CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS

Recredenciado pela Portaria Ministerial nº 1.162, de 13/10/16, D.O.U. nº 198, de 14/10/2016
AELBRA EDUCAÇÃO SUPERIOR - GRADUAÇÃO E PÓS-GRADUAÇÃO S.A.

Haniel Zaiine Côrtes

GERAÇÃO DE CENÁRIOS DE TESTE NA SINTAXE GHERKIN A PARTIR DO DIAGRAMA DE ATIVIDADES

Palmas – TO

2021

Haniel Zaiine Côrtes

GERAÇÃO DE CENÁRIOS DE TESTE NA SINTAXE GHERKIN A PARTIR DO DIAGRAMA DE ATIVIDADES

Projeto de Pesquisa elaborado e apresentado como requisito parcial para aprovação na disciplina de Trabalho de Conclusão de Curso I (TCC I) do curso de bacharel em Engenharia de Software pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. Esp. Fábio Castro Araújo.

Palmas – TO

2021

Haniel Zaiine Côrtes

GERAÇÃO DE CENÁRIOS DE TESTE NA SINTAXE GHERKIN A PARTIR DO DIAGRAMA DE ATIVIDADES.

Projeto de Pesquisa elaborado e apresentado como requisito parcial para aprovação na disciplina de Trabalho de Conclusão de Curso 2 (TCC II) do curso de bacharel em Engenharia de Software pelo Centro Universitário Luterano de Palmas (CEULP/ULBRA).

Orientador: Prof. Esp. Fábio Castro Araújo.

Aprovado em: _____/_____/_____

BANCA EXAMINADORA

_____

Prof. Esp. Fabio Castro Araujo

Orientador

Centro Universitário Luterano de Palmas – CEULP

_____

Prof. M.e Fabiano Fagundes

Centro Universitário Luterano de Palmas – CEULP

_____

Prof. Esp. Fernanda Pereira Gomes

Centro Universitário Luterano de Palmas – CEULP

Palmas – TO

2021

AGRADECIMENTOS

Gostaria de agradecer primeiramente à meus pais, Cléia Côrtes dos Reis e José Borges Neto por terem sido meus exemplos de vida, garra e determinação, e principalmente por sempre terem acreditado junto comigo que o sonho de ser Engenheiro de Software era possível. Agradeço também aos meus irmãos Agatha Cristine e Iago Giulio, mesmo mais velho sou quem mais aprendo e me divirto com vocês. É uma honra e alegria sem tamanho poder chamá-los de família.

Também gostaria de agradecer minha namorada Antônia Açucena que sempre se fez presente dos momentos mais tristes aos mais felizes, obrigado por tudo. Não posso esquecer do nosso doguinho Sr. Canino Alfredo, companheiro de todas as horas.

Também queria deixar meus agradecimentos para todos amigos que contribuíram e me apoiaram nessa caminhada, em especial aqui vai um salve para os integrantes do grupo “Dogs” tamo junto felas!.

Meus mais sinceros agradecimentos a todos os professores com quem tive aula, e que me ajudaram na minha formação. Em especial agradecer ao núcleo de computação do CEULP e ao orientador deste trabalho Fábio Castro.

Por fim, mas não menos importante, dedico esse trabalho às mulheres da minha vida, minha mãe Cléia Côrtes dos Reis, que tanto admiro e me inspiro. E à minha falecida avó, Maria das Mercês, de que tanto sinto falta.

RESUMO

CÔRTES, Haniel Zaiine. **Geração de cenários de teste na sintaxe Gherkin a partir do diagrama de atividades.** 2020. Trabalho de Conclusão de Curso (Graduação) – Engenharia de Software, Centro Universitário Luterano de Palmas, Palmas/TO, 2021[1].

Os testes de software são vistos como uma etapa crucial para o desenvolvimento de softwares com qualidade, porém essa etapa está suscetível a erros e falhas humanas. Com isso, a automação de testes busca minimizar essas falhas no processo de desenvolvimento bem como economizar recursos, buscando desenvolver softwares mais confiáveis e com mais qualidade. Nesse contexto, o presente trabalho teve como objetivo o desenvolvimento de um software capaz de gerar cenários de testes a partir de um diagrama de atividades. Para que isso fosse possível foi utilizado a ferramenta para desenhos UML, PlantUML para desenho dos diagramas e a linguagem Python como interpretador dos padrões do diagrama e para passagem de parâmetros e geração dos cenários em Gherkin. Para alcançar os objetivos descritos no trabalho, um conjunto de regras para escrita dos diagramas foi definido. Após isso foi desenvolvido um conjunto de funções em Python que recebe um diagrama de atividades e retorna um arquivo em extensão *.feature* contendo os cenários de testes na sintaxe Gherkin, esse conjunto de funções foi implementado em um servidor *web* django, proporcionando uma *interface* web para a ferramenta desenvolvida.

Palavras-chave: Testes de software, Gherkin, Automação de Testes

[1] Elemento incluído com a finalidade de posterior publicação do resumo na internet. Sua formatação segue a norma ABNT NBR 6023, por isto o alinhamento e o espaçamento diferem do padrão do texto.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Behavior Driven Development - BDD

Unified Modeling Language - UML

Application Programming Interface - API

SUMÁRIO

1 INTRODUÇÃO

A UML (*Unified Modeling Language* ou Linguagem Unificada de Modelagem) é uma linguagem gráfica para visualização, especificação e documentação de softwares. (Booch et al., 2006) a UML

> "proporciona uma forma padrão para preparação de planos e projetos de sistemas, apresentando tanto aspectos conceituais como regras de negócio e funcionalidades do sistema, quanto itens concretos como esquemas de banco de dados e componentes de software reutilizáveis.

Segundo Medeiros (2004), a UML indica formas que podem ser utilizadas para representar um software em diversos estágios de desenvolvimento. Medeiros também afirma que utilizando UML é possível pensar o software em um lugar e codificá-lo em outro. Ou seja, a UML possibilita uma forma de comunicação para desenvolvimento de software que não é um código e essa comunicação é feita a partir de diagramas UML, que são formas de representar uma ideia ou um objeto ou até mesmo um software em UML.

No contexto da Engenharia de Software os diagramas são considerados artefatos de software. Segundo (Booch et al., 2006) Um artefato pode ser visto como qualquer informação gerada, alterada ou utilizada durante o processo de desenvolvimento de software. Dentre esses artefatos pode-se destacar o diagrama de atividades, que é utilizado pela indústria de software para representar a interação dentro das funcionalidades de forma simples e clara (RUMBAUGH et al, 1998).

Durante o processo de concepção de um software vários artefatos são produzidos, desde as fases iniciais na modelagem, até o desenvolvimento e a fase final de testes, porém para manter todos os documentos atualizados há uma demanda de tempo para cada alteração no software que acarreta em revisão da documentação e reaplicação dos testes. Nesse cenário existe uma série de programas que auxiliam na geração de artefatos e no planejamento e execução dos testes.

Alguns exemplos são o PlantUML, que auxilia no desenho dos diagramas criando-os a partir de uma linguagem de texto simples, e o Gherkin, que é uma linguagem natural que busca alinhar o software com as regras do negócio. O Gherkin é um dos elementos principais quando se trata de BDD (*Behavior Driven*

Development) em automação. Sua função é padronizar a forma de descrever especificações de cenários, baseado na regra do negócio. O BDD ajuda as equipes a concentrar seus esforços na identificação, compreensão e construção de recursos valiosos que são importantes para as empresas e garante que esses recursos sejam bem desenhados e bem implementados (SMART, 2014). No quesito automação de testes o Gherkin se destaca pois permite criar cenários de testes legíveis de forma automatizada, economizando recursos e força de trabalho no desenvolvimento de software.

A etapa de testes de software demanda tempo e recursos, por isso muitas empresas de software consideram essa etapa “desejável” ao desenvolver softwares. A automação de testes permite que testes sejam feitos em menos tempo e com menos recursos alocados, evitando retrabalho e esforço desnecessário na equipe.

Segundo Crispin (2009), a automação de testes reduz a probabilidade de erros no desenvolvimento, provê *feedback* cedo e frequente, além de uma excelente documentação. Os testes aplicados da maneira correta podem contribuir significativamente para o sucesso de um projeto de software.

Documentar softwares é uma atividade necessária para o desenvolvimento, porém documentar, testar e manter a documentação atualizada são tarefas que demandam muito tempo. Durante o processo de documentação muitos diagramas são produzidos, entre eles o diagrama de atividades. Dado às informações, o presente trabalho levanta o seguinte questionamento: “Como automatizar a geração de cenários de testes a partir de um diagrama de atividades ?”.

