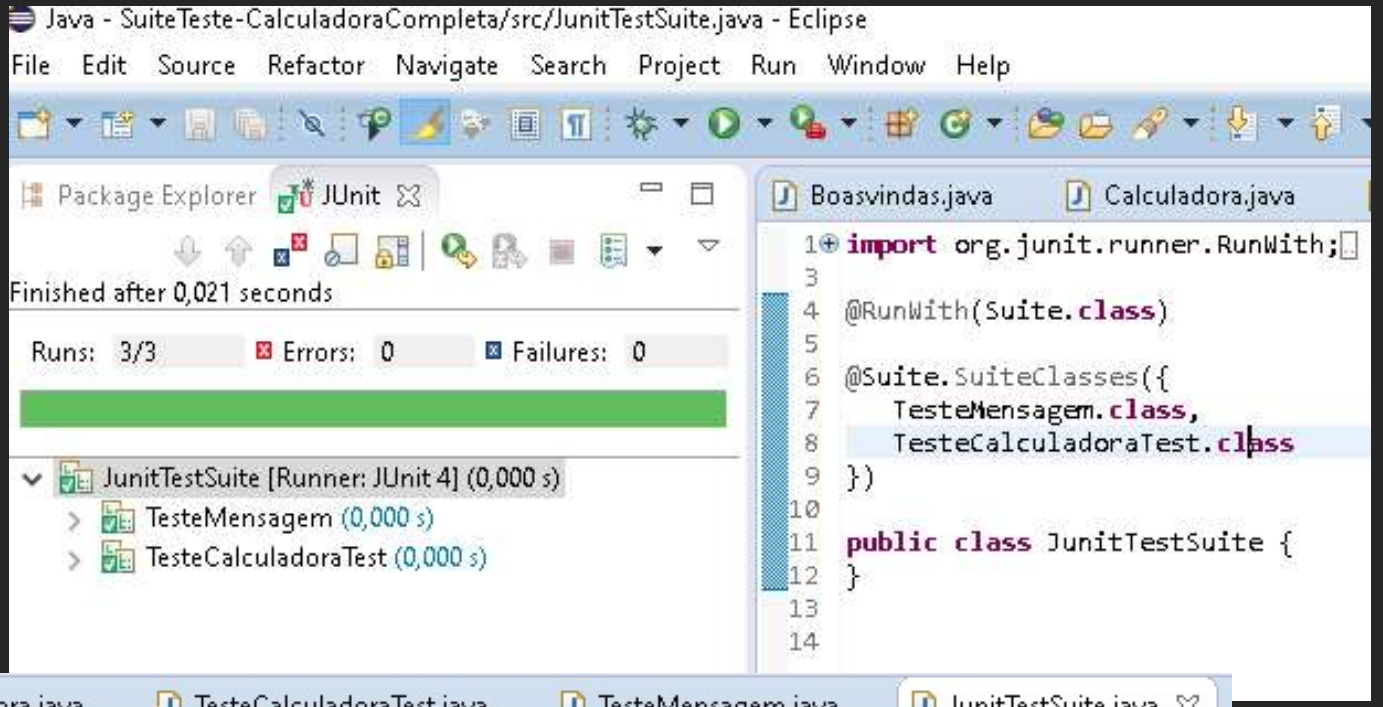


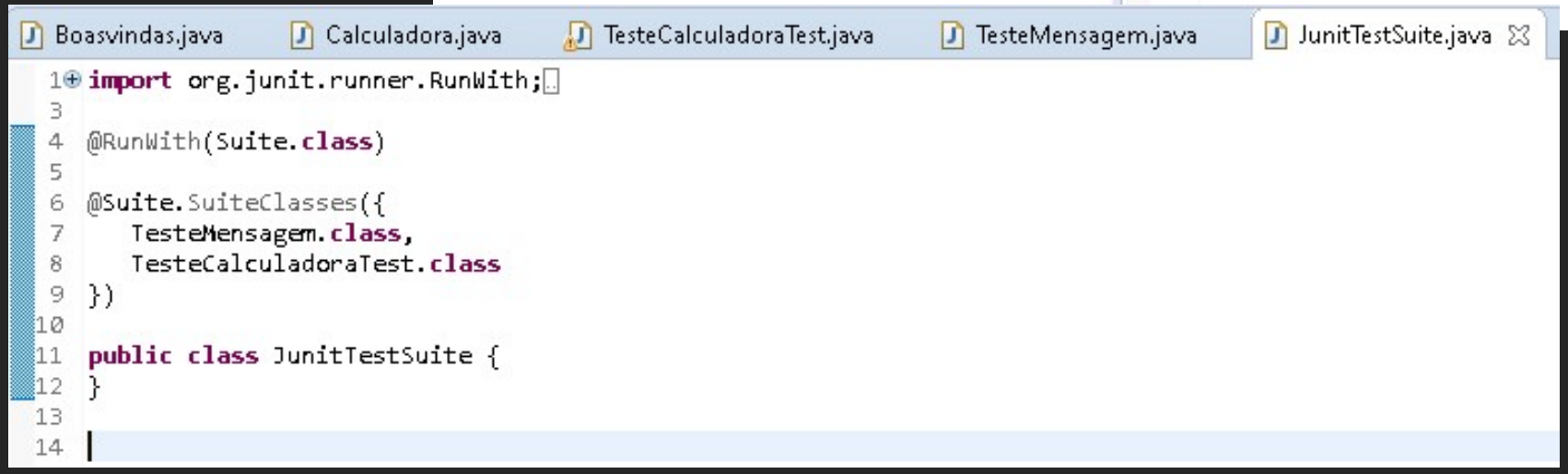
Vamos renomear a suíte para TesteCompletoCalculadora e incluir as chamadas de todas as Classes JUNIT*!*

