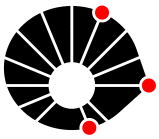

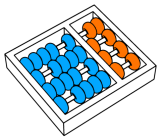

Aplicação de técnicas de testes de interface com BDD nos testes de API

F. Nicchio *L. Leal* *E. Martins*

Relatório Técnico - IC-PFG-21-20
Projeto Final de Graduação
2021 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Aplicação de técnicas de testes de interface com BDD nos testes de API

Fabiana Nicchio Lucas Leal Eliane Martins

Resumo

Este trabalho tem como objetivo comparar uma técnica de teste utilizando Behavior Driven Development (BDD) e Cucumber, com a forma como é testada uma Application Programming Interface (API) utilizada no mercado de pagamentos para levantar vantagens e desvantagem de cada uma das abordagens.

Para esta comparação, foi realizado um estudo de caso desta API. Foram implementandos os testes gerados por uma ferramenta pra gerar testes com base no pairwise, para então poder comparar o resultado da execução desses testes com os testes gerados de forma manual dessa mesma API pela empresa que a implementou.

Os resultados do trabalho apontam que ambas as abordagens possuem pontos positivos e negativos. Sendo a execução automatizada e a aplicação do BDD no processo de criação dos casos de teste os responsáveis pelas diferenças entre os métodos estudados.

1 Introdução

Métodos de teste de software são técnicas, procedimentos, padrões usados para guiar o procedimento de teste de maneira eficiente e eficaz. Dentre as técnicas temos exemplos como: teste de caixa branca ou preta, técnicas de teste estático, testes baseados em modelos de estado e outros.

Segundo Lee et al.[1] em média 24% das despesas relacionados a produção de software são voltadas pra teste e a maioria de seus entrevistados relatam ter dificuldades relacionadas às hailidades de testes e suporte de métodos e ferramentas de teste de software. A maioria dos entrevistados por Lee at al. [1] dizem ter interesse em testes automatizados, porém não possuem nenhuma estrutura ou planejamento para implementá-los. Trabalhos mais recentes [2] apontam que a indústria de software já adota algumas técnicas de automação de excução de testes, entretanto os processos de design e planejamento muitas vezes ainda são feitos de maneira manual.

Métodos já foram propostos para resolver alguns dos problemas relacionados ao design e planejamento de testes. Entre eles, podemos citar o Test Driven Development (TDD) que tem como objetivo criar testes unitário para alcançar 100% de cobertura de código, Behavior-Driven Development (BDD) que tem como objetivo horizontalizar a documentação do projeto e descrever cenários de modo a guiar o desenvolvimento da aplicação, assim como servir como referência para criação testes.

Neste estudo de caso foi comparado uma técnica de testes, pairwise, usando BDD e Cucumber (para automatizar a executar os testes), com a ténica que foram criados e executados os testes feitos pela equipe de qualidade da Evoluservices usados para validar um serviço para o uso em produção. Ambas as técnicas foram avaliadas e executadas na API de Link de Pagamento da empresa. Esta comparação envidencia pontos fortes e fracos de cada uma das abordagens. Com esses pontos levantados, é possível identificar melhorias em cada uma formas de validar a API e assim poder melhorar o processo de qualidade de software.

Esse relatório esá organizado da seguinte forma: (2) Fundamentação teórica, onde técnicas e ferramentas utilizadas no experimento são descritas com mais detalhes; (3) Descrição do estudo de caso e procedimento; (4) Resultados e (5) Conclusões.

1.1 Ferramentas utilizadas

Para a realização deste projeto, foram utilizadas algumas ferramentas que serão explicadas no decorrer do relatório. São elas:

- Swagger-Codegen
- Cucumber
- JUnit
- Allpairs (Satisfice)

2 Fundamentação

Na realização deste projeto, alguns conceitos e ferramentas foram utilizados como base. Alguns deles, como BDD, Cucumber e outras ferramentas são de importante compreensão para o entendimento por completo do projeto. Também é explicada a técnica de testes utilizada neste projeto.

2.1 Test Driven Development (TDD)

TDD é uma técnica de desenvolvimento inventada por Kent Beck no final dos anos 90 assim que surgiram as metodologias ágeis. É uma técnica muito efetiva que usa testes unitários para especificar, desenhar e verificar aplicações pelo seu código.

Quando os usuários de TDD implementam uma funcionalidade, eles começam implementando os testes que descrevem a funcionalidade, que consequentemente irá falhar por não ter sido implementado nada dela. Com o teste falhando, é implementado a parte da funcionalidade que fará o teste passar. Após isso, o código é refatorado de modo que facilite sua manutenção[6].