Buscando ampliar a aplicação dessas práticas, principalmente relacionadas ao automação de testes de software esse trabalho tem como objetivo desenvolver um software capaz de transcrever diagramas de atividades escritos pelo PlantUML em cenários de testes na sintaxe gherkin, com o intuito de otimizar o processo de documentação e testes de software relacionados ao diagrama de atividades. Neste sentido, observa-se especificamente: Definir e reconhecer padrões de escrita do diagrama de atividades a partir do PlantUML; Documentar o padrão de escrita dos diagramas de atividade e definir o padrão de saída para os casos de testes em Gherkin.

Tendo em vista esse cenário, é interessante o desenvolvimento de um software que seja capaz de identificar padrões em um artefato de software e gere cenários de testes a partir dele, permitindo que, dessa maneira, as equipes de

desenvolvimento e testes possam desempenhar seu trabalho com mais fluidez e, principalmente, evitando retrabalho.

2 REFERENCIAL TEÓRICO

2.1 TESTES DE SOFTWARE

Utilizando a definição técnica, teste de software é todo e qualquer procedimento que ajuda a determinar se o software atinge as expectativas para as quais foi criado (NETO, 2007).

A construção de softwares é um processo complexo pois varia muito de acordo com as características e dimensões do sistema. Devido a essa particularidade o desenvolvimento de software está sujeito a problemas que acarretam em um produto diferente do esperado ou especificado. Para que esses problemas não permaneçam, ou sejam solucionados, existe uma série de atividades conhecidas como “Verificação, Validação e Testes”.

Verificação e validação são etapas que consistem em identificar se o software está sendo construído da maneira adequada e se atende os requisitos especificados. (DELAMARO, 2013) destaca que o teste de software é uma atividade dinâmica e seu intuito é executar o programa ou modelo utilizando algumas entradas em particular e verificar se seu comportamento está de acordo com o esperado.

Essas etapas de testes são compostas por diversos elementos que auxiliam na formação e desempenho dessa atividade e, dentre esses elementos, pode-se destacar os casos de teste que segundo Craig e Jaskiel (2002) descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado, em outras palavras um caso de teste pode ser considerado um conjunto de condições utilizadas para responder algumas questões como “como irei testar?”, “o que será avaliado para que o software seja considerado bom ou ruim?”.

Outro elemento importante que deve ser destacado quando se fala em testes de software são os cenários de teste. O cenário de teste é um documento que descreve estórias (situações de teste) que descrevem o objetivo de um usuário ao

utilizar uma determinada funcionalidade do software e ajudam no trabalho do testador no momento de execução destes testes (UFPR, 2020). Cenários de testes também servem como uma documentação menos detalhada de um software.

Os cenários de teste apresentam o comportamento esperado do software e são estruturados seguindo o padrão Contexto → Ação → Resultado escritos em um formato especial chamado *Gherkin*. O *Gherkin* serve tanto como especificação quanto como entrada em testes automatizados, daí o nome “Especificações Executáveis”(BASE2, 2020).

2.2 BEHAVIOR DRIVEN DEVELOPMENT

O desenvolvimento orientado por comportamento (*Behaviour-Driven Development* - BDD) ajuda as equipes a concentrar seus esforços na identificação, compreensão e construção de softwares valiosos para as empresas e garante que esses softwares sejam bem desenhados e bem implementados (SMART,2014). O BDD concentra o trabalho em exemplos do mundo real (cenários) que ilustram como o sistema irá se comportar, esses exemplos servem como guia tanto para o conceito quanto para o desenvolvimento do software em um processo de integração contínua (CUCUMBER,2020). Esse modelo de desenvolvimento incentiva a equipe de desenvolvimento a expressar os requisitos de uma forma mais testável e legível, fazendo com que tanto a equipe quanto as partes interessadas possam entender facilmente o escopo do projeto (SMART,2014).

O BDD incentiva o trabalho em interações curtas e rápidas e, para isso, aborda os problemas dividindo-os em pequenas partes. O processo do BDD é composto essencialmente por três etapas (CUCUMBER,2020):

- *Discovery* ou Descoberta: o BDD utiliza de conversas estruturadas conhecidas como *discovery workshops* que focam em extrair exemplos do mundo real do sistema a partir da perspectiva do usuário. Essas conversas aumentam a compreensão da equipe de desenvolvimento quanto às reais necessidades do usuário, as regras que regem o sistema e o que realmente deve ser feito;
- *Formulation* ou Formulação: assim que os exemplos concretos são identificados na etapa de descoberta, uma documentação estruturada pode ser gerada, isso faz com que haja uma confirmação rápida de que a equipe tem um entendimento

comum sobre o projeto. Em BDD é utilizada uma linguagem comum a humanos e computadores. Ao escrever essa documentação é estabelecido uma linguagem compartilhada para falar sobre o sistema, isso ajuda a utilizar e abordar todos os temas a respeito do software;

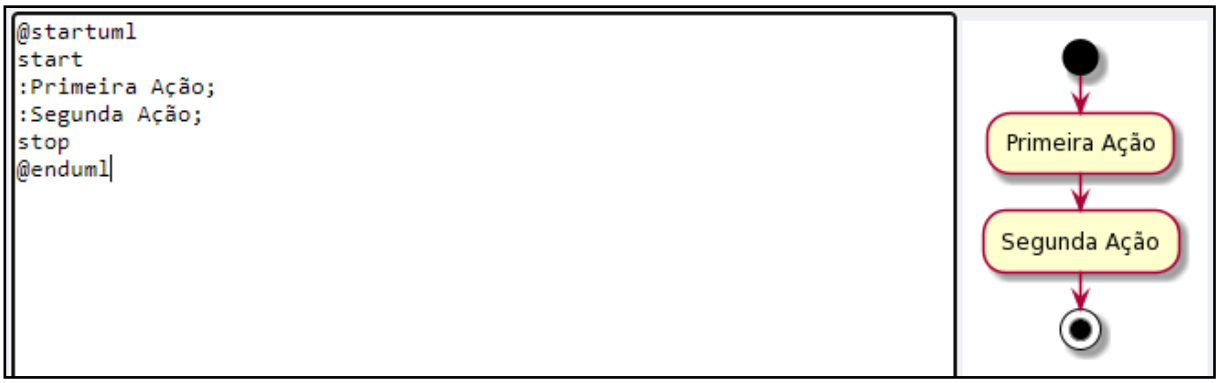
- *Automation* ou Automação: com as especificações prontas durante a formulação, na etapa de automação serão utilizadas como guia para a implementação do software. Um a um os exemplos se conectam ao sistema como um teste. Os exemplos automatizados funcionam como guias, isso ajuda a manter nosso trabalho de desenvolvimento no caminho certo. Além disso, os exemplos automatizados auxiliam na manutenção do código posteriormente, pois como descrevem o comportamento do sistema no momento simplifica o processo de realizar alterações com segurança.

2.3 PLANTUML

O PlantUML é usado para desenhar diagramas UML, usando uma descrição de texto simples e legível por humanos (PLANTUML, 2020). Dentre os diagramas possíveis pode-se destacar o diagrama de atividades.

Para desenhar um diagrama de atividades utilizando PlantUML o documento deve iniciar com a sintaxe @startuml e finalizar com @enduml. Todas as ações são descritas iniciando a ação com dois pontos e finalizando-a com ponto e vírgula. através da sintaxe :<NomeDaAção>; também é possível adicionar os estados iniciais e finais utilizando as palavras chave *start e stop* (PlantUML,2020).

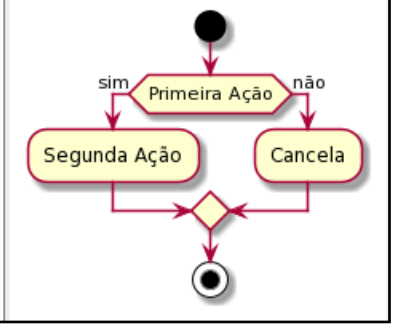
Figura 1 - Exemplo de utilização do PlantUML



A utilização de condicionais permite definir o fluxo de funcionamento do diagrama e seus desvios, no PlantUML são utilizadas a palavras-chave *if* (se) e *then* (então) para montar o condicional, e também é possível adicionar o comando *else* (senão):

Figura 2 - Exemplo de utilização de condicionais no diagrama de atividades em PlantUML

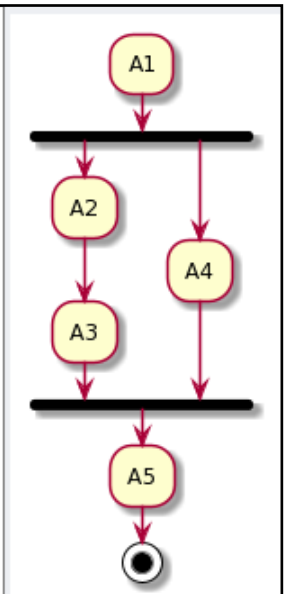
```
@startuml
start
if (Primeira Ação) then (sim)
    :Segunda Ação;
else (não)
    :Cancela;
endif
stop
@enduml
```



No PlantUML também é possível desenhar diagramas de atividades com bifurcações e uniões.

