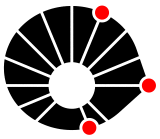
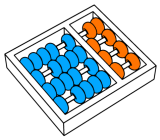




Avaliação de método combinando testes baseados em modelos e testes dirigidos pelo comportamento para APIs

T. A. Bispo *E. Martins*

Relatório Técnico - IC-PFG-23-14
Projeto Final de Graduação
2023 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Avaliação de método combinando testes baseados em modelos e testes dirigidos pelo comportamento para APIs

Thais Araujo Bispo* Eliane Martins†

Resumo

APIs (Application Programming Interface – API) oferecem um conjunto de operações que permitem a criação de aplicações e a comunicação entre serviços (sistemas). Os testes em APIs são essenciais para garantir o bom funcionamento e desempenho de aplicações. Neste projeto, os testes de API combinam o Teste Baseado em Modelos (Model Based Testing – MBT) e Testes Dirigidos pelo Comportamento (Behavior Driven Testing – BDT) para automatizar não só a execução, mas também a geração dos casos de teste. O MBT é um método para elaboração de testes que possui como premissa a modelagem do sistema, enquanto BDT possui como premissa a elaboração de casos de teste com base em cenários que representam o comportamento do sistema. Para avaliar a efetividade da combinação desses métodos, foi utilizado como estudo de caso uma API Rest.

1 Introdução

1.1 Motivação

Testes são essenciais para garantir o funcionando e desempenho dos sistemas, a fim de diminuir defeitos e falhas e facilitar a manutenção do sistema. Projetos de softwares sofrem com mudanças que devem ser refletidas nos cenários de testes realizados nos sistemas. A motivação desse projeto é ajudar no processo de elaboração e automatização de testes em APIs, em especial nos processos que há iterações (versionamento) das aplicações que afetam os cenários de testes.

1.2 Objetivo

Esse projeto pretende avaliar a combinação dos métodos Teste Baseado em Modelos (Model Based Testing – MBT) e Testes Dirigidos pelo Comportamento (Behavior Driven Testing – BDT) com base em critérios (cobertura de testes) para definir se a metodologia é efetiva para testes em APIs.

*Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 13083-852
†Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 13083-852

1.3 Metodologia

A metodologia utilizada nesse projeto foi a pesquisa em ação, o objetivo desse projeto é promover conhecimento e também a colaboração entre pesquisadores. As etapas do projeto são:

(A) Diagnosticar: Definir o tema e a proposta do relatório.

(B) Planejar: Estabelecer as etapas que irão compor o trabalho proposto, as ferramentas que serão usadas para qual objetivo com base no tema proposto.

(C) Executar: Realizar as etapas planejadas a fim de obter dados necessários para elaboração do resultado.

(D) Avaliar: Analisar os resultados atingidos e avaliar o impacto no tema proposto.

1.4 API Rest

API significa Interface de Programação de Aplicação (em inglês, Application Programming Interface) e é um conjunto de operações que permite a comunicação entre sistemas. A comunicação ocorre de forma transparente, ou seja, não é preciso conhecer a implementação das aplicações. Para que ocorra a comunicação com as APIs, existem protocolos cujo objetivo é a padronização das trocas de informações. Os protocolos mais comuns são: Rest (Representational State Transfer), SOAP (Simple Object Access Protocol) e RPC (Remote Procedure Call) [4]. Neste projeto, o objeto de estudo foi uma API Rest. Rest pode ser entendido como um estilo arquitetural (cliente-servidor) que atender critérios:

- **Arquitetura cliente-servidor**: protocolo utilizado é o HTTP e envolve cliente, servidor e recursos.
- **Sem monitoração de estado**: o estado não é armazenado no servidor entre as solicitações, o estado da sessão fica salvo no cliente.
- **Capacidade de cache**: o cliente salva os dados em cache para eliminar a necessidade de algumas interações e melhora o desempenho.
- **Sistema/Arquitetura em camadas**: entre a comunicação do cliente e servidor, pode haver camadas adicionais que envolve segurança, dados, balanceamento de cargas, entre outros.
- **Código sob demanda**: envio de código executável do servidor para o cliente, permitindo assim novas funcionalidades ao cliente.
- **Interface uniforme**: separação dos interesses entre a comunicação, onde é possível identificar os recursos nas solicitações e manipular os recursos por meio de representações.

2 Métodos

2.1 Teste baseado em modelos

O teste baseado em modelo consiste em modelar o comportamento esperado do software, e a partir do modelo construído ocorre a geração de testes de forma automatizada. Na elaboração de testes para essa metodologia é utilizado modelo de estados, no qual cada transição (aresta) resulta em um novo estado (vértice). Um caminho é formado por um conjunto de estados e transições a partir de um estado inicial, ele descreve um cenário de uso. Quando há muitos caminhos é necessário estabelecer critérios para seleção desses caminhos que pode ser por: cobertura de estados (abrange maior número de estados), cobertura de transições (abrange maior número de arestas), por pares de transições ou por caminhos primos (caminho único que não é obtido pela combinação de nenhum outro).

