Artigo

Desenvolvimento orientado a comportamento (BDD) com Cucumber

Veja neste artigo sobre como desenvolver testes de aceitação automatizado utilizando o framework Cucumber e a técnica BDD (Desenvolvimento orientado a comportamento).

O que é Behavior Driven Development (BDD)?

O **Behavior Driven Development (BDD)** ou desenvolvimento orientado por comportamento foi inventado por Dan North no ano de 2000, quando Dan percebeu que muitas equipes tinham dificuldades de adotar e usar eficazmente o TDD, criado como uma versão melhorada do desenvolvimento orientado por testes (TDD, criado por Kent Beck). O BDD não é uma metodologia de desenvolvimento de software, tão pouco um substituto para o XP, Scrum, Kanban, OpenUP, RUP ou qualquer metodologia que o mercado atualmente oferece, mas sim, **o BDD incorpora e melhora as ideias de muitas dessas metodologias**, ajudando assim e tornando a vida da equipe de software mais fácil. Portanto, o BDD é um conjunto de práticas de engenharia de software projetado para ajudar as equipes a construir e entregar mais rápido software de alta qualidade.

Para explicar o **funcionamento do BDD** vamos usar o seguinte exemplo: uma **equipe praticante de BDD** decide implementar uma nova funcionalidade e para isso, eles trabalham em conjunto com os usuários e outras partes interessadas para definir as histórias e cenários do que os usuários esperam dessa funcionalidade. Em particular, os usuários ajudam a definir um conjunto de exemplos concretos que ilustram resultados que a nova funcionalidade deve fornecer. Esses exemplos são criados utilizando um vocabulário comum e podem ser facilmente compreendidos pelos usuários finais e membros da equipe de desenvolvimento de software, e geralmente são expressos usando **Cenário**(Scenario), **Dado**(Given), **Quando**(When) e **Então** (Then).

Vejamos a **Figura 1** que mostra os **passos do BDD** utilizado pela equipe neste exemplo para especificação da nova funcionalidade.

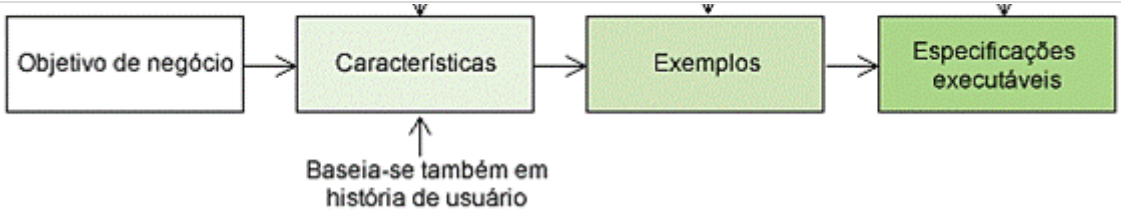

Figura 1. Os passos do BDD.

Com base no BDD, a equipe identificou e especificou o seguinte objetivo de negócio, definido com um exemplo concreto. Observe:

Cenário: Transferir dinheiro para uma conta poupança
Dado que eu tenho uma conta corrente com 1000.00
E que eu tenho uma conta de poupança com 2.000,00
Quando eu transferir 500,00 a partir de minha conta corrente para a minha conta poupança
Então eu deveria ter 500,00 em minha conta corrente
E eu deveria ter 2.500,00 em minha conta poupança

Depois de especificada a nova funcionalidade, sempre que possível estes exemplos concretos são automatizados sob a forma de especificações executáveis, que tanto valida o software quanto fornece uma documentação atualizada, técnica e funcional. Logo, existem várias ferramentas e frameworks que apoiam esta fase do BDD, transformando esses requisitos em testes automatizados que ajudam a orientar o desenvolvedor para que a nova funcionalidade seja desenvolvida corretamente e dentro do prazo.

Conhecendo o Cucumber

técnica BDD. Desde então o Cucumber cresceu e foi traduzido em várias linguagens, inclusive o Java, permitindo assim que vários de desenvolvedores desfrutem de suas vantagens. Diante disso, vejamos a **Figura 2**, que ilustra uma visão geral do Cucumber.

