Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Desenvolvimento Autônomo Orientado a Comportamentos Genéricos

Luiz Gustavo Pina de Sales

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Paulo Roberto Miranda Meirelles

2024

Resumo

Luiz Gustavo Pina de Sales. **Desenvolvimento Autônomo Orientado a Comportamentos Genéricos**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

O processo de desenvolvimento de software apresenta uma vasta diversidade de desafios e problemas que precisam ser superados, sendo um dos maiores, o tempo. A realização de testes durante o desenvolvimento é uma maneira de garantir que o programa esteja apresentando comportamentos condizentes com sua especificação. No entanto, é comum os desenvolvedores precisarem interromper o processo de evolução de uma aplicação para implementar casos de testes definidos pelo time de Garantia de Qualidade (Quality Assurance, ou QA), que dependem destas implementações para validarem o comportamento do software.

Portanto, este trabalho propõe um paradigma de desenvolvimento que torna o time de QA autônomo do time de desenvolvimento, na definição e execução de casos de teste, através da introdução de elementos genéricos e camadas de manipulação de casos de testes, de forma que não exija implementações diretas.

Tal paradigma é o nomeado AGBDD (Autonomous Generic Behavior Driven Development), que se apresenta como um sistema de passos pré-definidos, também conhecidos como Statements, genéricos que cobrem as funcionalidades gerais de um serviço. Neste trabalho será desenvolvido o Projeto AT4QA, para servir como um exemplo de implementação deste paradigma para Web APIs, através do desenvolvimento Software Livre, viabilizando a manutenção constante dos Statements e a customização necessária para atender qualquer cenário.

Palavras-chave: Testes de Software. BDD. Desenvolvimento Ágil. AGBDD. GAT. Autônomo. Genérico.

Abstract

Luiz Gustavo Pina de Sales. **Autonomous Generic Behavior Driven Development**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo.

The software development process presents a wide variety of challenges and problems that need to be overcome, one of the biggest, being time. Conducting tests during development is a way to ensure that the program is exhibiting behaviors consistent with its specification. However, developers often need to interrupt the evolution process of an application to implement test cases defined by the QA team, who depend on these implementations in order to validate the behavior of the software.

Therefore, this work proposes a testing paradigm that makes the QA team autonomous from the development team, in defining and executing test cases, through the introduction of generic elements and layers of test case manipulation, that do not require direct implementation.

This paradigm, named AGBDD (Autonomous Generic Behavior Driven Development), presents itself as a system of generic predetermined steps, also known as Statements, that cover the general functionalities of a service. In this work, we will present the development of Project AT4QA, to serve as an example of how to implement this paradigm for Web APIs, through Open Source development, enabling constant maintenance of the Statements and the necessary customization to meet any scenario.

Keywords: Software Testing. BDD. Agile Development. AGBDD. GAT. Autonomous. Generic.

Lista de abreviaturas

AGBDD	Desenvolvimento Autônomo Orientado a Comportamentos Genéricos (*Autonomous Generic Behavior Driven Development*)
GAT	Testes Autônomos e Genéricos (*Generic Autonomous Test*)
AT4QA	Testes Autônomos para Garantia de Qualidade (*Autonomous Testing for Quality Assurance*)
BDD	Desenvolvimento Orientado a Comportamentos (*Behavior Driven Development*)
URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
IME	Instituto de Matemática e Estatística
API	Interface de Programação de Aplicações (*Application Programming Interface*)
QA	Garantia de Qualidade (*Quality Assurance*)
USP	Universidade de São Paulo

Lista de figuras

Lista de tabelas

Sumário

Introdução

No ambiente de desenvolvimento de grandes sistemas de software, um dos maiores desafios é a capacidade de identificar problemas na implementação do código. Para tal, temos atualmente áreas designadas com o propósito de realizar testes para detectar tais problemas, como a área de Garantia de Qualidade, ou Quality Assurance, doravante QA, que possui como função a idealização e realização de casos de teste, nos quais, possíveis fraquezas na implementação de um software podem ser identificadas.

Comumente, profissionais da área de QA são de menor grau técnico que os profissionais da área de desenvolvimento, o que consequentemente, cria um vácuo na comunicação entre estes, podendo impactar tanto os casos de teste sendo idealizados, quanto a própria aplicação. Além disso, a carga de trabalho e a velocidade de produção de serviços e funcionalidades em um sistema de grande porte necessita constante avaliação e verificação de suas funcionalidades, o que torna a execução de tais casos de teste, de maneira manual, inviável para o estabelecimento de um patamar estável de qualidade no desenvolvimento ativo de software, e dificulta o acompanhamento do estado atual de viabilidade e funcionamento de um sistema. Diante de tal realidade, surge a necessidade da automatização do processo de realização de testes de software.

No entanto, métodos contemporâneos de automatização de testes envolvem a idealização de casos de teste pelo time de QA, e implementações de tais casos que precisam ser realizadas por profissionais do time de desenvolvimento, consequentemente, temos um atraso no processo de evolução do software, pois o time que já era responsável pelo desenvolvimento ativo de um produto, se torna também responsável pelo desenvolvimento de um sistema de testes, que pode apresentar seus próprios problemas e necessitar de manutenção eventualmente. Assim, temos que o time de desenvolvimento se torna responsável por gerenciar dois produtos em paralelo, e o time de QA se torna dependente do time de desenvolvimento para a validação de seus casos de teste.

Portanto, neste trabalho, será desenvolvida uma proposta de paradigma de desenvolvimento, e uma ferramenta que possui o intuito de exemplificar uma aplicação de tal paradigma, de modo que permita que o time de QA não dependa do time de desenvolvimento para realizar testes automatizados, tornando-se assim, autônomo na realização de suas funções. Tal paradigma, nomeado Desenvolvimento Autônomo Orientado a Comportamentos Genéricos, busca expandir o conceito de Desenvolvimento Orientado a Comportamentos (Behavior Driven Development ou BDD) para tentar solucionar este problema ao introduzir um sistema de Statements genéricos que cobrem as funcionalidades gerais de um tipo de aplicação, que no caso deste trabalho, será focado em Web APIs.

Objetivos

Teremos então como objetivos deste trabalho a definição de um paradigma de desenvolvimento que utilize um modelo de testes genéricos, que permita a realização de testes por parte de um time de QA de maneira autônoma e independente do time de desenvolvimento, e também o desenvolvimento do Projeto AT4QA (Autonomous Testing for Quality Assurance), que será uma ferramenta Web desenvolvida como Software Livre que aplicará este paradigma especificamente para Web APIs.

O paradigma AGDBB deverá então ser definido formalmente de modo que seja facilmente entendido e integrado em ambientes de desenvolvimento de software. O Projeto AT4QA será desenvolvido como um exemplo de aplicação deste paradigma especificamente para Web APIs, buscando sanar as possíveis dificuldades e dúvidas que um profissional da área de QA pode encontrar ao tentar idealizar e executar testes dentro do paradigma AGBDD, e também para facilitar a adesão de novos usuários a este, pois a ferramenta será disponibilizada com seu código fonte aberto, o que permitirá que a comunidade possa contribuir com sua evolução, com a expansão de sua compatibilidade com diferentes tipos de aplicações, e a adição de novas funcionalidades que garantam a relevância e manutenção da ferramenta.

O Projeto AT4QA será desenvolvido com base nos seguintes pilares:

- Comunicação: O projeto deve ser capaz de comunicar e tornar clara a proposta do paradigma AGBDD, sua utilidade, facilidade de uso e como este deve ser construído e aplicado em um ambiente de desenvolvimento de software, junto com um glossário que explique o uso e funcionamento dos Statements genéricos utilizados pela ferramenta;
- Edição: O projeto deve ser capaz de auxiliar o usuário na edição de casos de teste, de modo que a escrita de tais casos seja fácil e intuitiva, e que o usuário possa entender sem grandes problemas o que está sendo escrito e o que está sendo testado;
- Geração: O projeto deve ser capaz de gerar bases para testes de maneira automatizada, dando para o usuário um ponto de partida do qual seguir para a idealização dos seus casos de teste;
- Execução: O projeto deve ser capaz de executar testes de maneira automática, de modo que o usuário possa rapidamente verificar o estado atual de viabilidade e funcionamento de um sistema, e identificar possíveis problemas na implementação de um software;

Estrutura do Trabalho

Este trabalho irá percorrer todo o processo de conceitualização e desenvolvimento do paradigma AGBDD e do Projeto AT4QA. No Capítulo 1 serão definidos os conceitos centrais de testes de software, e como estes conceitos se relacionam com o paradigma AGBDD, além das ferramentas e tecnologias escolhidas para o desenvolvimento deste trabalho.

No Capítulo 2, será definido formalmente o paradigma AGBDD, e como este deve ser

aplicado em um ambiente de desenvolvimento de software. Além de definirmos também o Projeto AT4QA e como este deverá ser desenvolvido.

No Capítulo 3, será apresentado o desenvolvimento do Projeto AT4QA, onde mostraremos os passos que foram necessários para tornar o conceito da ferramente uma realidade.

No Capítulo 4, será apresentada a conclusão deste trabalho, onde iremos refletir sobre o processo e o caminho trilhado, analisando os resultados obtidos e as possíveis melhorias e expansões que podem ser realizadas no futuro.

Capítulo 1

Conceitos e Terminologia

Neste Capítulo teremos como foco a apresentação dos conceitos teóricos essenciais para o entendimento do trabalho em questão. Iniciaremos com uma revisão dos conceitos centrais de testes de software, em seguida iremos analisar especificamente os testes orientados a comportamentos, para que assim possamos definir os testes genéricos com clareza.

Além disso, iremos explorar as tecnologias utilizadas no desenvolvimento do Projeto AT4QA.