2.2 Teste dirigido pelo comportamento

O segundo método abordado neste projeto é a geração de testes dirigido pelo comportamento, em que a elaboração dos testes é baseado nos cenários. Essa abordagem ocorre com a técnica de desenvolvimento Behavior Driven Development - BDD, onde a criação do teste ocorre antes da escrita do código. O objetivo do BDD é facilitar a escrita de cenários para que profissionais com conhecimento técnico e profissionais que não possuem conhecimento técnico possam ter entendimento dos cenários descritos.

3 Descrição do método MBT e BDT

Na figura 1 há a junção dos dois métodos citados anteriormente, o método MBT ocorre nas etapas 2 e 3 e o método BDT a partir da etapa 4. Para cada fase temos as seguintes descrições:

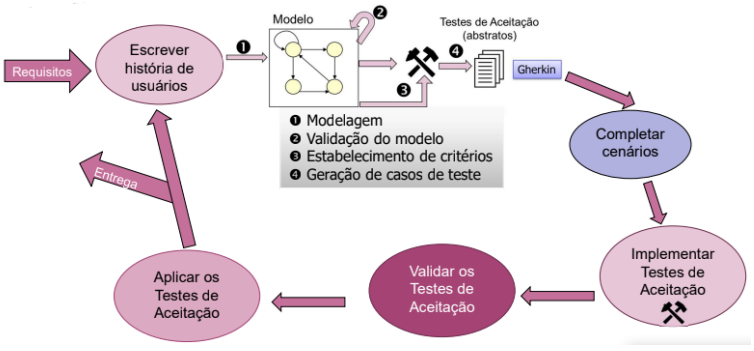


Figura 1: Fluxo do método proposto

(A) História do usuário: Escrita das histórias do usuário com base nos requisitos da aplicação.

(B) Modelagem: Criação de um modelo que representa a aplicação.

(C) Validação do modelo: Validação se o modelo proposto atende aos requisitos da aplicação.

(D) Geração dos cenários: Gerar os cenários de forma automatizada baseado no modelo criado.

(E) Complementar Cenários: Complementar os cenários gerados com parâmetros para diferentes testes.

(F) Implementar testes de aceitação: Escrever o comportamento esperado da aplicação.

(G) Execução dos testes de aceitação: Realizar os testes com base nos cenários obtidos.

(H) Validação dos testes de aceitação: Estabelecer critérios para definir a eficácia do método.

4 Ferramentas

A figura abaixo apresenta as ferramentas utilizadas para a elaboração deste relatório e na combinação dos métodos MBT e BDT. Nas fases de modelagem e validação do modelo foi utilizado a *Papyrus* e a *Moka*. Na geração dos cenários, foi utilizado a *Skyfire*). Para complementar dos cenários, implementação e executar os testes de aceitação, foi utilizado a *Karate Cucumber*.E, por fim, para a validação dos testes, foi utilizado a *Restats*. Uma breve descrição dessas ferramentas é fornecida a seguir:

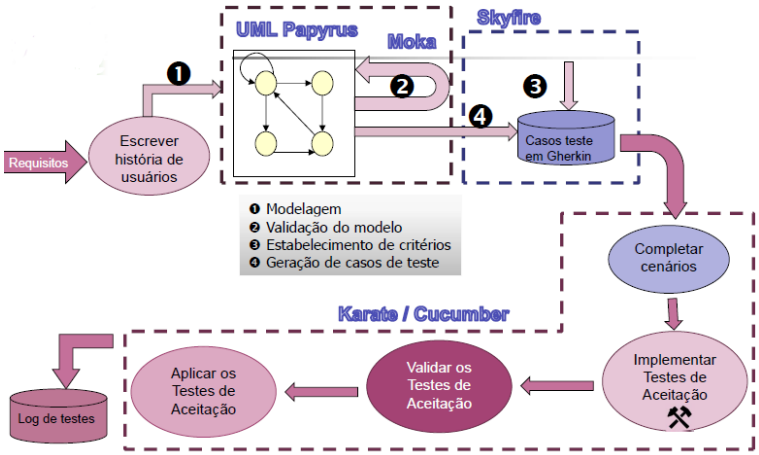


Figura 2: Ferramentas utilizadas no método

- Papyrus: Utilizado com a IDE Eclipse, é um plugin que permite a elaboração de modelos de software [8].