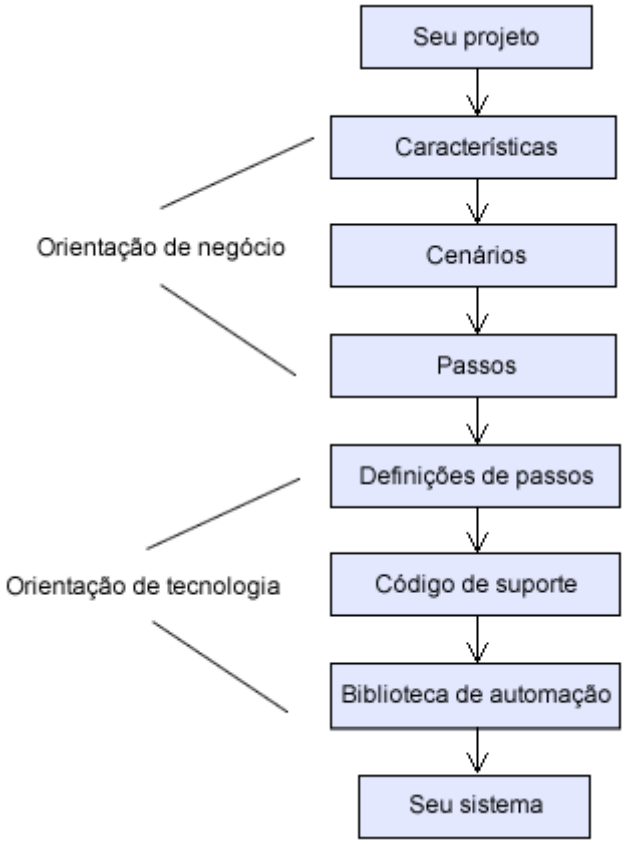

Figura 2. Visão geral do Cucumber.

Veja os passos para a utilização desse framework:

1. Descreva um comportamento em um texto simples;
2. Escreva uma definição dos passos em Java ou em outras linguagens;
3. Execute e veja os passos falhar;
4. Escreva o código para fazer os passos passar;

Como proposta deste artigo, faremos o desenvolvimento de testes de aceitação de duas funcionalidades utilizando o framework Cucumber em Java e a técnica BDD. Para começarmos, a técnica BDD se inicia na identificação do objetivo de negócio e como exemplo tomamos como objetivo de negócio a "Negociação bancária" que contém um Banco e Conta bancária. Vejamos as funcionalidades que devemos assegurar que funcionem.

1. **Primeira funcionalidade:** Consiste em possibilitar que o usuário realize as operações bancárias utilizando sua conta, que são:
   1.1. Fazer saque e depósito, considerando as seguintes restrições:
   1.1.1. Só libera o saque se o valor deste for menor ou igual ao valor do saldo disponível na conta;
   1.1.2. Só libera o depósito se o valor deste for menor ou igual ao valor do limite disponível na conta.
2. **Segunda funcionalidade:** Consiste em possibilitar o usuário a realizar operações básicas no banco, que são:
   2.1. Obter o total de dinheiro no banco;
   2.2. Obter o total de contas criadas no banco.

Conteúdo relacionado: Cursos de Engenharia de Software

Criando e configurando o projeto

Criaremos então um novo projeto Maven para adicionar e configurar as dependências necessárias dentro do arquivo `pom.xml`.

Além disso, desenvolveremos os testes de aceitação automatizados utilizando o

simple project" e clique em "Next" novamente. Na janela que aparece na **Figura 3** preencha os campos "Group Id " e "Artifact Id " e clique em "Finish".