1.1 Testes de Software

Durante o processo de desenvolvimento de um sistema de software, queremos ter certeza que o código funcione da maneira esperada, ou seja, que tenha comportamentos consistentes com a demanda que o projeto em questão busca sanar. Para tal, podemos realizar testes para garantir que o software esteja sendo desenvolvido de acordo com o que se espera de suas funcionalidades.

Tais testes podem ser realizados em diversas etapas do desenvolvimento, de maneiras diferentes, e com objetivos distintos. Uma aplicação pode dispor de testes unitários, que visam testar pequenas partes do código isoladamente, testes de integração, que visam testar a interação entre diferentes partes do projeto, testes de aceitação, que visam testar o comportamento do software como um todo. Temos também testes de regressão, que visam garantir que novas implementações não impactem negativamente em funcionalidades já existentes, e testes de performance, que visam garantir que o software possua um desempenho aceitável, entre outros.

Os testes que serão realizados em si podem ser manuais, com a interação direta de um indivíduo com o software, ou automatizados, em que temos um programa que realiza a interação com o software e avalia os resultados obtidos. Vale ressaltar que nem todos os testes podem ser automatizados, testes de usabilidade por exemplo, requerem a análise de como os usuários interagem e interpretam as interfaces do software, e portanto, não podem ser realizados de maneira automatizada.

Notamos então que o processo de testes de software é uma faceta essencial do desenvolvimento, e que possui uma gama extensiva de possibilidades, metodologias e objetivos.

1.2 Quality Assurance (QA)

Quality Assurance, ou Garantia de Qualidade, é a área do desenvolvimento que possui como objetivo a garantia de que o software em questão apresenta as funcionalidades e os comportamentos esperados da definição de sua demanda e criação.

Portanto, o time responsável pelo processo de QA é de suma importância no processo de desenvolvimento, atuando através da definição e realização de testes, e da análise de casos de uso do software, ou seja, este é o time responsável pela realização dos testes de software delineados anteriormente.

Em geral, os times de QA possuem um grau de conhecimento técnico menos minucioso que o time de desenvolvimento, proveniente de suas distintas áreas de atuação e focos de estudo. Portanto, temos que a comunicação entre estes dois times é fundamental para o sucesso do desenvolvimento de um software, comunicação esta que pode ser facilitada por meio de práticas de desenvolvimento de software que facilitem a compreensão e a definição de comportamentos esperados de um software.

1.3 Casos de teste

Uma das funções do time de QA é a definição de casos de teste, estes que são caracterizados por representar possíveis interações que o usuário pode ter com um software e o comportamento devido que este deve ter. Ou seja, um caso de teste é a idealização de um cenário de interação que um potencial usuário pode ter com software, e portanto, deve descrever qual o comportamento que o software deve ter em resposta a tal interação.

Sendo assim, podemos definir um caso de teste como o seguinte conjunto de passos:

1. Descrição do ambiente que necessitou a interação do usuário, criando assim a demanda de uso do software;
2. Descrição da interação entre o usuário e o software, em busca de um resultado que satisfaça sua demanda;
3. Descrição do comportamento resultante do software, em resposta a interação do usuário;

Portanto, a idealização dos casos de teste é a fundação de qualquer processo de testes de software, e é a partir destes que podemos avaliar se um software está se comportando de acordo com o esperado ou não.

1.4 Testes Automatizados

A realização de testes de maneira manual costuma demandar uma quantidade exorbitante de tempo, o que torna necessária a busca por meios de agilizar este processo.

A maneira mais comum de acelerar o processo de testes é através da implementação de testes automatizados. Tais testes, são comumente criados pelos próprios desenvolvedores como uma demanda adicional ao processo de evolução e manutenção de um software, e são caracterizados como módulos especiais do produto, ou um produto externo, que possuem como função executar partes do programa e validar seus comportamentos esperados.

Desta maneira, é possível que casos de testes sejam criados e avaliados sem que os elementos manuais do fluxo de execução de um software, como o preenchimento manual de formulários, impactem o tempo para o processo de validação de comportamentos.

Tais casos costumam ser inicialmente idealizados e definidos pelo time de QA, que os passa então para o time de desenvolvimento, que em seguida deve implementar o código responsável pela execução automatizada destes, posteriormente, o time de QA pode executar os testes de maneira automatizada e analisar os resultados obtidos apropriadamente.

No entanto, neste modelo o desenvolvedor precisa agora considerar um novo elemento para o processo de construção de um programa, a implementação de casos de teste, o que introduz então uma carga externa para o processo de desenvolvimento, que pode trazer seus próprios problemas e necessitar de manutenção e avaliação assim como o programa principal.

1.5 Metodologia Ágil

O desenvolvimento ágil é um paradigma de desenvolvimento de software que tem como objetivo introduzir flexibilidade e adaptabilidade ao processo de evolução de um software.

Formalizado em um manifesto por K. Beck *et al.*, 2001, é a partir deste modelo de desenvolvimento que a prática de testes de software se torna mais comum, pois para que seja possível ter flexibilidade e adaptabilidade, é necessário que sempre haja consciência do estado e funcionamento atual do software, para que as mudanças necessárias possam ser realizadas minimizando impactos negativos. Assim, podemos garantir que o processo de desenvolvimento de software esteja sempre preparado para eventuais mudanças em sua demanda ou em seu ambiente de execução.

Logo, temos que a prática de testes de software se torna parte essencial do desenvolvimento, popularizando a prática de desenvolvimento orientado a testes, conhecida como TDD (Test Driven Development).

1.6 Desenvolvimento Orientado a Testes (TDD)

O desenvolvimento orientado a testes é uma prática de desenvolvimento de software que tem como objetivo a definição de casos de teste, antes da implementação de funcionalidades de um software.

Considerada "redescoberta"por Kent Beck em 2002 (Beck, 2002), esta prática é derivada da metodologia utilizada para programação em cartões perfurados, onde eram descritos os comportamentos esperados de um programa a priori, em seguida era feita a programação nos cartões, e por fim, os resultados eram comparados com os definidos anteriormente.

Da mesma maneira, a aplicação de TDD delineada por Beck, tem como objetivo a conceitualização de casos de testes que determinam quais devem ser os resultados obtidos de uma execução do software antes de seu desenvolvimento, para que seja possível em seguida implementar o código da maneira mais simples possível para satisfazer os critérios estabelecidos inicialmente, e então, executar os testes e validar se os resultados obtidos são condizentes com o esperado.

Utilizando este método, temos sempre garantido que o desenvolvimento possui um direcionamento forte e claro, e que o desenvolvedor sempre tem em mente quais são os resultados esperados de sua implementação, o que facilita a manutenção e a evolução do software.

No entanto, como descrito anteriormente, o time de QA deve ser responsável pela definição dos casos de teste, e devido sua menor experiência técnica, a definição de comportamentos esperados pode ser um desafio sem ter uma noção clara do funcionamento de uma ferramenta. Para resolver este problema, temos o advento da prática de desenvolvimento orientado a comportamentos, ou BDD (Behavior Driven Development).

1.7 Desenvolvimento Orientado a Comportamentos (BDD)

Devido a discrepância de conhecimento técnico presente entre o time de desenvolvimento e o time de QA, é necessário encontrar meios de facilitar a comunicação das funcionalidades de um software. Uma dessas maneiras é o paradigma de testes orientados a comportamentos, definido por Dan North em seu artigo de 2006 "Introducing BDD"(North, 2006).

Expandindo sobre o TDD, o BDD tem como objetivo a definição de comportamentos esperados de um software em linguagem natural primeiro, para que o time de QA consiga expressar claramente quais são os resultados e objetivos esperados de um software antes que esse seja desenvolvido, de modo que, em seguida, o time de desenvolvimento possa seguir os comportamentos delineados anteriormente para a implementação de funcionalidades. Tal definição é realizada em um arquivo chamado Feature File, que é composto por Scenarios que representam os casos de teste, que em sua parte são compostos por Statements que representam cada passo de um caso de teste.

Feita a implementação, o time de QA pode então executar os testes que representam os comportamentos definidos, validando se os comportamentos obtidos são condizentes com o esperado.

O modelo de linguagem do BDD é delineado pelo uso de 3 palavras chaves principais:

- Given: Dado um ambiente pré-execução, esta palavra chave define o estado inicial em que o software se encontra;
- When: Quando o elemento em análise for executado, esta palavra chave define a ação que será realizada;

- Then: Então esperamos determinados resultados, esta palavra chave define os comportamentos esperados do software em resposta a ação;

Por meio do uso deste paradigma, temos que se torna possível que o time de QA possa delinear casos de teste para determinado software em linguagem natural, e que os desenvolvedores possam utilizar estes comportamentos esperados como um princípio de direcionamento, tornando o funcionamento do software claro não só para os desenvolvedores, mas para qualquer outro profissional que esteja interagindo com o processo de desenvolvimento, independente de seu grau de instrução técnica, devido ao uso de linguagem natural na definição do comportamento do software.

1.8 Feature Files

Feature files são os arquivos que irão conter as descrições comportamentais de um software em linguagem natural, utilizados como base para o desenvolvimento no BDD.

Idealmente cada Feature File deve representar um conjunto de casos de teste para uma funcionalidade específica do software, e deve conter uma descrição clara e concisa dos comportamentos esperados desta. Ou seja, cada Feature File representa os casos de teste de uma Feature, e será composto por Scenarios, que representam os casos em si. Cada Scenario é em sua vez composto por Statements, que representam os passos de um caso de teste.