Podemos utilizar um exemplo da aplicação do TDD para entender melhor sua execução. Num sistema onde deverá ser implementada uma função de validar um CPF, antes dela ser implementada, deverá ser escrito um teste que irá chamar a função em questão com um

CPF válido. O teste passará se a função retornar que o CPF é válido. Como o teste foi criado antes da implementação da função de validar CPF, o teste irá falhar.

Ao implementar a função que valida o CPF e executar o teste novamente, ele deverá passar. Para garantir que a função foi devidamente implementada, deverá ser criado um segundo teste que chama a função em questão com um caso de CPF inválido. Ao executar o teste, ele irá falhar e portanto, a função deverá ser alterada para que cubra o caso em questão. Feito isso, ao executar novamente o teste, ele irá passar.

Esse ciclo de interações entre testes e implementação da função deverá ser feito repetidamente até que não tenha mais casos para que sejam implementados testes.

2.2 Behavior-Driven Development (BDD)

O BDD é uma de técnica de Engenharia de Software desenvolvidas para auxiliar times a desenvolverem e entregar software com mais qualidade, mais valor e mais rápido[4].

Ele foi proposto em 2006 por Dan North com o intuito de melhorar o TDD, por isso o Behavior Driven Development também possui uma abordagem inicial de testes, mas difere por testar o comportamento do sistema da perspectiva do usuário final[5].

O objetivo do BDD é desenvolver o software certo da maneira correta. Ou seja, parte do objetivo do BDD é desenvolver o software com qualidade, bem estruturado, de fácil manutenibilidade, adaptável a novas mudanças e com a menor quantidade possível de bugs. E outra parte do objetivo é produzir um software devidamente alinhado com a estratégia de negócio, que seja benéfico para a corporação e deve ajudar os usuários a atingir seus objetivos de forma mais efetiva[6].

North sugere que a linguagem utilizada no BDD possa ser extraída das especificações fornecidas pelo cliente durante o levantamento dos requisitos. Assim, deve ser estruturado o suficiente de modo a quebrar as histórias em fragmentos[3].

O BDD é uma prática colaborativa, tanto usuários do sistema quanto os desenvolvedores conversam sobre as features desejadas para que assim, as ideias sejam criadas/desenvolvidas da especificação da feature, levando em consideração o conhecimento e experiência do usuário final[6]. Dessa conversa, surgem os cenários que os usuários esperam da feature.

Esses cenários são descritos em linguagem natural com uma sintaxe denominada por Gherkin, que falaremos mais adiante.

De forma semelhante ao TDD, o BDD garante os casos que devem ser levados em consideração ao implementar o sistema. A diferença entre as duas abordagens é qual é o foco da implementação. Enquanto no TDD o foco é a função que será implementada, no BDD o foco é como o usuário irá interagir com o sistema.

Apesar de cada um dos métodos terem abordagens diferentes, eles são complementares dado que o TDD é indicado para Testes de Unidade e Integração, enquanto o BDD é indicado para os Testes de Aceitação[8].

2.3 Gherkin

Gherkin é uma linguagem natural utilizada para descrever os cenários em BDD. Cada linha deve ser iniciada com uma de suas palavras chaves. As principais palavras chaves são **Dado**

que, **Quando**, **Então** e **E**.

Cada uma dessas palavras precede uma parte dos cenários. O **Dado que** se refere precedido quando for descrito a pré condição do cenário. O **Quando**, a interação que será feita, assim como a entrada do cenário. O **Então**, o resultado esperado da interação dada a pré-condição descrita[7]. E por fim, o **E** sempre que houver mais de cada uma das palavras chaves para que a leitura fique o mais próxima possível da linguagem natural. A estrutura que os cenários seguem com essas palavras chaves garantem que o cenário será descrito de forma coerente e entendível. Utilizando o exemplo da validação de CPF:

Cenário: Validar CPF válido do usuário
Dado que eu insiro um CPF válido
Quando prossigo com o cadastro
Então o sistema aceita meu CPF

Cenário: Validar CPF inválido do usuário
Dado que eu insiro um CPF inválido
Quando prossigo com o cadastro
Então o sistema informa que o CPF é inválido

A descrição dos cenários utilizando essas palavras chaves, faz com que os cenários fiquem numa linguagem mais simples e de fácil compreensão para o leitor, principalmente para aqueles que não entendem da parte técnica do sistema em desenvolvimento[9], permitindo que mais pessoas interessadas no sistema possam interagir na fase de construção do mesmo.

2.4 Cucumber

Cucumber é uma ferramenta que auxilia na escrita de testes automatizados utilizando BDD[10]. Os cenários escritos em Gherkin podem ser entendidos em forma de código para os testes automatizados assim como gerar relatórios das execuções[11].