- Moka: Plugin do Eclipse utilizada para validar a modelagem do software [12].
- Skyfire: Uma ferramenta de Teste Baseado em Modelos (MBT) que gera cenários de teste Cucumber a partir de um diagrama de estados da UML [5].
- Karate: Desenvolvida em Java, para criar e executar teste automatizados de API Rest [10].
- Restats: Ferramenta open-source de análise de cobertura de testes de APIs [9].

5 Aplicação do Método

5.1 Serviço em teste

Para ilustrar o uso do método proposto neste relatório, foi utilizado um serviço Rest que é parte de um conjunto de serviços usados como padrão nos testes de APIs. A primeira subseção realiza uma breve apresentação sobre Engenharia de Linha de Produtos de Software, seguido da apresentação do serviço selecionado para este estudo.

5.1.1 Engenharia de Linha de Produto

API utilizada para o estudo de caso deste projeto é de um serviço que apoia o desenvolvimento baseado no modelo de Engenharia de Linha de Produto, onde o produto é um software. "A engenharia de linha de produto consiste em projetar, implementar e evoluir um conjunto de produtos com alto grau de similaridade entre si, de forma prescritiva, a partir de um conjunto de artefatos básicos." [3]

Para esse tipo de desenvolvimento, é necessário ter o entendimento do conjunto de objetos e operações comuns em sistemas. A primeira etapa para criação da linha de produto é a análise de domínio. Está análise é essencial e fundamental para a Engenharia de Linha de Produto, pois reconhece os objetos e operações similares de uma determinada área, onde há a oportunidade de reuso de código. Existem várias metodologias para essa análise, como ODM, FAST, DSSA, PuLSE, KobrA. Neste estudo, apresenta-se o método FODA (Feature-Oriented Domain Analysis), escolhido por ser um método maduro e bem documentado [3], brevemente descrito a seguir.

O método FODA é dividido em 4 etapas, conforme mostra a figura 3. Análise de Contexto consiste na definição do escopo que será trabalhado na análise do domínio[3]. Essa etapa é importante, pois define o escopo do domínio que será tratado na linha de software. Análise de Características essa etapa, são definidos as características comuns que pertencem ao domínio. Essas características comuns podem ser classificas como opcionais, mandatórias ou alternadas. Também devem ser informadas as restrições e dependências entre elas. A Modelagem Entidade-Relacionamento representa o conhecimento do domínio com a elaboração de um modelo entidade-relacionamento e Análise Funcional é o documento resultado dessa atividade, contendo o diagrama de casos de uso, diagrama de atividade e diagrama de estados.

Por se tratar de uma atividade não trivial, a Engenharia de Linha de Produto se divide em duas: Engenharia de Família cujo objetivo é o estudo de produto (não de domínio) e Engenharia de Aplicação, cujo objetivo é o estudo da infraestrutura que depende do produto. Por fim, a Característica (ou feature) é uma propriedade de sistema relevante para alguma parte interessada e é usada para obter funcionalidades comuns ou variáveis entre sistemas de uma mesma família de produtos.

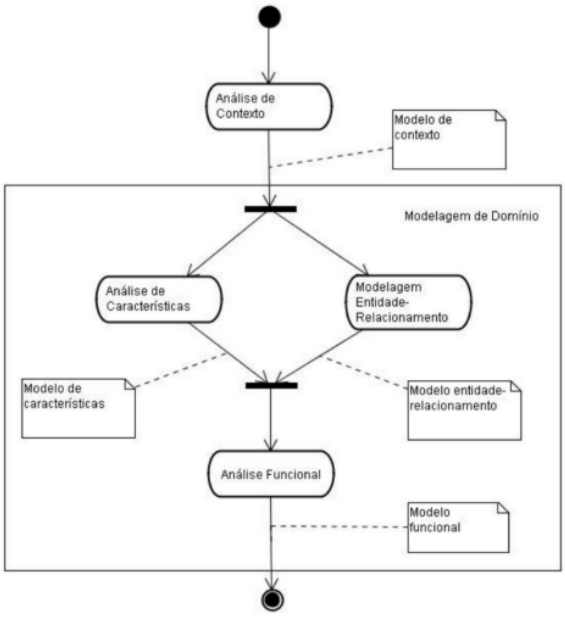


Figura 3: Visão da metodologia FODA

5.1.2 Feature Service

Feature Service [6] é a aplicação utilizada neste relatório. Nessa aplicação, é possível definir um produto, que, como dito anteriormente, é um software, podendo ser e-commerce, sistema bancário, sistema de ensino entre outras possíveis aplicações.

Na figura 4, é ilustrada a hierarquia de uma Linha de Produto de E-Shop, onde as features são: catálogos de produtos, forma de pagamento, níveis de segurança e busca por produto. Neste exemplo, podemos notar que a feature *Busca* é opcional, ou seja, em algumas instanciações de E-shop não haverá a funcionalidade de *Busca*. Assim como a forma de pagamento é um exemplo que informa que pelo menos uma forma deve ser utilizada, como *Transferência Bancária* ou *Cartão de Crédito*.

