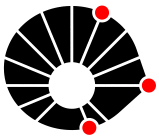

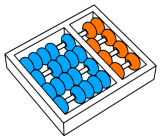

Automatização do desenvolvimento dirigido ao comportamento com o uso de teste baseado em modelo

L. Y. Koike *E. Martins*

Relatório Técnico - IC-PFG-22-45
Projeto Final de Graduação
2022 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Automatização do desenvolvimento dirigido a comportamento com o uso de teste baseado em modelo

Leonardo Y. Koike Eliane Martins

Resumo

Com a popularidade de metodologias ágeis dentro do área de engenharia de software, o desenvolvimento dirigido a comportamento, BDD, se tornou uma prática recorrente nas equipes de desenvolvimento. No entanto, olhando seu processo, identificamos atividades manuais que dificultam a sua execução no cotidiano. Por isso, o teste baseado em modelos, MBT, se propõem a automatizar atividades como a geração de cenários de testes e a implementação dos testes de aceitação para facilitar o uso do BDD.

1 Introdução

1.1 Contexto

Com a evolução da tecnologia e aumento de empresas de software, a busca por entregar sistemas, com qualidade e que atendam as reais necessidades de seus clientes, tem crescido cada vez mais. Por esse motivo, times de desenvolvimento e os analistas de negócios (BAs) têm procurado metodologias e processos que atendam a essas necessidades, ainda seguindo práticas de agilidade.

Por esses motivos, o desenvolvimento dirigido ao comportamento, do ingles BDD (Behavior Driven Development)[1][2], surge para ditar processos para entregar software com qualidade. Ele consiste de ciclos que têm como etapas: o entendimento do problema, definição de requisitos e comportamentos do sistema, escrita de testes de aceitação, validação destes e, por fim, a demonstração e entrega das funcionalidades.

Ligado a todas as fases, existe a descrição do comportamento através de uma linguagem estruturada e abstrata, que geralmente é o Gherkin[4]. Através dessa linguagem, todas partes envolvidas no projeto têm a compreensão do comportamento do sistema. Essa documentação também é importante para descrever os requisitos que devem ser implementados e testados, para assim garantir a qualidade. Esse processo de teste é chamada de teste dirigido ao comportamento, BDT (Behavior Driven Test).

1.2 Motivação

No entanto, ao utilizar o processo BDD no cotidiano de um time, encontramos alguns pontos que merecem melhorias:

- A formulação manual de especificações executáveis do Gherkin, vinda de um requisito, depende da capacidade e experiência de um testador. Por isso, ela está sujeita a falhas na transcrição dos cenários.
- Falta de métrica de cobertura, logo não é possivel saber quanto os casos de testes criados atendem uma funcionalidade[5].
- Com a evolução de funcionalidade, existe uma dificuldade no rastreamento de casos de testes e seus respectivos requisitos[3]. Além de aumentar o número de cenários, não existe uma maneira simples para validar quais deles ainda são válidos.

1.3 Objetivos

Será estudado o uso de modelos para mapear o comportamento a fim de melhorar o processo BDD. Essa técnica é chamada de testes baseados em modelos, MBT (Model Based Test). Ele se propõem a:

- Facilitar na evolução e manutenção de uma funcionalidade, refletindo qualquer mudança de requisito nos modelos.
- Produzir cenários abstratos em Gherkin automaticamente.
- Métrica de cobertura sobre o modelo desenhado.

1.4 Metodologia

Será utilizada uma interface de programação de aplicação, API, já existente para adotar a metodologia MBT. Então realizaremos o processo de: modelagem do software, geração de cenários abstratos em Gherkin e a codificação dos testes de aceitação. Ao final, avaliaremos os resultados apontando os benefícios e malefícios do seu uso.

1.5 Estrutura do texto

A descrição deste projeto segue a seguinte ordem: fundamentação teórica dos processos utilizados, diagnóstico do problema a ser explorado, planejamento para execução do método, aplicação desta metodologia, observações dos dados obtidos, discussão desses resultados e, por fim, a conclusão do projeto e trabalhos futuros.