Portanto, temos que um Feature File é estruturado da seguinte maneira:

- Feature: Descrição da funcionalidade que será testada;
- Scenario: Descrição de um caso de teste específico da funcionalidade em questão;
- Statements: Passos que compõem um caso de teste;

Com estes componentes, podemos descrever de maneira clara e concisa os comportamentos esperados de um software, de modo que seja possível interpretar e entender como um software funciona sem a necessidade de conhecimento técnico.

1.9 Scenarios

Scenarios são os casos de teste que compõem um Feature File, estes tendem a representar interações específicas entre um usuário e um software, e descrevem quais os comportamentos esperados de um software em resposta a tais interações.

Como dito anteriormente, um Scenario é composto por Statements, e possui 3 partes principais:

- Ambiente: Descrição do ambiente inicial em que o caso de teste ocorre, ou seja, o cenário em que o usuário possui uma demanda do uso do software;
- Ação: Descrição da interação que o usuário irá realizar com o software;
- Resultados: Descrição dos comportamentos esperados em resposta a interação;

Portanto, um Scenario consegue tornar explícito quais são os comportamentos esperados de um software em resposta a uma interação específica, e torna possível a validação destes comportamentos após a implementação.

1.10 Statements

Statements são os passos que compõem um Scenario, e representam a descrição geral dos componentes de uma interação específica entre um usuário e um software, ou seja, cada Statement irá descrever um elemento específico sobre um caso de uso em que um usuário necessita interagir com o software.

Um Statement pode ser de 3 tipos:

- Given: Este tipo de Statement descreve situações em que o usuário e/ou o software se encontram antes de sua interação;
- When: Este descreve o momento da interação entre o usuário e o software, e como esta ocorre;
- Then: Já este descreve os comportamentos esperados do software em resposta a interação;

Sendo assim, temos que os Statements permitem a descrição categorizada e linear de um caso de teste, facilitando a interpretação e validação dos comportamentos esperados de um software.

1.11 Web APIs (Application Programming Interfaces)

Uma API é uma interface pela qual uma aplicação consegue estender suas funcionalidades para disponibilizar informações ou serviços de diversos tipos para desenvolvedores, ou seja, uma Web API torna possível que desenvolvedores ou usuários consigam interagir com um serviço Web para extrair informações, dados ou funcionalidades de maneira simples, para que não seja necessário interagir com o código do serviço diretamente.

Diversos serviços Web disponibilizam APIs para que desenvolvedores possam interagir com seus serviços, podendo então extrair métricas ou outros diversos tipos de dados. Portanto, Web APIs são uma faceta essencial do ecossistema moderno de interação entre serviços, e são uma ferramenta fundamental para a construção de aplicações modernas.

Estas são normalmente documentadas em formato YAML ou JSON, o que permite fácil comunicação de como o serviço oferecido funciona e como é possível interagir com ele.

1.12 Back-end

O back-end é a parte de um software que é responsável pelo processamento de dados e informações. Ele não é visível para o usuário, sendo responsável por receber as informações

enviadas por este, para que sejam processadas e, enfim, retornadas para o front-end.

1.13 Front-end

O front-end é a parte de um software que é responsável pela interação direta com o usuário. Ele é a parte visível da aplicação, e é responsável por receber as informações enviadas pelo back-end e apresentá-las por meio de alguma interface para o usuário.

1.14 Gherkin

Gherkin[1] é uma definição gramática para a descrição de casos de testes, que torna estes facilmente tratáveis de maneira automatizada, caracterizada pelo seu uso de 3 tipos centrais de Statements, Dado (Given), Quando (When) e Então (Then).

Com estes, podemos descrever o ambiente inicial em que se espera que um caso de teste ocorra, posteriormente, descrevemos como a interação em análise ocorre, e por fim, podemos detalhar quais são as consequências esperadas desta.

Os casos de teste definidos utilizando esta gramática são armazenados em arquivos .feature, que chamaremos de Feature Files.

1.15 Python

Python é uma linguagem de programação de alto-nível, conhecida por sua facilidade de leitura e escrita, e por sua vasta biblioteca de funções, que tornam a implementação de diversos tipos de sistemas mais simples. Apesar do custo de performance proveniente de sua natureza interpretada, sua facilidade de uso e de manipulação de estruturas de dados a tornou a escolha ideal para o projeto em questão.

Portanto, a implementação do Projeto AT4QA será feita utilizando a linguagem Python, com a ajuda principal da biblioteca Pytest BDD.

1.15.1 Pytest

Pytest[2] é uma biblioteca pública da linguagem Python que busca implementar um arcabouço para testes simples e intuitivo. Inicialmente desenvolvida como uma expansão do projeto PyPy, com o intuito de tornar a escrita de testes mais fácil para este, a biblioteca evoluiu para um projeto independente que é amplamente utilizado para a implementação de testes utilizando Python em geral.

Ela conta com diversas funcionalidades que facilitam a escrita de testes de software, e portanto foi escolhida para a implementação do Projeto AT4QA.

---

[1] https://cucumber.io/docs/gherkin/

[2] https://docs.pytest.org/en/stable/

1.15.2 Pytest BDD

Pytest BDD[3] é uma expansão da biblioteca Pytest que implementa parte da interpretação da gramática Gherkin, com o objetivo de facilitar o processo de desenvolvimento utilizando BDD. Ela define as mecânicas de interpretação de um Feature File escrito em Gherkin, e a execução de testes a partir destes, portanto, com ela é possível escrevermos Feature Files descrevendo comportamentos esperados de teste, e interpretá-los diretamente em testes executáveis de maneira automatizada, fazendo então tradução direta de um comportamento em linguagem natural para a execução de código.

Portanto, a biblioteca Pytest BDD é uma parte fundamental do Projeto AT4QA.

1.16 JSON

JSON, ou JavaScript Object Notation, é um formato de armazenamento de dados que é facilmente legível por humanos e máquinas, definido por pares chave e valor. Ele será utilizado como um método alternativo de exportação dos relatórios de execução de testes, além de ser utilizado nos Feature Files para definir determinados parâmetros de execução de testes.

1.17 Flask

Flask é uma biblioteca para Python que facilita a construção de aplicações Web. Atuando como um arcabouço para a implementação de um servidor Web, ela permite a definição de rotas, a construção de interfaces Web, e a comunicação com o servidor de maneira simplificada.

Ela será utilizada para a construção das funcionalidades Web do Projeto AT4QA.

1.18 YAML

YAML, ou Yet Another Markup Language, é uma linguagem de serialização de dados que possui como objetivo ser facilmente interpretada. Ela é comumente utilizada para a escrita de documentação de serviços Web, e portanto, será uma das linguagens suportadas pelo módulo de geração de Feature Files do Projeto AT4QA, em conjunto com JSON.

1.19 GitHub

GitHub é uma plataforma de hospedagem de código que utiliza o sistema de controle de versão Git. Ela permite que um desenvolvedor tenha fácil controle sobre as diferentes versões de um código, e também permite a colaboração de diferentes desenvolvedores em um mesmo projeto. Esta será a ferramenta utilizada para controle de versão do Projeto AT4QA.

---

[3] https://pytest-bdd.readthedocs.io/en/stable/

1.20 Ace

Ace é uma biblioteca pública disponível para a linguagem JavaScript que implementa um editor de texto customizável. Ela será utilizada para a construção do editor de Feature Files, que será descrito em mais detalhes no Capítulo 3. Esta biblioteca permite a implementação rápida e direta de um editor de código em uma página Web.

Agora que temos definidos os termos e tecnologias que serão utilizadas nos Capítulos futuros, podemos então seguir para a definição formal do AGBDD e a conceitualização deste trabalho.

Capítulo 2

Concepção do Desenvolvimento Autônomo Orientado a Comportamentos Genéricos

Definidos os termos em que estamos trabalhando, a proposta deste TCC será estabelecer um paradigma de desenvolvimento, que torne possível a execução de testes de software automatizados de maneira autônoma por parte do time de QA, sem depender de implementações específicas de casos de teste pelo time de desenvolvimento, e uma ferramenta que aplique este paradigma, tornando sua utilidade clara e facilitando sua adesão.

A utilidade de tal sistema se torna evidente perante os problemas apresentados na nossa discussão sobre testes de software e o paradigma orientado a comportamentos, pois nele temos que ao desenvolvedor é dado um papel duplo, de evoluir e manter uma aplicação, e também de implementar os casos de teste delineados pelo time de QA. Logo, temos a necessidade de um novo paradigma com o qual seja possível que o time de QA possa conceitualizar seus cenários de teste e executá-los, sem que ocorra um desvio do foco do desenvolvimento do software para os testes, externalizando a implementação destes.

Portanto, nas seguintes seções iremos percorrer a história de como surgiu a necessidade advinda do autor de solucionar esse problema, definir um paradigma de desenvolvimento e um modelo de testes que possa ser uma solução, e também definir uma ferramenta que possa implementar este paradigma.

2.1 Origem do Problema

Durante a atuação do autor como estagiário na empresa de desenvolvimento de software Opus Software, este foi encarregado de implementar um sistema de testes automatizados utilizando o paradigma BDD, com o auxílio da biblioteca Pytest BDD. Durante o desenvolvimento deste sistema, eram utilizados Feature Files para a definição de cenários de teste para cada API, e a implementação destes cenários era feita em uma classe de teste equivalente a cada Feature File.

Com o tempo, tornou-se claro que a implementação de tais cenários de teste constava com uma quantidade considerável de similaridade entre as APIs, tornando o processo de implementação repleto de código redundante. Tal redundância trouxe a tona o fato de que existiam aspectos gerais do processo de se realizar testes de APIs que poderiam ser extraídos de maneira explícita do conjunto de testes, ou seja, após criar uma massa de testes específicos, foi possível identificar elementos comuns entre estes que poderiam ser abstraídos em um novo sistema, que tornasse os elementos em comum o foco do processo de idealização de casos de teste.