5.2 Histórias de Usuário

A escrita de histórias de usuários descreve um requisito da aplicação a partir do ponto de vista do usuário final. A escrita de Histórias de Usuários seguiu o formato *Eu como* persona

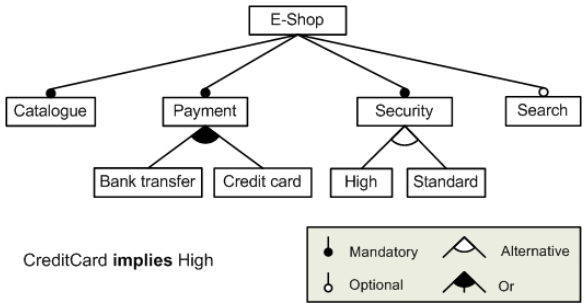


Figura 4: Linha de produto de software: E-shop [7]

gostaria de ação deseja *de forma que* resultado esperado. As histórias escritas atendem os requisitos funcionais: Criar um produto, Criar uma característica para o produto, Remover uma característica e Remover o produto.

Como	Ação	Resultado
Um administrador	criar produto A	obter a instanciação do produto A
Um administrador	criar uma característica 1 para produto A	produto A possui a característica 1
Um administrador	remover uma característica 1	produto A não contém a característica 1
Um administrador	remover o produto A	remoção do produto A na lista de produtos

Tabela 1: Histórias de usuário

5.3 Modelagem

O foco na modelagem é nas funcionalidades descrita na forma de história. Para esse relatório o modelo elaborado reflete a segunda história de usuário da tabela 1.

O nome dos eventos reflete os métodos que constam na API e para cada condição é dado um nome que representa o domínio do problema. No estado inicial há alguns produtos registrados no sistema as transições possíveis são a inserção de uma característica para um produto existente e a tentativa de inserção de uma característica para um produto não existente.

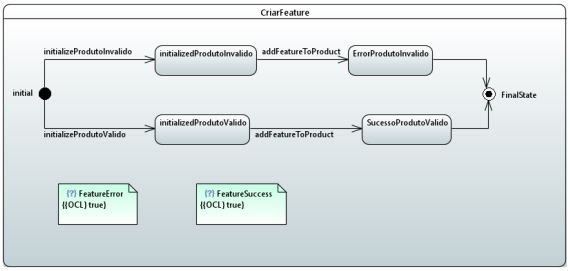


Figura 5: Modelo de estado criação de uma característica

5.4 Validação do Modelo

Para validação da modelagem da história, foi utilizado o plugin da *Papyrus* na IDE Eclipse. A versão do arquivo UML gerado é 2.5.0. [8]. Com auxílio da Moka, foi desenvolvida uma simulação do modelo criado. Com a orientação da documentação deste plugin na IDE Eclipse, o modelo foi completado com dados. Para isso, foi necessário criar um diagrama que contemplasse os métodos e as classes que envolviam a história de usuário selecionada. Em seguida, foram informados valores (lista de produtos, lista de características e o produto que receberia a característica) para poder realizar a simulação.

5.5 Geração dos cenários com diferentes critério

A geração dos cenários ocorreu de forma automatizada através da ferramenta *Skyfire* e os parâmetros informados são: um caminho para o arquivo UML com modelo do sistema, o critério de cobertura, a descrição dos cenários gerados e o nome do arquivo que irá salvar os cenários gerados. Além do auxílio do Cucumber que é ferramenta voltada para o desenvolvimento BDT e utilizando a classe abstrata CucumberTestGenerator e o método generateCumcuberScenario, foi possível gerar os cenários.

Os cenários seguem o formato da escrita Gherkin, uma linguagem de alto nível cujo objetivo é a compressão das especificações funcionais do sistema, os cenários no padrão Gherkin são formados pelas keywords: Given, When, Then, And e But.

Foram gerados casos de teste usando os quatro critérios de cobertura. Na figura abaixo temos o cenário resultado do critério *Node Coverage*, no qual ocorre a criação de uma característica.

```
1   Feature: CriarProdutoComFeature
2
3   Scenario: initializeProdutoInvalido addFeatureToProduct
4   Given initializeProdutoInvalido
5   When addFeatureToProduct
6   Then FeatureError
7
8
9   Scenario: initializeProdutoValido addFeatureToProduct
10  Given initializeProdutoValido
11  When addFeatureToProduct
12  Then FeatureSuccess
```

Figura 6: Cenários de teste em Gherkin gerado pelo critério Node Coverage

Critério	N Cenários	N máximo de passos	N mínimo de passos
Node Coverage	2	3	3
Edge Coverage	2	3	3
Edge-pair Coverage	2	3	3
Prime-path Coverage	2	3	3