2 Fundamentação teórica

2.1 Mais sobre BDD

O processo BDD é conhecido no mercado por aproximar o time de desenvolvimento com seus clientes no trabalho de definir os requisitos que tragam valor para o produto. Ele possui passos para orientar a equipe na busca de definições para o desenvolvimento. Segundo John Smart[1], ele descreve o processo do BDD constituído de cinco etapas: ilustração, formulação, automação, validação e demonstração.

A ilustração compreende um esforço conjunto para explorar os requisitos. Isso acontece a partir de discussões sobre o comportamento esperado, elaboração de exemplos e situações de sucesso e falha dos quais o sistema deve tratar. Essa fase é fundamental, pois nela há o compartilhamento de conhecimento entre todos os integrantes.

A formulação é utilizada para documentar as especificações executáveis com base na estória de usuário. Aqui começamos ver o Gherkin, que expressa os requisitos em uma linguagem que pode ser compreendida por todos, técnicos e não técnicos. Segue um exemplo:

```
Feature: Criação de incidente

 Scenario: Criação de um novo incidente no sistema
   Given Usuário logado no sistema de incidente
   When Clica em criar novo incidente
   And Escreve uma descrição do ocorrido
   And Clica no botão salvar
   Then Incidente é criado no sistema
```

A automatização consiste em codificar testes de aceitação de acordo com o Gherkin gerado, criando os casos de testes executáveis.

A validação é a execução dos testes automatizados, onde é analisada a qualidade do sistema. Aqui podem ser identificados problemas negociais que precisam ser corrigidos pelos desenvolvedores.

Por fim, a demonstração é apresentação do resultado para avaliação dos clientes do projeto.

2.2 Mais sobre MBT

O MBT faz o uso de modelo para expressar o comportamento do sistema. Essa representação por modelo facilita tanto a visualizar todo o comportamento de uma funcionalidade, entendendo sua complexidade, como o seu rastreamento com os requisitos. Através de dessa ilustração abstrata, a equipe pode compartilhar o conhecimento e alterar o modelo a fim de contemplar os requisitos. Nessa fase, existe uma preocupação de validar se o modelo está estruturalmente correto de acordo com os padrões da lingugagem de modelagem utlizada, e também se corresponde aos requisitos que representa[6].

Uma notação muito usada é a linguagem de modelagem unificada, UML (Unified Modeling Language), e muitas ferramentas que fazem a leitura desses modelos exigem o uso dela para poder gerar os casos de testes. Por isso se faz necessário o conhecimento dessa linguagem e de software de modelagem como o Papyrus[8] para criar os modelos do sistema.

Com base no desenho do sistema, são utilizados software que leem esses modelos e, de acordo com estratégias de leitura desses modelos, geram automaticamente diferentes tipos de casos de testes abstratos. Em seguida, é necessário codificar esses casos de testes abstratos para serem executados. Ferramentas como o Cucumber[7] ou o Behave[9] cumprem essa função de codificar testes com base na leitura do Gherkin, tornando-os executáveis. Por fim, é feito a validação do software executando esses testes.

2.3 Modelo de estado para os testes

O modelo de máquina de estados é um tipo de modelagem que é familiar para os desenvolvedores, por facilitar a apresentação da lógica de um sistema[6]. Dentro de um sitema, ela é composta por:

- Estados: representam a situação da aplicação, contendo informações atreladas a ela
- Transições: apresentam ações que mudam uma aplicação de um estado para outro. Ela pode representar chamadas a uma API ou uma interação com uma tela, por exemplo.
- Pseudoestados: é um conjunto de elementos que alteram o fluxo de uma máquina de estado. Por exemplo, elemento de escolha é um pseudoestado, que a partir de uma transição e uma condição, podem gerar mais possibilidades de transições.

Esse tipo de estrutura é exigido por ferramentas de geração de cenários de testes. Um exemplo de ferramenta é o Skyfire[7] que aceita como entrada um máquina de estado como uma representação do sistema para gerar os cenários abstratos em Gherkin.