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
  TesteMensagem.class,
  TesteCalculadoraTest.class
})

public class JunitTestSuite {
}
```

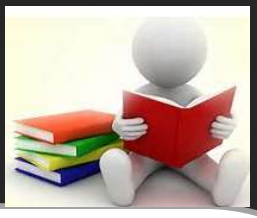
Depois, é só selecionar a Suite hora de executar os testes e ela mostrará o resultado de cada Classe/Método!

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
  TesteMensagem.class,
  TesteCalculadoraTest.class
})

public class JunitTestSuite {
}
```





ESTUDO DE CASO SIMULADO



Os desenvolvedores querem poder disparar os testes via linha de comandos (prompt)!

É possível? Perguntaram ao C.

C respondeu que é necessário criar um job executor de testes (test runner).
Veja como fazer...

Vamos criar uma nova Classe, chamada Executoradetestes, no mesmo projeto onde está a Suite, as Classes do sistema e as Junit Classes*!*

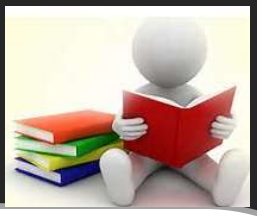
```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class Executoradetestes {
  public static void main(String[] args) {
    Result resultado = JUnitCore.runClasses(JunitTestSuite.class);

    for (Failure failure : resultado.getFailures()) {
      System.out.println(failure.toString());
    }

    System.out.println(resultado.wasSuccessful());
  }
}
```

Você pode executar como uma aplicação JAVA no próprio Eclipse ou executar via linha de comando, após compilação de todas as Classes e Junits no JAVA (comando JAVAR <nome da classe>).

```
C:\JUNIT_WORKSPACE>javac Calculadora.java Boasvindas.java
TesteCalculadoraTest.java TesteMensagem.java Executoradetestes.java
```

```
C:\JUNIT_WORKSPACE>java Executoradetestes
```

ESTUDO DE CASO SIMULADO



Você está em um projeto de um software e precisa criar uma função que colha 3 digitações de números inteiros, compare os três e diga qual o maior e qual o menor número digitado.

1º) Crie a Classe JAVA e o Método de Captura de Digitação de um número, mais o Método exibir o Maior e outro para exibir o Menor número entre três digitados.

2º) Crie a Classe de Testes e seus Métodos, com JUNIT.

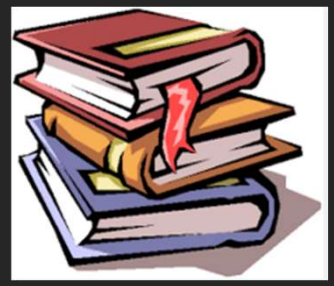
3º) Crie depois a Suíte de teste.
Crie a classe executora dos testes.

4º) Depois que tudo estiver funcionando, transfira a Classe e as JUNITs para o projeto que foi aplicado no treinamento (Calculadora Digital) e ajuste a Test Suite Class e a Test Runner para acomodar essa nova classe e seus testes.

APROVEITE PARA AUTOMATIZAR TESTES UNITÁRIOS COM JUNIT EM TODOS OS SEUS PROJETOS JAVA, DAQUI POR DIANTE!

D ú v I D A S

Referência bibliográficas



BIBLIOGRAFIA :

- MOLINARI, Leonardo. Inovação e Automação de Testes de Software, 1ª edição. Érica, 2010.

TESTE DE SOFTWARE - TDD

FIM

PROFESSOR:
RENATO JARDIM PARDUCCI

PROFRENATO.PARDUCCI@FIAP.COM.BR