Tabela 2: Dados sobre cenários gerados a partir do modelo

5.6 Parametrização dos cenários

A etapa de parametrização dos cenários é necessária para que seja possível a execução dos testes e para que se possa obter diferentes retornos conforme os cenários gerados. Nessa etapa temos que informar os dados da API Rest (Feature Service) que foi utilizado como estudo de caso para esse relatório, informando valores para complementar os testes na API Rest.

```
Feature: CriarProdutoComFeature

  Scenario Outline: initializeProdutoInvalido addFeatureToProduct
    Given initializeProdutoInvalido <produto>
    When addFeatureToProduct <feature>
    Then FeatureError <errorCode>

    Examples:
    | produto  | feature       | errorCode |
    | ESHOP_1  | VIDEO_LESSONS | 500       |

  Scenario Outline: initializeProdutoValido addFeatureToProduct
    Given initializeProdutoValido <produto>
    When addFeatureToProduct <feature>
    Then FeatureSuccess <successCode>

    Examples:
      | produto  | feature | successCode |
      | ESHOP_0 | PAYMENT  | 500         |
```

Figura 7: Exemplo de cenário parametrizado

5.6.1 Implementação e execução dos casos de teste

Karate é um projeto open-source para automatização de testes. A ferramenta permite realizar teste de API, testes de desempenho de API, testes de interface e simulações de API. Para a execução dos testes foi utilizado este software. A sintaxe utilizada é similar ao Gherkin, como ilustrado na figura **??**. No teste executado temos ambos os cenários gerados pela *Skyfire* initalizeProdutoValido e initializeProdutoInvalido. Ao comparar a figura 7 com a figura 8, podemos notar que, no lugar initialize foi informado a URL responsável pela criação de uma característica. Também foi necessário informar o parâmetro body no formato form-data. Por fim, a validação ocorre com o status de retorno da requisição.

Como premissa para execução dessa ferramenta, é necessário ter o arquivo *.feature*, que pode ser obtido com a saída da *Skyfire*, e informar os parâmetros da API Rest: URL, método HTTP, status e response esperado do teste.

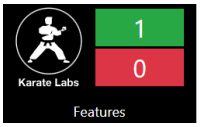
```
Feature: CriarProdutoComFeature

  Scenario Outline: initializeProdutoInvalido addFeatureToProduct
    Given url 'http://localhost:8080/products/' + '<produto>' + /features/ + '<feature>'
    And form field description = '<description>'
    When method POST
    Then status <statusAddFeauture>
    And print response

    Examples:
    | produto  | feature       | description         | statusAddFeauture |
    | ESHOP_1  | VIDEO_LESSONS | aulas em videos     | 500               |
    | ESHOP_0  | PAYMENT       | formas de pagamento | 201               |
```

Figura 8: Implementação dos casos de testes

A ferramenta *Karate* gera uma pasta chamada*karate-reports* com o compilado dos resultados dos testes executados informando quantidade de testes com retorno de sucesso, falhas, cenários e o tempo (em milésimos de segundos) da duração dos testes.



Tags | Timeline

Feature	Title	Passed	Failed	Scenarios	Time (ms)
src/test/java/org/example/criarFeature.feature	CriarProdutoComFeature	2	0	2	1218

Figura 9: Exemplo do arquivo summary gerado pela ferramenta Karate

6 Monitoração e Observação

Esta seção mostra a avaliação dos testes executados com base na cobertura dos testes. Essa análise foi realizada através da ferramenta *Restats*. Nela, há o critério de cobertura em oito métricas descritas na tabela 3. A *Restats* possui como parâmetros de entrada o Swagger da aplicação e os arquivos resultados da *Karate*, onde para cada cenário há um arquivo txt com as informações de request e response. Essas informações são declaradas no arquivo *config.json* [2].

Na figura 10, temos um trecho do stats.json (o arquivo completo encontra-se no anexo A), um dos arquivos de saída da *Restats*, que detalha a saída de uma métrica, seguindo o mesmo modelo para as demais métricas. O Campo *documented* refere-se às operações presentes no Swagger da aplicação, *documentedAndTested* informa o número de operações que foram testadas e que constam na documentação, e, por fim, *totalTested* é o número total de operações testadas, podendo estar inclusas ou não na documentação da API.

Outro indicador informado pela *Restats* é o nível de Modelo de Cobertura de Testes chamado de TCL; este indicador possui 8 níveis de avaliação [2], que são:

(A) TCL0: sem requisitos de cobertura.

(B) TCL1: cobertura do caminho.

(C) TCL2: cobertura da operação.

(D) TCL3: cobertura por tipo de conteúdo de solicitação.

(E) TCL4: cobertura de parâmetros e classes de código de status.

(F) TCL5: cobertura por código de status.