Cada linha de cada cenário, iniciada por uma palavra chave, até seu fim antes da próxima palavra chave, é chamada de definição de passo (step definition, como é mais conhecida). Cada *step definition* é uma interação no sistema descrita em forma de código para que seja executada de acordo com a descrição do passo[11]. Na seção 3.2 será dado um exemplo que permitir entender melhor o Cucumber.

2.5 Outras ferramentas utilizadas

OpenAPI [1] É uma forma de documentação da especificação de API REST. Arquivos que contêm especificações OpenAPI são escritos em json ou yaml e possuem informações relevantes para quem for utilizar a API em questão, como possíveis respostas para cada requisição, formato dos parâmetros, url da requisição, entre outros.

[1] Mais informações em https://swagger.io/docs/specification/about/

Swagger Codegen [2] Uma das ferramentas utilizadas foi o Swagger Codegen que gera SDKs para API de acordo com a especificação OpenAPI usada como entrada. O SDK gerado é capaz de lidar com as requisições, sendo necessário apenas passar os valores dos parâmetros. Sendo assim, o usuário não precisa se preocupar em saber qual o método deve ser usado, qual a url que deve será feita a requisição e assim por diante.

JUnit [3] É um framework para escrever testes em Java. Com ele, é possível verificar saídas e comparar se está dentro do esperado ou não, com respostas rápidas e precisas. O JUnit possui diversas outras funcionalidades, como realizar testes de unidade, apresentação de resultados, entre outros. Porém usaremos apenas os métodos que comparam resultados com o que é esperado. O JUnit é uma ferramenta opensource com uma grande comunidade e suporte.

Allpairs [4] É uma ferramenta que gera casos de testes de acordo com sua entrada, composta pelos parâmetros e valores possíveis para cada parâmetro, utilizando a técnica de teste pairwise. A ferramenta é executada através de linha de comando tendo como entrada um arquivo txt com os parâmetros e seus possíveis valores e tem como saída um planilha xls com os casos de testes gerados.

2.6 Técnicas de teste

Antes de iniciarmos a apresentação de técnicas de teste é importante que os leitores sejam apresentados a alguns conceitos e definições que são comuns a comunidade de teste de software. Esses conceitos aparecerão inúmeras vezes ao longo do texto, não só nessa sessão, e sem o seu devido entendimento a compreensão do texto será dificultada. Para entender de forma clara será dito, é interessante revisar alguns conceitos:

2.6.1 Conceitos

Verificação Consiste no processo que determina se o software implementou as especificações previamente estabelecidas para ele[14].

Validação É o processo que avalia se o software implementado está de acordo com o uso que será feito dele[14].

Teste de Software É o processo de executar o programa alvo com o objetivo de garantir a qualidade do mesmo[12]. É realizada uma verificação do comportamento do programa com um conjuntos de testes de acordo com as especificações do programa[13].

Domínio de entrada É definido por todas as possibilidades de valores que são possíveis de ser usados na entrada do programa que será testado[14].

[2]Mais informações em https://swagger.io/tools/swagger-codegen/
[3]Mais informações em https://junit.org/junit5/
[4]Mais informações em https://www.satisfice.com/download/allpairs

Bloco Ao analisar as possibilidades de entradas de um programa, é possível dividir o domínio de entrada em blocos de entradas, de tal forma que cada bloco contém um conjunto de valores para teste[14].

Característica Cada bloco definido no domínio de entrada possui valores que podem ser agrupados por características que possuem em comum[14].

2.6.2 Técnicas

As técnicas de testes são diferentes procedimentos utilizados para selecionar um conjunto de testes que serão executados na aplicação alvo. Algumas dessas técnicas serão apresentadas aseguir, utilizaremos um exemplo de uma SCC (Sistema de Controle de Caldeira) para facilitar a explicação. O SCC tem os seguintes comandos:

- temp: mudar a temperatura da caldeira.
- desl: desliga a caldeira.
- cncl: cancela o comando.

No caso de temp, o sistema pede ao operador que indique o incremento de temperatura que deseja. O valor do incremento está no intervalo [-10, 10] e só são permitidos incrementos de 5 graus. O incremento não pode ser igual a 0. O operador pode ainda escolher se deseja entrar com os comandos via GUI ou via arquivo. Caso os comandos estejam em um arquivo, o operador fornece como entrada o nome do arquivo.

ACoC - All Combinations Coverage O ACoC é uma técnica de teste que tem como objetivo gerar casos de testes que são formados por todas as combinações possíveis de blocos de valores de acordo com os parâmetros de entrada[15].

No nosso exemplo da SCC, temos 5 parâmetros de entrada de modo que cada parâmetro tem as seguintes quantidades de blocos de valores:

- temp: 5
- dels: 2
- cncl: 2
- entrada: 2
- nome_arq: 2

Assim sendo, podemos calcular a quantidade de combinações de entrada:

C= 5 * 2 * 2 * 2*2 = 80

Portanto, a ACoC nos fornece um total de 80 testes para serem executados.