3 Diagnóstico

O uso do BDD, principalmente para a área de qualidade, apresenta problemas ao longo do tempo. A medida que um sistema cresce, existe um grande conjunto de artefatos de testes que já foram criados[3] manualmente. E algo comum em aplicações de grande escala, é a prática de atualizar ou melhorar determinadas funcionalidades. Somando essas duas situações, é necessário um esforço da equipe para identificar cada cenário e se ele continua válido. Resumindo, isso gera um custo de tempo que cresce a medida que um sistema evolui.

Além da falta de automatização nesse processo, o teste dirigido ao comportamento, BDT (Behavior Driven Test), não possui nenhum parâmetro de cobertura para mostrar quanto os cenários de testes cobrem efetivamente os requisitos listados em uma estória de usuário. Ou seja, essa atualização manual dos cenários fica sujeita a falhas.

A solução para essas questões levantadas é utilizar outra metodologia dentro do BDD que automatizem pontos fundamentais para o restante processo, como a geração dos cenários em Gherkin. Assim o MBT surge para, através de um modelo de estados e o uso de ferramentas, mapear todos os fluxos possíveis e, em seguida, gerar os artefatos de testes para realizar a cobertura da funcionalidade, tudo isso automaticamente[10].

4 Planejamento

Para avaliar a metodologia MBT, será utilizada uma parte de uma API para execução de todo os métodos. A API escolhida é o software Instatus[12], que serve para fazer o monitoramento de incidentes de outros sistemas.

Abaixo seguem as etapas do ciclo:

- Primeiramente, é preciso entender o funcionamento do sistema, todo a sua lógica e suas regras negociais. Para isso, é essencial ler a documentação ou materias de apoio para obter tais informações. Como teremos uma API como alvo do estudo, é possível enviar requisições para testar seu comportamento. Para isso é recomendado o uso de ferramentas como Postman ou Insomnia.

- Em um próximo momento, é necessário construir modelos que representem o funcionamento do sistema. Por isso é recomendado modelar diagramas de classe, para entender a estrutura dos dados retornados; diagramas de sequência, para entender como o sistema faz requisições para API, e por fim um diagrama de estados, que representa como o sistema funciona como um todo[11]. Nesta etapa é necessário ter um conhecimento de protocolo de transferência de dados web, HTTP (Hypertext Transfer Protocol), para criar o diagrama de sequência de chamadas a REST API, e em UML, para construir o modelo de um sistema.

  Como ferramenta, será preciso um software de modelagem. Nesse caso será utilizado o Papyrus para uma máquina de estados finitos. É importante destacar que Papyrus não aceita, dentro do UML, parâmetros de guarda dentro dos estados e transições de estados, somente invariantes de estados. Desta forma, precisamos criar uma máquina de estados que contenham essa característica do software para fazê-lo funcionar.
  O modelo deverá ter alguns elementos básicos: estados, transições e pseudoestados, como escolha e ponto de inicio e saída da máquina de estado. Dentro das transições e das invariantes de estado, é fundamental, para as outras fases, identificar nomear as ações e verificações que o modelo deve conter. Isso é importante tanto para todos compreenderem o que acontece nos fluxos, como também para a codificação.

- Com uma máquina de estados que representa o sistema, o próprio Papyrus valida se o modelo criado está dentro dos padrões do UML. Então a medida que o modelo é feito, podemos observar que ele orienta os usuários a ajustarem seus erros.

- A geração de casos de testes é feita utilizando o software Skyfire. Para esse projeto, será utilizado o codigo fonte que está pronto para uso, com todas as configurações feitas, do Skyfire[14]. Desta forma, basta referenciar o arquivo UML para entrada do programa e ele mesmo irá gerar os casos de testes. O Skyfire basicamente trata a maquina de estados que informamos como um grafo, onde cada estado é um nó e as transições, arestas do grafo. Além disso, o Skyfire se utiliza de diferentes estratégias de cobertura desse grafo: Node, Edge, EdgePair e PrimePath[7]. Como resultado, serão gerados arquivos separados para cada estratégia de cobertura executada, cada uma contendo todos os cenários em Gherkin. Esses cenários gerados apresentarão nos steps os mesmos nomes, de acordo com o modelo, das transições como passos de execução e o nome da invariante de estado como o passo de validação do cenário. Logo os cenários de Gherkin serão abstratos.