Figura 3 - Exemplo de utilização do *fork e join* no diagrama de atividades em PlantUML

```
@startuml
:A1;
 fork
   :A2;
   :A3;
 fork again
   :A4;
 endfork
 :A5;
 stop
@enduml
```



Com isso pode-se perceber o poder de desenho que o PlantUML possui. Recentemente sua versão foi atualizada trazendo algumas melhorias no processo de desenhos e organização dos diagramas tornando-os mais legíveis e funcionais.

2.4 DIAGRAMA DE ATIVIDADES

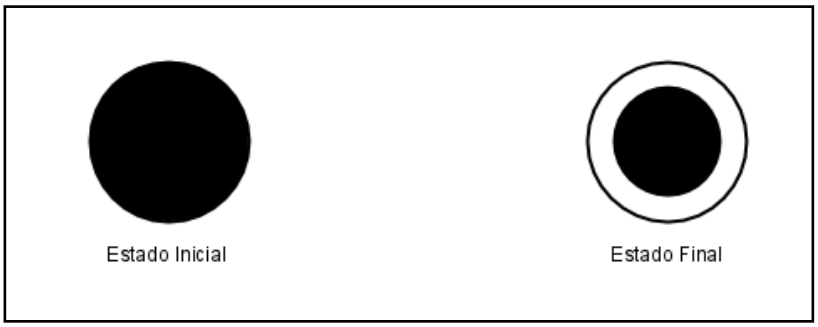
No UML um diagrama de atividade fornece uma visualização do comportamento de um sistema descrevendo a sequência de ações em um processo sendo que os diagramas de atividades também podem mostrar fluxos paralelos ou simultâneos e fluxos alternativos (IBM, 2020). Uma atividade é modelada como uma

sequência estruturada de ações, controladas potencialmente por nós de decisão e sincronismo (GUDWIN, 2016).

O diagrama de atividades é comumente composto pelos seguintes elementos básicos:

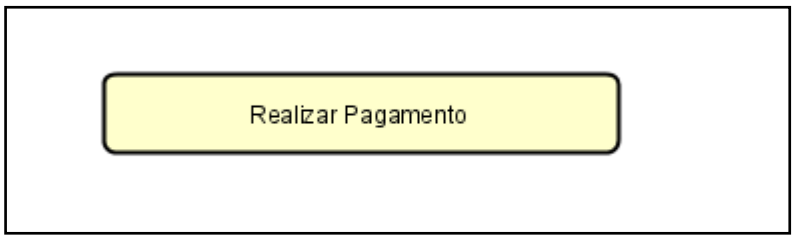
- Estados iniciais e finais: os diagramas de atividades possuem pelo menos um estado inicial e um final(Podem haver vários). O Estado inicial indica o início da atividade e o Estado final indica seu fim.

Figura 4 - Demonstração do estado inicial e final no diagrama de atividades



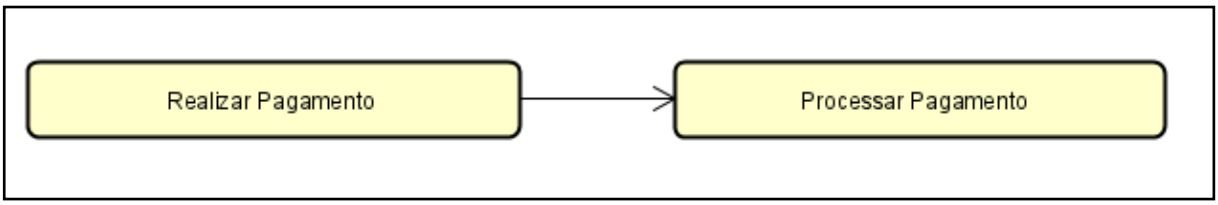
- Atividade: atividades são elementos de contêiner que descrevem o nível mais alto do comportamento em um diagrama de atividades.(IBM,2020) As atividades descrevem o comportamento que deve ser realizado.
- Ação: ação representa uma unidade discreta de funcionalidade em uma atividade. (IBM,2020) Uma Ação é representada por um retângulo com bordas arredondadas, uma ação representa um passo a ser executado para que a atividade seja concluída. Quando a execução de uma ação é encerrada ela transfere a execução para a próxima ação(transição).

Figura 5 - Exemplo atividade



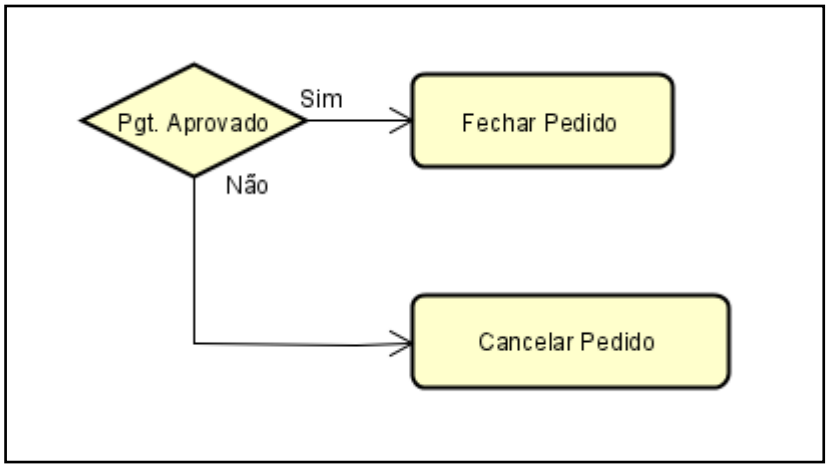
- Transições: representadas por uma seta contínua que indica o fluxo de execução da atividade e o caminho a ser seguido para sua conclusão.

Figura 6 - Exemplo transição



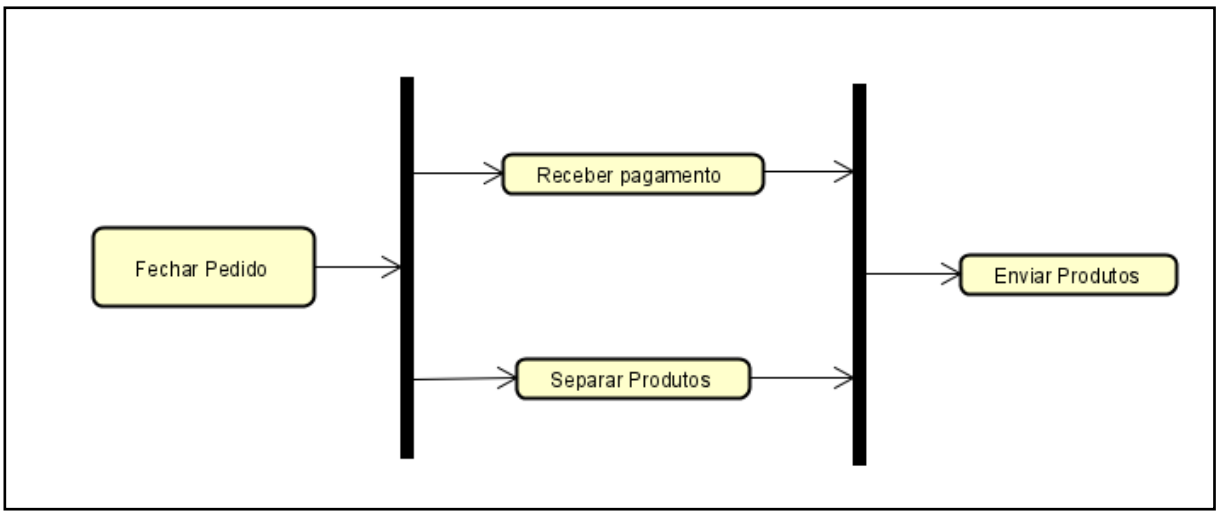
- Decisões: representadas por Losangos, são utilizadas para controlar os desvios de fluxo, decisões são comumente aplicadas quando se tem uma questão condicional em dependendo de uma condição o fluxo segue para um lado ou para outro.