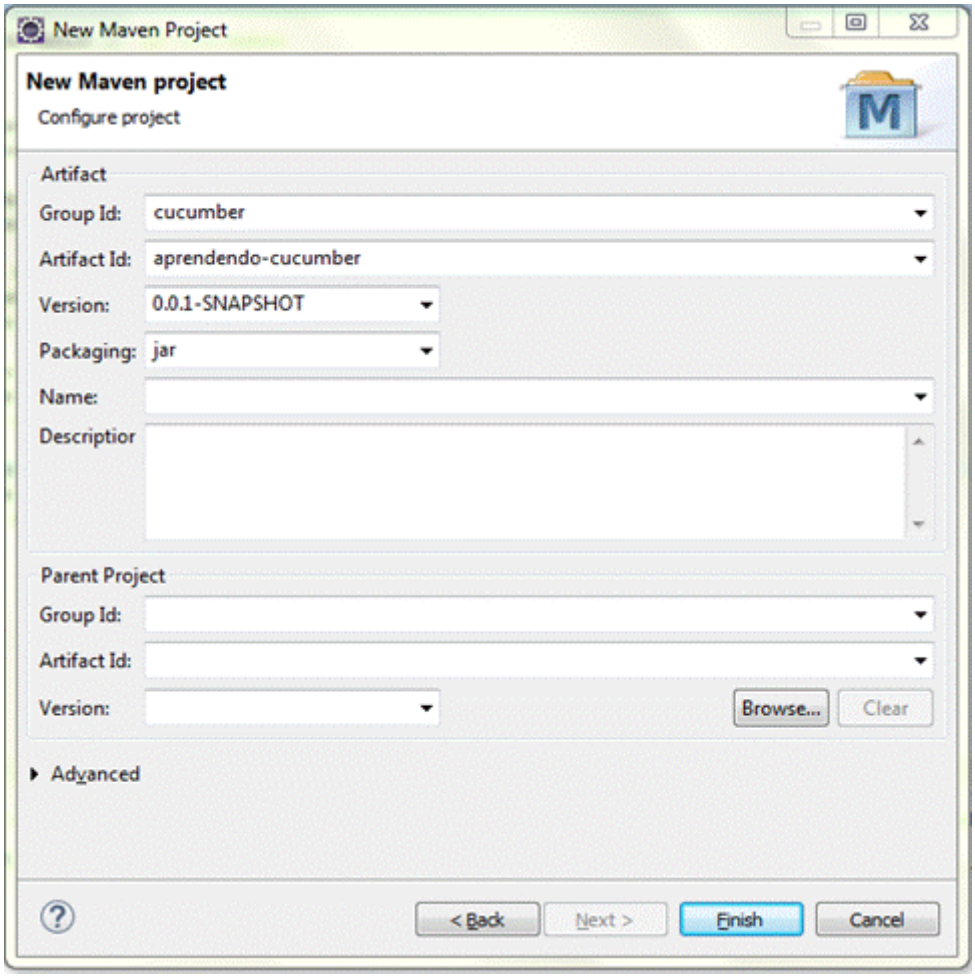

Figura 3. Janela do eclipse com as configurações do projeto Maven.

Para adicionar as dependências no POM do Maven abra o `pom.xml` e acrescente o código da **Listagem 1** entre as tags `<project>`. Logo após execute o “Maven Update Project” utilizando o “ALT+F5”.

Listagem 1. Dependências do projeto

```
6                       <!-- Versão do plugin maven -->
7                       <version>3.3</version>
8                       <configuration>
9                               <!-- Versão do java -->
10                              <source>1.8</source>
11                              <target>1.8</target>
12                      </configuration>
13              </plugin>
14      </plugins>
15  </build>
16  <properties>
17          <cucumber.version>1.2.0</cucumber.version>
18  </properties>
19  <dependencies>
20          <!-- JUnit -->
21          <dependency>
22                  <groupId>junit</groupId>
23                  <artifactId>junit</artifactId>
24                  <version>4.11</version>
25                  <scope>test</scope>
26          </dependency>
27          <!-- Cucumber -->
28          <dependency>
29                  <groupId>info.cukes</groupId>
30                  <artifactId>cucumber-java</artifactId>
31                  <version>${cucumber.version}</version>
32                  <scope>test</scope>
33          </dependency>
34  <dependency>
35  <groupId>info.cukes</groupId>
36                  <artifactId>cucumber-junit</artifactId>
37                  <version>${cucumber.version}</version>
38                  <scope>test</scope>
39          </dependency>
40          <dependency>
41                  <groupId>info.cukes</groupId>
42                  <artifactId>gherkin</artifactId>
43                  <version>2.12.2</version>
```

Precisamos agora criar a estrutura semelhante à **Figura 4** (pacote, classe e arquivos) dentro do nosso projeto Maven.

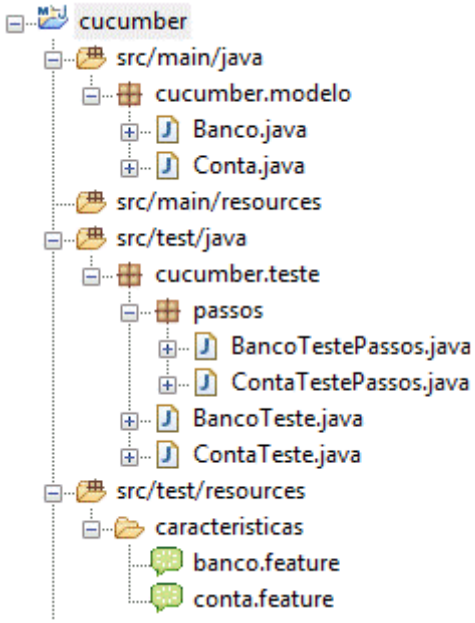

Figura 4. Estrutura do projeto.

