

# PASSO A PASSO PARA ESCREVER UMA FUNCIONALIDADE COM BDD, TDD UTILIZANDO CUCUMBER E JUNIT.

Primeiro escrevemos o arquivo de feature também chamado de arquivo de características.

```
1 # language: pt
2 @ContaTestRunner
3 Funcionalidade: Saque de um determinado valor da conta
4 Regras: O Saldo disponível da conta é composto pelo saldo atual mais o limite,
5 portanto um saque pode ser realizado caso seja menor ou igual ao saldo mais limite.
6
7 Cenario: Saque realizado com sucesso
8 Dado uma conta do dono "Peterson" de numero 12345 com o limite 200 e saldo 50
9 Quando o dono realizar o saque no valor de 50 na conta
10 Entao o dono tera o saldo no valor de 200 na conta
```

Depois criamos a Classe com o Glue Code, responsável por realizar os testes para cada cenário. Geralmente criamos ela da seguinte maneira para facilitar:

```
public class ContaSaqueTest {

    Dado uma conta do dono "Peterson" de numero 12345 com o limite 200 e saldo 50
    Quando o dono realizar o saque no valor de 50 na conta
    Entao o dono tera o saldo no valor de 200 na conta

}
```

Colocamos os mesmos textos do arquivo de feature, pois eles irão se tornar métodos.






```
public class ContaSaqueTest {  
  
    @Dado("^uma conta do dono \"(.*)\" de numero (\\d+) com o limite (\\d+) e saldo (\\d+)$")  
    public void uma_conta_do_dono_de_numero_com_o_limite_e_saldo(String dono, int numero, Double limite,  
        Double saldo) {  
        // Aqui a conta eh criada de acordo com os parametros enviados  
        // por argumento, depois atribuida a variavel de instancia  
    }  
  
    @Quando("^o dono realizar o saque no valor de (\\d+) na conta$")  
    public void o_dono_realiza_o_saque_no_valor_de_na_conta(Double valorSaque) throws Throwable{  
        // Aqui eu a consigo realizar alguma codificacao que quebra  
        // o teste caso nao seja possivel sacar  
    }  
  
    @Entao("^o dono tera o saldo no valor de (\\d+) na conta$")  
    public void o_dono_tem_o_saldo_na_conta_no_valor_de(Double saldoEsperado) {  
        // Aqui eu verifico se o saldo esperado eh igual  
        // ao saldo atual do objeto conta  
    }  
}
```

Os passos, Dado, Quando e Então se tornam métodos e são amarrados com o arquivo de features. Através de expressão regular de tipos o método recebe o parâmetro correto.

Agora basta realizar a codificação mínima para cada método, teoricamente a partir daqui estamos usando TDD, pois iremos atender uma funcionalidade de comportamento com a validação de testes de unidade.

```
private Conta conta;  
  
@Dado("^uma do dono \"(.*)\" de numero (\\d+) com o limite (\\d+) e saldo (\\d+)$")  
public void uma_conta_do_dono_de_numero_com_o_limite_e_saldo(String dono, int numero, Double limite,  
    Double saldo) {  
    // Aqui a conta eh criada de acordo com os parametros enviados  
    // por argumento, depois atribuida a variavel de instancia  
    conta = new Conta(dono, numero, limite, saldo);  
  
    // Posso me certificar se a conta foi criada  
    Assert.assertNotNull(conta);  
}
```

Posso rodar os testes e verificar que todos estão funcionando, inclusive o que eu acabei de codificar. Os outros funcionam porque não possuem nenhuma validação.

- ▲  Funcionalidade: Saque de um determinado valor da conta (0,006 s)
  - ▲  Cenário: Saque realizado com sucesso (0,006 s)
    -  Dado uma do dono "Peterson" de numero 12345 com o limite 200
    -  Quando o dono realizar o saque no valor de 50 na conta (0,000 s)
    -  Entao o dono tera o saldo no valor de 150 na conta (0,006 s)

Agora posso partir para a implementação do próximo método.

