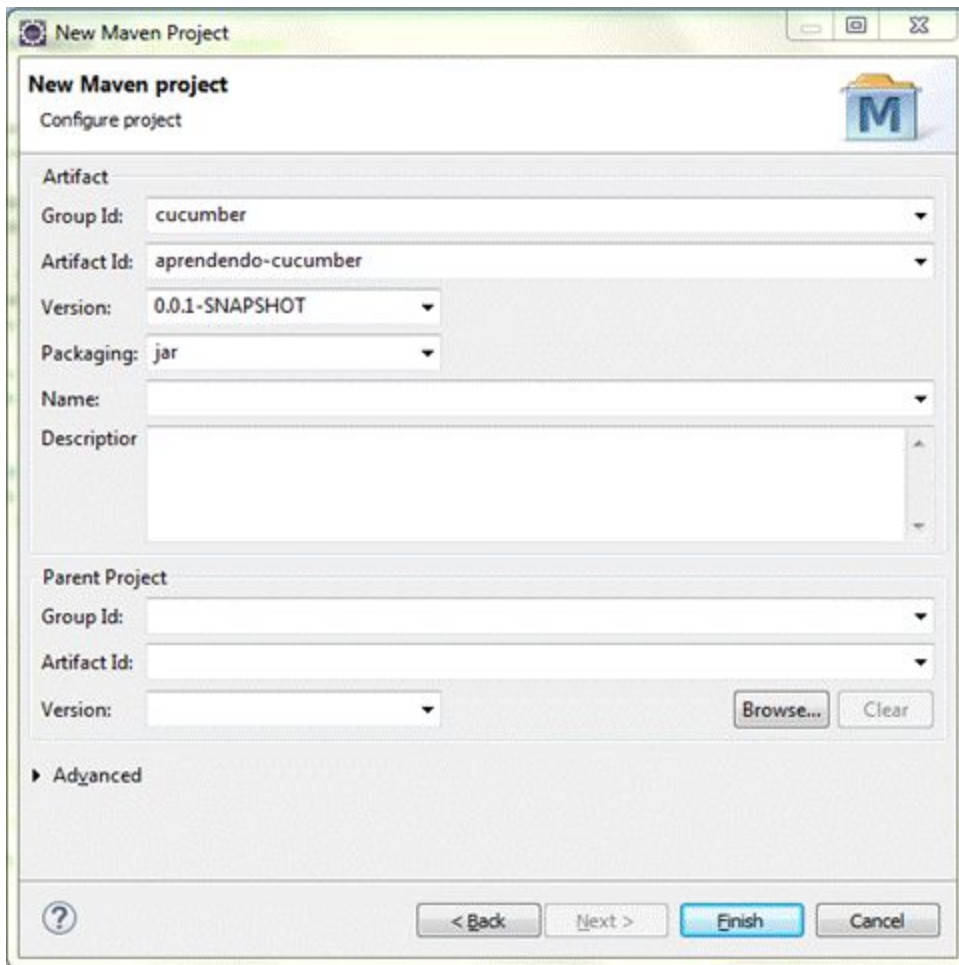


## Configuração do projeto no Eclipse

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 Cucumber.**

Primeiro inicie a IDE Eclipse e depois selecione "File > New> Other ..." e "Maven > Maven Project". Clique em "Next" e na tela seguinte marque o checkbox "Create a 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".



The screenshot shows the 'New Maven Project' dialog box in Eclipse. The title bar says 'New Maven Project'. Inside, the 'New Maven project' section has a 'Configure project' link and a Maven icon. The 'Artifact' section contains the following fields: 'Group Id' (cucumber), 'Artifact Id' (aprendendo-cucumber), 'Version' (0.0.1-SNAPSHOT), and 'Packaging' (jar). There are also empty fields for 'Name' and 'Description'. The 'Parent Project' section has empty fields for 'Group Id', 'Artifact Id', and 'Version', along with 'Browse...' and 'Clear' buttons. An 'Advanced' section is collapsed. At the bottom, there are buttons for '< Back', 'Next >', 'Finish' (highlighted in blue), and 'Cancel'.