Pairwise Nesta técnica, há dois pontos importantes, onde todos os pares dos 2 fatores mais importantes são levados em consideração e alguns testes com fatores não tão importantes são selecionados para cobrir os casos em que eles não foram levados em consideração em função desses 2 fatores mais importantes previamente selecionados[16].

Para exemplificar essa técnica, foram levantados valores possíveis para cada bloco de cada parâmetro de entrada do SCC, com suas respectivas restrições. Um exemplo de restrição é o caso da entrada ser feita via GUI e o nome do arquivo for inválido, neste caso o nome do arquivo será ignorado. Essas informações foram inseridas na ferramenta que gera os casos de teste automaticamente, disponível em https://sqamate.com/tools/pairwise, baseia na técnica pairwise.

casos	temp	desl	cncl	entrada	nome_arq
1	-10	S	S	GUI	
2	-5	N	N	arq	válido
3	5	N	S	arq	inválido
4	10	S	N	GUI	
5	0	S	N	arq	inválido
6	0	N	S	GUI	
7	10	N	S	arq	inválido
8	5	S	N	GUI	
9	-5	S	S	GUI	
10	-10	N	N	arq	inválido

Tabela 1: Tabela com os casos gerados de acordo com o Pairwise

BCC - Base Choice Coverage O primeiro passo para o BCC ser aplicado é definir qual é o caso de teste base. Esse caso de teste base é definido com os valores que respeitem os critérios previamente definidos, podendo ser valores mais simples, mais usados, menores, etc. Esse critério vai ser definido de acordo com que for considerado mais importante para os testes. Feito isso, cada parâmetro do seu teste base deve ser alterado com outros valores, um parâmetro de cada vez. Após alterar um parâmetro, ele deve voltar para o valor do seu teste base, o próximo parâmetro deve ser alterado e assim por diante[17].

É possível ver claramente o uso do BCC utilizando o exemplo da SCC. Os valores são variados a cada parâmetro do teste base, com um parâmetro sendo alterado por cada caso de teste.

ECC - Each Choice Coverage A ideia do ECC é que deve ser usado um valor de cada bloco em pelo menos um teste no seu conjunto de testes[18]. Os casos de teste são gerados pela combinação dos valores não utilizados de cada parâmetro, caso não tenha mais nenhum valor não utilizado para algum dos parâmetros, valores já utilizados podem ser repetidos[18]. Essa técnica foi proposta por Ammann and Offutt[14] que também sugerem não ser uma técnica muito desejável por gerar testes que não levam em consideração a informação semântica de cada parâmetro.

casos	temp	desl	cncl	entrada	nome_arq
1	-10	N	N	arq	válido
2	-10	N	N	arq	inválido
3	-10	N	N	GUI	
4	-10	N	S	arq	inválido
5	-10	S	N	arq	inválido
6	5	N	N	arq	inválido
7	0	N	N	arq	inválido

Tabela 2: Tabela com os casos gerados de acordo com o BCC

No exemplo da SCC, podemos observar que os valores de cada parâmetro são utilizados de forma que cada bloco de cada parâmetro é representado nos casos e quando algum parâmetro já tenha utilizado valores de todos os blocos, eles são repetidos.

casos	temp	desl	cncl	entrada	nome_arq
1	-10	S	S	GUI	
2	-5	N	N	arq	válido
3	5	S	S	GUI	
4	-10	N	S	arq	inválido
5	0	N	N	GUI	

Tabela 3: Tabela com os casos gerados de acordo com o ECC

3 Descrição do estudo empírico

3.1 Objetivos

O objetivo dos testes que foram implementados e executados é poder comparar os resultados dos testes realizados dentro de uma empresa, sem o uso de uma técnica de teste formal com um método utilizando o BDD aplicando a técnica de testes pairwise e usando o Cucumber.

O estudo de caso foi feito com uma API utilizada no mercado para facilitar pagamentos de produtos e serviços por meio de e-commerce. A API em questão permite que parceiros façam consultas das possíveis formas de criar pagamentos para os clientes finais.

As duas formas de testar a API em questão tiveram resultados diferentes, deixando o uso do pairwise em desvantagem com relação à cobertura dos testes.

3.2 Estudo de caso

O estudo de caso que é a API de Link de Pagamentos da empresa Evoluservices Meios de Pagamento Ltda.

A Evoluservices é uma empresa subadquirente (que faz o intermédio entre estabelecimentos e grandes adquirentes, como Rede e Cielo) de grande porte, de aproximadamente

130 funcionários, que está há 18 anos no mercado trabalhando com diferentes meios de pagamento, principalmente pagamentos com cartão de crédito.