Com isso, propomos a implementação de um modelo de testes genéricos, em que os Statements eram definidos de antemão, buscando cobrir funcionalidades gerais das chamadas de APIs, que haviam sido reconhecidas no processo da implementação dos testes individuais, e novos cenários seriam escritos utilizando estes Statements genéricos. Desta maneira, todas as APIs poderiam ser testadas compartilhando o mesmo código, sem a necessidade de classes de teste específicas para cada, além disso, agora o time de QA não dependia da implementação de testes por parte de um time técnico, e podia testar seus casos de maneira autônoma, pois o código já havia sido implementado de maneira genérica.

Após alguns meses de uso desse sistema, foi realizada uma pesquisa de satisfação com o time de QA, e foi constatado que o sistema genérico de testes acelerou consideravelmente o processo de QA. No entanto, o sistema ainda apresentava problemas de legibilidade e manutenção, pois os Statements genéricos eram altamente parametrizados.

Isso porque para lidar com as diversas possibilidades inerentes do uso de APIs, era necessário haver um alto nível de controle dos elementos envolvidos em cada teste, o que devido a complexidade deste processo, tornou necessária a presença de diversos parâmetros em cada Statement. Além disso, a escrita de cenários de teste ainda era uma tarefa complexa, o que tornava os Feature Files muito extensos.

Outro elemento apontado era a legibilidade do relatório, pois era difícil identificar qual era o caso de teste específico em que um erro ocorre quando um cenário de uma mesma API envolvia muitos parâmetros genéricos iguais, e como apontado anteriormente, haviam muitos parâmetros por Feature File, dificultando a identificação de casos de teste específicos.

Havia também uma certa dificuldade de iniciar o processo da escrita, pois como o conjunto de Statements genéricos era extenso, sendo funcionalmente necessário aprender uma nova linguagem de testes para seu uso, o que tornava a adesão ao sistema um processo relativamente confuso, devido a dificuldade de identificar quais Statements seriam preferíveis para determinada situação.

Através desta pesquisa, foram reconhecidos 3 problemas principais com o sistema inicial de testes genéricos:

- Comunicação: O time apresentava dificuldades em identificar quais Statements eram ideais para determinadas situações, e qual o ponto de partida que deveria ser tomado ao iniciar a escrita de um novo teste, o que apontava para uma falha na comunicação e na clareza do glossário de Statements genéricos;
- Manutenção: A manutenção dos testes era complexa, pois os Statements genéricos eram altamente parametrizados, o que tornava a identificação de pontos de correção

e a correção em si uma tarefa complexa e laboriosa;

- Legibilidade: A legibilidade dos Feature Files e dos relatórios era prejudicada pela alta parametrização dos Statements genéricos, o que tornava difícil a identificação de erros específicos e a manutenção dos casos de teste;

Para sanar estes problemas, foi definido então o sistema de Dialética Diegética, este sistema é caracterizado pelo uso de arquivos adicionais denominados máscaras e dicionários, que permitem a criação e utilização de Statements customizados, utilizando os Statements genéricos definidos como base. As máscaras seriam arquivos com sintaxe similar a dos Feature Files, mas que poderiam conter parâmetros repetidos injetados diretamente nos Statements, e também poderiam utilizar Statements não definidos no conjunto de Statements genéricos, isso se tornava possível através de um arquivo auxiliar chamado dicionário, que seria responsável por traduzir um Statement customizado em um conjunto de Statements genéricos para resumir um processo recorrente ou extenso. Com essas ferramentas, o modelo de testes genéricos consegue preservar o uso da linguagem natural, apesar de sua natureza genérica e altamente parametrizada, e torna a manutenção e a escrita de cenários de teste mais acessível e intuitiva para o time de QA.

Desta forma, tornou-se evidente que a implementação de um sistema de testes genéricos pode ser uma solução para o problema de dependência do time de QA em relação ao time de desenvolvimento, e que este sistema poderia evoluir em uma plataforma de testes genéricos que poderia ser utilizada por diversos times de forma colaborativa.

É com este intuito que iremos então definir um paradigma de testes genéricos e uma ferramenta que o implemente a seguir, para que possamos tornar a adesão a este sistema mais fácil e intuitiva para outros times de desenvolvimento e QA, que possam identificar desafios similares aos descritos acima, e que podem se beneficiar do uso de um sistema equivalente de acordo com as características específicas de seu ambiente.

2.2 Definição do Paradigma

Para que possamos definir uma expansão do paradigma BDD que contemple os elementos genéricos, temos primeiro que definir o que exatamente são esses testes genéricos, e para tal iremos explorar os elementos que foram identificados acima e como os testes genéricos se manifestaram neste processo, definindo então um novo modelo de testes automatizados caracterizado por seu uso de Statements genéricos.

2.2.1 Definindo o Modelo de Testes Genéricos

Reconhecemos então que o modelo genérico surgiu a partir da transição de um modelo de testes tradicional para um modelo ágil, ou seja, inicialmente o software era desenvolvido com testes de unidade tradicionais, sendo posteriormente testado manualmente e verificado pelo time de QA. Eventualmente, temos a necessidade de testes automatizados com o intuito de adesão ao paradigma BDD no desenvolvimento, o que torna necessário a implementação de testes por desenvolvedores para as aplicações já existentes e para as novas funcionalidades. Temos então o advento do modelo genérico, sendo introduzido inicialmente para facilitar essa transição, com o intuito de acelerar o processo de

implementação para funcionalidades já existentes, ou seja, atuando principalmente na área dos testes regressivos.

Podemos então identificar que os testes genéricos se apresentam como uma maneira de facilitar o processo transitório de um modelo tradicional de testes para um modelo automatizado, onde não é necessária a interrupção da evolução do software para a implementação de testes, e em que o time de QA é capaz de realizar suas atividades de maneira autônoma.

Com isso, podemos definir o modelo de testes genéricos de maneira formal como um conjunto de regras e comportamentos definidos por Statements, que cobrem os comportamentos comuns de um sistema sob análise. Estes Statements devem ser definidos de forma que possam ser utilizados para a composição de casos de teste que sejam representativos de um caso de uso do software em questão, e que possam ser combinados em Statements customizados, para que o time de QA seja capaz de definir comportamentos complexos e específicos, enquanto preservando a comunicação do comportamento do software através da linguagem natural.

Realizando um estudo dos comportamentos gerais de um sistema, é teoricamente possível aplicar este modelo em qualquer área do desenvolvimento de software, no entanto, é preciso reconhecer que o nível de complexidade e de comunicação inerentes do sistema em análise pode influenciar na praticidade de implementação deste modelo. Por exemplo, podemos definir comportamentos genéricos no desenvolvimento de jogos, como o input do jogador e a movimentação de personagens, no entanto, a interação de diversos elementos em um ambiente 3D pode tornar a definição de Statements genéricos inapropriada, apesar de não ser impossível a princípio.

Podemos então determinar os seguintes pilares fundamentais do modelo de testes genéricos:

1. Generalidade: É necessário estudar e reconhecer o objeto de seu desenvolvimento, seja através da implementação de testes específicos, ou pelo estudo direto do sistema que se busca produzir, para reconhecer elementos comuns que possam ser generalizados em um nível comportamental, tornando possível a definição de Statements genéricos que possam ser utilizados para a concepção de qualquer caso de uso do software;

2. Autonomia: Deve ser possível que o time de QA consiga definir e executar seus próprios testes, sem depender de implementações específicas por parte do time de desenvolvimento, para que o time de QA possa focar em testar o sistema de forma automatizada sempre que necessário, garantindo consciência constante do estado do sistema;

3. Acessibilidade: O sistema de testes genéricos deve ser de fácil adesão, tomando cuidado para evitar que Statements genéricos sejam redundantes ou que possuam funcionalidade ambígua, podendo haver medidas auxiliares para facilitar a escrita e a manutenção de Feature Files, como verificadores de sintaxe e editores específicos que verificam o uso dos Statements definidos;

4. Atomicidade: Cada Statement genérico deve representar uma ação única e específica, e sua definição deve tornar clara a ação sendo realizada, para evitar ambiguidades e

efeitos colaterais indesejados. Apenas os Statements customizados devem representar ações complexas.

5. Legibilidade: Os resultados dos testes devem ser claros e de fácil identificação de erros, portanto os relatórios resultantes da execução dos testes devem priorizar a legibilidade e auxiliar o usuário a identificar qual foi o caso específico em que houve um erro, e deve tomar cuidado para que o alto nível de parametrização dos Statements genéricos não prejudique a legibilidade dos resultados;

6. Colaboração: O modelo de testes genéricos necessita de manutenção constante, pois o sistema em análise pode sofrer mudanças e evoluções bruscas a medida que o cenário e o ambiente tecnológico se modificam e evoluem, portanto, o desenvolvimento Software Livre é fundamental para garantir que o modelo é capaz de cobrir os cenários necessários de um sistema.

Sendo assim caracterizado, nomeamos o modelo de testes genéricos como Testes Genéricos e Autônomos, ou Generic Autonomous Testing (GAT). Vale notar que este modelo não se propõe como uma extinção imediata da implementação de testes por desenvolvedores, mas sim como uma externalização deste processo de maneira colaborativa, através do desenvolvimento Software Livre, com o intuito de disponibilizar ferramentas expansivas de testes que possam eventualmente reduzir ao máximo possível a necessidade de implementação de testes específicos e que possa ser constantemente atualizada para novas tecnologias e mudanças no ambiente de desenvolvimento.

