PROCESSO DE DESENVOLVIMENTO DOCUMENTAÇÃO DE PROJETO		
Nome do curso: Testes Automatizados	Aula 7 – BDD – PARTE 2	Responsável: Aline Freitas

Olá, seja bem-vindo(a)!

Neste curso, você aprendeu que o Behaviour Driven Development (BDD) é uma técnica utilizada para desenvolver projetos de software de uma forma transparente, aumentando a produtividade do time, e ao mesmo tempo focando na estratégia de um software com valor para o cliente. Portanto, você vai conhecer algumas ferramentas e técnicas para implementar a metodologia ágil BDD. Preparado(a)? Vamos lá!

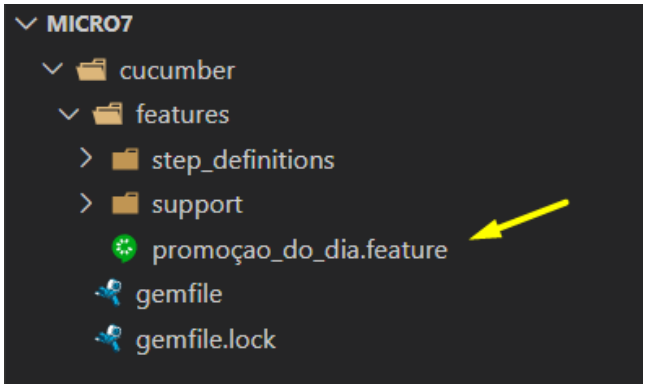
Pronto, agora que você tem o Cucumber instalado e as estruturas básicas de diretórios feitas, tudo está pronto para escrever uma especificação com Cucumber. Nesse momento, especifique o cenário para uma estória de usuário. Quando a técnica do BDD (desenvolvimento guiado por comportamento) é utilizada juntamente com o framework Cucumber, você utiliza exemplos concretos para especificar o que quer que o sistema faça.

No Cucumber, os cenários são definidos nos arquivos com extensão ".feature", e esses arquivos devem estar dentro do diretório (pasta) **Features.**

Então, imagine que o cenário de teste seja verificar qual a promoção do dia na Loja XYZ.

Dentro do projeto no VSCode, deve-se criar um arquivo **"promoção_do_dia.feature"** no diretório **"features"**. O print, a seguir, mostra a tela do projeto VSCode, onde há a pasta "Micro7".

Dentro dela, há pastas “cucumber” e “features”, que está expandida e mostra as pastas “step_definitions” e “support”, e os arquivos “promoção_do_dia.feature”, “gemfile” e “gemfile.lock”. Com isso, uma seta amarela aponta para o arquivo “promoção_do_dia.feature”.



Dentro do arquivo **“promoção_do_dia.feature”**, você vai descrever o seguinte cenário de teste:

```
1 #language:pt
2
3 Funcionalidade: Promoção do dia!
4       Queremos saber qual a promoção do dia na loja XYZ
5
6 Cenário: Hoje é dia de desconto em eletrônicos
7       Dado que hoje é “quarta-feira”
8       Quando eu pergunto qual é a promoção do dia
9       Então a resposta deve ser “desconto de 10% em monitores”
```

Ah, não esqueça de salvar o arquivo, hein?

Com o arquivo .feature criado, o Cucumber nomeia suas especificações como funcionalidade (features). Uma funcionalidade tem um título, uma descrição, também chamada de narrativa, e um ou mais cenários. Agora, você entenderá esse processo com mais detalhes:

A primeira linha **#language:pt** define o idioma da sintaxe Gherkin. Portanto, essa linha serve para dizer que estamos escrevendo uma feature com Cucumber e utilizando o idioma português. Caso não tivesse colocado essa linha, o Cucumber tentaria interpretar essa feature, utilizando o idioma padrão dele, que é inglês. A diferença básica entre utilizar um idioma ou outro são as palavras reservadas do Gherkin.

Na linha 3, temos a palavra-chave **"Funcionalidade"** com o nome da funcionalidade do sistema que que será testada. Para isso, a descrição da funcionalidade sendo especificada, deve ser curta.

Na linha 4, temos uma breve descrição do que a funcionalidade faz. É um texto livre que você pode usar para descrever o que for necessário para a descrição dessa funcionalidade. Assim, o Cucumber não executa essa descrição, apenas apresenta uma documentação de teste.