(G) TCL6: cobertura dos atributos do body.

(H) TCL7 cobertura do fluxo da operação.

O grupo de cenários realizados atingiu o nível TCL2, correspondendo à cobertura dos caminhos e das operações.

Métrica	Porcentagem de Cobertura	Porcentagem de cobertura (Swagger refatorado)
Path Coverage	0.09	1.0
Operation Coverage	0.05	1.0
Status Class Coverage	0.0	0.0
Status Coverage	0.0	0.0
Response Type Coverage	0.0	0.0
Request Type Coverage	0.125	0.5
Parameter Coverage	0.0	0.0
Parameter Value Coveraga	N/A	N/A

Tabela 3: Métrica obtidas pela Restat

7 Discussão

Podemos observar que houve uma baixa cobertura nos testes. É possível concluir que dois fatores colaboraram para esse resultado: a documentação (Swagger) da API do objeto de estudo deste relatório e a baixa complexidade do serviço selecionado. Dos 10 endpoints que compõem a API, apenas um foi contemplado pelos testes. A ferramenta *Restats* realiza a análise sobre o total dos endpoints presentes no Swagger da API. Ao alterá-lo para conter apenas a requisição testada, o valor da cobertura foi maior (TCL2). Apesar disso, algumas métricas permaneceram sem cobertura. Isso ocorreu devido à ausência das especificações

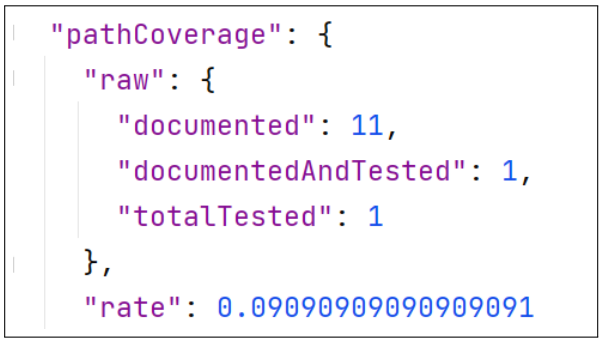


Figura 10: Exemplo de saída stats.json

de mensagens de retorno da API. Ao estudar o Swagger, podemos notar que apenas os métodos GET e PUT tem um código de status de retorno definido, o que impacta na análise da *Restats* e contribui para o baixo valor da cobertura.

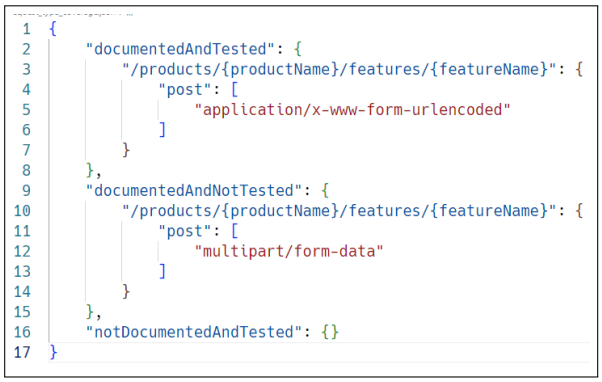
Os arquivos de saída da *Restats* são salvos na pasta *reports*, o diretório desta pasta deve ser informado em *config.json*, como informado anteriormente, e estão no formato JSON, onde para cada métrica (listados na tabela 3) existe um arquivo detalhado que informa as operações e suas categorias (*documentedAndTested*, *documentedAndNotTested* e *notDocumentedAndTested*).

Por exemplo, no arquivo *path_coverage.json* é informado que a operação */products/{productName}/features/featureName* consta na documentação (Swagger) e foi testada e na categoria *documentedAndNotTested* não consta nenhuma operação, portanto a cobertura informado para essa métrica é 1.0.

Figura 11: Arquivo path_coverage.json

Um caso diferente ocorre para o arquivo *request_type_coverage.json* que avalia a cobertura por conteúdo de uma requisição. Neste caso a ferramenta mostra que houve conteúdo que consta na documentação da API, mas que não foi testado. Era esperado um teste no qual *quest_type* fosse do tipo *multipart/form-data*, por isso a cobertura obtida foi de 0.5.

Por fim, também não foi possível atingir uma cobertura superior ao TCL2, pois para isso seria necessário complementar os testes de modo a cobrir o conteúdo das requisições e respostas, a fim de alcançar o nível TCL3. Não era esperado um nível de cobertura de testes acima do TCL3, para este estudo inicial, pois a história de usuário selecionada para este relatório abrange apenas uma operação. Para melhorar o nível de cobertura, é necessário, naturalmente, modelar e testar as demais funcionalidades da API *Feature*



```
{
    "documentedAndTested": {
        "/products/{productName}/features/{featureName}": {
            "post": [
                "application/x-www-form-urlencoded"
            ]
        }
    },
    "documentedAndNotTested": {
        "/products/{productName}/features/{featureName}": {
            "post": [
                "multipart/form-data"
            ]
        }
    },
    "notDocumentedAndTested": {}
}
```

Figura 12: Arquivo request_type_coverage.json

Service. Além disso, os resultados mostram a necessidade de combinar outras técnicas de teste que permitam uma cobertura mais abrangente do conteúdo das requisições e respostas.