- Por fim, temos a implementação dos casos de testes. Aqui será utilizado uma biblioteca do python, o *behave*. Através dele é preciso fazer a implementação dos passos (step definitions), onde cada passo é uma ação dentro do sistema como chamadas a API.

Essa etapa requer o conhecimento do uso da linguagem Python, biblioteca *behave* e *requests* para implementar a estrutura do projeto e realizar as chamadas a API.

Ao completar o primeiro ciclo, será realizada mais um incremento para contemplar cenários com erros dentro do sistema e observar o uso desse método para incrementos ao sistemas.

Ao encerrar essas atividades, existem questões que devem ser discutidas para entender o resultado deste estudo:

- Estrutura dos casos de testes gerados respeitam o Gherkin corretamente?
- Cenários gerados possuem descrições simples?
- Qual o esforço para aplicar o MBT?

5 Aplicação do método

5.1 Aplicação utilizada

Como dito, iremos usar a API do sistema Instatus nesse projeto. Essa API serve para fazer o monitoramento de outros sistemas, como um website, através da criação de incidentes. Por isso ele cria um dashboard com os incidentes que estão ativos, onde cada incidente pode ter sua descrição, status ou componente atualizado. Além disso é possível criar novos incidentes e deletear outros.

Essa aplicação segue a arquitetura REST para uma API, onde as respostas e requisições são codificadas no formato JSON[13]. Ela também possui uma autenticação via token que deve ser sempre enviada junto das requisições.

5.2 Modelagem

Para realizar chegar até a máquina de estados do sistema, foi preciso antes construir outras modelagens para entender o funcionamento do sistema, como a relação entre os dados e as chamadas da API[11]. Desta forma foram feitos dois diagramas, um de classes e outro de sequência para apresentar essas lógicas.

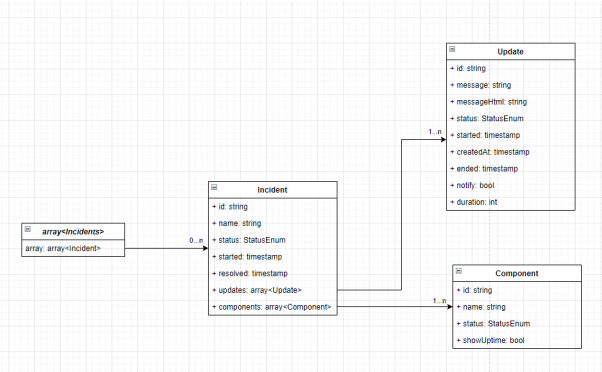

Figura 1: Diagrama de classes para representar os dados

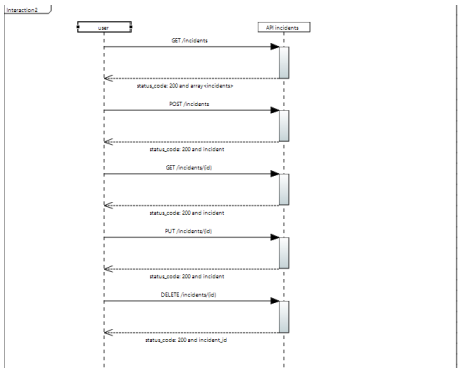

Figura 2: Diagrama de sequência de chamadas para API

A partir disso foi possível fazer a modelagem da máquina de estados[11] com base na lógica do sistema. Aqui temos a primeira versão realizada, contendo os comportamentos básicos do sistema.