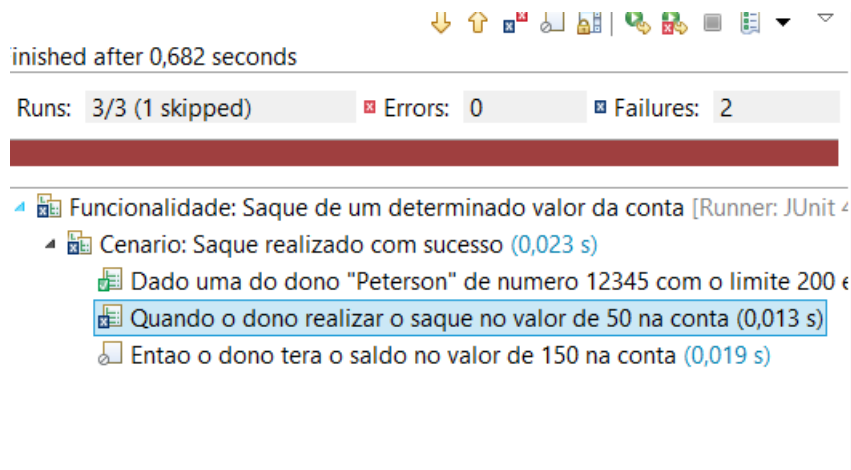
```
@Quando("^o dono realizar o saque no valor de (\\d+) na conta$")
public void o_dono_realiza_o_saque_no_valor_de_na_conta(Double valorSaque) throws Throwable{

    // Aqui eu a consigo realizar alguma codificacao que quebra
    // o teste caso nao seja possivel sacar

    boolean conseguiuSacar = conta.sacar(valorSaque);
    Assert.assertTrue(conseguiuSacar);

}
```

Posso também rodar o teste:

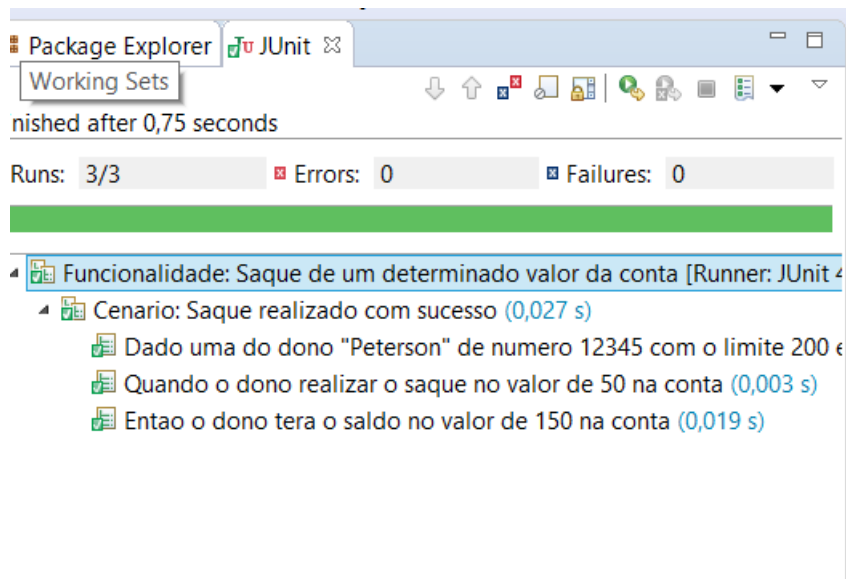


The screenshot shows a JUnit test runner interface. At the top, it says "inished after 0,682 seconds". Below that, it shows "Runs: 3/3 (1 skipped)", "Errors: 0", and "Failures: 2". A red bar indicates a failure. The test suite is "Funcionalidade: Saque de um determinado valor da conta [Runner: JUnit 4]". The test case is "Cenario: Saque realizado com sucesso (0,023 s)". The test data is "Dado uma do dono 'Peterson' de numero 12345 com o limite 200 €". The test step is "Quando o dono realizar o saque no valor de 50 na conta (0,013 s)". The expected result is "Entao o dono tera o saldo no valor de 150 na conta (0,019 s)".

Dessa vez o teste falhou pois, o método sacar não possui implementação. Vamos então implementar:

```
17
18 public boolean sacar(Double valor) {
19     if ((saldo + limite) <= valor) {
20         // Nao pode sacar
21         return false;
22     } else {
23         // Pode sacar
24         saldo = saldo - valor;
25         return true;
26     }
27 }
28
```

Agora rodando os testes novamente:



Partindo agora para o próximo método:

```
@Entao("^o dono tera o saldo no valor de (\\d+) na conta$")
public void o_dono_tem_o_saldo_na_conta_no_valor_de(Double saldoEsperado) {

    // Aqui eu verifico se o saldo esperado eh igual
    // ao saldo atual do objeto conta

    Assert.assertEquals(saldoEsperado, conta.getSaldo());
}
```

Este método simplesmente garante de que o saldo da conta está conforme o valor esperado.

```
38
39 @Entao("^o dono tera o saldo no valor de (\\d+) na conta$")
40 public void o_dono_tem_o_saldo_na_conta_no_valor_de(Double saldoEsperado) {
41
42     // Aqui eu verifico se o saldo esperado eh igual
43     // ao saldo atual do objeto conta
44
45     Assert.assertEquals(saldoEsperado, conta.getSaldo());
46 }
47
```