Diante do projeto criado e configurado, precisamos agora adicionar o conteúdo em cada arquivo do nosso projeto.

Adicionando as classes e features dos testes

Vamos adicionar as classes Java e features (características) dos testes. Como estamos utilizando a técnica BDD, criaremos as features descrevendo os cenários contendo exemplos concretos baseando-se nas histórias de usuário descrito nos itens 1 e 2 da visão geral. Antes de começarmos a adicionar os arquivos, certifique-se que o plugin do cucumber esteja instalado na IDE Eclipse (veja na

nosso projeto.

Listagem 2. Código do arquivo `conta.feature`

```
# language: pt
  @ContaTeste
  Funcionalidade: Testar as operacoes basicas de conta
    O sistema deve prover o saque e deposito na conta de forma corre
    Seguindo as seguintes restrições:
    1) Só libera o saque, se o valor do saque for menor ou igual ao
        do saldo disponível na conta
    2) Só libera o deposito, se o valor do deposito for menor ou igu
        valor do limite disponível na conta

    Esquema do Cenario: Testar saque e deposito
      Dado a conta criada para o dono "<dono>" de numero <numero>
       com o limite <limite> e saldo <saldo>
      Quando o dono realiza o deposito no valor de <deposito> na cor
      E o dono realiza o primeiro saque no valor de <primeiro_saque>
      E o dono realiza o segundo saque no valor de <segundo_saque> r
      Entao o dono tem o saldo no valor de <saldo_esperado> na conta

Exemplos:
| dono |numero|limite|saldo|deposito|primeiro_saque|segundo_saque|sa
|Brendo| 111  | 1000 | 0   | 100    | 10           | 10          | 8
|Hiago | 222  | 1000 | 0   | 200    | 10           | 10          | 1
```

Note que estamos utilizando uma linguagem padrão para especificação de testes de aceitação chamada “Grerkin”, do Cucumber. Outro ponto a destacar é que estamos utilizando a sintaxe desse framework em Português, indicado por “# language: pt”. A anotação `@ContaTeste` está anotada neste arquivo, pois o teste fará a chamada para execução deste mediante esta tag.

Vejamos o que cada palavra-chave do Gherkin utilizada neste arquivo realiza:

no escopo máximo da especificação, mas poderia ser anotada acima de outras palavras-chaves do Gherkin;

- Funcionalidade: Nesta palavra-chave encontra-se uma descrição de alto nível de um recurso de software. Foi adicionado abaixo desta palavra-chave uma descrição (opcional, mas recomendável) que pode abranger várias linhas;
- Esquema do Cenario: Informa ao Cucumber que este cenário irá utilizar um conjunto de dados para executar exemplos N vezes descrito em seu escopo, que no nosso caso irá executar este cenário duas vezes, pois contém dois registros abaixo da palavra-chave “Exemplos”;
- Passos: Um passo geralmente começa com “Dado”, “Quando” ou “Entao”.
  - Dado: É utilizado para descrever um contexto inicial do cenário. Quando o Cucumber executa a palavra-chave “Dado”, espera-se que o cenário esteja em um estado definido, por meio e exemplo de uma criação ou configuração de objetos;
  - Quando: Utilizado para descrever um evento ou ação. Pode-se descrever, por exemplo, uma pessoa interagindo com o sistema ou pode ser um evento desencadeado por um sistema;
  - E: É semelhante ao “Dado”, “Quando” e “Entao”, pois ele é empregado quando um deles já foi declarado dentro de um mesmo cenário;
  - Entao: É utilizado para descrever um resultado esperado.
- Exemplos: É utilizado para estabelecer um conjunto de dados a serem executados nos passos definido no “Esquema do Cenario”;
- """: Informando que o valor é uma string;
- |:É utilizado em “Tabelas de Dados” para separar um conjunto de valores, como é declarado no arquivo anterior, abaixo da palavra-chave “Exemplos”.

adicione a **Listagem 3** no arquivo “banco.feature” do nosso projeto.