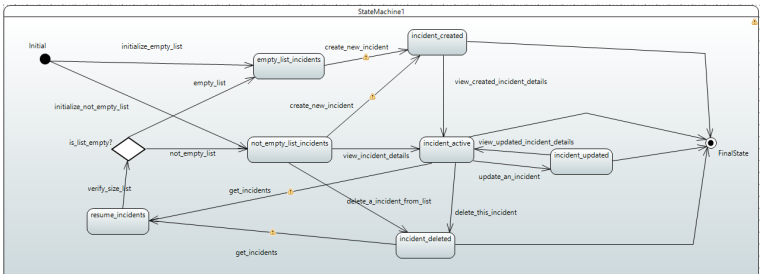

Figura 3: Primeiro incremento máquina de estados Instatus API

Para a segunda versão, o diagrama de classe e de sequência não sofreram mudanças. Foi modificado somente a máquina de estados para contemplar novos fluxos, contendo cenários de erro.

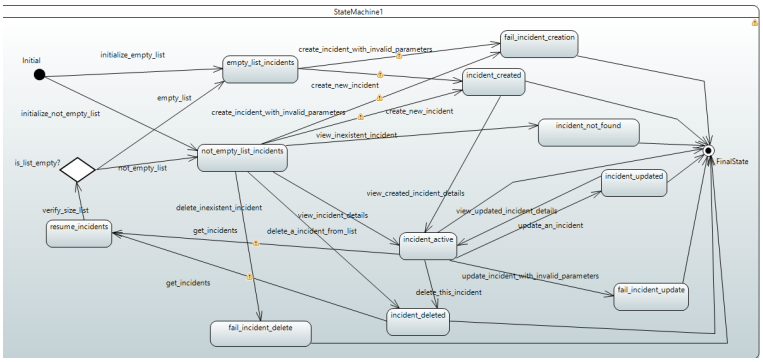

Figura 4: Segundo incremento máquina de estados Instatus API

5.3 Validação do modelo

Tanto o primeiro como o segundo incremento, foi utilizada a validação do máquina de estado do Papyrus. Nas figuras 3 e 4, podemos ver que o Papyrus identificou alguns alertas nas transições entre estados no modelo devido ao nome estar repetido. No entanto isso foi não impeditivo para executar os próximos passos.

5.4 Geração dos casos de testes

No primeiro incremento, o Skyfire gerou os seguintes cenários e passos para cada estratégia de cobertura:

Estratégia de cobertura	Qtde de cenários	Qtde de passos
Node	5	35
Edge	7	55
EdgePair	10	104
PrimePath	34	436

Tabela 1: Tabela de quantidade de testes gerados por estratégia no primeiro incremento

Durante a execução do Skyfire, é gerado um arquivo de log onde é possível observar a cobertura de casos percorrendo o modelo.
No segundo incremento, foi executado novamente o Skyfire para gerar novos cenários:

Estratégia de cobertura	Qtde de cenários	Qtde de passos
Node	9	43
Edge	11	69
EdgePair	20	164
PrimePath	63	713

Tabela 2: Tabela de quantidade de testes gerados por estratégia no segundo incremento

O gherkin gerado possui as mesmas nomeclaturas utilizadas aos elementos dentro do UML. Aqui está um exemplo de um dos testes gerados:

```
Feature: Instatus

 Scenario: initialize_not_empty_list view_incident_details delete_this_incident
  Given initialize_not_empty_list
  When view_incident_details
  Then incidentActiveSuccess
  When delete_this_incident
  Then deleteSuccess
```

5.5 Implementação dos testes de aceitação

Com base nos nomes dos passos que foram definidos no diagrama de estados, é preciso realizar o mapeamento daquilo que cada um desses passos abstratos representam dentro do sistema (*step definitions*). Desta forma, é preciso fazer uma relação dos passos com as chamadas a API. Por exemplo, quando temos o passo '*view_incident_details*', é preciso fazer uma requisição para API no endpoint '*/incidents/id*' para recuperar os dados do incidente.