Seu setor de TI é focado em criar novos produtos que permitam pagamentos com cartão de crédito e criar novas formas para facilitar o pagamento dos valores referentes às transações feitas com cartão de crédito. Sua base de código é escrita principalmente em Java e está estruturada em um monolito com alguns microsserviços auxiliando em algumas das atividades.

Um de seus produtos é chamado de link de pagamento e é uma maneira de estabelecimentos comerciais poderem enviar cobranças de valores a seus clientes referente a produtos e/ou serviços prestados, de forma que os mesmos possam pagar essas cobranças por meios digitais com cartão de crédito.

A API do link de pagamento permite que estabelecimentos integrados ao sistema da Evoluservices, possam criar os links e enviar para seus clientes através de seus próprios sistemas. Esta API está em produção de maneira estável e é utilizada por diversos estabelecimentos parceiros desde o fim de 2018.

Para que seja possível criar este link, o estabelecimento precisa consultar as formas possíveis de pagamento de acordo com o valor desejado, para poder criar o link com as opções disponíveis respeitando as condições de seu contrato.

Para nosso estudo de caso, foi utilizada a requisição que faz esta consulta dos métodos disponíveis de pagamento. A API possui uma autenticação do tipo Basic Auth, onde o usuário e senha que serão inseridos no header da requisição, e na requisição em questão, exige que seja colocado em seus parâmetros o código do estabelecimento que irá realizar a cobrança e o valor dessa cobrança. A saída de sucesso mostrará as bandeiras disponibilizadas para aquele valor, assim como o número máximo de parcelas permitidas.

Dentro do setor de TI, existe um time de qualidade responsável por toda a documentação, criação e execução dos testes nas funcionalidades que são implementadas. Esta equipe não segue uma técnica específica de teste e não utiliza o BDD em todos as áreas do sistema. Ela leva em consideração as regras de negócio que estão envolvidas na funcionalidade e avaliam cada parâmetro de forma independente. A forma em que geram os testes manuais se assemelha muito ao BCC e ao ECC, explicadas na seção 2.3.

Para analises futuras, um exemplo de teste criado pela Evoluservices para o caso de resposta de sucesso:
Título: Consulta válida
Ação: Inserir valor válido
Ação: Inserir merchantCode válido
Ação: Enviar a requisição
Resultado: Deve onter como retorno as seguintes informações:
Tipo : Crédito ou Recorrente
Opções de crédito: máximo de parcelas, bandeiras
Opções de recorrente: máximo de parcelas = 1, bandeiras
Resultado: As informações retornadas devem refletir o plano do estabelecimento em questão

Pelo exemplo, é claro como as regras de negócio são a base dos testes. Detalhes da resposta da requisição como o status, não é considerado no caso. Porém, é considerado

como é formada essa resposta e as possibilidades de variação da resposta de acordo com regras de negócio que vão muito além da da especificação OpenAPI disponibilizada.

3.3 Procedimento

Para a realização do projeto, foi necessário seguir alguns passos para configurar e implementar os testes e ter então, uma ferramenta pronta para ser usada nos testes.

O diagrama abaixo mostra o passo a passo que será explicado em seguida.

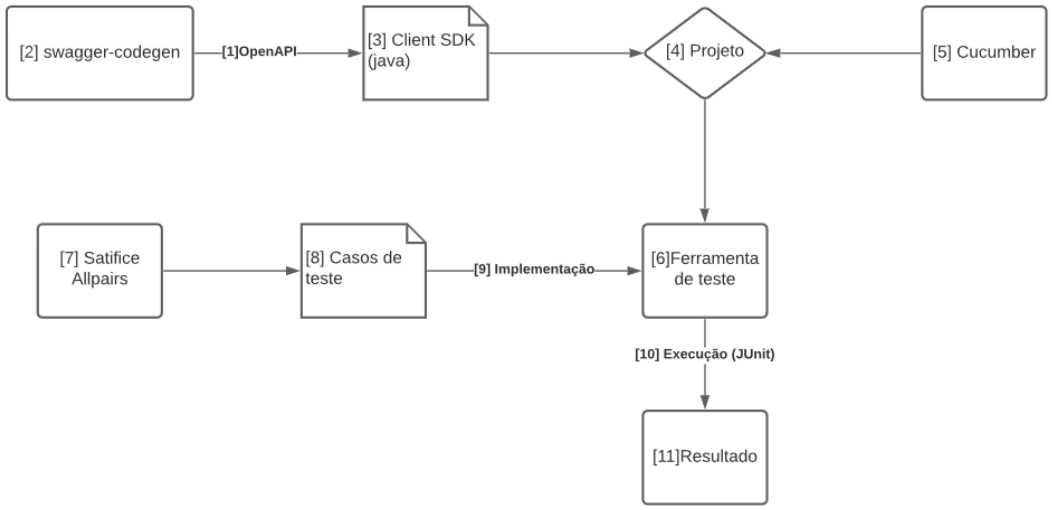

Figura 1: Passo a passo do procedimento realizado.