Listagem 3. Código do arquivo banco.feature

```
# language: pt
@BancoTeste
Funcionalidade: Testar as operacoes basicas de banco
  O sistema deve prover operações básicas de banco de forma correta.

  Contexto: Cria todas as contas e associa ao banco
    Dado que as contas sao do "Banco do Brasil"
      | dono                      | numero | saldo |
      | Abias Corpus Da Silva     | 111    | 100   |
      | Antônio Morrendo das Dores | 222    | 200   |
      | Carabino Tiro Certo       | 333    | 200   |

  Cenario: Verifica o total de contas criadas
    Dado o calculo do total de contas criadas
    Entao o total de contas e 3

  Cenario: Verifica o total de dinheiro no banco
    Dado o calculo do total de dinheiro
    Entao o total de dinheiro no banco e 500
```

Note que no arquivo banco.feature estávamos anotando a tag `@BancoTeste`, que será chamado pela classe de teste responsável por chamar esta especificação executável.

Observe que surgiram novas palavras-chave do Gherkin neste arquivo e que valem destaque:

- Contexto: É utilizando para definir um contexto inicial para cada cenário declarado no arquivo .feature;

Agora que definimos as nossas especificações executáveis para validar cada cenário da visão geral faremos a implementação das classes de testes.

Para começar, acrescente o código da **Listagem 4** na classe `ContaTeste.java`. Esta classe tem como objetivo fazer a chamada para a execução dos passos (os testes de aceitação) contidas na classe `ContaTestePassos.java`, que implementaremos posteriormente.

Listagem 4.Código da classe ContaTeste.java

```
1 @RunWith(Cucumber.class)
2 @CucumberOptions(features = "classpath:caracteristicas", tags = "@Cc
3 glue = "cucumber.teste.passos", monochrome = true, dryRun = false)
4 public class ContaTeste {
5 }
```

Observe que na classe `ContaTeste.java` existe uma anotação chamada `@RunWith` (`Cucumber.class`): isso diz ao JUnit que o Cucumber irá assumir o controle da execução dos testes nesta classe. Outra anotação definida na classe é a `@CucumberOptions`, onde podemos definir parâmetros customizáveis utilizados pelo Cucumber na execução dos testes. Veja a seguir uma descrição sobre cada parâmetro desta anotação:

- `Features`: É utilizada para ajudar o Cucumber na localização das features (especificação executáveis), que no caso está localizada em uma pasta dentro do projeto chamada “caracteristicas”;
- `Tags`: É utilizada para definir as tags neste parâmetro, uma vez uma mesma tag definida neste atributo e no (s) arquivo (s) .feature. Quando o Cucumber

- Glue: É utilizada para ajudar o Cucumber na localização das classes que contém os passos para os testes de aceitação, que no caso estão localizadas no pacote `cucumber.teste.passos`;
- Monochrome: É utilizado para definir a formatação do resultado dos testes na saída da console. Se marcado como ”true”, o resultado dos testes sai na forma legível, mas se “false”, não sai tão legível;
- DryRun: esta opção pode ser definida como “true” ou “false“. Se estiver marcado como “true”, isso significa que o Cucumber só verifica se cada etapa definida no arquivo .feature tem código correspondente. Considerando ainda “true”, se na execução de um arquivo .feature o Cucumber não achar nenhum código (Java) correspondente a esse arquivo, então o Cucumber gera o código correspondente. Se marcado como “false”, o Cucumber não faz essa verificação.

Na sequência, vamos adicionar a **Listagem 5**na classe `ContaTestePassos.java`, que será chamada pelo Cucumber mediante a chamada da classe `ContaTeste`, para executar os teses de aceitação definidos no arquivo conta.feature.