No primeiro incremento, é preciso um esforço de mapear todos os passos e relacioná-los a ações dentro do sistema. Já no segundo incremento, é preciso apenas mapear os novos passos, pois os outros são reutilizados.

5.6 Validação dos casos de teste

Para realizar a validação dos casos, é preciso utilizar a ferramenta para codificar esses *steps definitions*. Para isso foi utilizado a biblioteca *behave* do Python. Aqui segue um exemplo dessa codificação:

```
from behave import *

@when('delete_this_incident')
def step_imp(context):
    incident_id = context.response[GET_ONE][JSON]["id"]
    resposta = context.session.delete(
        f"{API_URL}/incidents/{incident_id}", headers=AUTH
    )
    context.response[DELETE] = {
        STATUS: resposta.status_code,
        JSON: resposta.json(),
        VAR: incident_id,
    }

@then('deleteSuccess')
def step_imp(context):
    assert context.response[DELETE][STATUS] == 200
    assert context.response[DELETE][JSON]["id"] == context.response[DELETE][VAR]
```

No primeiro incremento, foi preciso codificar todos os passos. Porém no segundo incremento, foi somente implementado os novos passos que foram inseridos no modelo.

5.7 Aplicação dos casos de teste

Aqui foram rodados os testes automaticamente com o uso do *behave*, tendo como entrada os casos de testes em Gherkin. Dessa forma, o *behave* faz a leitura dos passos, converte esses passos em ações através do código dos *steps definitions* e os executa. Essa parte é feita uma vez ao final de cada incremento, assim que todos os passos estiverem codificados.

Essa execução vai gerar o resultado dos testes, mostrando quais casos de testes passaram e quais falharam.

Para executar todos os testes, basta rodar o seguinte comando:

```
# na pasta raiz do repositorio
$ behave
```

Para analisar cada um dos conjuntos de casos e seus resultados, foi utilizado o seguinte comando:

```
# na pasta raiz do repositorio
$ behave -i features/{nome_arquivo}.feature > output/{estrategia}_output.txt
```

Desta forma são gerados arquivos de output isolados para cada conjunto de teste. Aqui segue um exemplo do retorno da execução do *behave* para o critério Edge:

```
1 feature passed, 0 failed, 0 skipped
10 scenarios passed, 0 failed, 0 skipped
104 steps passed, 0 failed, 0 skipped, 0 undefined
Took 2m8.213s
```

Ao executar todos os testes, foi obtido o seguinte resultado:

Critério	Passos validados	Passos com falha	Passos pulados	Tempo de execução
Node	35	0	0	47s
Edge	55	0	0	1m8s
EdgePair	104	0	0	2m8s
PrimePath	166	23	247	5m16s

Tabela 3: Tabela de resultado dos testes no primeiro incremento

Critério	Passos validados	Passos com falha	Passos pulados	Tempo de execução
Node	43	0	0	3m21s
Edge	69	0	0	2m12s
EdgePair	159	1	4	4m34s
PrimePath	553	18	142	15m40s

Tabela 4: Tabela de resultado dos testes no segundo incremento

6 Monitorização e observação

Aplicando o MBT, é possível destacar que as etapas de modelagem com os incrementos da funcionalidade produziram um aumento significativo na quantidade de casos de testes e de passos de execução. Entre cada critério de cobertura, há uma diferença também na quantidade de cenários e passos entre eles.

Sobre a execução dos testes, foram executados todos os cenários, no entanto houveram erros durante alguns passos. Esses erros em sua maioria foram encontrados no testes gerados pelo critério PrimePath de cobertura.

7 Discussão

7.1 Sobre o uso de MBT em relação ao BDT

Durante as fases de ilustração e entendimento do problema, o uso do MBT trouxe uma vantagem que é a representação visual. A adição de um recurso visual e abstrato facilita no entendimento do requisito e na identificação de fluxos que devem ser pensados na implementação e teste da feature. No entanto, uma desvantagem é a necessidade de desenvolver essa habilidade para modelar sistemas, porque existe um cuidado principalmente de validar se um modelo está de acordo com os requisitos desejados.