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

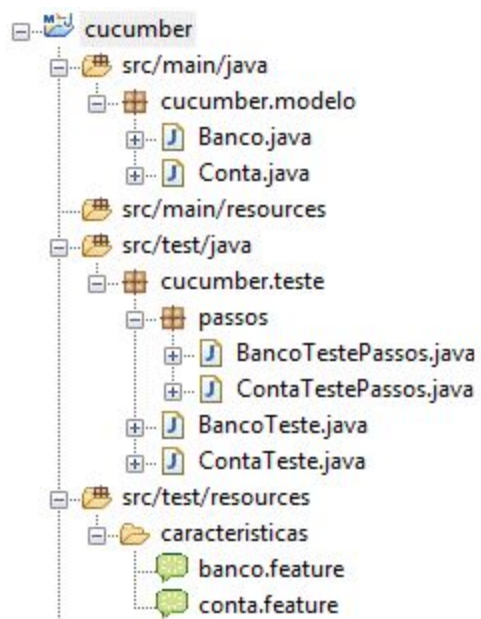
```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <!-- Versão do plugin maven -->
      <version>3.3</version>
      <configuration>
        <!-- Versão do java -->
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
<properties>
  <cucumber.version>1.2.0</cucumber.version>
</properties>
<dependencies>
  <!-- JUnit -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <!-- Cucumber -->
  <dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
```

```

        <version>${cucumber.version}</version>
        <scope>test</scope>
    </dependency>
</dependency>
<groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>${cucumber.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>gherkin</artifactId>
    <version>2.12.2</version>
    <scope>test</scope>
</dependency>
</dependencies>

```

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 seção **Link** onde fazer o download do mesmo).

Em seguida faremos a primeira especificação do cenário de teste de aceitação utilizando o Cucumber, então adicione a **Listagem 2** no arquivo “conta.feature” do 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 correta.
    Seguindo as seguintes restrições:
    1) Só libera o saque, se o valor do saque for menor ou igual ao valor do saldo disponível na conta
    2) Só libera o deposito, se o valor do deposito for menor ou igual ao 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 conta
        E o dono realiza o primeiro saque no valor de <primeiro_saque> na conta
        E o dono realiza o segundo saque no valor de <segundo_saque> na conta
        Entao o dono tem o saldo no valor de <saldo_esperado> na conta

    Exemplos:
```

dono	numero	limite	saldo	deposito	primeiro_saque	segundo_saque	saldo_esperado	
Brendo	111	1000	0	100	10	10	80	
Hiago	222	1000	0	200	10	10	180	

Note que estamos utilizando uma linguagem padrão para especificação de testes de aceitação chamada “Gherkin”, 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:

- #: Utilizado para escrever comentário;
- @: Simbologia para marcar uma tag: perceba que estávamos utilizando a tag 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 Cenário: 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 Cenário”;
- """: 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”.

Agora que nossa primeira especificação executável já está pronta, faremos a segunda especificação tomando como base o item 2 da visão geral. Então 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 utilizado para definir um contexto inicial para cada cenário declarado no arquivo .feature;
- Cenário: É utilizado para especificar um exemplo concreto que ilustra uma regra de negócio, basicamente constituída por uma lista de passos.

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

```

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

```

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 executar, esta classe só executará em conjunto apenas os arquivos .feature marcados com a mesma tag;
- **Glue:** É utilizada para ajudar o Cucumber na localização das classes que contêm 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 testes de aceitação definidos no arquivo conta.feature.

#### **Listagem 5.** Código da classe ContaTestePassos.java

```

public class ContaTestePassos {

    private Conta conta;

```

```

    @Dado("^a conta criada para o dono \"(.*)\" de numero (\\d+) com o limite (\\d+) e saldo (\\d+)$")
    public void a_conta_criada_para_o_dono_de_numero_com_o_limite_e_saldo(String dono, int
numero, Double limite,
        Double saldo) throws Throwable {
        // Definição de conta
        conta = new Conta(dono, numero, limite, saldo);
    }

    @Quando("^o dono realiza o deposito no valor de (\\d+) na conta$")
    public void o_dono_realiza_o_deposito_no_valor_de_na_conta(Double valorDeposito) throws
Throwable {
        assertTrue("O dono " + conta.getDono() + " não tem limite disponível na conta para este valor
de deposito",
            conta.depositar(valorDeposito));
    }

    @E("^o dono realiza o primeiro saque no valor de (\\d+) na conta$")
    public void o_dono_realiza_o_primeiro_saque_no_valor_de_na_conta(Double valorSaque) throws
Throwable {
        assertTrue("O dono " + conta.getDono() + " não tem saldo disponível na conta para este valor
de saque",
            conta.sacar(valorSaque));
    }