Listagem 5. Código da classe `ContaTestePassos.java`

```
1  public class ContaTestePassos {
2
3          private Conta conta;
4
5          @Dado("^a conta criada para o dono \"(.*?)\" de numero (\\d+)
6          public void a_conta_criada_para_o_dono_de_numero_com_o_limite
7                       Double saldo) throws Throwable {
8                // Definição de conta
9                conta = new Conta(dono, numero, limite, saldo);
10         }
11
```

```
16          }
17
18          @E("^o dono realiza o primeiro saque no valor de (\\d+) na cc
19          public void o_dono_realiza_o_primeiro_saque_no_valor_de_na_cc
20                assertTrue("O dono " + conta.getDono() + " não tem salc
21                            conta.sacar(valorSaque));
22          }
23
24          @E("^o dono realiza o segundo saque no valor de (\\d+) na cor
25          public void o_dono_realiza_o_segundo_saque_no_valor_de_na_cor
26                assertTrue("O dono " + conta.getDono() + " não tem salc
27                            conta.sacar(valorSaque));
28          }
29
30          @Entao("^o dono tem o saldo no valor de (\\d+) na conta$")
31          public void o_dono_tem_o_saldo_na_conta_no_valor_de(Double sa
32                assertEquals("O dono " + conta.getDono() + " está com c
33                            conta.getSaldo());
34          }
35    }
```

Observe que na classe `ContaTestePassos` estamos utilizadas as anotações `@Dado`, `@Quando`, `@E` e `@Entao`, que correspondem ao mesmo conteúdo e as palavras-chave do Gherkin definidas nos arquivos .feature. Outro ponto a destacar é que em todos os métodos da classe `ContaTestePassos` definimos algumas expressões regulares, como (\\d+) (extrai valor decimal), \"(.*?)\" (extrai qualquer valor string). Isso diz ao Cucumber para extrair os valores definidos no arquivo .feature a qual a classe corresponde, e em tempo de execução injetar esses valores nos parâmetros de cada método correspondente.

Por fim, note que dentro de cada anotação existe no início uma expressão regular "^" e no final "$": as duas expressões estabelecem o início e fim da leitura do Cucumber em cada linha da especificação.

contidas na classe `BancoTestePassos.java` (este iremos implementar mais adiante).

Listagem 6.Código da classe `BancoTeste.java`

```
1  @RunWith(Cucumber.class)
2  @CucumberOptions(features = "classpath:caracteristicas", tags = "@Ba
3  glue = "cucumber.teste.passos", monochrome = true, dryRun = false)
4  public class BancoTeste {
5  }
```

Na sequência, vamos acrescentar a **Listagem 7**na classe `BancoTestePassos.java`, que será chamada pelo Cucumber (mediante a chamada da classe `BancoTeste`) para executar os passos (teses de aceitação) definidos no arquivo banco.feature.

Listagem 7.Código da classe `BancoTestePassos.java`

```
1  public class BancoTestePassos {
2
3          private Banco banco;
4          private int totalContas;
5          private Double totalDinheiro;
6
7          @Dado("^que as contas sao do \"(.*?)\"$")
8          public void que_as_contas_sao_do(String nome, List<Conta> lis
9                  // Definição do banco e associando as contas
10                 banco = new Banco(nome, listaDeContas);
11
12         }
13
14         @Dado("^o calculo do total de contas criadas$")
15         public void o_calculo_do_total_de_contas_criadas() throws Thr
16                 totalContas = banco.getListaDeContas().size();
17         }
```

```
22          }
23
24          @Dado("^o calculo do total de dinheiro$")
25          public void o_calculo_do_total_de_dinheiro() throws Throwable
26                totalDinheiro = banco.getListaDeContas().stream().mapTo
27                (c -> c.getSaldo()).sum();
28          }
29
30          @Entao("^o total de dinheiro no banco e (\\d+)$")
31          public void o_total_de_dinheiro_no_banco_e(Double totalDinhei
32            throws Throwable {
33
34                assertEquals("O cálculo do total de dinheiro no banco "
35                 + " está incorreto",
36                 totalDinheiroEsperado, totalDinheiro);
37          }
38    }
39
```

Agora implementaremos as classes principais que de fato serão testadas em conjunto com as especificações declaradas anteriormente.

Adicionando as classes principais

Vamos começar adicionando a **Listagem 8** na classe `Conta.java`, que representa uma entidade do mundo real "Conta bancária". Esta tem como responsabilidade fornecer métodos úteis que serão utilizadas nos testes de aceitação, em particular para atender os requisitos do item 1 da visão geral.