Outro ponto de vantagem identificado pelo MBT, é sua agilidade, praticidade e flexibilidade na geração de cenários de testes com o uso da ferramenta Skyfire. Dentro do desenvolvimento incremental, as modificações no modelo refletiam automaticamente em todos os cenários que precisavam ser validados. Com isso, os cenários contemplavam casos antigos, e que ainda precisam ser testados, como os novos.

Existem também pontos negativos que são a escrita dos cenários em Gherkin. Com o Skyfire, os casos em Gherkin não possuem uma descrição simples do cenário, principalmente para aqueles com grande fluxo de ação. Além disso, em um mesmo cenários, podem aparecer diversas validações (a palavra *'Then'* no Gherkin) no mesmo fluxo. Na questão da implementação de testes de aceitação com o *behave*, o MBT apresenta vantagens pela reutilização de código das *step definitions*. Por isso quando temos um desenvolvimento incremental, basta implementar somente os novos passos e validações.

7.2 Os critérios de cobertura do Skyfire

Durante a fase de desenvolvimento incremental, é possível notar o quanto os critérios de cobertura atendem completamente o modelo. Isso porque com a modificação do modelo, a quantidade de cenários e passos foram alterados para atender a cobertura proposta do modelo. O critério Node gerou menos cenários e com poucos passos. Isso se deve porque esse tipo de critério tenta cobrir todos os estados do modelo. Já o critério Egde, visa cobrir todas as transições do modelo. Como a máquina de estados criada possui poucos estados mas uma grande conexão entre eles, é possivel observar que gerou mais cenários que o critério anterior. O critério EdgePair visa cobrir pares de transições dentro de um mesmo cenário, por isso gerou mais cenários devido a alta possibilidade de combinação de caminhos e mais passos, consequentemente. Por último, o critério PrimePath visa cobrir caminhos que percorram todo o modelo, dessa forma existem diversas combinações para esse critério, além de gerarem os cenários com grande quantidade de passos. Um ponto que merece atenção é a geração de cenários muito complexos, mas que são gerados para atender esse critério de cobertura. Em conjunto a isso, o modelo que foi criado não possui parametros e nem guardas, o que dificulta a modelagem de máquinas de estado com comportamento deterministico. Isso acabou se refletindo na criação de cenários com fluxos infactíveis no comportamento do sistema. Como exemplo disso:

```
Feature: Instatus
```

```
Scenario: initialize_not_empty_list view_incident_details
               get_incidents verify_size_list empty_list
 Given initialize_not_empty_list
 When view_incident_details
 Then incidentActiveSuccess
 When get_incidents
 And verify_size_list
 And empty_list
 Then verifyEmptyList
```

No cenário acima, podemos identificar uma inconsistência em relação a quantidade de incidentes na lista. Por inicializar a lista de incidentes não vazia, fazer operações que não alteram essa quantidade e no final validar se essa mesma lista é vazia, isso gerar um erro de execução no próprio teste. Por isso, apesar da geração de cenários automático, existe o exercício de analisar as regras negociais nesses testes, principalmente devido a falta de informação nos modelos construídos. Nesse caso, esse tipo de testes deveria ser descartado.

7.3 Execução dos testes

Durante a fase de validação, foram identifcadas falhas na execução dentro de conjuntos de testes com diversos passos, principalmente os gerados pelo critério PrimePath. Essas falhas ocorreram por utilizar uma API em um serviço web e não em um ambiente local. Por isso, os testes ficam sujeitos a inconsistências nas respostas, onde algumas podem ter sucesso e outras erros, que podem estar atreladas a situação do servidor, como exceção do limite de tempo de espera e quantidade mensagem para atender. Segue um exemplo encontrado ao rodar um dos testes com falha:

```
features/steps/steps.py:79
      Traceback (most recent call last):
        File "/python3.10/site-packages/behave/model.py",
            line 1329, in run
          match.run(runner.context)
        File "/python3.10/site-packages/behave/matchers.py",
            line 98, in run
          self.func(context, *args, **kwargs)
        File "features/steps/steps.py", line 82, in step_given_init_not_empty
          assert resposta.status_code == 200
      AssertionError
```

No exemplo acima, ao inicializar o cenário com uma lista não vazia de incidentes, houve um erro na requisição para inserir um dado na lista para atender o passo. Esse tipo de situação não é esperado uma resposta diferente do codigo 200, seguindo a documentação da aplicação.

