Teste é o processo de executar um programa com a intenção de descobrir um erro.

Testes podem ser feito utilizando-se diferentes técnicas. Vai depender da metodologia aplicada na empresa ou no projeto.
No modelo ágil, o profissional de teste começa seu trabalho já na concepção e construção da análise do software. É aqui que entra TDD.

POR QUE DEVEMOS TESTAR?

Testes quando são bem planejados, poupam um bocado de tempo e dores de cabeça futuras, minimizando retrabalhos e desgastes com o cliente.

MITO: teste custa caro. Pressman já apresentou em seu Livro de Engenharia de Software que o custo do defeito é progressivo, ou seja, encontrar o defeito na fase de engenharia de requisitos custa 1 enquanto encontrar o defeito durante a fase de uso custa 100 vezes mais, então utilizar o teste, reduz custo e não aumenta.

Benefícios do teste

1. Qualidade do seu produto
2. Evita perda da confiança do cliente.
3. Ajuda a identificar trechos de código que foram mal escritos.
4. O Software mais fácil de ser mantido e evoluído.
5. Diminuindo o custo com bugs

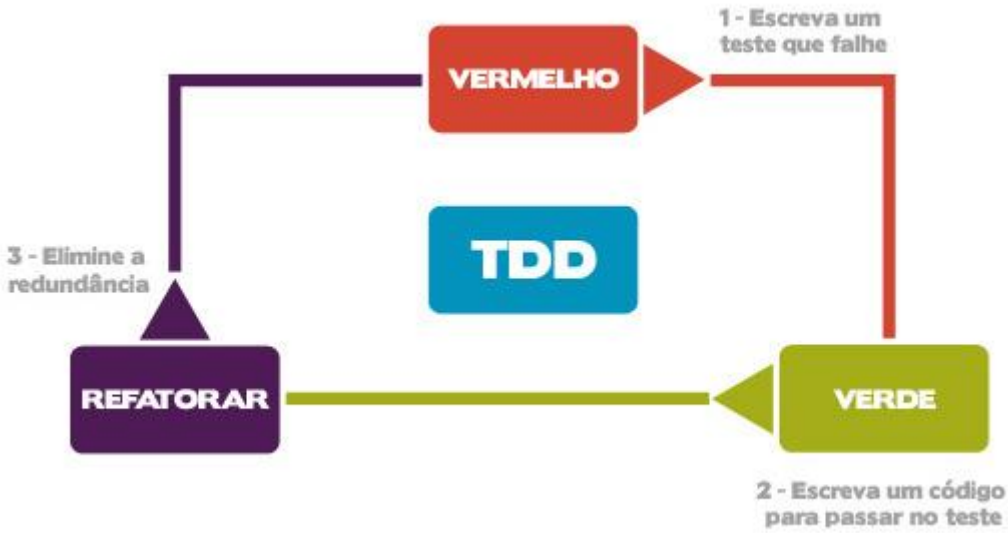
Testes manuais	**Testes automatizados**
Perda de tempo	Ganho de tempo
Geralmente, ele executa testes enquanto desenvolve o algoritmo completo. Ele escreve um pouco, roda o programa, e o programa falha. Nesse momento, o desenvolvedor entende o problema, corrige-o, e em seguida executa novamente o mesmo teste.	O desenvolvedor que automatiza seus testes perde tempo apenas 1 vez com ele; nas próximas, ele simplesmente aperta um botão e vê a máquina executando o teste pra ele, de forma correta e rápida.

À medida que a tecnologia muda, e mais organizações se movem para o desenvolvimento ágil, os testes devem adaptar-se rapidamente. A automação de teste é essencial.

TDD - Test Driven Development
Desenvolvimento guiado por testes

É uma das práticas de desenvolvimento de software sugeridas por diversas metodologias ágeis.
TDD sugere que o desenvolvedor escreva o teste antes mesmo da implementação.

Ciclo do TDD: (vermelho-verde-refatora ou red-green-refactor)



Criamos um teste que falhe -> **Fazemos a codificação para passar no teste** -> **Refatoramos nosso código (melhorar o código)**

TDD maximiza a quantidade de feedback sobre o código que está sendo produzido, fazendo o programador perceber os problemas antecipadamente e, por consequência, diminuindo os custos de manutenção e melhorando o código.

TESTES DE UNIDADE

O teste unitário, serve para validar a menor porção do código, ou seja, a unidade.Geralmente, em sistemas orientados a objetos, essa unidade é a classe.