Figura 7 - Exemplo condicional



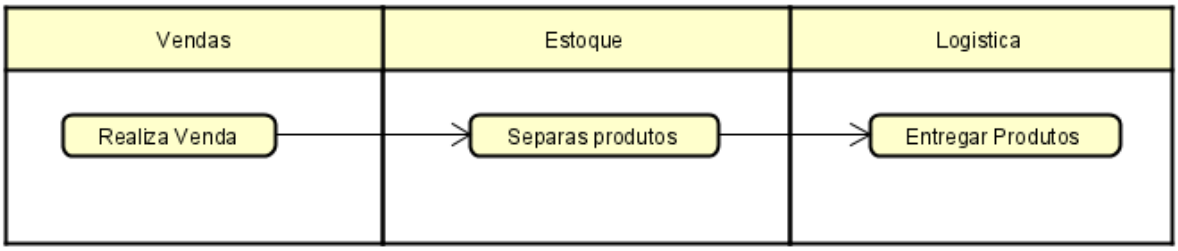
- Bifurcação e União: representado por uma barra sólida que pode estar tanto na horizontal quanto na vertical usado para representar atividades paralelas ou concorrentes.

Figura 8 - Exemplo bifurcação e união (*fork and join*)



- Raias: elemento utilizado para demonstrar como as atividades percorrem diversos agentes dentro de um mesmo processo, cada raia recebe o nome do agente que representa.

Figura 8 - Exemplo raias



Com isso o diagrama de atividades se mostra uma documentação crucial para determinados cenários e escopos de software, pois por apresentar de forma visual o comportamento e o fluxo de funcionamento de atividades do sistema, além de trazer muito valor agregado possibilita a integração com outros cenários como por exemplo com BDD e Gherkin.

2.5 GHERKIN

"Gherkin é um conjunto de regras gramaticais que torna o texto simples e estruturado" (GHERKIN, 2020). Por ser uma linguagem natural o gherkin permite que diversos integrantes do projeto possam entender dessa forma o processo fica mais simples pois todas as partes interessadas podem identificar o que o software fará. Gherkin é comumente utilizado fornecendo bases para testes automatizados e para documentar como um sistema realmente se comporta.

A estrutura do Gherkin é composta pelas seguintes palavras-chave e padrões:

- *Feature* ou Funcionalidade: o objetivo da *Feature* é fornecer uma descrição de alto nível de uma funcionalidade do software e agrupar seus cenários relacionados.(GHERKIN, 2020). A primeira palavra chave de um documento gherkin deve ser *"Feature"* seguido por dois pontos ":" e um texto curto que descreve a funcionalidade.
- *Example* ou Cenário: este é um *exemplo concreto* que *ilustra* uma regra de negócios. Ele consiste em uma lista de etapas. (GHERKIN, 2020) Além de ser uma especificação e documentação, um exemplo também é um teste. Os Cenários seguem sempre o mesmo padrão:

- Descreve-se o contexto inicial ou pré requisito: Representado pela palavra-chave *Given* ou Dado. O objetivo dessa etapa é colocar o sistema em um estado conhecido antes que o usuário comece a interagir com ele. (GHERKIN,2020)
- Descreve-se um evento: Representado pela palavra-chave *When* ou Quando. Etapa que tem como objetivo descrever um evento ou ação, pode tanto ser do usuário ou de um outro sistema.(GHERKIN,2020)
- Descreva o resultado esperado: Representado pela palavra-chave *Then* ou Então. É utilizado para representar ou descrever o resultado esperado da etapa deve usar uma afirmação para comparar o resultado real (o que o sistema realmente faz) com o resultado esperado (o que a etapa diz que o sistema deve fazer). (GHERKIN,2020)

- *Background* ou Fundo: o Fundo ou *Background* permite adicionar algum contexto aos cenários que o seguem, ele pode conter uma ou mais *Given*, que são executadas antes de cada cenário. (GHERKIN,2020)
- *Scenarios Outlines* ou Esquema do cenário: os Esquemas de cenários são usados para executar os mesmos cenários várias vezes, com diferentes combinações de valores.(GHERKIN,2020)

Exemplo da estrutura de um arquivo *.feature:

Figura 9 - Exemplo arquivo *.feature

```
1  Cenário: Haniel posta foto no Instagram
2    Dado que eu estou logado como Haniel
3    Quando eu tento postar a foto
4    Então eu devo ver "Sua foto foi publicada."
```

O Gherkin é um dos principais elementos quando se trata de *BDD* pois auxilia na padronização dos cenários de testes, o que facilita todo o processo de concepção e desenvolvimento de um software utilizando o *BDD*.

3 METODOLOGIA

Nesta seção encontram-se dados relacionados aos materiais e métodos que serão utilizados para o desenvolvimento do trabalho.

3.1 MATERIAIS:

Para a realização do trabalho foram utilizadas: a linguagem Python, o PlantUML e o Visual Studio Code.

Python é uma linguagem de programação interpretada, interativa e orientada a objetos. Ele incorpora módulos, exceções, tipagem dinâmica, tipos de dados dinâmicos de nível muito alto e classes (Python, 2020). Python será utilizado como linguagem para codificação do software a escolha dessa tecnologia foi devido ao seu baixo grau de abstração e por apresentar uma estrutura de código mais “limpa” facilitando a leitura e manutenção do código.

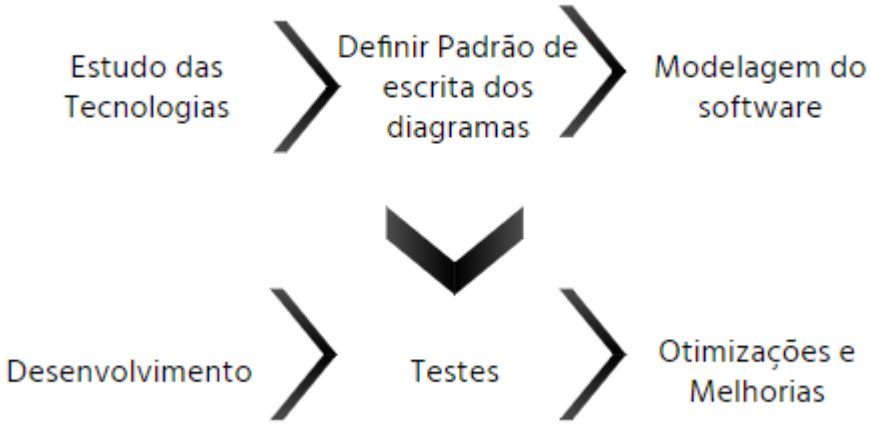
PlantUML é usado para *desenhar* diagramas UML, usando uma descrição de texto simples e legível por humanos (PLANTUML, 2020). O PlantUML utiliza-se de uma notação própria que, através de textos, torna possível desenhar diagramas UML. O PlantUML será utilizado para descrição e desenho dos diagramas de atividades, a partir desses diagramas desenhados no PlantUML será possível gerar cenários de teste em Gherkin.

O Visual Studio Code é um editor de código simplificado com suporte para operações de desenvolvimento como depuração, execução de tarefas e controle de versão (Visual Studio, 2020). Por ter uma boa performance e uma interface limpa o VS Code foi o editor de código escolhido para o projeto, além de possuir algumas ferramentas que auxiliam no desenvolvimento e aumentam a produtividade.

3.2 MÉTODOS:

Nesta seção estão descritas as etapas referentes ao desenvolvimento do trabalho.

Figura 10 - Etapas do desenvolvimento do trabalho



A primeira etapa do projeto consistiu no estudo das tecnologias envolvidas para conseguir aplicar e utilizar da melhor maneira possível as ferramentas envolvidas no projeto.

Na segunda etapa foi definido um padrão de desenho dos diagramas de atividades no PlantUML para que seja possível identificar os elementos e criar cenários de testes em gherkin da melhor maneira possível. Foi levado em consideração as particularidades do diagrama de atividades para montar uma estrutura que possa associar os elementos do diagrama com os casos de teste.

Na etapa de modelagem do software a ser desenvolvido, foi descrito como o software iria se comportar, quais suas entradas, suas saídas e o que ele iria fazer com essas entradas e como isso seria feito computacionalmente.

No desenvolvimento, o produto começa a ser construído de fato, onde os artefatos de software como diagramas foram transcritos para códigos Python no editor de código Visual Studio Code.

Na etapa de testes nenhuma metodologia formal foi utilizada, porém testes de caixa preta foram realizados buscando a validação dos resultados obtidos (Cenários de testes) com os resultados esperados. Após isso, analisando os resultados obtidos alguns ajustes foram aplicados dentre esses : lógica de execução do software, *layout* do arquivo .feature, formato de exportação do arquivo.

4 RESULTADOS

Esta seção tem por objetivo descrever e apresentar o fluxo do trabalho realizado para o desenvolvimento da ferramenta. Nesta seção é descrita a padronização da escrita dos diagramas, a modelagem e a implementação do software.

4.2 ESCRITA DOS DIAGRAMAS

Para que fosse possível gerar casos de testes em Gherkin a partir de um diagrama de atividades, foi preciso num estágio inicial definir o padrão de escrita dos diagramas na linguagem PlantUML, para que o software pudesse reconhecer o padrão de escrita. Na tabela abaixo estão destacadas as regras que foram criadas e servem de referência para a escrita dos diagramas em PlantUML.