Na linha 6, **Cenário: Hoje é dia de desconto em eletrônicos** é um cenário que desejamos executar. Portanto, é onde você deve descrever o comportamento do usuário ao usar o sistema que será desenvolvido.

Nas últimas três linhas, onde estão as palavras-chaves: **Dado**, **Quando** e **Então**, têm esses passos (steps) para reproduzir o comportamento do cenário, pois serão eles que o Cucumber executará automaticamente.

Agora, você vai aprender a executar o cenário de teste.

Para rodar o cenário, você precisa informar ao Cucumber, para executá-lo, e o comando que devemos informar no terminal do Windows, que é "**cucumber**". Assim, você precisa estar dentro do diretório "c:\micro7\cucumber. Então, vamos lá, execute o comando:

cucumber

Esse é o retorno da execução Cucumber, no terminal do Windows.

```
# language: pt
Funcionalidade: Promoção do dia!
  Queremos saber qual a promoção do dia na loja XYZ

  Cenário: Hoje é dia de desconto em eletrônicos # features/promocao_do_dia.feature:6
    Dado que hoje é "quarta-feira" # features/promocao_do_dia.feature:7
    Quando eu pergunto qual é a promoção do dia # features/promocao_do_dia.feature:8
    Então a resposta deve ser "desconto de 10% em monitores" # features/promocao_do_dia.feature:9

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.154s

You can implement step definitions for undefined steps with these snippets:

Dado("que hoje é {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
end

Quando("eu pergunto qual é a promoção do dia") do
  pending # Write code here that turns the phrase above into concrete actions
end

Então("a resposta deve ser {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
```

end

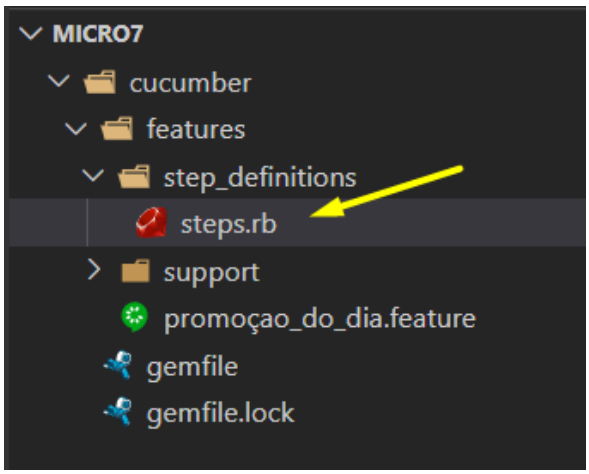
```
c:\Micro7\cucumber>cucumber
*** WARNING: You must use ANSICON 1.31 or higher (https://github.com/adoxa/ansicon/) to get coloured output on Windows
# language: pt
Funcionalidade: Promoção do dia!
  Queremos saber qual a promoção do dia na loja XYZ

  Cenário: Hoje é dia de desconto em eletrônicos             # features/promoçao_do_dia.feature:6
    Dado que hoje é "quarta-feira"                           # features/promoçao_do_dia.feature:7
    Quando eu pergunto qual é a promoção do dia              # features/promoçao_do_dia.feature:8
    Então a resposta deve ser "desconto de 10% em monitores" # features/promoçao_do_dia.feature:9

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.166s

You can implement step definitions for undefined steps with these snippets:

Dado("que hoje é {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
end

Quando("eu pergunto qual é a promoção do dia") do
  pending # Write code here that turns the phrase above into concrete actions
end

Então("a resposta deve ser {string}") do |string|
  pending # Write code here that turns the phrase above into concrete actions
end
```

Note que o Cucumber finaliza a execução do cenário e dos steps como "**undefined**". Assim, ele entende que não existe códigos Ruby implementados, para automatizar o comportamento do usuário no cenário de teste, e retorna ao terminal os métodos, já em código Ruby, para você finalizar no seu teste automatizado.

O próximo passo é implementar os steps definitions, que são os passos que o Cucumber interpretará e executará. Para isso, retorne ao VSCode e crie o arquivo **"steps.rb"** no diretório **features/step_definitions/.**

A seguir, no print da tela do projeto VSCode, há a pasta "Micro7" e, dentro dela, há as pastas "cucumber" e "features", que está expandida e mostra a pasta "step_definitions", que está expandida e mostra um arquivo em destaque chamado "steps.rb" e "support"; e os arquivos "promoção_do_dia.feature", "gemfile" e

“gemfile.lock”. Na imagem a seguir, há uma seta amarela apontada para o arquivo “steps.rb”.