Este modelo se demonstra especialmente eficaz na transição de um modelo manual para automatizado e no desenvolvimento de testes de regressão, pois permite autonomia do time de QA na experimentação de novos cenários de teste, consequentemente expandindo os tipos de testes que podem ser feitos e aumentando a cobertura destes. Além disso, devido sua natureza customizável e automatizada, garante um maior nível de consciência do estado atual de um projeto, pois sempre é possível realizar testes novos e verificar se o comportamento esperado ainda é satisfeito, em paralelo com a evolução do software, sendo possível também atrelar o modelo de testes genéricos a um sistema de integração contínua, em que os Feature Files são executados automaticamente a cada nova versão do software.

Portanto, é preciso ter conhecimento da área de aplicação do modelo de testes para garantir que este modelo é adequado para determinados processos. No caso deste trabalho, teremos como foco o desenvolvimento de Web APIs, onde a aplicação do modelo de testes genéricos teve origem, se mostrando como prática e eficaz. Na Figura 2.1 temos uma ilustração abstrata do funcionamento do modelo GAT.

2.2.2 AGBDD

Dada a experiência adquirida com a implementação do sistema de testes genéricos, torna-se possível definir uma expansão do paradigma BDD, que inclua em sua definição o modelo GAT. Sendo assim, podemos considerar como uma base os critérios definidos no BDD, como discutidos no Capítulo 1, incluindo o processo de definição dos comportamentos a priori, e o desenvolvimento voltado para a garantia da satisfatibilidade dos comportamentos esperados, com o intuito de verificá-los ao fim do desenvolvimento.

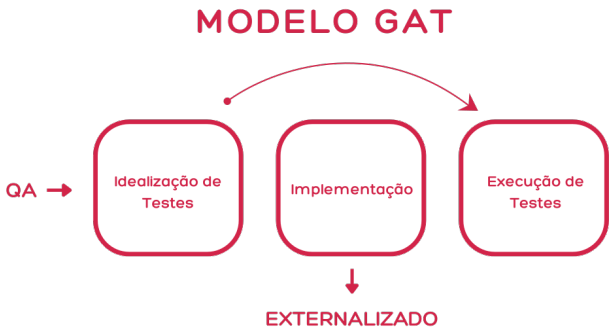

Figura 2.1: *Abstração do modelo GAT*

Definidos os pilares fundamentais do GAT, notamos que o modelo se manifesta como uma expansão natural do BDD, onde a única diferença é que os comportamentos definidos como guia para o desenvolvimento serão escritos com base em uma gramática de Statements pré-definidos, em Feature Files que podem ser executados de forma automatizada. Sendo assim, temos os benefícios do BDD, como a comunicação eficaz entre times de técnicos e não técnicos, os benefícios do TDD com os comportamentos esperados do software claramente definidos antes de seu desenvolvimento, e por fim os benefícios do GAT, que permitem a execução de testes de maneira autônoma e genérica, sem a necessidade de implementações específicas por parte do time de desenvolvimento, e com maior facilidade de transição para um modelo automatizado, pois a aplicação do GAT não requer a interrupção do desenvolvimento para a implementação de testes. Podendo então ser aplicada de maneira gradual, com a adição de novos cenários sendo testados pelo time de QA à medida que são desenvolvidos. Logo, torna-se evidente que a implementação do modelo GAT dentro do paradigma BDD apresenta benefícios consideráveis para o processo de desenvolvimento de software.

Definimos, então, o paradigma que representa a união do GAT com o BDD, como Desenvolvimento Autônomo Orientado a Comportamentos Genéricos, ou Autonomous Generic Behavior-Driven Development (AGBDD). A Figura 2.2 apresenta uma abstração do modelo AGBDD, que ilustra a relação entre estes elementos.

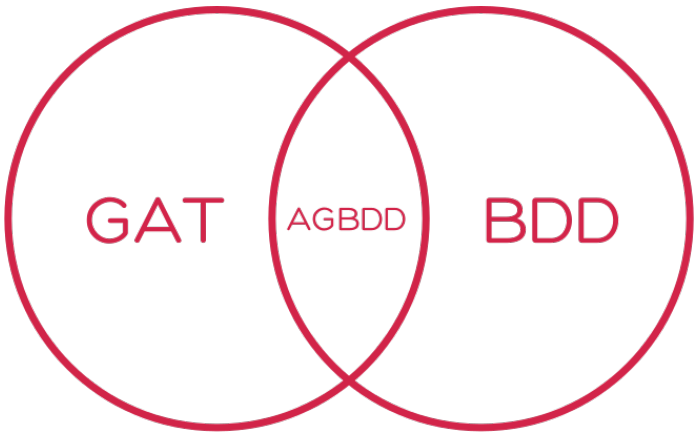

Figura 2.2: *Abstração do modelo AGBDD*

2.3 Exemplo de Aplicação

Seguindo da definição do paradigma AGBDD, podemos então definir uma ferramenta que o implemente dentro de uma área específica de desenvolvimento de software, no caso, o desenvolvimento de Web APIs. Para tal, podemos definir os seguintes aspectos que devem ser contemplados pela ferramenta:

1. Statements Genéricos: Iremos definir um conjunto de Statements genéricos que serão utilizados para descrever comportamentos comuns de Web APIs, e que poderão ser utilizados para a definição de casos de teste que representem o ato de realizar chamadas para uma API. Estes Statements devem ser claros e concisos, e devem representar ações únicas e específicas, para que possam ser utilizados de maneira modular e atuar genericamente para qualquer API;
2. Retorno a Linguagem Natural: Os Statements genéricos devem ser descritos com a maior semelhança possível com a linguagem natural, e deve ser possível criar Statements customizados que possam ser utilizados para a definição de cenários de teste;
3. Auxiliar de Edição: Também será desenvolvido um auxiliar de edição de Feature Files que tenha compatibilidade com os modelos das máscaras e dicionários, tornando assim o processo de escrita dos casos de teste mais fácil e acessível para os times de QA;
4. Gerador de Testes com base em YAML e JSON: Grande parte da documentação do funcionamento de APIs encontra-se nos formatos YAML e JSON, sendo assim, temos que a implementação de um sistema de geração de testes simples, que consiga interpretar arquivos YAML e JSON, e gerar pelo menos o esqueleto de um Feature File já atuará como uma ferramenta de grande utilidade para os times de QA, dando um ponto de partida para o desenvolvimento;
5. Serviço Software Livre Web: Para maximizar o potencial de compartilhamento das funcionalidades do modelo genérico e dar acesso a diversos desenvolvedores a um produto pronto que poderia ser introduzido facilmente no seu processo de desenvolvimento de software, a disponibilização da ferramenta em um serviço Software Livre Web seria uma maneira eficaz de atingir este objetivo, pois permitiria que qualquer desenvolvedor pudesse acessar e contribuir com a ferramenta, garantindo grande nível de manutenção e evolução do produto, evitando que eventuais mudanças nos comportamentos genéricos de uma API tornem a ferramenta obsoleta.

A ferramenta que irá implementar os elementos acima será denominada AT4QA, ou Autonomous Testing for Quality Assurance (Testes Autônomos para Garantia de Qualidade), e terá como objetivo garantir que os elementos fundamentais do AGBDD sejam contemplados, sendo então um exemplo geral de aplicação deste paradigma para Web APIs.

Com estes aspectos definidos, podemos então seguir para a implementação da ferramenta, que será descrita no próximo Capítulo.

Capítulo 3

Desenvolvimento do Projeto AT4QA

Como discorrido anteriormente, este projeto teve origem durante um estágio na empresa Opus Software em que o autor atuou no desenvolvimento de testes automatizados com base no paradigma BDD. Portanto, a parte inicial do desenvolvimento foi feita durante o estágio, e contempla a definição dos Statements genéricos para testes de Web APIs, a implementação do sistema de interpretação dos Feature Files utilizando a biblioteca Pytest BDD, a implementação de um sistema de execução de testes em paralelo, e a implementação do sistema de máscaras e dicionários.

Após este desenvolvimento inicial, foi feita a proposta de externalizar o projeto para a comunidade de software livre como parte do TCC do autor. Com isso, foi feita a separação do código do projeto em um repositório público no GitHub, dentro da organização da Opus Software, e foi realizado um fork para que o autor pudesse contribuir com o projeto. A partir deste ponto, o desenvolvimento foi feito de forma individual para o projeto de TCC em questão.

A seguir, serão detalhados os principais pontos do desenvolvimento do projeto com mais detalhes.

3.1 Interpretação de Feature Files

A interpretação de Feature Files é feita utilizando a biblioteca Pytest BDD, com ela utilizamos a estrutura do Pytest para interpretar os Feature Files escritos em Gherkin e executar os testes. Os testes são separados em 3 pastas principais:

- features: A pasta features contém os Feature Files, ela será a pasta principal de onde o sistema irá extrair os casos de teste. Os Feature Files devem ser escritos em Gherkin para que o sistema possa interpretá-los, logo os arquivos desta pasta devem ter a extensão .feature;
- step_defs: A pasta step_defs contém os arquivos de implementação dos passos dos testes, também denominados Step Definitions, ou Definições de Passos. Nesta pasta, teremos arquivos Python que irão realizar a interpretação de cada Statement genérico, ou seja, os arquivos desta pasta serão responsáveis por tomar um Statement que

esteja definido e atribuí-lo a determinado comportamento em código, fazendo assim com que cada Statement represente uma ação computacional que será realizada ao ser utilizado em um caso de teste. No caso deste trabalho, esta foi dividida em duas subpastas:

steps: Esta contém os arquivos que agrupam os Statements genéricos com base em suas funcionalidades comuns, por exemplo, o arquivo test_Storage.py possui a ligação entre os Statements relacionados a armazenamento de valores e as classes que implementam este armazenamento;

custom: Esta pasta contem o arquivo test_Custom.py, que recebe todos os arquivos da pasta steps como plugins, e é o arquivo central que sempre será executado, pois ele atua como um núcleo para todos os Statements genéricos, assim temos um nível de abstração e modularização maior, podendo definir os Statements genéricos em arquivos separados e agrupá-los em um único arquivo central, além de facilitar a execução do pytest tendo um único arquivo a ser executado na linha de comando;

- test_classes: A pasta test_classes contém os arquivos de implementação das classes de teste, que são classes Python que contém os códigos que realizam as ações atribuídas nos Step Definitions, ou seja, são classes que contém as implementações dos métodos que serão executados ao serem chamados por um Statement.