Tabela 01 - Relação de regras para escrita dos diagramas

Regra	Descrição
As atividades do diagrama devem conter algumas das tags da sintaxe Gherkin.	As tags da sintaxe Gherkin (Dado, Quando, Então, E) devem estar destacadas no início das atividades. (Diagrama Exemplo Regra 01)
O título da feature deverá estar entre a tag 'floating note'	O título da feature deve estar entre a sintaxe plantUML 'floating note'
O título da feature deve estar exatamente abaixo da tag 'floating note'	O título da feature deve ser a primeira informação do diagrama.
A rule da feature deverá estar entre a tag 'floating note'	Caso o cenário de testes possua Rules (Regras), elas devem ficar destacadas entre a *tag* plantUML 'float note'. (Diagrama Exemplo Regra 02)

A tabela 01, descreve como cada regra deve ser aplicada ao escrever diagramas de atividades utilizando o PlantUML para gerar cenários de testes na sintaxe Gherkin. A primeira regra define que para gerar os cenários de teste na sintaxe gherkin é preciso que as *tags* da sintaxe estejam presentes nas atividades do diagrama de atividades. Também é definido que o título da *feature* deve estar entre a *tag* do PlantUML *'floating note'* e deve ser exatamente a primeira informação dentro da *tag*. Por fim, a última regra imposta é que para os casos onde existe uma *Rule* no cenário de teste ela deve ser destacada também dentro da tag '*floating note'*.

O diagrama descrito na figura 11, exemplifica a utilização das regras descritas para a escrita dos diagramas de atividade.

Figura 11 - Diagrama exemplo 01

A figura 11, apresenta dois itens. O item 1 apresenta o código escrito na notação do plantuml para gerar o diagrama de atividades apresentado no item 2.

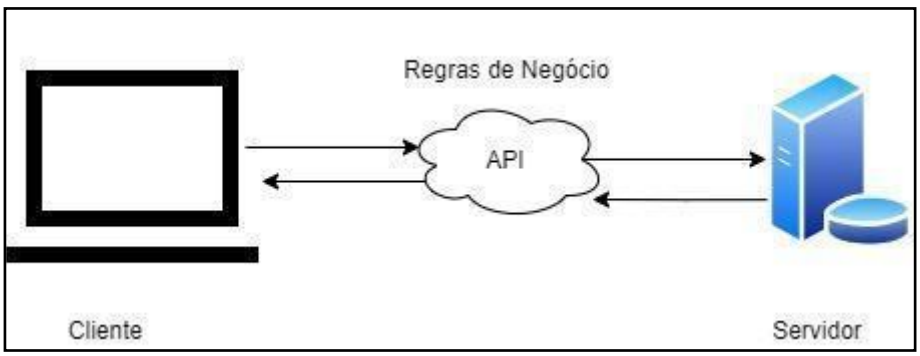
A figura 11 possibilita perceber e entender onde as regras estão sendo aplicadas no diagrama. Cada atividade carrega consigo a *tag* da sintaxe Gherkin. (Dado, Quando, Então). Além disso, o título da *feature* está entre a *tag* 'floating note'.

```
@startuml
floating note
    Postar foto no Instagram
end note
start
    :Dado que estou logado;
    :Quando eu tento postar uma foto;
    :Então devo ver "Sua foto foi publicada";
end
@enduml
```

4.3 MODELAGEM DO SISTEMA

O software foi desenvolvido baseado na arquitetura cliente-servidor. Os clientes se comunicarão com o servidor através de requisições HTTP (*Hypertext Transfer Protocol*) feitas para uma API (*Application Programming Interface*) que foi desenvolvida para mediar essa comunicação e para hospedar as regras de negócio do software como identificar erros de escrita dos diagramas, processar diagramas, e etc.

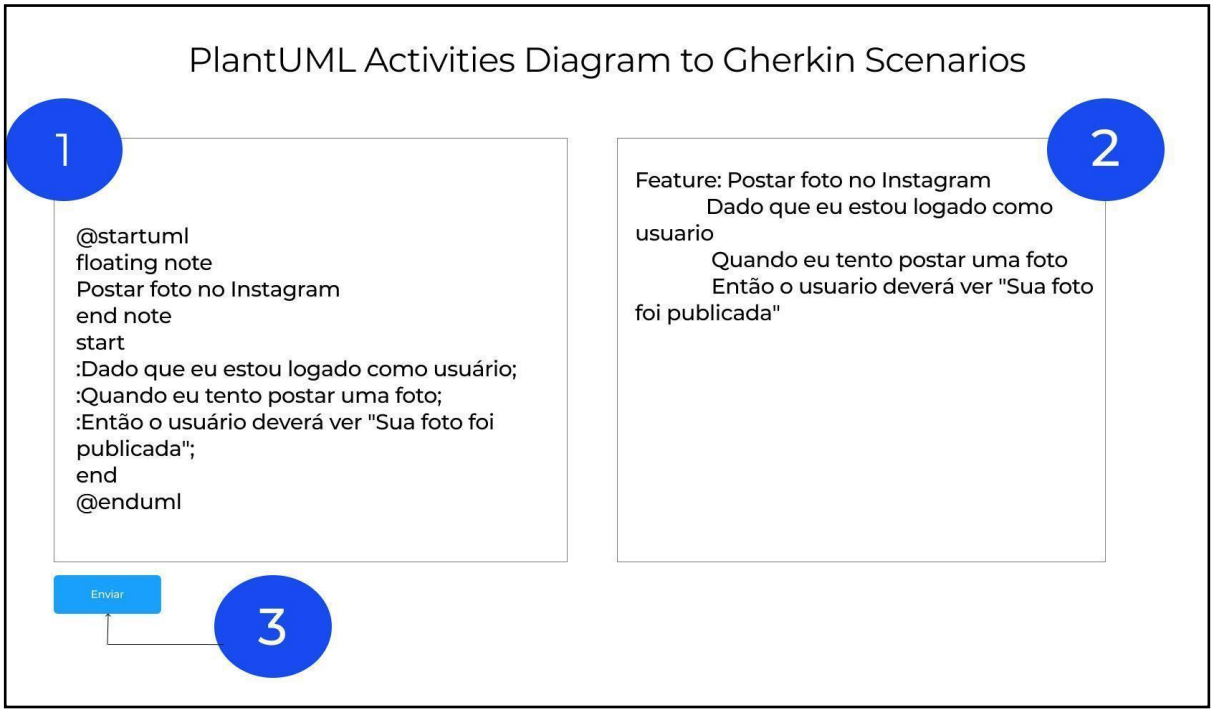
Figura 12 - Representação da arquitetura cliente-servidor



A escolha dessa arquitetura foi devido à característica de descentralização e escalabilidade que ela possui. Isso pode possibilitar que o software seja mais

acessível a um número maior de clientes ao mesmo tempo, além de facilitar quesitos como manutenção, evolução e até mesmo a distribuição do software.

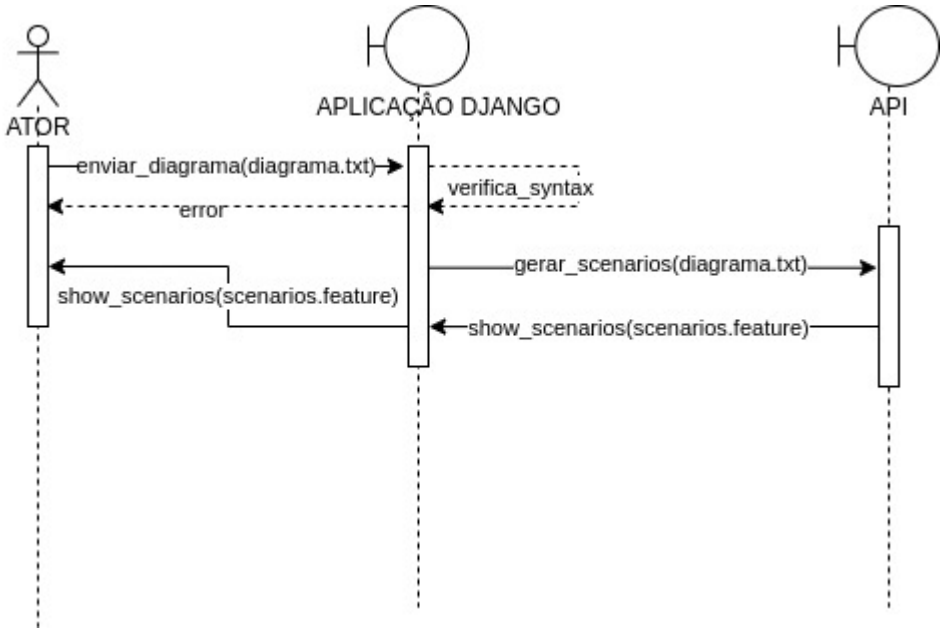
Figura 13 - Esboço do software e suas funcionalidades



A figura 13 representa a tela principal do software. O item 1 em destaque corresponde ao local onde o código fonte do diagrama escrito na notação do PlantUML será inserido. O item 2 exibe o resultado (um cenário na sintaxe Gherkin) gerado a partir do diagrama inserido na seção 1. Por fim, o item 3 representa o botão que será utilizado para enviar o diagrama do item 1 para processamento. Os resultados apresentados na seção 2, são exibidos somente após o usuário clicar no item 3 “Enviar”.

