Behavior Driven Development (BDD)

Jailma Januário da Silva

Leonardo Massayuki Takuno

Resumo

Os objetivos desta parte são: (I) Conhecer sobre a abordagem de desenvolvimento orientado por comportamento - do inglês Behavior Driven Development (BDD). (II) Entender como criar arquivos de funcionalidades na linguagem Gherkin (III) Criar um projeto utilizando pytest-bdd.

Introdução

Esta parte tem como objetivo apresentar uma abordagem de Desenvolvimento Orientado a Comportamento, do inglês *Behavior Driven Development* (BDD), que é uma abordagem de desenvolvimento ágil que encoraja a colaboração de pessoas técnicas e não técnicas dentro do time de desenvolvimento do software. Criado por Dan North (2003) como resposta às limitações do Test Driven Development (TDD), o BDD tem foco na linguagem e nas interações utilizadas no processo de desenvolvimento de software. Segundo LEAL (2019), o BDD é uma derivação do *Test Driven Development* (TDD) e do *Domain Driven Design* (DDD), a qual se baseia na definição de cenários que exprimem o comportamento de uma funcionalidade do software. Estes cenários são descritos em uma linguagem de domínio específico, do inglês *domain specific language* (DSL) de alto nível, a qual é livre de termos técnicos (RITTER, 2018).

De acordo com Dan North (2003), o BDD foi pensado de maneira que a escrita dos cenários fossem simples para facilitar o entendimento, flexíveis e estruturados. Com isso, o BDD facilita a comunicação entre os *stakeholders*, que são as pessoas que estão envolvidas com o desenvolvimento do software e da garantia de qualidade (Quality Assurance - QA), e também, os

analistas de negócios, *product owner* e usuários finais. Cabe ainda ressaltar que, por ser uma especificação em linguagem estruturada, o BDD permite a automação da execução dos cenários.

A linguagem Gherkin

A maioria das ferramentas que utilizam a abordagem BDD, como o Behave e o Pytest-BDD, utilizam um formato de linguagem chamado Gherkin. Gherkin é uma linguagem de domínio específica, estruturada, que foi criada para descrição dos comportamentos. A linguagem natural utilizada pelo Gherkin facilita o entendimento por parte dos envolvidos e por ser estruturada é possível automatizar. Assim, o Gherkin permite que analistas de negócios, project owners e outros envolvidos descrevam o comportamento do software sem precisar detalhar como ele será implementado.

Um arquivo em Gherkin contém requisitos para uma funcionalidade escrita em arquivo texto com extensão '.feature'. Inicialmente o arquivo inclui a palavra-chave 'Feature', que é utilizada para descrever uma funcionalidade. Em seguida, inclui-se os cenários ou a descrição de comportamento. E cada arquivo deve incluir apenas uma funcionalidade. Observe a Figura 10.1 para um exemplo de arquivo com a funcionalidade de uma calculadora.

Figura 10.1. Funcionalidade de uma Calculadora

Fonte: do autor 2022.

Como se pode observar, pela Figura 10.1, o arquivo de requisitos utiliza-se de termos em inglês, como por exemplo *Given*, *And*, *When* e *Then* para definir um conjunto de passos para a definir um comportamento. Para detalhar melhor cada termo:

- *Feature* fornece uma descrição de alto nível nas funcionalidades do software.

- *Given* especifica uma pré-condição.
- *When* define alguma ação.
- *And* é utilizado para incluir passos adicionais junto com as etapas *Given*, *When* e *Then*.

Este é um padrão conhecido como *Given-When-Then* que é a ordem natural de um cenário, cuja tradução é Dado-Quando-Então.

O que é o pytest-bdd

O pytest-bdd é um plugin do pytest para implementar a abordagem BDD no desenvolvimento ou teste do código. O pytest-bdd utiliza a linguagem Gherkin para a escrita das funcionalidades e geração dos arquivos de testes. Para utilizar o pytest-bdd é preciso utilizar a instrução.

pip install pytest-bdd

Com o pytest-bdd instalado é possível por meio do pytest associar um arquivo feature com o teste que será executado.