    @E("^o dono realiza o segundo saque no valor de (\\d+) na conta$")
    public void o_dono_realiza_o_segundo_saque_no_valor_de_na_conta(Double valorSaque) throws
Throwable {
        assertTrue("O dono " + conta.getDono() + " não tem saldo disponível na conta para este valor
de saque",
            conta.sacar(valorSaque));
    }

    @Entao("^o dono tem o saldo no valor de (\\d+) na conta$")
    public void o_dono_tem_o_saldo_na_conta_no_valor_de(Double saldoEsperado) throws Throwable
{
        assertEquals("O dono " + conta.getDono() + " está com o saldo incorreto na conta",
saldoEsperado,

```



```

        conta.getSaldo());
    }
}

```

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.

Agora adicione o código da **Listagem 6** na classe “BancoTeste.java”, que tem como objetivo fazer a chamada para a execução dos passos (os testes de aceitação) contidas na classe “BancoTestePassos.java” (este iremos implementar mais adiante).

#### Listagem 6. Código da classe BancoTeste.java

```

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

```

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

```

public class BancoTestePassos {

    private Banco banco;
    private int totalContas;
    private Double totalDinheiro;

```

```

    @Dado("^que as contas sao do \"(.*)\"$")
    public void que_as_contas_sao_do(String nome, List<Conta> listaDeContas) throws Throwable {
        // Definição do banco e associando as contas
        banco = new Banco(nome, listaDeContas);
    }

    @Dado("^o calculo do total de contas criadas$")
    public void o_calculo_do_total_de_contas_criadas() throws Throwable {
        totalContas = banco.getListaDeContas().size();
    }

    @Entao("^o total de contas e (\\d+)$")
    public void o_total_de_contas_e(int totalContasEsperado) throws Throwable {
        assertEquals("O cálculo do total de contas está incorreto", totalContasEsperado, totalContas);
    }

    @Dado("^o calculo do total de dinheiro$")
    public void o_calculo_do_total_de_dinheiro() throws Throwable {
        totalDinheiro = banco.getListaDeContas().stream().mapToDouble(c -> c.getSaldo()).sum();
    }

    @Entao("^o total de dinheiro no banco e (\\d+)$")
    public void o_total_de_dinheiro_no_banco_e(Double totalDinheiroEsperado) throws Throwable {

        assertEquals("O cálculo do total de dinheiro no banco " + banco.getNome() + " está incorreto",
            totalDinheiroEsperado, totalDinheiro);
    }
}

```

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

```
public class Conta {

    private String dono;
    private Integer numero;
    private Double saldo;
    private Double limite;

    public Conta(String dono, int numero, Double limite, Double saldo) {
        this.dono = dono;
        this.numero = numero;
        this.saldo = saldo;
        this.limite = limite;
    }

    public boolean sacar(Double valor) {
        if (saldo <= valor) {
            // Não pode sacar
            return false;
        } else {
            // Pode sacar
            saldo = saldo - valor;
            return true;
        }
    }

    public boolean depositar(Double quantidade) {

        if (limite <= quantidade + saldo) {
            // Não pode depositar
```

```
        return false;
    } else {
        // Pode depositar
        saldo += quantidade;
        return true;
    }
}

public String getDono() {
    return dono;
}

public void setDono(String dono) {
    this.dono = dono;
}

public Integer getNumero() {
    return numero;
}

public void setNumero(Integer numero) {
    this.numero = numero;
}

public Double getSaldo() {
    return saldo;
}

public void setSaldo(Double saldo) {
    this.saldo = saldo;
}

public Double getLimite() {
    return limite;
}

public void setLimite(Double limite) {
```

```
        this.limite = limite;
    }
}
```

Em seguida, acrescente a **Listagem 9** na classe “Banco.java”, que representa uma entidade do mundo real “Banco”. Este tem como responsabilidade fornecer 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

```
public class Banco {

    private String nome;
    private List<Conta> listaDeContas;
    public Banco(String nome, List<Conta> listaDeContas) {
        this.nome = nome;
        this.listaDeContas= listaDeContas;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public List<Conta> getListaDeContas() {
        return listaDeContas;
    }
    public void setListaDeContas(List<Conta> listaDeContas) {
        this.listaDeContas = listaDeContas;
    }
}
```

## 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 direito do mouse clicar na opção "Run As > JUnit Test". O mesmo é valido 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.

Obrigado e até mais!

## **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>