O primeiro passo foi conseguir a especificação da API em OpenAPI. A Evoluservices fornece para seus parceiros integradores o arquivo yaml com sua especificação OpenAPI para facilitar a integração dos parceiros com a API de link de pagamentos. É possível encontrar esse arquivo em sua documentação[5].

Com a especificação em mãos, foi utilizada a ferramenta Swagger Codegen, assinalado como [2] no esquema (foi utilizada a versão 3.0.28 do Swagger Codegen). A especificação foi utilizada como entrada desta ferramenta, que gerou um projeto em Java com o Client SDK da API especificada, representado em [3] na figura.

Este projeto gerado permite que sejam realizadas requisições para a API de Link de Pagamento de forma que seja necessário apenas acrescentar os parâmetros nos objetos DTO de entrada. Ou seja, não é preciso montar a requisição desde o começo, já sabendo qual método (POST, GET, DELETE) deve ser utilizado para cada tipo de requisição, qual é a autenticação que é usada na API e assim por diante.

Este projeto será a base de tudo o que será feito, como mostrado em [4].

[5]https://evoluservices.github.io/evoluservices-docs-apis

Com essa base feita, foi configurado o framework Cucumber, em [5], para realizar a execução dos cenários de testes. Ele foi configurado no projeto através do gradle com a versão mais recente disponível (versão 6.10.4).

Com o client SDK e o Cucumber configurado, o projeto ficou pronto para iniciar a escrita e implementação dos nossos cenários de aceite. Ou seja, temos uma ferramenta pronta para iniciar os testes.

Antes de iniciar a escrita e implementação dos casos, eles foram gerados de acordo com a técnica pairwise utilizando a ferramenta Allpairs em [7].

Para gerar os casos de teste, foram considerados os parâmetros que seria possível serem alterados de acordo com que o nosso client SDK permite. Ou seja, usuário, senha, valor e o código do estabelecimento. Para cada um desses parâmetros, foram selecionados valores que representem cada um dos possíveis blocos dos respectivos parâmetros. Sendo eles:

PASSW	AMOUNT	CODE
teste	1	6783WK
123	1000	2222LK
	asd	1

Tabela 4: Tabela com os parâmetros e os valores possíveis em cada um dos parâmetros

Em PASSW, estão representado uma senha correta e uma incorreta para o parceiro que irá realizar a requisição utilizando a autenticação da API. A senha correta será representada como *teste* e a incorreta como *123*. O domínio deste parâmetro entrada permite qualquer quantidade de caracteres e quaisquer tipos de caracteres.

Para AMOUNT, foram considerados 3 valores de tal forma que um valor é válido ('1000' centavos), um valor é inválido ('1' centavo, por ser um valor muito pequeno para realizar uma transação) e um valor incorreto ('asd', por não ser possível inserir valores diferentes de números). O domínio do AMOUNT permite apenas números e que sejam maiores que 1, o que confirma a escolha de valores para este parâmetro.

Com relação ao CODE, é o código do estabelecimento que irá criar o link de pagamento, que pode será válido ('6783WK'), inválido ('2222LK') ou incorreto ('1'). O parâmetro CODE é o que possui maiores restrições, ele é formado por 6 caracteres, sendo os 4 primeiros números, e os 2 últimos letras. No caso, '2222LK' mostra um CODE que obedece o padrão, porém é inexistente e o '1' mostra um CODE que é incorreto.

Com essas informações, Allpairs gerou os casos abaixo, que estão representados por [8] no esquema da figura 1.

Os valores da Tabela 2 com '~' na frente são valores que são indiferentes de quais serão usados no parâmetro no cenário em questão. Isso se dá por que os pares possíveis para serem feitos com os outros parâmetros já foram todos apresentados nos casos anteriores.

Para cada caso, foi criado um cenário de aceitação em Gherkin. Para facilitar o entendimento, iremos mostrar um cenário implementado. Então temos nosso caso de teste seguindo o padrão do Gherkin:
Scenario: Verificar os métodos de pagamento com sucesso
Given eu tenho um parceiro valido

case	PASSW	AMOUNT	CODE
1	teste	1	6783WK
2	123	1000	2222LK
3	teste	asd	1
4	123	1	2222LK
5	teste	1000	6783WK
6	123	asd	6783WK
7	123	1	1
8	teste	1000	2222LK
9	˜teste	asd	2222LK
10	˜123	1000	1

Tabela 5: Tabela com os casos gerados pelo Allpairs

And uma senha valida
And um valor valido
And um merchantCode valido
When eu envio a requisicao com esses parametros
Then eu tenho meus metodos de pagamento

Para cada um dos passos do teste, foi implementado um método que será executado, fazendo o que o passo se propõe, representado por [9] no esquema da figura 1.