Estrutura de um projeto em pytest-bdd

Para o primeiro exemplo, observe a estrutura de pastas de acordo com a Figura 10.2.

Figura 10.2. Estrutura de pastas de um projeto com pytest-bdd

Fonte: do autor 2022.

O arquivo calculadora.py tem apenas a função de soma, como se pode ver na Codificação 10.1. Fica como exercício estender o código para outras funcionalidades da calculadora.

Codificação 10.1. calculadora.py

```
def soma(a, b):
```

```
    return a + b
```

Fonte: do autor 2022.

Para o arquivo calculadora.feature, observe a Figura 10.1, que foi descrita na seção 10.2 sobre a linguagem Gherkin. O arquivo calculadora.feature descreve o comportamento da funcionalidade da calculadora com relação a soma de dois números. Observe pela especificação do requisito, que serão fornecidos dois números, 2 e 7, e deve verificar se a soma resultou em 9.

Com o arquivo calculadora.feature criado, agora será necessário gerar o arquivo de teste. Para isso utilize a seguinte instrução:

pytest-bdd generate features/calculadora.feature > tests/test_calculadora_soma.py

O pytest-bdd utiliza a palavra chave generate para ler a especificação do arquivo calculadora.feature e devolve como saída o arquivo de teste. Utiliza-se o sinal de maior (>), que é um direcionador de fluxo do sistema operacional, para gravar o arquivo test_calculadora_soma.py dentro da pasta tests. Observe o código gerado pela Codificação 10.2.

Codificação 10.2. test_calculadora_soma.py

```
"""Calculadora feature tests."""

from pytest_bdd import (
    given,
    scenario,
    then,
    when,
)


@scenario('features/calculadora.feature', 'Soma dois números')
def test_soma_dois_números():
    """Soma dois números."""


@given('eu também tenho o número 7 como entrada para a calculadora')
```

```
def
eu_também_tenho_o_número_7_como_entrada_para_a_calculadora():
    """eu também tenho o número 7 como entrada para a calculadora."""
    raise NotImplementedError


@given('eu tenho o número 2 como entrada para a calculadora')
def eu_tenho_o_número_2_como_entrada_para_a_calculadora():
    """eu tenho o número 2 como entrada para a calculadora."""
    raise NotImplementedError


@when('eu solicito que realize a soma')
def eu_solicito_que_realize_a_soma():
    """eu solicito que realize a soma."""
    raise NotImplementedError


@then('o resultado deve ser 9')
def o_resultado_deve_ser_9():
    """o resultado deve ser 9."""

    raise NotImplementedError
```

Fonte: do autor 2022.

Agora que o arquivo de teste foi gerado, basta implementar as regras utilizando a linguagem python. Neste exemplo, é preciso realizar a importação do arquivo calculadora.py e implementar as etapas especificadas pelo teste. Note que para cada passo do arquivo de requisitos, gerou-se uma função que indica o passo da regra de negócio. Observe a implementação completa do teste na Codificação 10.3.

Codificação 10.3. test_calculadora_soma.py - 2

```
"""Calculadora feature tests."""

from pytest import fixture
```

```
from pytest_bdd import (
    given,
    scenario,
    then,
    when,
)

from app.calculadora import soma

@fixture
def contexto():
    return {'num1': 0, 'num2': 0, 'resultado': 0}


@scenario('../features/calculadora.feature', 'Soma dois números')
def test_soma_dois_números():
    """Soma dois números."""


@given('eu também tenho o número 7 como entrada para a calculadora')
def
eu_também_tenho_o_número_7_como_entrada_para_a_calculadora(co
ntexto):
    """eu também tenho o número 7 como entrada para a calculadora."""
    contexto['num2'] = 7


@given('eu tenho o número 2 como entrada para a calculadora')
def
eu_tenho_o_número_2_como_entrada_para_a_calculadora(contexto):
    """eu tenho o número 2 como entrada para a calculadora."""
    contexto['num1'] = 2


@when('eu solicito que realize a soma')
```

```
def eu_solicito_que_realize_a_soma(contexto):
    """eu solicito que realize a soma."""
    contexto['resultado'] = soma(contexto['num1'], contexto['num2'])


@then('o resultado deve ser 9')
def o_resultado_deve_ser_9(contexto):
    """o resultado deve ser 9."""

    assert contexto['resultado'] == 9
```

Fonte: do autor 2022.