Ao rodar o teste, ele quebra pois o código do método getSaldo não leva em consideração o limite.

```
50
57 public Double getSaldo() {
58     return saldo;
59 }
60
```

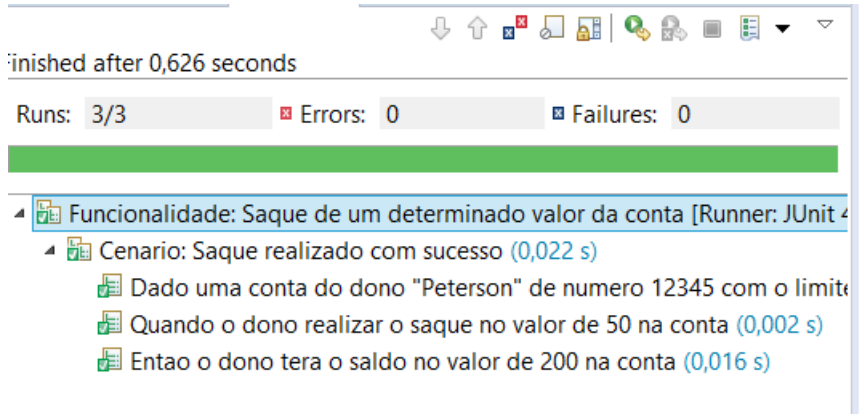
Codificação correta:

```

56
57 public Double getSaldo() {
58     return saldo + limite;
59 }
60

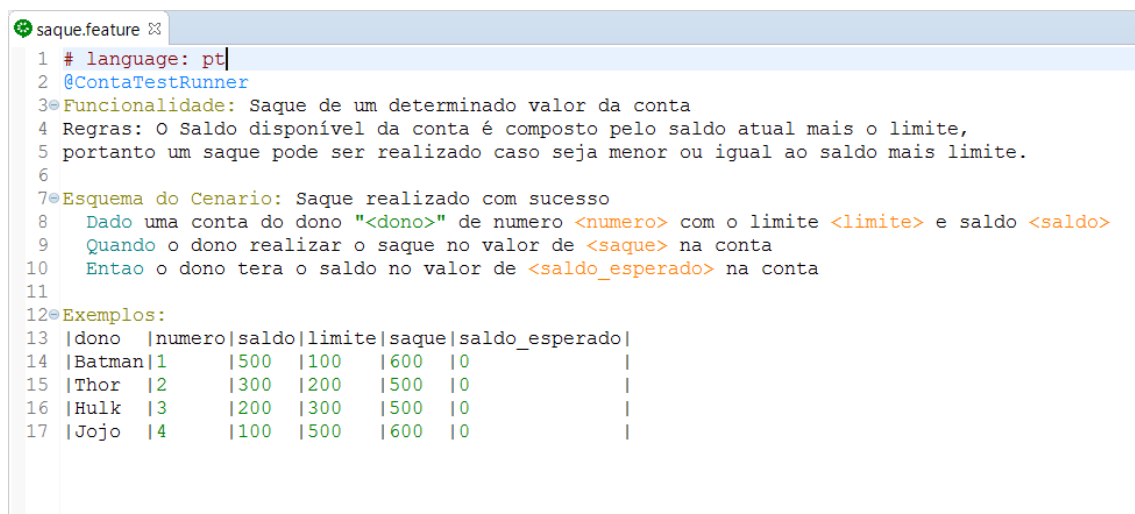
```

Teste passando:



Agora que nós temos este cenário com sucesso podemos refatorar nosso arquivo de feature para usar um recurso do Gherkins para facilitar na execução de cenários de testes mais ricos.