Listing 1: Implementação para a execução do teste descrito

```
@Given("eu tenho um parceiro valido")
public void eu_tenho_um_parceiro_valido() {
    call.getApiClient().setUsername(USUARIO);
}

@Given("uma senha invalida")
public void uma_senha_invalida() {
    call.getApiClient().setPassword(SENHA);
}

@Given("um valor valido")
public void um_valor_valido() {
    input.setAmount("10000");
}

@Given("um merchantCode valido")
public void um_merchant_code_valido() {
    input.setMerchantCode("6873WK");
}
```

```
@When("eu envio a requisicao com esses parametros")
public void eu_envio_a_requisicao_com_esses_parametros() {
    try {
        resposta = call.paymentMethodsCall(input.getAmount(),
              input.getMerchantCode(), progList, progReq).execute();
    } catch (ApiException | IOException e) {
        e.printStackTrace();
    }
}

@Then("eu tenho os metodos de pagamento na resposta")
public void eu_tenho_os_metodos_de_pagamento_na_resposta()
                          throws JsonProcessingException {
    try {
        System.out.println(resposta.body().string());
    } catch (IOException e) {
        e.printStackTrace();
    }
    assertThat("Metodos de pagamento nao disponibilizados",
                          resposta.code(), is(200));
}
```

Com o cenário implementado (visto em [10]), basta o cenário ser executado para termos o resultado do mesmo. Durante a execução, é utilizado JUnit para fazer a verificação do que foi obtido de resposta das requisições com o que é esperado para cada requisição.

Em [11], teremos o resultado da execução de todos os cenários implementados.Todos os cenários passaram como esperado de uma API que está há um tempo no mercado sendo usada sem problemas pelos parceiros.

4 Resultados

4.1 Testes existentes

Para analisar os resultados, foram feitas comparações com o número de possíveis respostas da requisição e quantidade de casos de testes criados. Não será possível comparar problemas encontrados na API já que nenhum foi encontrado na realização deste projeto.

No desenvolvimento da API em questão, foram criados 8 testes oficiais que se enquadram no escopo deste projeto, além de testes exploratórios realizados na época pela equipe de qualidade da Evoluservices. Tais testes oficiais foram feitos manualmente e adicionados em um gerenciador de testes como uma das formas de documentação da API que existem.

Para a realização deste projeto, a ferramenta Allpairs gerou 10 casos que só conseguiram reproduzir 5 das 7 possíveis saídas da API para a requisição testada.

4.2 Discussão dos resultados

O pairwise não permitiu que fossem testadas todas as possibilidades de saída da requisição. Desta forma, se fosse só utilizada a técnica em questão, não seria possível garantir que o escopo testado da API seria validada por completo.

O pairwise não leva em consideração o valor da informação de cada parâmetro e o fato das regras de negócio não terem sido consideradas para gerar o casos de teste, fizeram com que o pairwise ficasse em desvantagem para validar a requisição.

Num caso como este, as regras de negócio ditam a forma como o sistema deve funcionar e ignorar alguma dessas regras na verificação pode ser prejudicial para o resultado da verificação, o que foi o que aconteceu. Alguns casos não foram contemplados no pairwise, casos estes que o conhecimento das regras de negócio levaria em conta neste momento de verificação.

4.3 Ameaças à validade ao estudo de caso

Para realizar o estudo de caso deste trabalho, foi realizado uma revisão bibliográfica de técnicas de teste, assim como de fundamentos importantes como o BDD. Porém, próximo ao fim do trabalho, foi identificado pontos que poderiam ter sido revisados com uma revisão mais profunda de teoria de testes.

Uma profunda revisão sistemática poderia ter alertado sobre os pontos de falhas no início da execução, podendo ser possível alterar a estratégia deste estudo de caso. Os atuais resultados não são invalidados por isso, porém poderiam ser mais precisos e elaborados.

5 Conclusões

O uso do pairwise mostrou não ser uma técnica vantajosa para o teste da requisição comparado à forma que já são criado os testes na Evoluservices. Como não obtivemos todas as respostas possíveis pela API, não é possível afirmar que esses testes garantem a cobertura das respostas da API.

O fato do pairwise não levar em conta as regras de negócio da API, a deixa em desvantagem. Os testes realizados pela Evoluservices tem uma forte dependência com as regras de negócio e isso influencia muito na criação dos testes, consequentemente, nas possibilidades de respostas que a API pode oferecer. Este ponto fortalece o argumento de Bach et al.[19] de que pairwise não é uma boa técnica de teste, pois ele não leva em consideração as interações entre as variáveis.