8 Conclusão e Trabalhos Futuros

Neste projeto, houve a combinação dos métodos MBT e do BDT para automatizar a geração dos cenários de testes e execução deles.Visando facilitar a realização de testes automatizados por testadores de forma a abstrair os aspectos de programação em um ambiente que costuma ser altamente heterogêneo em termos de linguagens de programação, que são os testes de APIs. Além da automatização das atividades de testes, desde a geração até a execução, neste trabalho também mostramos que o testador pode avaliar a efetividade dos testes produzidos, em termos de cobertura da API, o que mostrará que aspectos precisam ser melhorados nos testes.

Para a automatização dos passos do método proposto foram utilizadas 5 ferramentas neste estudo: (i) a *Papyrus*, um plugin do Eclipse, que foi usado para a criação do modelo de estados; (ii) a *Skyfire*, que importa o modelo construído na *Papyrus* e gera os casos de teste a partir deste modelo. Os casos de teste são gerados no formato de cenários em Gherkin, (iii) os cenários podem ser usados na ferramenta *Cucumber*, que auxilia na criação de casos de testes executáveis.(iv) *Karate*, que permite criar casos de teste executáveis para APIs Rest e (v) *Restats*, que permite avaliar a cobertura da API obtida pelos casos de teste executados.

O método proposto foi executado voltado para o requisito funcional da API Rest, a criação de uma característica no contexto da Feature Service. Com a ferramenta *Papyrus* houve a modelagem da história de uso. O método recomenda a criação de casos de teste por histórias. Para cada história foi criado o modelo de estados usando a ferramenta *Papyrus*.

As etapas em que houve maiores esforços foram: na elaboração do modelo houve um esforço maior para adequação da saída da *Papyrus* para *Skyfire* pois nessa etapa foi necessário adicionar manualmente o xml para versão 4.0.0 para que fosse possível utilizar o arquivo na *Skyfire*; outras adequações, que foram realizadas são nos arquivos de request e

response (formato txt) para *Restats* para cada cenário executado, porque há um padrão esperado pelo software. Por fim o entendimento da API Rest Feature Service e o contexto de engenharia de linha de software, demandou esforços e tempo não estimado previamente no início desse estudo.

Como sugestões para futuros trabalhos que envolve o tema desse projeto é indicado que complete os retornos da API Rest utilizada nesse projeto, pois no atual estado há apenas retorno com status 500 e sem mensagens com tratamento de erros. Sugere-se também abranger os requisitos funcionais que a aplicação permite para poder gerar a modelagem por história, geração dos cenários e por fim a execução dos cenários e análise de sua cobertura, dado que a API Rest possui a vantagem de se tratar de uma linha de engenharia de software onde há liberdade para definir requisitos funcionais.

Por fim, para melhorar a efetividade do método proposto neste relatório para geração e execução de teste em API, recomenda-se abranger mais história de usuário na modelagem, de modo que seja possível exercitar mais operação da API. Além disso, para obter resultados completos da *Restats*, é recomendado utilizar técnica de testes de domínio, a fim de cobrir melhor o conteúdo das requisições.

Referências

[1] L. M. Alberto, S. Sergio e C. R. Antonio, *Test Coverage Criteria for RESTful Web APIs*, 2019.

[2] C. Davide, Z. Amedeo, P. Michele e C. Mariano, *Restats: A Test Coverage Tool for RESTful APIs*, 2021.

[3] L. C. E. Ana e R. F. M. Cecília, *Um Estudo para Implantação de Linha de Produto de Software baseada em Componentes*, 2009.

[4] Martins, Eliane, *Aula10-TestesAPIComKarate2.2023. Apresentação do Power Point.* Disponível em: `https://drive.google.com/file/d/1vCz_I8F6kQwvunyY_hYq2zHl6WfyExnS/view?usp=drive_link`. Acesso em: 01 Jun 2023.

[5] Skyfire. *Projeto github*. Disponível em: `https://github.com/mdsol/skyfire`. Acesso em: 13 Fev 2023.

[6] API Rest Feature Service. *Projeto github*. Disponível em: `https://github.com/JavierMF/features-service`. Acesso em: 17 Mar 2023.

[7] WIKIPEDIA. *Feature Model*. Disponível em: `https://en.wikipedia.org/wiki/Feature_model#/media/File:E-shopFM.jpg`. Acesso em: 20 Abr 2023.