Portanto, a interpretação dos Feature Files é feita da seguinte forma: o sistema lê os arquivos da pasta features, interpreta cada Statement destes, para cada Statement ele irá buscar o Step Definition correspondente na pasta step_defs, e então irá executar o método atrelado a este Statement na classe de teste correspondente. Com isso, o sistema poderá executar os casos de teste definidos nos Feature Files de forma automatizada.

3.1.1 Tags

Além disso, temos também o sistema de Tags, estas são palavras-chave precedidas por um @, que atuam como marcações que podem ser atribuídas nos Feature Files, tanto na Feature em si quanto em Scenarios específicos, para que o sistema possa filtrar quais casos de teste devem ser realizados em determinada execução. Assim, evitamos que todos os casos de teste sejam executados sempre e garantimos que serão executados apenas os casos de interesse no momento.

As Tags podem ser separadas pelas palavras-chave 'and' e 'or', onde 'and' indica que os Scenarios devem conter todas as Tags especificadas, e 'or' indica que os Scenarios devem conter ao menos uma das Tags especificadas. Por exemplo, se temos um Feature File com 3 Scenarios, onde o primeiro Scenario possui a Tag @success, o segundo Scenario possui a Tag @positive, e o terceiro Scenario possui as Tags @success e @positive, e executamos o sistema com a especificação 'success and positive', apenas o Scenarios 3 será executado, pois o Scenario 1 não possui a Tag @positive. Agora se usarmos a especificação 'success or positive', os Scenarios 1, 2 e 3 serão executados, pois todos possuem ao menos uma das Tags especificadas.

3.1.2 Conftest.py

Temos também o arquivo conftest.py, onde são definidas configurações gerais da execução de testes, nele podemos especificar comportamentos que devem ser tomados antes ou após a execução de todos os testes. No caso deste projeto, foi definido neste arquivo a configuração para que o sistema possa realizar abertura e fechamento de conexões com bancos de dados, fazendo com que o sistema possa manter uma única conexão aberta para todos os testes realizados em uma execução, e evitar problemas de performance com a abertura e fechamento de conexões a todo momento. Ou seja, em uma execução de testes, todos os testes dentro da mesma execução compartilharão a mesma conexão com o determinado banco de dados.

3.1.3 Pytest.ini

A biblioteca Pytest BDD também provê do arquivo pytest.ini, onde são definidas configurações gerais do Pytest, como por exemplo a pasta onde estão os Feature Files, configurações de execução paralela, entre outras. Nele foi definido o uso de paralelismo na execução dos testes, com atribuição automática de núcleos de processamento, garantindo que os testes estejam utilizando o máximo de recursos disponíveis na máquina onde estão sendo executados. Esta é uma consideração importante para garantir a eficiência da execução dos testes, pois com a execução paralela podemos garantir que os testes sejam executados de forma mais rápida, apresentando um aumento exponencial de velocidade de execução ao ser implementado, onde Feature files que levavam 20 minutos passaram a levar 2 minutos com a execução paralela.

A execução paralela também introduz um novo desafio, que é a garantia de que os testes sejam executados de forma isolada, de modo que não haja interferência entre estes, ou seja, garantindo que um teste não altere o estado de outro. Para garantir isso, é necessário tomar cuidado com o uso da memória de cada teste para evitar que não haja variáveis que preservem dados antigos que possam ser utilizados em testes posteriores, ou que haja variáveis que possam ser alteradas por um teste e afetar o resultado de outro teste. Para tal, foram implementadas medidas de isolamento garantindo que todas as variáveis sejam atribuídas ou só sejam utilizadas após serem limpadas em cada teste.

Com esse sistema, tornamos possível a execução automatizada de testes a partir de um Feature File escrito em Gherkin.

3.2 Statements Genéricos

Feita a interpretação dos Feature Files, o próximo passo foi a definição dos Statements genéricos que seriam utilizados para a composição dos casos de teste. Para tal, foi determinado quais eram os elementos que apresentavam código repetido, e como estes elementos poderiam ser abstraídos para que fosse utilizado o mesmo código nestes diferentes casos de teste que apresentavam comportamento similar.

O primeiro passo foi abstrair o ato de realizar uma chamada a uma API em si, notamos que este é composto sempre pelos mesmos elementos básicos, ou seja, sempre é necessário definir o método HTTP, a URL, os headers, e o corpo da requisição. Portanto, foi definido

o Statement genérico que tornava estes elementos básicos parâmetros de entrada, e a partir disso, o sistema poderia realizar a chamada a API com base nestes parâmetros de maneira genérica, sem criar uma implementação específica e um Statement específico para cada chamada de API.

O segundo passo foi abstrair o ato de realizar a validação de uma resposta de uma API, notamos que este também é composto sempre pelos mesmos elementos básicos, ou seja, sempre é necessário definir o status code esperado, os headers esperados, e o corpo da resposta esperado. Portanto, foram definidos 2 Statements genéricos um para a análise específica do status code, pois este é um elemento fundamental da resposta que muitas vezes é a única informação que queremos analisar, e outro para a análise da resposta em si, que analisa os headers e o corpo da resposta. Com isso, o sistema poderia realizar a validação da resposta de uma API de maneira genérica.

Em seguida, o mesmo processo de abstração foi aplicado para os elementos restantes que notamos ser utilizados em diversos casos de teste, como por exemplo a validação de um valor específico em uma chamada de banco de dados, o armazenamento de um valor retornado para uso comparativo posterior, entre outros.

Ao fim deste processo de abstração, tínhamos um conjunto de Statements genéricos que poderiam ser utilizados para a definição de casos de teste de forma independente pelo time de QA, que utilizassem a mesma implementação. Com isso, garantimos que o código de testes seja mais limpo, organizado e reutilizável, além de garantir que os testes sejam mais fáceis de serem escritos e mantidos. Não é de interesse deste Capítulo ou do corpo central deste trabalho a listagem completa dos Statements genéricos devido a grande quantidade destes, logo, o glossário completo de Statements genéricos pode ser encontrado no Anexo A.

3.3 Dialética Diegética

Após a definição dos Statements genéricos, foi implementado o sistema de máscaras e dicionários que são utilizados para a definição de Statements customizados que não estejam contemplados nos Statements genéricos, utilizando-os como base. Isso permite resumir processos repetitivos ou complexos em um único Statement customizado e diminui a parametrização necessária para a escrita da máscara, tornando-a mais fácil de manter e editar do que um Feature File genérico em si.

A máscara possui a mesma sintaxe básica de um Feature File, porém permitindo a injeção de valores diretamente nos Statements e também permitindo o uso de Statements customizados. Estes precisam estar definidos em um dicionário, que é um arquivo especial que deve ser providenciado junto com a máscara para que o sistema possa interpretar os Statements customizados.

O dicionário é um arquivo com sintaxe similar a de um Feature File, no entanto, os nomes dos cenários são substituídos pelo Statement customizado que será definido, e os passos são substituídos por Statements genéricos que ele deverá resumir, incluindo os parâmetros que serão utilizados, parâmetros estes que podem ser fixados na definição do Statement customizado, ou podem ser definidos como variáveis no Statement customizado

e atribuídos na máscara.

Para realizar esta tradução tanto a máscara como o dicionário são lidos e interpretados pelo sistema, traduzidos para dicionários Python, e então os Statements customizados são interpretados e substituídos por sua definição no dicionário, gerando então um Feature File final que será executado em si.

Desta maneira, o sistema permite que o profissional de QA precise interagir apenas com a máscara resumida e não com o Feature File genérico, tornando o processo de escrita de testes mais simples e rápido, além de garantir que as definições dos casos de teste sejam mais legíveis.

Descrições mais detalhadas das sintaxes da máscara e do dicionário podem ser encontradas no Anexo B.

3.4 Serviço Web

Para poder disponibilizar este sistema de maneira mais acessível, também foi desenvolvido um serviço Web que permite a execução dos testes de forma mais amigável, sem a necessidade de conhecimento técnico para a execução dos testes. Este serviço Web foi desenvolvido utilizando a biblioteca Flask, que é uma biblioteca Python para desenvolvimento de aplicações Web.

O serviço foi desenvolvido para atuar como uma interface gráfica para o sistema de execução de testes, ele é composto pelas seguintes páginas:

- **Home**: A página inicial do serviço, onde os módulos do sistema são apresentados;
- **AGBDD**: Uma página que contém informações gerais sobre a definição do AGBDD e do GAT;
- **Tester**: Nesta página será possível fazer o upload de um ou mais Feature Files, definir as Tags que serão utilizadas para filtrar os testes e então executá-los. Após a execução, será possível visualizar o relatório dos testes, com a quantidade de testes que foram sucedidos, a quantidade de falhas e as descrições específicas de cada caso;
- **Glossary**: Aqui será possível encontrar uma lista com todos os Statements genéricos disponíveis para o usuário, com uma breve descrição de cada um e exemplos de uso;
- **Generator**: Nesta página será possível fazer o upload de um ou mais arquivos YAML ou JSON, que contenham documentação de uma API, para então gerar um esqueleto base de Feature File com base nesta documentação providenciada;
- **Editor**: Nesta página teremos um editor de Feature Files, onde será possível fazer upload de um Feature File, e então editá-lo, com funcionalidades voltadas para facilitar a escrita por parte dos profissionais de QA como contador de parâmetros, verificador de sintaxe, edição direta de parâmetros e correção da indentação;
- **Translator**: Nesta página será possível fazer o upload de uma máscara e/ou dicionário, e então traduzir máscaras selecionadas para um Feature File genérico.