Por outro lado, o uso do BDD garante a padronização dos casos, assim como casos escritos de forma mais clara e legível. Essa padronização garante uma documentação mais organizada e mais compreensível para outros integrantes do processo de desenvolvimento que não sejam da área técnica, que não sejam da área técnica, que queiram participar do processo como um todo.

Os testes realizados pela equipe de qualidade para esta API não seguem um padrão tão bem definido como o BDD, o que os deixa em desvantagem. Alguns testes, como o caso apresentado na explicação do estudo de caso, possuem mais de um cenário num mesmo

teste. O uso de BDD seria vantajoso para casos como esse, deixando explícitos os cenários e evitando execução confusa e incorreta do caso.

Um segundo ponto que o estudo de caso leva vantagem nessa comparação, é que os casos automatizados com o Cucumber permitem que mais testes possam ser executados comparado ao tempo que os testes manuais levam. Não pode ser desconsiderado o tempo gasto para configurar a ferramenta, assim como o tempo gasto para implementar os testes em questão. Porém, estes mesmos testes podem ser executados diversas vezes depois da configuração e implementação do projeto, sem a necessidade de um colaborador dedicado nisso, sendo possível ter retorno deste tempo empregado na configuração e implementação.

Devido ao tempo, algumas ideias não foram possíveis de ser realizadas e que poderiam ter grande valor para o trabalho feito. A técnica de teste utilizada poderia ter sido usada em todas as requisições da API para ter dados mais completos da API como um todo e assim ter um escopo mais bem definido. Também seria interessante comparar outras técnicas de teste, como por exemplo a Each Choice Coverage (ECC) e a Base Choice Coverage (BCC) que possuem uma proximidade maior com a forma como são criados os testes na Evoluservices e podem gerar resultados interessantes.

Referências

[1] Lee, Jihyun, Sungwon Kang, and Danhyung Lee. "Survey on software testing practices."IET software 6.3 (2012): 275-282.

[2] Hynninen, Timo, et al. "Software testing: Survey of the industry practices."2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, 2018.

[3] Anderle, Angelita. "Introdução de BDD (Behavior Driven Development) como melhoria de processo no desenvolvimento ágil de software."(2015).

[4] Irshad, Mohsin, Ricardo Britto, and Kai Petersen. "Adapting Behavior Driven Development (BDD) for large-scale software systems."Journal of Systems and Software 177 (2021): 110944.

[5] Kfouri, Tiago de Oliveira. "Análise de projetos no GitHub que utilizam Behavior Driven Development."(2019).

[6] Smart, John. BDD in Action: Behavior-driven development for the whole software lifecycle. Simon and Schuster, 2014.

[7] Fischer, Tomas, and Dana Dghyam. "Formal model validation through acceptance tests."International Conference on Reliability, Safety, and Security of Railway Systems. Springer, Cham, 2019.

[8] Rocha, Fabio G., et al. "Agile Teaching Practices: Using TDD and BDD in Software Development Teaching."Proceedings of the XXXIII Brazilian Symposium on Software Engineering. 2019.

[9] da Silva, José Pedro, and Sousa Borges. "Live Acceptance Testing using Behavior Driven Development."(2020).

[10] Nečas, Ivan. BDD as a specification and QA instrument. Diss. Masaryk University, Faculty of Informatics, 2011.

[11] Fischer, Tomas, and Dana Dghyam. "Formal model validation through acceptance tests."International Conference on Reliability, Safety, and Security of Railway Systems. Springer, Cham, 2019.

[12] Pan, Jiantao. "Software testing."Dependable Embedded Systems 5 (1999): 2006.

[13] Bertolino, Antonia, and Eda Marchetti. "A brief essay on software testing."Software Engineering, 3rd edn. Development process 1 (2005): 393-411.

[14] Ammann, Paul, and Jeff Offutt. Introduction to software testing. Cambridge University Press, 2016.

[15] Khurshid, Sarfraz, and Darko Marinov. "Reducing Combinatorial Testing Requirements Based on Equivalences with Respect to the Code Under Test."SQAMIA 2018 (2018).

[16] Burroughs, Kirk, Aridaman Jain, and Robert L. Erickson. "Improved quality of protocol testing through techniques of experimental design."Proceedings of ICC/SUPERCOMM'94-1994 International Conference on Communications. IEEE, 1994.

[17] Grindal, Mats, Jeff Offutt, and Sten F. Andler. "Combination testing strategies: a survey."Software Testing, Verification and Reliability 15.3 (2005): 167-199.

[18] Grindal, Mats. Evaluation of Combination Strategies for Practical Testing. Vol. 62. Technical Report HS-IKI-TR-04-003, 2004.

[19] Bach, James, and Patrick J. Schroeder. "Pairwise testing: A best practice that isn't."Proceedings of 22nd Pacific Northwest Software Quality Conference. 2004.