Agora, copie os métodos, que a execução do comando **cucumber** retornou no terminal do Windows, e cole-os no seu arquivo **“steps.rb”** no VSCode. Além disso, localize e acione a opção “salvar arquivo”.

```
Dado("que hoje é {string}") do |string|
    pending # Write code here that turns the phrase above into concrete actions
end
Quando("eu pergunto qual é a promoção do dia") do
    pending # Write code here that turns the phrase above into concrete actions
end
Então("a resposta deve ser {string}") do |string|
    pending # Write code here that turns the phrase above into concrete actions
end
```

Nesse momento, que você tem os steps em métodos Ruby, você deverá pedir ao Cucumber para executar esses cenários novamente. Para tanto, volte ao terminal do Windows e execute o comando cucumber mais uma vez.

A partir desse comando, pode-se verificar que o Cucumber encontrou os códigos Ruby que dão vida à especificação, mas, o resultado do cenário, agora, passou a

ser “pending” “pendente”, ou seja, já temos a estrutura que testa. Então, nesse momento, falta somente implementar.

Resultado dos cenários e passos após o comando:

scenario (1 pending)
steps (2 skipped, 1 pending)

Esse ciclo é o mesmo que você estudou na técnica do TDD. Ou seja, executando o teste com falha. Por isso, é necessário que você implemente o código em Ruby dos passos que serão executados, ou dos passos que estão como “pending”, “pendente”.

No arquivo **“steps.rb”** no VSCode, implemente os seguintes códigos:

```
class Loja
  def promoçao_do_dia(dia)
  end
end

Dado("que hoje é {string}") do |dia|
 @hoje = dia
end

Quando("eu pergunto qual é a promoção do dia") do
  @valor_obtido = Loja.new.promoçao_do_dia(@hoje)
end

Então("a resposta deve ser {string}") do |promoçao_esperada|
  expect(@valor_obtido).to eql promoçao_esperada
end
```

Não esqueça de salvar o arquivo **“steps.rb”** no VSCode.

Por fim, recapitulando o que você acabou de fazer, primeiro, foi implementado o comportamento, em que a classe **Loja** representa a unidade de código que deve

ser desenvolvida, para, em seguida, serem implementados os steps que ativam e acessam os recursos da classe **Loja**, para que esta seja testada.

Em último lugar, agora que você tem os steps em métodos Ruby, solicite ao Cucumber para executá-lo novamente nosso teste. Para isso, volte para o terminal do Windows, execute o comando cucumber e os resultados dos cenários. Assim, os passos, após o comando, serão:

scenario (1 failed)
steps (1 failed, 2 passed)

Os dois primeiros steps "**passos"** estão passando, mas o último está falhando, não é mesmo? Calma! Isso acontece porque o método **"promoção_do_dia"** só recebe o valor **"quarta-feira"** como descrito na funcionalidade, dentro do arquivo **promoção_do_dia.feature**, mas não possui uma regra para tratar essa informação e fazer com que o comportamento especificado devolva o valor esperado para o cenário.

Para resolvermos isso, você deve alterar a definição do método **def promoção_do_dia** da classe **Loja**, no arquivo **steps.rb**, adicionando o seguinte código Ruby:

```
class Loja
  def promoçao_do_dia(dia)
    if dia == "quarta-feira"
  return "desconto de 10% em monitores"
 end
  end
end
```

Novamente, solicite ao Cucumber para executá-lo no teste. No terminal, execute o comando cucumber mais uma vez.

No terminal do Windows, você terá o resultado que o 1 cenário passou e 3 passos passaram. É possível observar os seguintes resultados dos cenários e passos, após o comando:

scenario (1 passed)
steps (3 passed)

Então, para finalizarmos, nesta aula, você aprendeu como instalar o framework Cucumber, além de instalar o pacote Ruby e automatizar um cenário de teste, usando o processo do BDD para desenvolver uma funcionalidade simples, guiado pelo comportamento do usuário no sistema.
Por hoje é só!
Até mais!

Referência:
https://code.visualstudio.com/
http://rubyinstaller.org/downloads/
https://www.falandoemtestes.com.br/2018/03/27/cucumber-configuracao-windows/