[8] ECLIPSE DEV. *Papyrus*. Disponível em: `https://eclipse.dev/papyrus/`. Acesso em: 17 Mar 2023.

[9] KARATE. *Projeto gitub*. Disponível em: `https://github.com/SeUniVr/restats`. Acesso em: 03 Mar 2023.

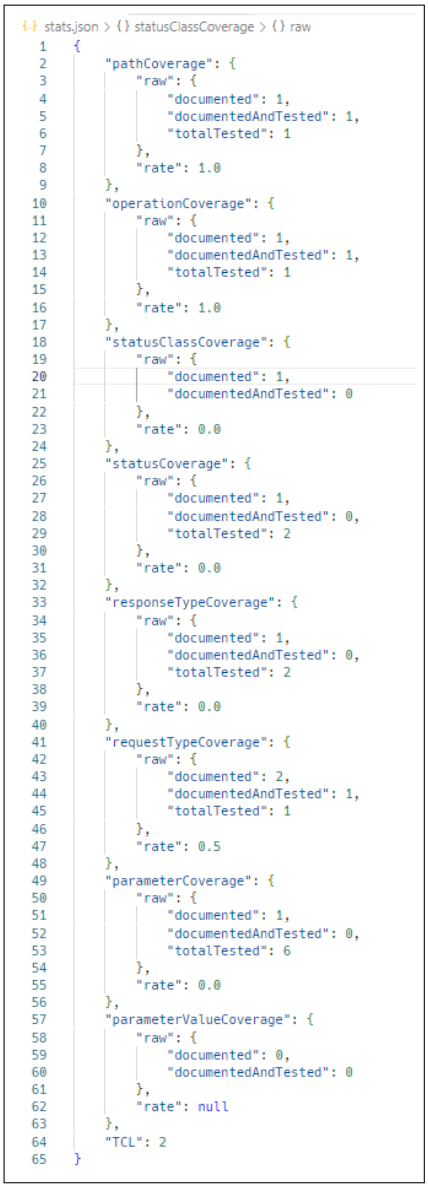
[10] KARATE. *Getting Started*. Disponível em: `https://karatelabs.github.io/karate/`. Acesso em: 03 Mar 2023.

[11] ECLIPSE FOUNDATION. *Papyrus UserGuide ModelExecution*. Disponível em: `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`. Acesso em: 17 Mar 2023.

[12] MBT_BTD. *Repositório github com os programas desenvolvidos para esse relatório*. Disponível em: `https://github.com/thaisaraujo/mbt_bdt`. Acesso em: 14 Jul 2023.

Anexos

A Arquivo stats.json



stats.json > {} statusClassCoverage > {} raw


```
{
    "pathCoverage": {
        "raw": {
            "documented": 1,
            "documentedAndTested": 1,
            "totalTested": 1
        },
        "rate": 1.0
    },
    "operationCoverage": {
        "raw": {
            "documented": 1,
            "documentedAndTested": 1,
            "totalTested": 1
        },
        "rate": 1.0
    },
    "statusClassCoverage": {
        "raw": {
            "documented": 1,
            "documentedAndTested": 0
        },
        "rate": 0.0
    },
    "statusCoverage": {
        "raw": {
            "documented": 1,
            "documentedAndTested": 0,
            "totalTested": 2
        },
        "rate": 0.0
    },
    "responseTypeCoverage": {
        "raw": {
            "documented": 1,
            "documentedAndTested": 0,
            "totalTested": 2
        },
        "rate": 0.0
    },
    "requestTypeCoverage": {
        "raw": {
            "documented": 2,
            "documentedAndTested": 1,
            "totalTested": 1
        },
        "rate": 0.5
    },
    "parameterCoverage": {
        "raw": {
            "documented": 1,
            "documentedAndTested": 0,
            "totalTested": 6
        },
        "rate": 0.0
    },
    "parameterValueCoverage": {
        "raw": {
            "documented": 0,
            "documentedAndTested": 0
        },
        "rate": null
    },
    "TCL": 2
}
```

B Swagger da API Rest Feature Service

```
{
  "swagger": "2.0",
  "info": {
    "title": "Feature Service Refatoria Feauture",
    "version": "1.0"
  },
  "host": "localhost:8080",
  "basePath": "/",
  "schemes": [
    "http"
  ],
  "paths": {
    "/products/{productName}/features/{featureName}": {
      "post": {
        "operationId": "addFeatureToProduct",
        "produces": [
          "application/json"
        ],
        "consumes": [
          "application/x-www-form-urlencoded",
          "multipart/form-data"
        ],
        "parameters": [
          {
            "name": "productName",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "featureName",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "description",
            "in": "formData",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "default": {
            "description": "successful operation"
          }
        }
      }
    }
  }
}
```