Como se pode perceber, foi necessário criar um contexto por meio de uma fixture. Esse contexto serve para comunicar uma função de teste com outra. A única função que inicia pelo prefixo test_ é a função test_soma_dois_números(), o qual é o ponto de entrada do teste e também o responsável por conectar o arquivo de feature com as funções que implementam os passos *given*, *when* e *then*.

Para executar o teste observe a Figura 10.3.

Figura 10.3. Executando o teste com pytest

Fonte: do autor 2022.

Agrupando vários exemplos de testes

Para o segundo exemplo, observe a estrutura de pastas de acordo com a Figura 10.4. Este exemplo é uma extensão do primeiro exemplo, portanto, segue a mesma estrutura.

Figura 10.4. Estrutura de pastas de um projeto com pytest-bdd

Fonte: do autor 2022.

Para o arquivo calculadora.feature, observe a Figura 10.5.

Figura 10.5. Arquivo de requisitos da calculadora

Fonte: do autor 2022.

As palavras *Scenario Outline* permitem agrupar cenários de maneira mais concisa, sendo possível realizar testes fornecidos como entrada pelo uso da palavra *Examples*. Os valores de exemplos serão substituídos por parâmetros delimitados por < e >. Então, um *Scenario Outline* deve necessariamente conter um ou mais seções *Examples*.

Observe que a Figura 10.5 apresenta um *Scenario Outline* para o cálculo de soma de dois números, e logo abaixo na seção Examples há um conjunto de testes de entrada e também um resultado para ser validado na função de teste.

Com o arquivo calculadora.feature criado, agora será necessário gerar o arquivo de teste. Para isso utilize a seguinte instrução:

pytest-bdd generate features/calculadora.feature > tests/test_calculadora_soma.py

Assim como apresentado anteriormente, o pytest-bdd utiliza a palavra chave generate para ler a especificação do arquivo calculadora.feature e devolve como saída o arquivo de teste. Utiliza-se o sinal de maior (>), que é um direcionador de fluxo do sistema operacional, para gravar o arquivo test_calculadora_soma.py dentro da pasta tests. Observe o código gerado pela Codificação 10.4.

Codificação 10.4. test_calculadora_soma.py - 3

```
"""features/calculadora.feature feature tests."""
from pytest_bdd import (
    given,
    scenario,
    then,
```

```
    when,
)
@scenario('features/calculadora.feature', 'Soma dois números')
def test_soma_dois_números():
    """Soma dois números."""

@given('Eu entro um número <num1> na calculadora')
def eu_entro_um_número_num1_na_calculadora():
    """Eu entro um número <num1> na calculadora."""
    raise NotImplementedError

@given('Eu também entro com o valor <num2> na calculadora')
def eu_também_entro_com_o_valor_num2_na_calculadora():
    """Eu também entro com o valor <num2> na calculadora."""
    raise NotImplementedError

@when('Eu solicito para realizar a soma.')
def eu_solicito_para_realizar_a_soma():
    """Eu solicito para realizar a soma.."""
    raise NotImplementedError

@then('Então a soma deve ser <resultado>')
def então_a_soma_deve_ser_resultado():
    """Então a soma deve ser <resultado>."""

    raise NotImplementedError
```

Fonte: do autor 2022.

O arquivo calculadora.py tem apenas a função de soma, como se pode ver na Codificação 10.5.

Codificação 10.5. calculadora.py - 2

```
def soma(a, b):
    return a + b
```

Fonte: do autor 2022.

Agora que o arquivo de teste foi gerado, basta implementar as regras utilizando a linguagem python. Neste exemplo, é preciso realizar a importação do arquivo calculadora.py e implementar as etapas especificadas pelo teste. Note que para cada passo do arquivo de requisitos, gerou-se uma função que indica o passo da regra de negócio. Observe a implementação completa do teste na Codificação 10.6.