O diagrama de sequência abaixo representa o fluxo de funcionamento do software e apresenta a sequência de passos ou métodos a serem executados para gerar os cenários de teste. No diagrama tem-se o ator usuário e duas outras *lifelines* (Linhas de vida) sendo que uma representa a API responsável por fazer a manipulação e troca de informações e mensagens e a outra representa o servidor onde essa API será hospedada e funcionará.

Figura 14 - Diagrama de Sequência 01



Na figura 14, o usuário envia o código fonte do diagrama de atividades na notação do PlantUML. A *API* verifica se existem erros de sintaxe e se o diagrama enviado respeita as regras propostas. Caso exista um erro, retorna uma mensagem para o usuário, caso não identifique erros, envia o diagrama para processamento. Por fim, o software retorna os cenários de teste escritos na linguagem Gherkin para o usuário.

4.4 IMPLEMENTAÇÃO

Nesta seção é descrita a implementação do software. O software foi desenvolvido utilizando a linguagem de programação Python na versão 3.9, sem a utilização de bibliotecas ou extensões adicionais. Como ambiente de desenvolvimento foi utilizado o Visual Studio Code. Para gerar os diagramas foi escrito as seguintes funções:

Tabela 02 - Relação de funções implementadas

Função	Descrição
`read_diagram()`	Lê o arquivo .txt .
`split_lines()`	Separa o arquivo .txt por quebra de linha.

`check_conditional_exist()`	Verifica se existem condicionais no diagrama.
`get_feature_name()`	Obtém o nome da funcionalidade.
`remove_feature_tittle()`	Remove o nome da funcionalidade do texto.
`remove_syntax_tags()`	Remove sintaxe do plantUML.
`generate_feature()`	Gera arquivo .feature para casos onde não existe condicional.
`format_scenario_line()`	Formata as linhas escritas no arquivo .feature
`clean_scenario_name()`	Formata as linhas escritas no nome do cenário.
`check_conditionals()`	Gera dicionário {condicional:atividades}
`generate_feature_conditional()`	Gera arquivo .feature para casos onde existem condicionais.
`check_given_before()`	Verifica os dados ou atividades executadas antes do condicional.
`remove_givens()`	Remove atividades executadas antes do condicional.
`generate_scenario()`	Função que centraliza toda operação, recebe um arquivo .txt e chama as outras funções descritas acima para gerar um cenário de teste.

Dentre a lista de funções apresentadas acima pode-se destacar a função `generate_scenario()`. Essa função é imprescindível para o software, pois ela é responsável por centralizar a geração de cenários de teste em Gherkin.

Figura 15 - Função `generate_scenario()`

```
149     def generate_scenario(text_diagram):
150         feature_name = get_feature_name(text_diagram)
151         remove_feature_tittle(feature_name,text_diagram)
152         remove_syntax_tags(text_diagram)
153         if check_conditional_exist(text_diagram):
154             gherkin_givens = check_given_before(text_diagram)
155             remove_givens(gherkin_givens, text_diagram)
156             scenarios_dict = check_conditionals(text_diagram)
157             generate_feature_conditional(feature_name,scenarios_dict, gherkin_givens)
158         else:
159             generate_feature(feature_name,text_diagram)
```

Essa função recebe como parâmetro o diagrama de atividades no formato de texto. A primeira etapa da função é recuperar o nome da feature. Para isso é

executada a chamada da função `get_feature_name()` e passa como parâmetro o texto do diagrama.

Após obter o nome da feature é executada a chamada da função *remove_feature_tittle(),* uma vez armazenada o título da feature, essa função remove os valores e tags correspondentes do texto do diagrama escrito em plantUML.

O próximo passo da função `generate_scenario()` é remover os caracteres referentes à sintaxe do plantUML. Isso é feito por meio da função `remove_syntax_tags()`, *q*ue declara um conjunto de tags referente à sintaxe do plantUML e busca por elas no texto do diagrama. Assim que encontra as remove.

Figura 16 - Função *`remove_syntax_tags()`*

```
53    def remove_syntax_tags(text_diagram):
54        # Remove plantUML syntax tags from file
55        tags = ['(', ')', 'if', 'then', 'yes', 'else', 'elseif', '@startuml', 'floating note',
56            'end note', 'start', 'end', 'endif', '@enduml','stop']
57
58        for tag in tags:
59
60            if tag in text_diagram:
61                text_diagram.remove(tag)
62        return (text_diagram)
```

Após remover as tags da sintaxe do plantUML é executada a verificação se o diagrama de atividades possui condicionais. A função `check_conditional_exist()` é responsável por essa verificação. Caso existam condicionais a função retorna *`True`*, senão retorna `False`.

Caso exista a *tag* `if` no início de alguma linha do diagrama, é um sinal de que existem condicionais, então o fluxo para diagramas com condicionais é executado.

Figura 17 - Função *`check_conditionals()`*

```
90    def check_conditionals(text_diagram): #R
91        '''
92        This function checks whether the diagram has conditionals, and groups the flow paths
93        In the structure: {conditional:scenario}
94        '''
95        scenarios_dict = {}
96        main_key = ''
97
98        for i in text_diagram:
99
100           if i.startswith('if'):
101               scenario_key = clean_scenario_name(i)
102               scenarios_dict[scenario_key] = []
103               # Setting main key to dict(conditional reason).
104               main_key = scenario_key
105
106           elif i.startswith('else') or i.startswith('elseif'):
107               scenario_key = clean_scenario_name(i)
108               scenarios_dict[scenario_key] = []
109
110           scenarios_dict[scenario_key].append(i)
111
112       return scenarios_dict
```

A função `check_conditionals()` percorre o texto do diagrama e identifica tags que são referentes a condicionais e as armazena como chaves de dicionário e o fluxo de atividades correspondente a esse condicional como valores da chave, seguindo o padrão {condicional: cenário}.

Por fim é chamada a função intitulada `generate_feature_conditional()` que é responsável por exportar os cenários para um arquivo de texto (.feature). A função utiliza a manipulação de arquivos do Python para abrir um arquivo em branco chamado "*gherkin_scenarios.feature*" e escreve as atividades do diagrama seguindo o padrão e formatando para se adequar à sintaxe Gherkin.