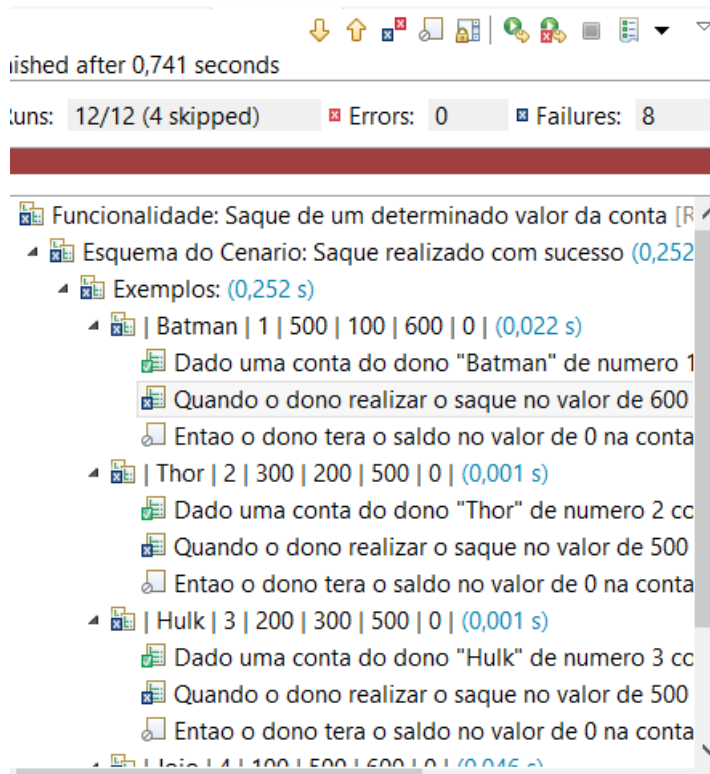
Esquema de Cenário ou em Inglês Scenario Outline, serve para podermos inserir exemplos para o cenário.



Agora os nossos testes serão executados quatro vezes, que são a quantidade de exemplos do Esquema do Cenário.

Desta maneira podemos escrever cenários mais ricos e pensar em mais possibilidades para validar o cenário.

Não precisamos mudar nada nas classes de testes basta rodar novamente.



Ops! =(

Agora os testes quebraram no passo do Quando...

O que aconteceu é que o código do método sacar esta da seguinte maneira:

```

17
18 public boolean sacar(Double valor) {
19     if ((saldo + limite) <= valor) {
20         // Nao pode sacar
21         return false;
22     } else {
23         // Pode sacar
24         saldo = saldo - valor;
25         return true;
26     }
27 }

```

Dessa forma ele só irá permitir sacar se caso o valor não for exatamente igual ao da conta, ou menor. Ou seja, os nossos heróis não podem esvaziar suas contas e salvar o planeta.

Vamos corrigir isso:

```

17
18 public boolean sacar(Double valor) {
19     if ((saldo + limite) < valor) {
20         // Nao pode sacar
21         return false;
22     } else {
23         // Pode sacar
24         saldo = saldo - valor;
25         return true;
26     }
27 }

```

Vamos rodar o teste:

finished after 0,741 seconds

Runs: 12/12    Errors: 0    Failures: 0

Funcionalidade: Saque de um determinado valor da conta [Run]

- Esquema do Cenario: Saque realizado com sucesso (0,150 s)
  - Exemplos: (0,150 s)
    - Batman | 1 | 500 | 100 | 600 | 0 | (0,007 s)
    - Thor | 2 | 300 | 200 | 500 | 0 | (0,000 s)
    - Hulk | 3 | 200 | 300 | 500 | 0 | (0,000 s)
    - Jojo | 4 | 100 | 500 | 600 | 0 | (0,000 s)

Agora sim! Veja que todos os exemplos foram executados. Agora vamos colocar um cenário mais comum que não esvazia a conta e verificar se tudo continua funcionando. Farei isso adicionando mais um exemplo.

```

11
12 Exemplos:
13 |dono|numero|saldo|limite|saque|saldo_esperado|
14 |Batman|1|500|100|600|0|
15 |Thor|2|300|200|500|0|
16 |Hulk|3|200|300|500|0|
17 |Jojo|4|100|500|600|0|
18 |Heitor|5|1000|1000|500|1500|

```












Vamos lá, rodando os testes:

Finished after 0,684 seconds

Runs: 15/15

Errors: 0

Failures: 0

- ▲  Funcionalidade: Saque de um determinado valor da conta [Run]
- ▲  Esquema do Cenário: Saque realizado com sucesso (0,176 s)
  - ▲  Exemplos: (0,176 s)
    - ▷  | Batman | 1 | 500 | 100 | 600 | 0 | (0,007 s)
    - ▷  | Thor | 2 | 300 | 200 | 500 | 0 | (0,000 s)
    - ▷  | Hulk | 3 | 200 | 300 | 500 | 0 | (0,000 s)
    - ▷  | Jojo | 4 | 100 | 500 | 600 | 0 | (0,001 s)
    - ▲  | Heitor | 5 | 1000 | 1000 | 500 | 1500 | (0,001 s)
      -  Dado uma conta do dono "Heitor" de numero 5 com
      -  Quando o dono realizar o saque no valor de 500 na
      -  Entao o dono tera o saldo no valor de 1500 na conta

Tudo Funcionando!

O Código que está no git da IWorks Education, contém os exemplos completos de saque, depósito e transferência.