Codificação 10.6. test_calculadora_soma.py - 4

```
"""features/calculadora.feature feature tests."""

from pytest_bdd import (
    given,
    scenario,
    then,
    when,,
    parsers,
)

from src.calculadora import soma


@scenario('..features/calculadora.feature', 'Soma dois números')
def test_soma_dois_números():
    """Soma dois números."""

@given(parsers.parse('Eu entro um número {num1:d} na calculadora'), target_fixture="contexto")
def eu_entro_um_número_num1_na_calculadora(num1):
    """Eu entro um número <num1> na calculadora."""
    return {'num1': num1, 'num2': 0, 'resultado': 0}

@given(parsers.parse('Eu também entro com o valor {num2:d} na calculadora'))
```

```
def eu_também_entro_com_o_valor_num2_na_calculadora(contexto, num2):
    """Eu também entro com o valor <num2> na calculadora."""
    contexto['num2'] = num2

@when('Eu solicito para realizar a soma.')
def eu_solicito_para_realizar_a_soma(contexto):
    """Eu solicito para realizar a soma.."""
    contexto['resultado'] = soma(contexto['num1'], contexto['num2'])

@then(parsers.parse('Então a soma deve ser {resultado:d}'))
def então_a_soma_deve_ser_resultado(contexto, resultado):
    """Então a soma deve ser <resultado>."""

    assert contexto['resultado'] == resultado
```

Fonte: do autor 2022.

Como se pode observar pela Figura 10.6, existem várias alterações realizadas para o tratamento de números inteiros. Cada parâmetro delimitado por < e > nas cláusulas @given, @when e @then foram alteradas. Por exemplo, o parâmetro **<num1>** foi substituído por **{num1:d}**. Isto significa que o parâmetro num1 é do tipo inteiro (d). Note que para o pytest entender que o parâmetro num1 é do tipo inteiro, ele é preciso ser traduzido. Para isso, é necessário utilizar a instrução parser.parser(), do pacote pytest_bdd.

Outro ponto a ser observado é que uma fixture foi criada dentro da cláusula **@given**, através da palavra reservada **target_fixture** que gerou uma variável denominada contexto que armazena um dicionário.

E por fim, a fixture contexto e os parâmetros são transmitidos como parâmetros formais nas funções de testes.

Para executar o teste observe a Figura 10.6.

Figura 10.6. Executando o teste no pytest

Fonte: do autor 2022.

Considerações finais

Esta parte apresentou conceitos de Behave Driven Development, que é uma maneira de desenvolver software utilizando requisitos de alto nível que podem ser especificados por pessoas técnicas e não técnicas envolvidas no desenvolvimento do projeto. Para isso, apresentou-se um plugin do pytest chamado pytest-bdd que é próprio para gerar o código de teste associado ao arquivo de requisitos. As vantagens de se utilizar o pytest-bdd é que os testes seguem o mesmo padrão utilizado no pytest para construção de testes. A utilização de *fixtures* é uma maneira de trocar informações entre funções de testes. E a parametrização auxilia em diversos casos de testes que tenham mesmos comportamentos.

Referências

NORTH, D. **Introducing bdd**, 2003. Versão em inglês. Disponível em: <https://dannorth.net/introducing-bdd/>. Acesso em: 31 jul. 2022.

LEAL, F. B.. **Uma abordagem usando features BDD e modelo de objetivos para o desenvolvimento ágil de software**. Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) 84 p. Orientadora: Profa. Dr. Genaina Nunes Rodrigues. Brasília: Universidade de Brasília, Instituto de Ciências Exatas, Departamento de Ciência da Computação, 2019.

RITTER, R.. **Reúso de cenários BDD para minimizar o esforço de migração de testes para a plataforma Android**. Dissertação (Mestrado). 58 p. Orientadora: Profa. Dra. Érika Fernandes Cota. Porto Alegre: Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, 2018.

PYTEST. **Documentação versão de python 3.7+**. 2015. Disponível em: <https://docs.pytest.org/en/7.1.x/index.html>. Acesso em: 11 jun. 2022.

PYTHON. **Documentação versão de python 3.10.5.** The python standard library. Development tools. unittest.mock - mock object library. Versão em inglês. Disponível em: <https://docs.python.org/3/library/unittest.mock.html>. Acesso: 08 jul. 2022.