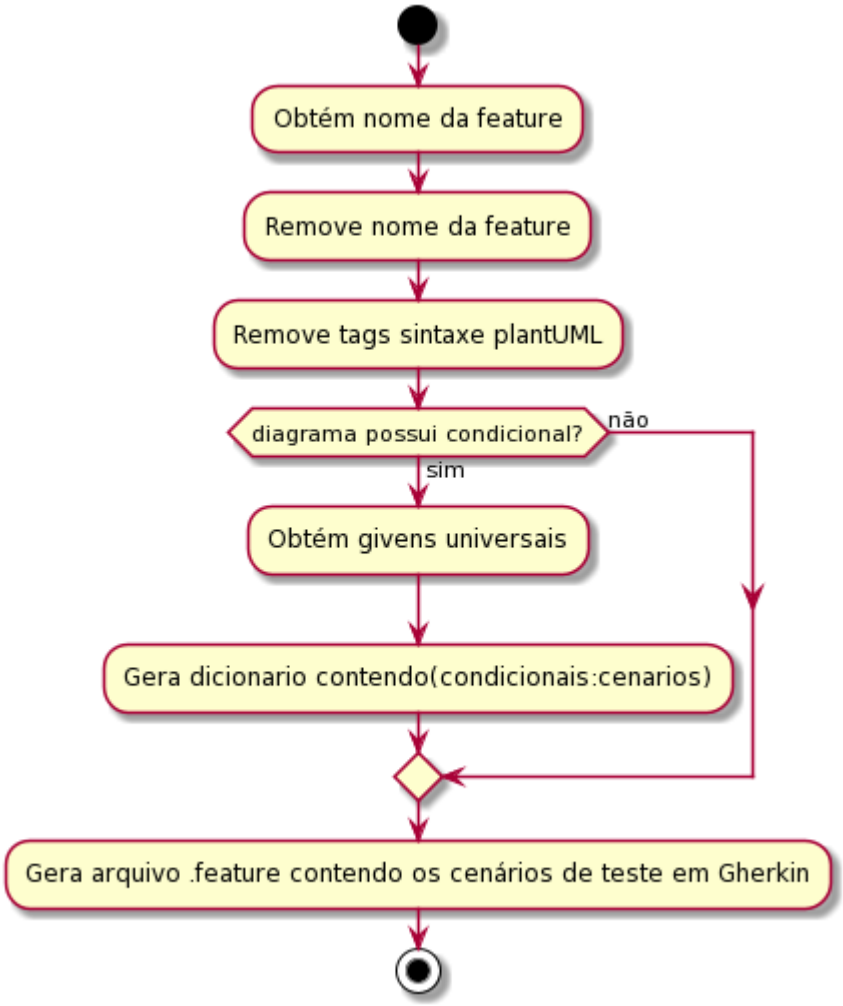
Figura 18 - Função *`generate_feature_conditional()`*

```
114     def generate_feature_conditional(tittle, scenarios_dict, givens): #R
115
116         f = open("gherkin_scenarios.feature", "w", encoding='utf-8')
117         f.write('# language: pt'+'\n')
118         f.write(tittle+'\n')
119         for key in scenarios_dict.keys():
120             f.write("Cenário: "+key+'\n')
121             for _ in givens:
122                 f.write("\t"+format_scenario_line(_)+'\n')
123             for _ in scenarios_dict[key]:
124                 if _ == scenarios_dict[key][0]:
125                     f.write("\t"+"Quando "+format_scenario_line(_)+'\n')
126                 else:
127                     f.write("\t"+format_scenario_line(_)+'\n')
128             f.write('\n')
129         f.close()
```

Com isso, chega-se ao fim o fluxo de geração de cenários para os casos em que existem condicionais no diagrama. Para os casos onde não existe condicional no diagrama a geração é feita pela função `generate_feature()`.

O diagrama de atividades a seguir apresenta o fluxo de execução do software para transcrever diagramas de atividade escritos no plantUML em cenários de testes na sintaxe Gherkin.

Figura 19 - Diagrama de atividades - Fluxo de execução do software



A primeira atividade executada pelo algoritmo é obter o nome da feature, após obter o nome o algoritmo remove o trecho do código correspondente. Após isso remove as *tags* referentes à sintaxe do plantUML. Feito isso, o algoritmo verifica se o diagrama possui condicionais com uma busca pela *tag* "*if",* caso não exista gera o arquivo .feature. Porém, caso existam condicionais, as atividades antes dele são consideradas *Givens* e o algoritmo mapeia o condicional gerando um dicionário que possui como chaves os condicionais e como valor as atividades correspondentes.

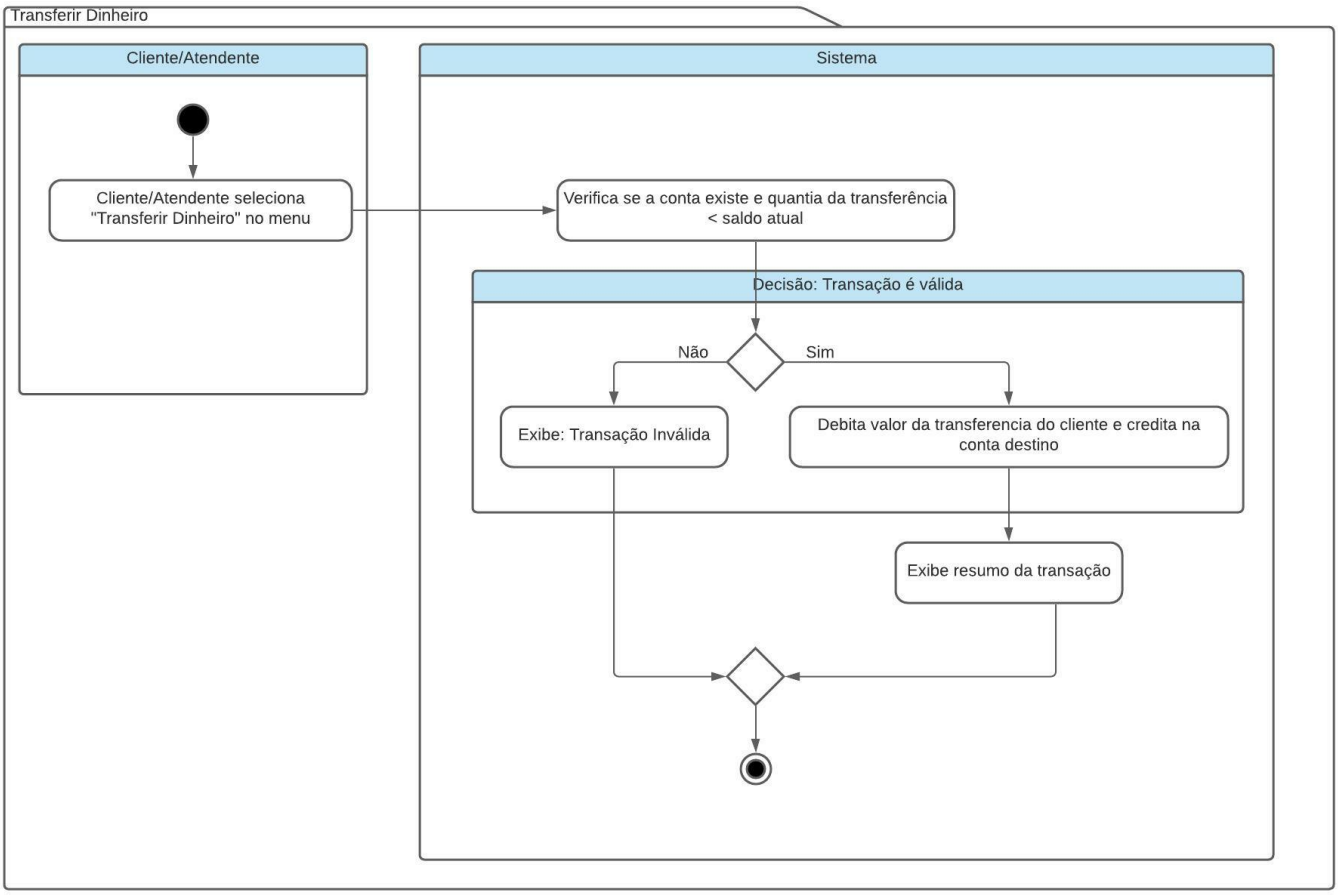
4.5 APLICAÇÃO EM UM EXEMPLO

Para exemplificação dos resultados obtidos pelo software desenvolvido foi utilizado um diagrama de atividades retirado da documentação do ambiente IBM *Rational Software Architect.*

Este exemplo de diagrama de atividade é obtido da amostra do diagrama de atividade *PiggyBank*, um sistema bancário modelado pela IBM para fins educacionais a respeito de modelagem de software. O caso de uso *TransferMoney*

descreve as etapas que ocorrem quando um usuário transfere dinheiro de uma conta para outra (IBM, 2021).

Figura 20 - Diagrama Exemplo IBM



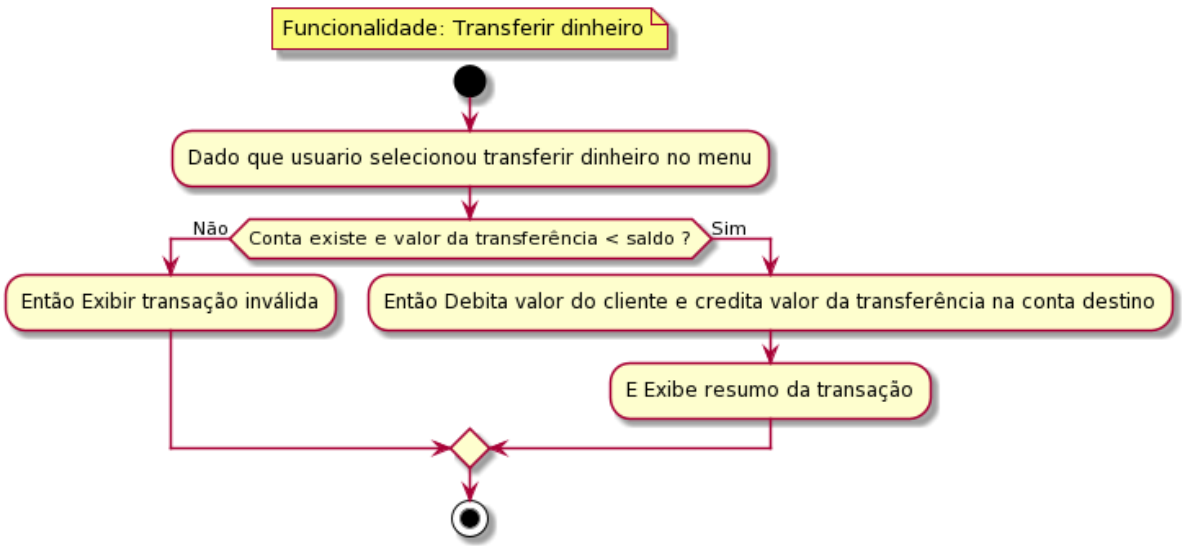
O diagrama de atividades (Figura 20) apresenta um exemplo de uma ação para transferir dinheiro. A atividade inicia com o cliente selecionando transferir dinheiro no menu do sistema *PiggyBank*. Após isso o sistema do banco valida se a conta destino existe e se o saldo do cliente é maior que o valor da transferência. Se essas duas premissas forem verdadeiras então o sistema debita do cliente o valor da transferência e credita na conta destino e após isso exibe o resumo da transferência. Se a premissa for falsa, ou seja, ou a conta destino não existe ou o saldo atual do cliente é menor que o valor da transferência, o sistema exibe que o método de transação é inválido.