8 Conclusão e trabalhos futuros

Como proposta para esse projeto, foi aplicado o uso da metologia MBT junto de ferramentas como Papyrus, Skyfire[15] e o Behave[16] para o contexto do desenvolvimento ágil de software. O uso de modelos durante esse processo muda o paradigma de documentação de requisitos de uma estória de usuário, mas traz ganhos em algumas etapas, principalmente por automatizar e flexibilizar atividades com alto investimento de tempo dentro do BDT (geração e execução de casos de testes). Ao mesmo tempo, adotar essa nova abordagem significa também adotar novas habilidades e preocupações para o ciclo de desenvolvimento.

Apesar de ser um estudo de caso, esse projeto ainda pode evoluir dentro da codificação dos testes de aceitação, adicionando técnicas de testes como tabela de dados no gherkin. Outro ponto a se melhorar, é utilizar uma API que execute no ambiente local do teste, para obter maior controle sobre o contexto do teste, como acesso ao banco de dados.

Referências

[1] Smart, John. "A day (or a sprint) in the life of a BDD team", *https://johnfergusonsmart.com/329-2/*. (2018). (Accessed: December, 2022)

[2] Silva, Thiago Rocha. "Definition of a behavior-driven model for requirements specification and testing of interactive systems."2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 2016.

[3] Binder, Robert. "Model-based Testing: Taking BDD/ATDD to the Next Level", *https://pt.slideshare.net/robertvbinder/taking-bddtothenextlevel*. Slides from presentation at the Chicago Quality Assurance Association, 2014. (Accessed: December, 2022)

[4] Härlin, Magnus. "Testing and Gherkin in agile projects."Linköping University Department of Computer and Information Science Department of Computer and Information Science (2016).

[5] Apfelbaum, Larry, and John Doyle. "Model based testing."Software quality week conference. 1997.

[6] Sousa, Manuela Maria Ferreira da Costa. Model-Based Testing-Automação de testes com base em modelos. Instuto Superior de Engenharia do Porto. 2018.

[7] Li, Nan, Anthony Escalona, and Tariq Kamal. "Skyfire: Model-based testing with cucumber."2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2016.

[8] Bordeleau, Francis, and Edgard Fiallos. "Model-Based Engineering: A New Era Based on Papyrus and Open Source Tooling."OSS4MDE@ MoDELS. 2014.

[9] Österholm, Viktor. "Overview of Behaviour-Driven Development tools for web applications."Åbo Akademi University Faculty of Science and Engineering Department of Information Technologies (2021).

[10] Zanin, Aline. "Teste baseado em modelos em projetos ágeis, uma abordagem baseada em linguagem de domínio específico."(2019).

[11] Rauf, Irum, et al. "Modeling a composite RESTful web service with UML."Proceedings of the Fourth European Conference on Software Architecture: Companion Volume. 2010.

[12] Instatus, Inc. Instatus API "https://instatus.com/help/api/incidents"(Accessed: December, 2022)

[13] Lima, Washington Luiz da Silva. Aplicando BDD em testes de REST API: uma experiência prática. MS thesis. Universidade Federal do Rio Grande do Norte, 2022.

[14] Juliani, Victor. "Use of Model-Based Testing in combination with Behavior-Driven Development". Universidade estadual de Campinas, Instituto de Computação. 2021.

[15] Koike, Leonardo. Repositorio para uso do Skyfire "https://github.com/leokoike/ready-to-use-skyfire", 2022.

[16] Koike, Leonardo. Repositorio de testes para o sistema Instatus "https://github.com/leokoike/instatus-bdd", 2022.