3.5 HomePage

A página inicial do Projeto AT4QA terá como objetivo apresentar aos usuários os módulos disponibilizados pelo sistema, de maneira a explicar brevemente a utilidade de cada um e direcionar o usuário para a página que deseja visitar.

A Figura 3.1 mostra a interface da página inicial do sistema.

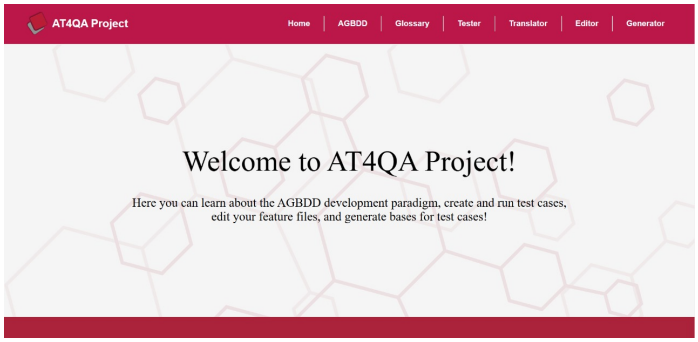

Figura 3.1: *Página inicial do sistema*

3.6 AGBDD

A página AGBDD terá como propósito disponibilizar ao usuário informações básicas sobre o modelo GAT e o paradigma AGBDD, servindo como ponto de partida para que os usuários busquem mais informações, contendo um link os direcionando para este trabalho, onde poderão encontrar uma descrição mais detalhada da teoria.

A Figura 3.2 mostra a interface da página AGBDD do sistema.

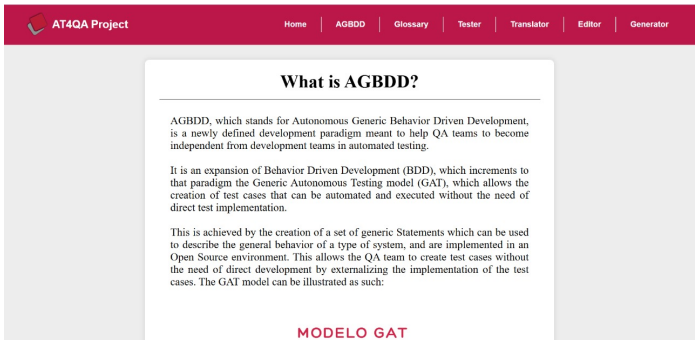

Figura 3.2: *Página de introdução ao AGBDD*

3.7 Glossary

A página Glossary terá o glossário que contém a lista de todos os Statements genéricos disponíveis para o usuário, com uma breve descrição de cada um e exemplos de uso. Esta página servirá como referência para os profissionais de QA que desejam utilizar o sistema, para que possam entender quais Statements genéricos estão disponíveis e como utilizá-los.

A Figura 3.3 mostra a interface da página Glossary do sistema.

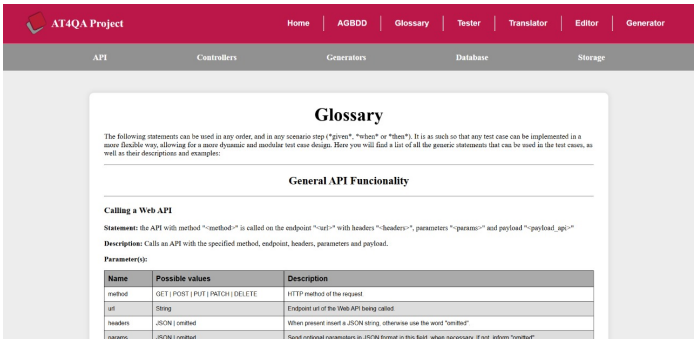

Figura 3.3: *Página contendo glossário de Statementes genéricos*

3.8 Execução de Testes

A execução de testes será feita utilizando o sistema de interpretação descrito anteriormente, onde o sistema Web irá permitir o upload de Feature Files, a inserção de Tags e o download dos relatórios resultantes dos testes. Também serão listados todos os Feature Files presentes na pasta features, e será possível fazer a deleção de Feature Files determinados.

Ao executar os testes, o sistema irá percorrer todos os Feature Files com base nas Tags inseridas, executar os casos de teste filtrados mostrando o progresso da execução em uma janela de console, e ao fim, irá gerar dois relatórios, um em formato JSON e outro em formato HTML, que serão disponibilizados para download.

As Figuras 3.4, 3.5, 3.6 e 3.7 mostram a interface do sistema de execução de testes.

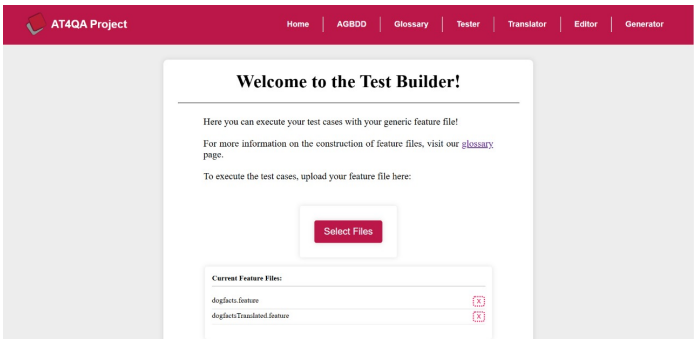

Figura 3.4: *Página de execução de testes - Área de upload*

3.9 Tradutor de máscaras

O tradutor de máscaras será implementado de maneira similar ao gerador de Feature Files, também utilizando uma classe Python, que irá receber as máscaras e dicionários, interpretá-los como dicionários Python, e então percorrê-los para substituir os Statements customizados da máscara de acordo com suas definições encontradas no dicionário equivalente.

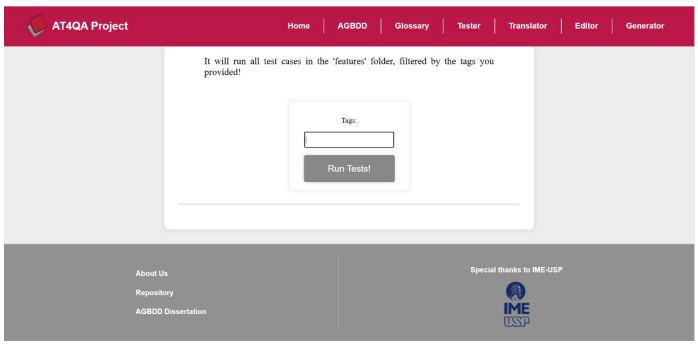

Figura 3.5: *Página de execução de testes - Antes de executar os testes*

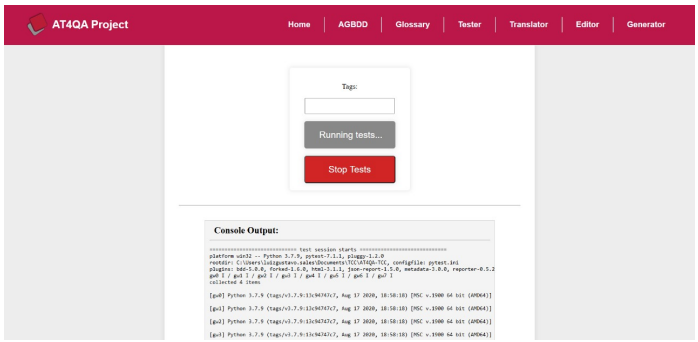

Figura 3.6: *Página de execução de testes - Durante a execução dos testes*

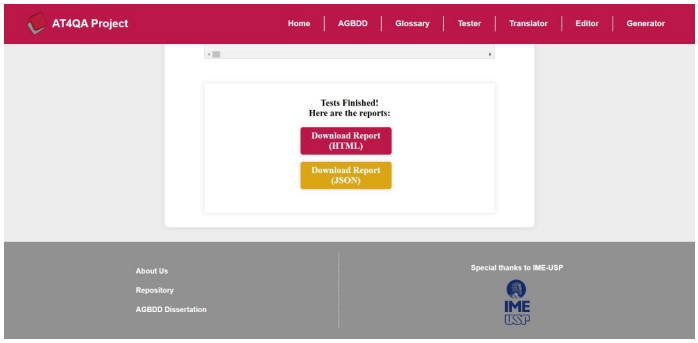

Figura 3.7: *Página de execução de testes - Após a execução dos testes*

Será então possível fazer upload dos arquivos .mask e .dict, selecionar as máscaras que devem ser traduzidas e os dicionários utilizados para tal, através de checkboxes, e então, utilizar o botão 'Translate!' para que os arquivos selecionados sejam traduzidos em Feature Files.

Feita a tradução, os arquivos serão automaticamente salvos na pasta features.

As Figuras 3.8 e 3.9 mostram a interface do sistema de tradução de máscaras.