Listagem 8. Código da classe `Conta.java`

```
1  public class Conta {
2
3          private String dono;
```

```
8         public Conta(String dono, int numero, Double limite, Double s
9               this.dono = dono;
10              this.numero = numero;
11              this.saldo = saldo;
12              this.limite = limite;
13
14        }
15
16        public boolean sacar(Double valor) {
17              if (saldo <= valor) {
18                    // Não pode sacar
19                    return false;
20              } else {
21                    // Pode sacar
22                    saldo = saldo - valor;
23                    return true;
24              }
25        }
26
27        public boolean depositar(Double quantidade) {
28
29              if (limite <= quantidade + saldo) {
30                    // Não pode depositar
31                    return false;
32              } else {
33                    // Pode depositar
34                    saldo += quantidade;
35                    return true;
36              }
37        }
38
39        public String getDono() {
40              return dono;
41        }
42
43        public void setDono(String dono) {
44              this.dono = dono;
45        }
```

```
51          public void setNumero(Integer numero) {
52                this.numero = numero;
53          }
54
55          public Double getSaldo() {
56                return saldo;
57          }
58
59          public void setSaldo(Double saldo) {
60                this.saldo = saldo;
61          }
62
63          public Double getLimite() {
64                return limite;
65          }
66
67          public void setLimite(Double limite) {
68                this.limite = limite;
69          }
70    }
```

Em seguida, acrescente a **Listagem 9** na classe `Banco.java`, que representa uma entidade do mundo real "Banco". Este tem como responsabilidade fornece métodos úteis que serão utilizados posteriormente nos testes de aceitação, em particular para atender os requisitos do item 2 da visão geral.

Listagem 9. Código da classe `Banco.java`

```
1   public class Banco {
2
3          private String nome;
4          private List<Conta> listaDeContas;
5          public Banco(String nome, List<Conta> listaDeContas) {
6                 this.nome = nome;
7                 this.listaDeContas= listaDeContas;
```

```
12              this.nome = nome;
13        }
14        public List<Conta> getListaDeContas() {
15              return listaDeContas;
16        }
17        public void setListaDeContas(List<Conta> listaDeContas) {
18              this.listaDeContas = listaDeContas;
19        }
20
21  }
22
```

Resultado dos testes

Nesta seção iremos executar e verificar os resultados dos testes de aceitação utilizando o JUnit em conjunto com o Cucumber.

Para executarmos todos os testes definidos em nosso projeto selecione o projeto, depois clique com o botão direito do mouse selecionando a opção “Run As > JUnit Test”. Se tudo ocorreu com sucesso, teremos um resultado semelhante à **Figura 5**.