O importante aqui é que os testes depois de desenvolvidos, pode ser utilizado diversas vezes, já que ele testa a funcionalidade, então, se algum dia um outro desenvolvedor alterar esse código, ele poderá executar a bateria de testes automatizados existente, garantindo que não ocorreu uma regressão de funcionalidade durante as melhorias ou as integrações de módulos.

“O teste de unidade ajuda o desenvolvedor a garantir a qualidade interna do código, dando feedback sobre o design dos módulos e permitindo uma manutenção com menor custo”.

Existem diversos frameworks para as mais diversas linguagens. O livro sugerido adota JUnit. O **JUnit** é um framework de teste para Java, que permite a criação de testes unitários. Além disso, está disponível como plug-in para os mais diversos IDE'S como Eclipse, Netbeans etc.

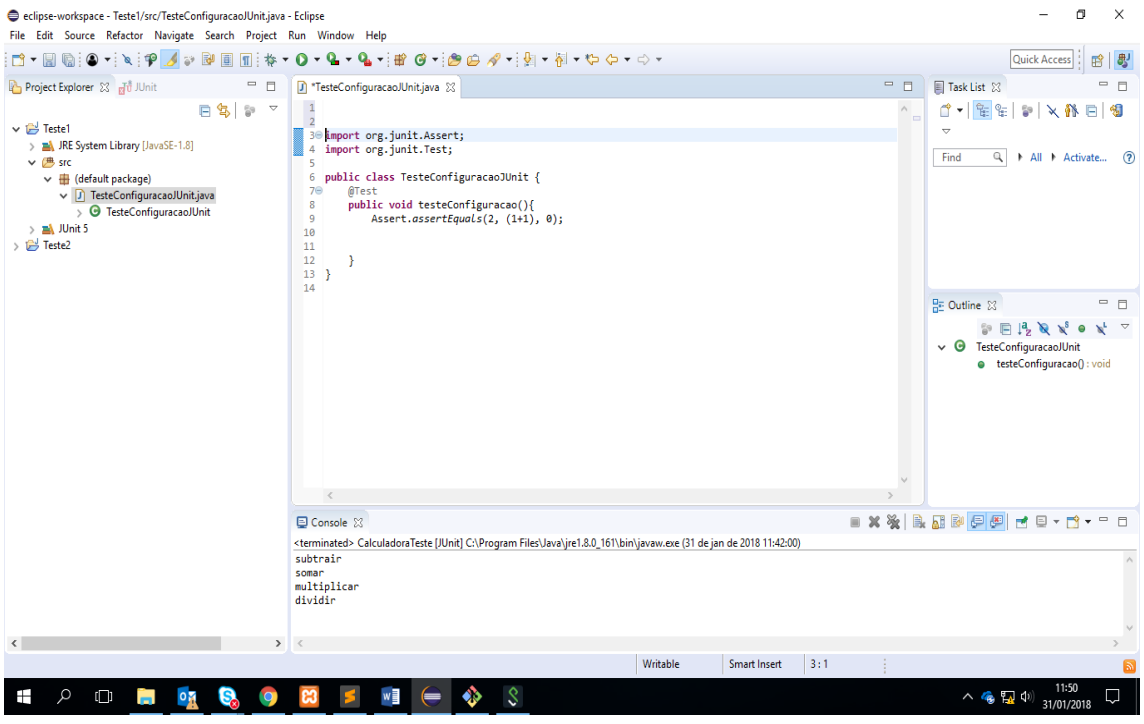
Vantagens do Junit

1. Permite a criação rápida de código de teste possibilitando um aumento na qualidade do desenvolvimento e teste;
2. Uma vez escritos, os testes são executados rapidamente sem que, para isso, seja interrompido o processo de desenvolvimento;
3. Checa os resultados dos testes e fornece uma resposta imediata;
4. É livre e orientado a objetos.