Para que o software pudesse transcrever o diagrama, foi preciso escrevê-lo utilizando a sintaxe PlantUML e aplicar as regras definidas por este trabalho para a escrita dos diagramas. resultado é apresentado no diagrama (Figura 21).

Figura 21 - Diagrama Exemplo IBM em notação PlantUML

A figura (Figura 21) apresenta o resultado após escrever o diagrama de atividades para a funcionalidade Transferir dinheiro. O formato visual desse diagrama está disposto na imagem abaixo.

Figura 22 - Diagrama Exemplo IBM em PlantUML



Após a transcrição do diagrama de atividades exemplo da Figura 20 para um modelo do plantUML o software pode reconhecer, identificar os padrões do diagrama e gerar cenários de teste.

Após submeter o diagrama da figura acima no software, foi exportado o arquivo *gherkin_scenarios.feature* contendo o seguinte conteúdo:

Figura 23 - Arquivo de saída *.feature*

```
# language: pt
Funcionalidade: Transferir dinheiro
Cenário:  Conta existe e valor da transferência < saldo ?  Não
    Dado que usuario selecionou transferir dinheiro no menu
    Quando if (Conta existe e valor da transferência < saldo ?) then (Não)
    Então Exibir transação inválida

Cenário:  Sim
    Dado que usuario selecionou transferir dinheiro no menu
    Quando else (Sim)
    Então Debita valor do cliente e credita valor da transferência na conta do usuario
    E Exibe resumo da transação
```

Pode-se destacar que o arquivo contém os cenários de testes na sintaxe Gherkin referente ao diagrama de atividades submetido. O diagrama possui dois cenários, um em queo saldo da conta é menor que o valor informado na transferência, nesses casos deve-se exibir “Transação inválida". E outro onde o saldo é maior que o valor da transferência, então deve-se debitar o valor do cliente e creditar na conta do destinatário e exibir o resumo da transação.

A quantidade de cenários presentes no arquivo *.feature* é definida a partir do condicionais presentes no diagrama, para este exemplo em específico existiam apenas dois caminhos de atividades possíveis, um para se a conta destino existisse e se o saldo fosse suficiente, e um outro caminho para caso essa premissa fosse falsa. Se existissem outros caminhos possíveis para o diagrama a quantidade de cenários possíveis consequentemente aumentaria, o que resultaria em um número maior de cenários de testes a serem gerados no arquivo *.feature*.

5 CONSIDERAÇÕES FINAIS

O presente trabalho teve como objetivo desenvolver um software capaz de transcrever um diagrama de atividades descrito no plantUML em cenários de teste na sintaxe Gherkin.

Para chegar ao resultado foi adotado um processo de desenvolvimento que consistia primeiramente no estudo das tecnologias envolvidas, definição do padrão de escrita dos diagramas, modelagem e desenvolvimento do software.

O resultado final consiste em um software que define alguns padrões de escrita (de diagramas de atividades) e permite gerar cenários de testes na sintaxe gherkin a partir do diagrama de atividades escrito no PlantUML.

Para que isso fosse possível foi utilizado o Python como linguagem para codificação do software, Visual Studio Code como editor de texto e PlantUML para elaboração dos diagramas utilizados durante o desenvolvimento.

O software desenvolvido gera os cenários de teste fazendo uma espécie de limpeza no código do diagrama em PlantUML. O software percorre o código do diagrama de atividades diversas vezes buscando por *tags* úteis para a geração dos cenários de teste. Sempre que uma informação útil é encontrada o software armazena em uma variável, remove essa informação do código fonte do diagrama e retorna o novo código fonte. Ao fim de todas as verificações no código fonte do diagrama restam apenas as estruturas condicionais e suas respectivas atividades.

Como trabalhos futuros propõe-se a inserção de mais elementos do diagrama de atividades como Raias, Bifurcações, Uniões, como elementos reconhecíveis pelo software. Além disso, melhorar a formatação do arquivo *.feature* é um ponto importante para que fique mais intuitivo e limpo.

Por fim, cogita-se também a adição de novos diagramas ao ambiente de transcrição de diagramas plantUML para cenários Gherkin. Como por exemplo o diagrama de sequência, o que possibilitaria uma maior abrangência do software e auxiliaria na difusão tecnológica do software já que ele poderia gerar cenários de testes a partir de um numero maior de entradas ou diagramas.

REFERÊNCIAS

BASE2. **Como escrever um bom BDD?.** Disponível em: https://www.base2.com.br/2020/06/03/como-escerver-um-bom-bdd/. Acesso em: 27 Nov 2020.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: guia do usuário**. Elsevier Brasil, 2006.

CUCUMBER. **Behaviour-Driven Development.** Disponível em: https://cucumber.io/docs/bdd/. Acesso em: 03 Dez 2020.

CRAIG, Rick David; JASKIEL, Stefan P. **Systematic software testing**. Artech house, 2002.

CRISPIN, Lisa; GREGORY, Janet. **Agile testing: A practical guide for testers and agile teams**. Pearson Education, 2009.

DELAMARO, Marcio; JINO, Mario; MALDONADO, Jose. **Introdução ao teste de software**. Elsevier Brasil, 2013.

GHERKIN. **Gherkin Reference**. disponível em: https://cucumber.io/docs/gherkin/reference/. Acesso em: 10 set. 2020.

GUDWIN, Ricardo R. Diagramas de Atividade e Diagramas de Estado. **FEEC-UNICAMP, DCA**, 2016.

IBM. **Diagrama de Atividades**. Disponível em: https://www.ibm.com/support/knowledgecenter/pt-br/SS5JSH_9.5.0/com.ibm.xtools.modeler.doc/topics/cactd.html. Acesso em: 29 Nov 2020.

IBM. **Exemplo de um diagrama de atividades**. Disponível em: https://www.ibm.com/docs/pt-br/rsm/7.5.0?topic=diagrams-example-activity-diagram. Acesso em: 18 Jun 2021.

MEDEIROS, Ernani. Desenvolvendo Software com UML definitivo 2.0. **São Paulo. Person**, 2004.

NETO, A., & Dias, C. (2007). **Introdução a teste de software.** *Engenharia de Software Magazine*, *1*, 22.

ONEDAYTESTING. **GHERKIN: INTRODUZINDO SEUS CONCEITOS E BENEFICIOS.** Disponível em: https://blog.onedaytesting.com.br/gherkin/. Acesso em: 11/11/2020.

PLANTUML, **PlantUML F.A.Q**. Disponível em: https://plantuml.com/faq. Acesso em: 10 set. 2020.

PYTHON, **What is Python.** Disponível em: https://docs.python.org/3/faq/general.html#what-is-python. Acesso em: 01/10/2020.

RUMBAUGH, James; BOOCH, Grady; JACOBSON, Ivar. **The unified modeling language user guide**. Addison-wesley, 1998.

SMART, John Ferguson. **BDD in Action**. New York: Manning Publications, 2014.

UFPR. **Artifact: Cenário de teste.** Disponivel em: https://www.agtic.ufpr.br/pds-ufpr/ProcessoDemoiselllePlugin/workproducts/casoDeTeste_24F453B9.html#:~:text=O%20cen%C3%A1rio%20de%20teste%20%C3%A9,%22como%22%20deve%20ser%20testado.. Acesso em: 27 Nov 2020.

VISUAL STUDIO. **Visual Studio F.A.Q.** Disponível em: https://code.visualstudio.com/docs/supporting/faq. Acesso em: 01/10/2020.