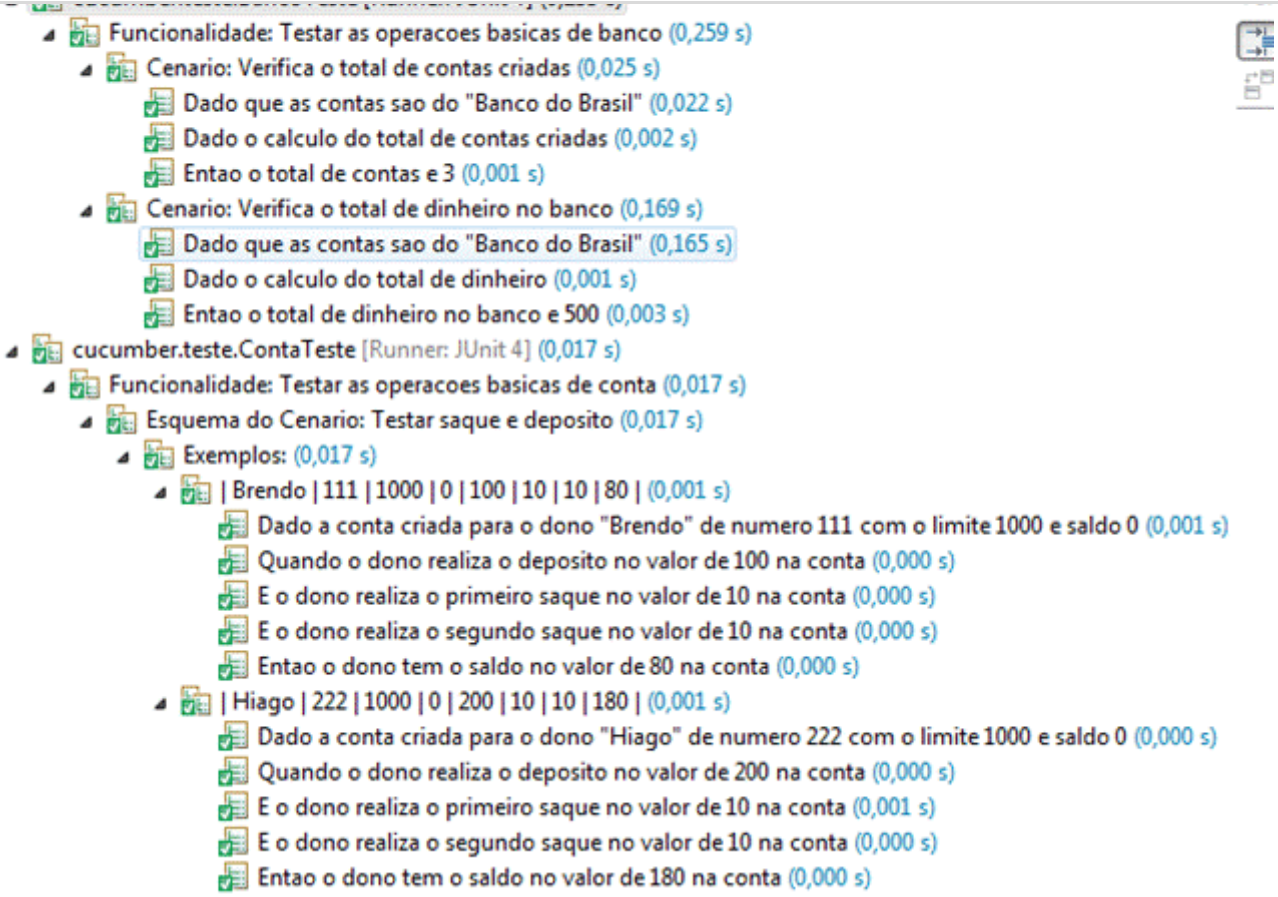

Figura 5. Resultado dos testes com JUnit e Cucumber

> Observação: Caso queria executar apenas os testes da classe `BancoTeste`, é só selecionar a classe e depois com o botão direto do mouse clicar na opção "Run As > JUnit Test". O mesmo é válido para a classe `ContaTeste`.

Por fim, conseguimos por meio da utilização do Cucumber e a técnica BDD validar o nosso código, que por sinal, atende perfeitamente os requisitos definidos na visão geral.

Links Úteis

- Curso de PHP e MVC: Como fazer um CRUD 1:N:

- **Criando meu primeiro projeto no Java**:

  Neste curso você aprenderá a criar o seu primeiro programa com Java, e não, ele não será um simples “Hello, World!”. :) Para isso, vamos começar ensinando como instalar o Java e preparar o ambiente de desenvolvimento.

Saiba mais sobre Java ;)

- **Guias Java** :

  Encontre aqui os Guias de estudo que vão ajudar você a aprofundar seu conhecimento na linguagem Java. Desde o básico ao mais avançado. Escolha o seu!

- **Carreira Programador Java**:

  Nesse Guia de Referência você encontrará o conteúdo que precisa para iniciar seus estudos sobre a tecnologia Java, base para o desenvolvimento de aplicações desktop, web e mobile/embarcadas.

- **Formulário de cadastro com JSF e Bootstrap**:

  Aprenda neste exemplo como criar interfaces ricas com Bootstrap e JSF. Saiba como o Pass-through elements pode te ajudar a ter mais controle sobre o HTML gerado pelos componentes nativos.

Bibliografia

ROSE, Seb; WYNNE, Matt; HELLESøY, Aslak.**The Cucumber for Java Book:** Behaviour-Driven Development for Testers and Developers. United States Of America: The Pragmatic Programmers, LLC., 2015.

YE, Wayne.**Instant Cucumber BDD How-to.** Birmingham: Packt Publishing, 2013.

https://cucumber.io/docs/reference#html

SMART, JOHN FERGUSON.**BDD in Action:** Behavior-Driven Development for the whole software lifecycle . : Manning Publications Co, 2015.

Plugin do Cucumber

https://marketplace.eclipse.org/content/cucumber-jvm-eclipse-plugin

Tecnologias:

Comece agora

Por **Brendo**
Em 2015

RECEBA NOSSAS NOVIDADES

Informe o seu e-mail

Receber Newsletter

Suporte ao aluno

Ver minhas dúvidas

Olá, eu sou o Rodolfo, seu professor. Qual a sua dúvida sobre este conteúdo?

Você não está sozinho. Poste aqui a sua dúvida e todo o time de professores será