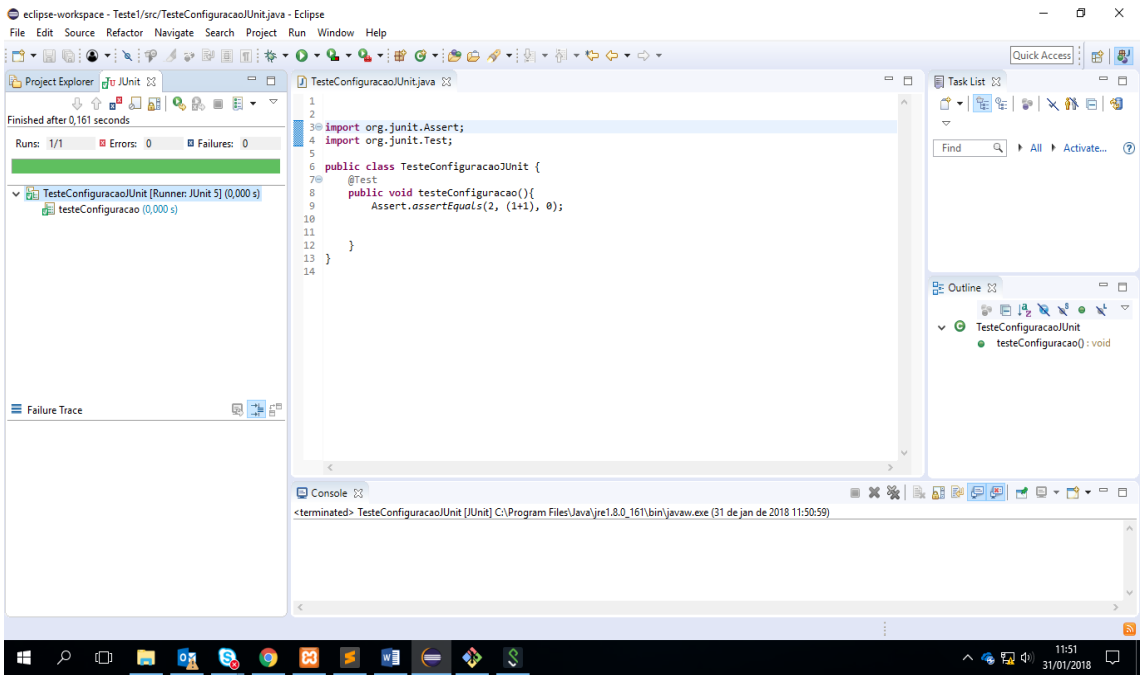
A melhor parte: se algum dia um outro desenvolvedor alterar esse código, ele poderá executar a bateria de testes automatizados existente, e descobrir se a sua alteração fez alguma funcionalidade que já funcionava anteriormente parar de funcionar. **(testes de regressão).**
Muito mais que uma moda, teste de software tem se tornado uma necessidade crescente devido a também crescente busca por qualidade.

PRATICA:

Primeiro configurei o JUnit no eclipse, criando o projeto Teste1 com a classe TesteConfiguraçãoJunit.java para testar as configurações.

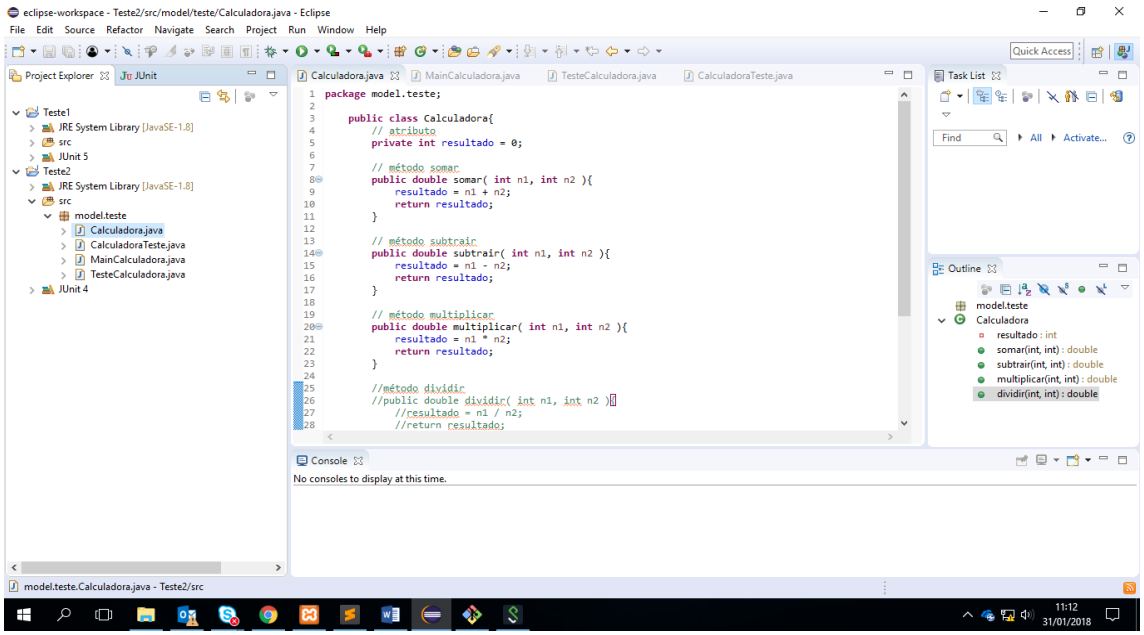


Ok verde, veja abaixo:

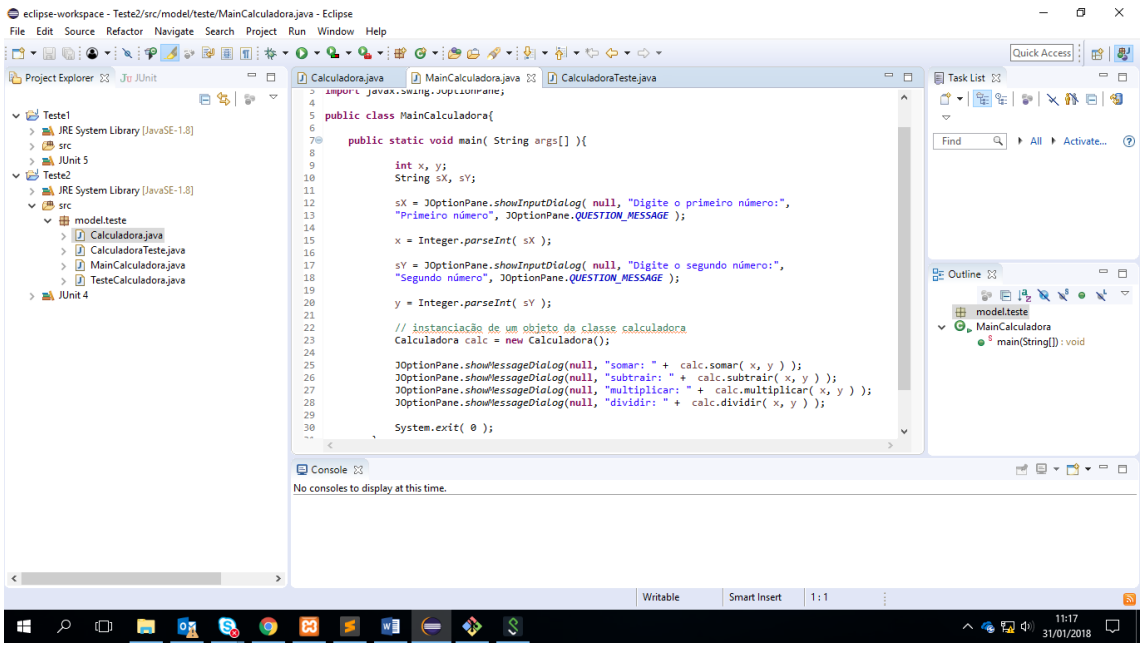


Segundo, criei Projeto Teste2 com as classes:

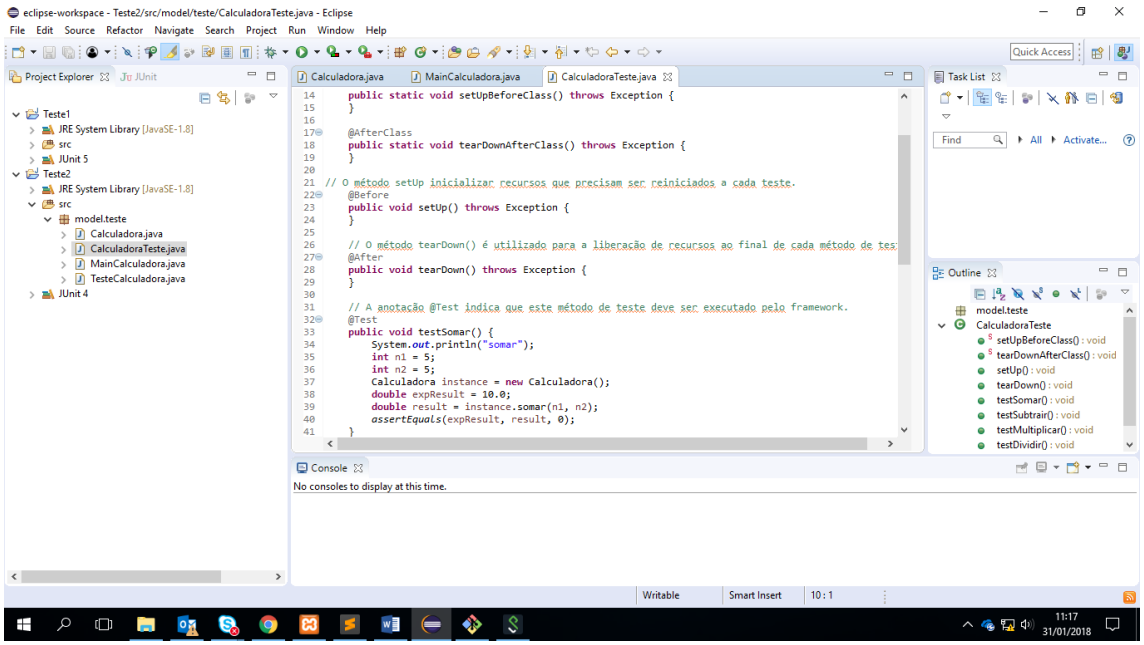
Calculadora.java



MainCalculadora.java



CalculadoraTeste.java

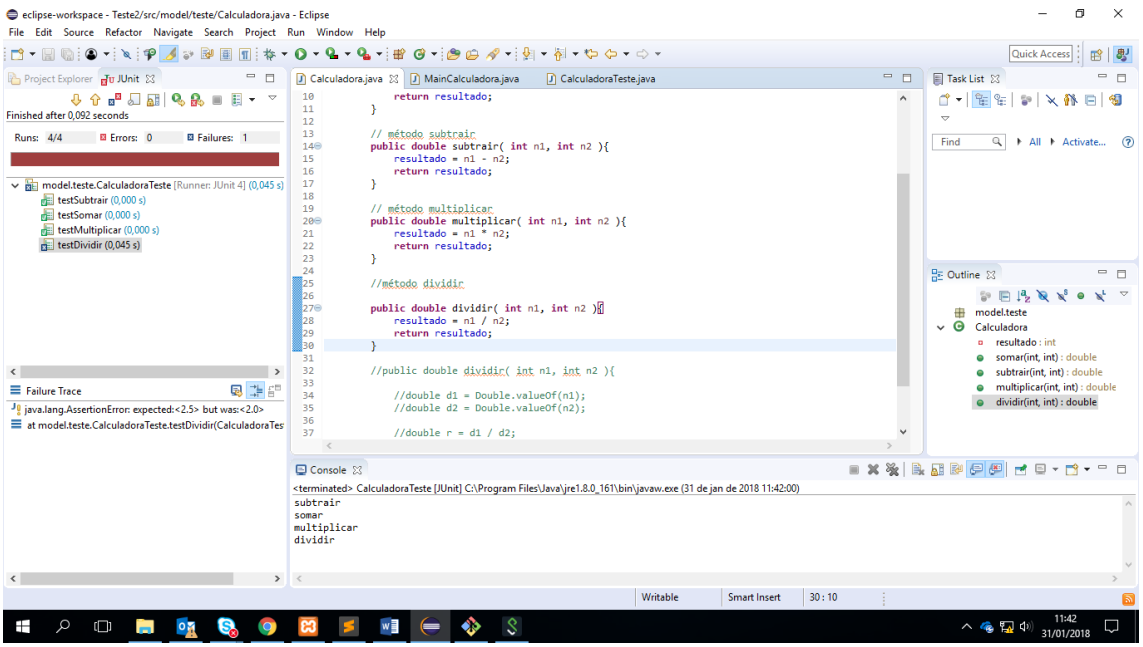


A notação @Test indica que o metodo de teste vai ser executado pelo framework.

Executando os testes:

Metodo dividir deu erro, pois não aceita retornos em decimal. Veja o print abaixo:

```
public double dividir( int n1, int n2 ){
            resultado = n1 / n2;
            return resultado;
        }
```



Note que se estivesse assim (código abaixo Refatorado), passaria pois o metodo divisão aceita retornos em decimal. Veja o print abaixo:

```
public double dividir( int n1, int n2 ){

            double d1 = Double.valueOf(n1);
            double d2 = Double.valueOf(n2);

            double r = d1 / d2;
            return r;
```