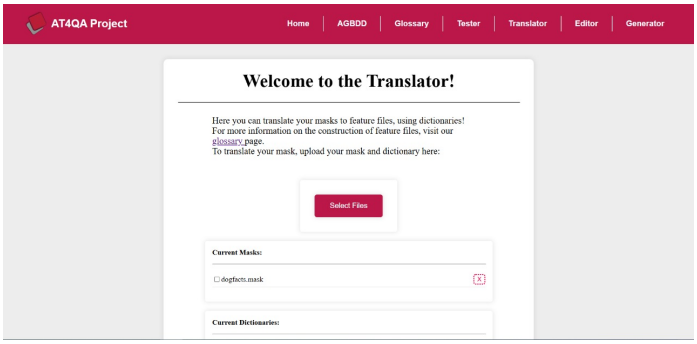

Figura 3.8: *Página de tradução de máscaras - Área de upload*

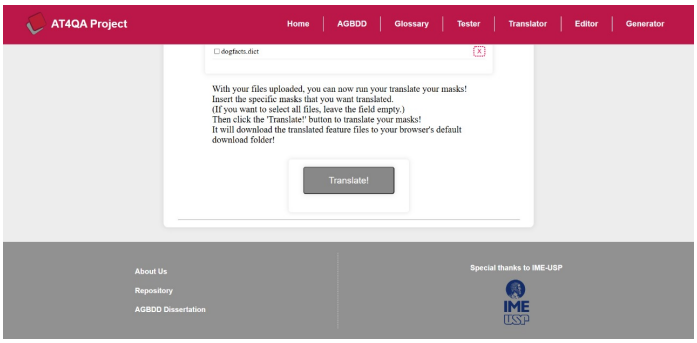

Figura 3.9: *Página de tradução de máscaras - Área de tradução*

3.10 Editor de Feature Files

O editor de Feature Files será feito utilizando a biblioteca Ace, que é uma biblioteca JavaScript para edição de código, e será integrado ao sistema Web para permitir a edição de Feature Files diretamente no navegador. O editor prevê funcionalidades como contador de parâmetros, verificador de sintaxe, edição direta de parâmetros e correção da identação, para facilitar a escrita de Feature Files por parte dos profissionais de QA.

A interface do sistema Web irá permitir o upload de um arquivo Feature File, onde as funcionalidades descritas acima estarão disponíveis, e então será possível fazer o download do arquivo editado.

As Figuras 3.10 e 3.11 mostram a interface do sistema de edição de Feature Files.

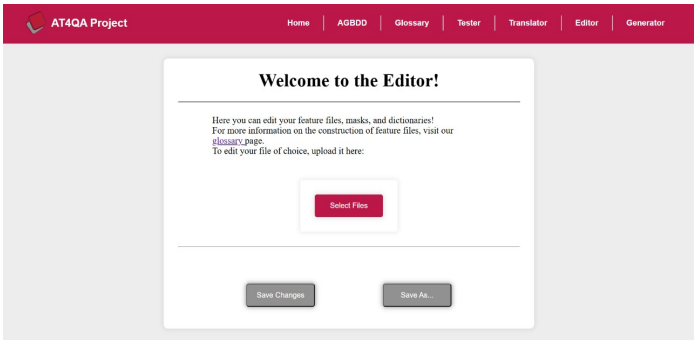

Figura 3.10: *Página de edição de testes - Área de upload*

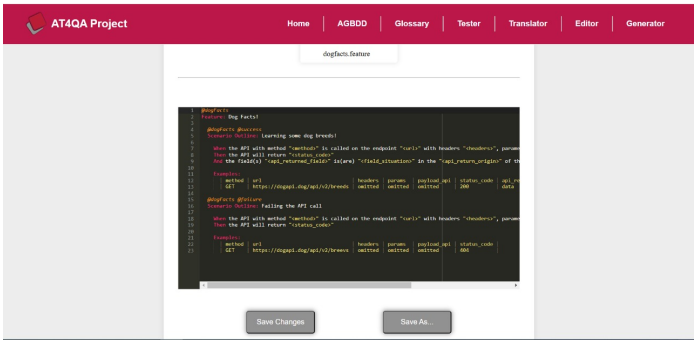

Figura 3.11: *Página de edição de testes - Área de edição e download*

3.11 Geração de Testes

A geração de testes será feita utilizando uma classe Python, que irá receber um arquivo YAML ou JSON, interpretá-los como dicionários Python, e então gerar um esqueleto de Feature File com base nestes dicionários. O esqueleto gerado irá cobrir os casos simples apresentados na documentação, como caso de sucesso e default, por exemplo. Desta maneira, podemos ter um ponto de partida e um exemplo de uso de statements genéricos para que os profissionais de QA possam utilizar como base para a escrita de testes mais complexos.

O sistema de geração de Feature Files terá uma interface similar a de tradução de máscaras, onde será possível fazer o upload de um arquivo YAML ou JSON para a pasta documentation, escolher qual o arquivo que será usado como base para a geração, definir quais os endpoints que serão testados, e então gerar os Feature Files, que serão disponibilizados para download.

As Figuras 3.12 e 3.13 mostram a interface do sistema de geração de testes.

3.12 Repositórios

O repositório principal do desenvolvimento feito durante o estágio na Opus Software pode ser acessado no link: https://github.com/Opus-Software/AT4QA.

Já o repositório do desenvolvimento feito para este trabalho pode ser acessado no link:

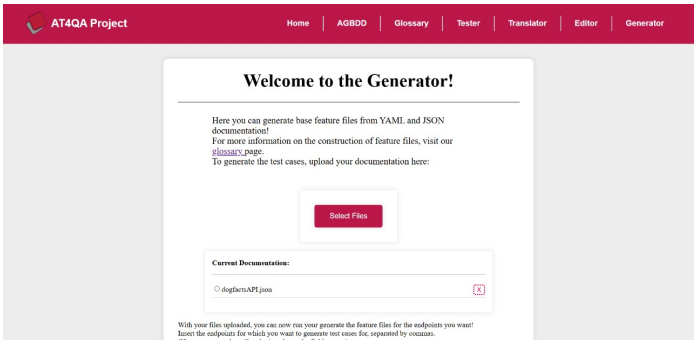

Figura 3.12: *Página de geração de testes - Área de upload*

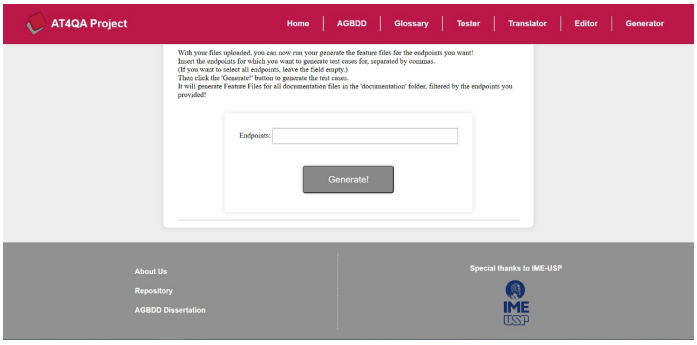

Figura 3.13: *Página de geração de testes - Área de geração*

https://github.com/LuizGustavoP/AT4QA-TCC.

Finalizado o desenvolvimento do projeto, as mudanças realizadas no repositório deste trabalho serão replicadas no repositório principal da Opus Software, onde o desenvolvimento do projeto será continuado indefinitivamente pela comunidade de software livre.

3.13 Considerações Finais

O Projeto AT4QA pôde ser desenvolvido com sucesso, implementando todas as funcionalidades básicas que devem ser atendidas por uma ferramenta de automação de testes fundamentada no AGBDD, incluindo a execução de testes, a tradução de máscaras, a geração de casos de teste e a edição de Feature Files.

No entanto, este não é o fim do desenvolvimento, pois como discorrido anteriormente, este é um projeto de software livre, e como tal, está aberto para contribuições da comunidade, podendo evoluir futuramente para incluir e atender muitos outros requisitos e funcionalidades não discutidas neste trabalho. Com isso, podemos então seguir para o Capítulo final, onde iremos discutir as conclusões deste trabalho.

Capítulo 4

Conclusão

Ao longo deste trabalho, foi apresentada uma proposta de um novo modelo de testes, um novo paradigma de desenvolvimento e uma ferramenta que implementa um exemplo de aplicação deste paradigma. As áreas de testes de software e desenvolvimento ágil estão em constante evolução e passam por frequentes transformações que mudam completamente a forma como as equipes de desenvolvimento de software trabalham.

Com isso em mente, acreditamos que as propostas apresentadas neste trabalho possam ser uma pequena parte dessa incessável metamorfose, contribuindo para a constante evolução das práticas de desenvolvimento de software e testes de software. O processo de desenvolvimento do Projeto AT4QA, a definição do modelo GAT e a definição do paradigma AGBDD apresentou ótimos resultados internamente durante o estágio do autor, e esperamos que possa ser tão útil para outras equipes de desenvolvimento, que buscarem adotar as práticas aqui definidas.

Sendo um projeto desenvolvido como Software Livre, o desenvolvimento do Projeto AT4QA está aberto para contribuições de qualquer pessoa interessada em colaborar. Ainda temos muitos elementos que serão adicionados no futuro com a continuidade do desenvolvimento, como um sistema de tutorial para facilitar a comunicação das funcionalidades, localização de Statements para diferentes idiomas, revisões de experiência de usuário e interfaces, expansão da compatibilidade do gerador, entre outros.

Logo, acreditamos que este projeto possui um grande potencial para se tornar uma ferramenta aberta, colaborativa e de grande utilidade para equipes de desenvolvimento de software.

Anexo A

Anexo 1 - Glossário de Statements Genéricos

The following statements can be used in any order, and in any scenario step (*given*, *when* or *then*). It is as such so that any test case can be implemented in a more flexible way, allowing for a more dynamic and modular test case design. Here you will find a list of all the generic statements that can be used in the test cases, as well as their descriptions and examples:

A.1 General API Funcionality

Calling a Web API

Statement: the API with method "<method>"is called on the endpoint "<url>"with headers "<headers>", parameters "<params>"and payload "<payload_api>"

Description: Calls an API with the specified method, endpoint, headers, parameters and payload.

Parameter(s):

Tabela A.1: *Parâmetros para chamadas de API*

Name	Possible values	Description
method	GET | POST | PUT | PATCH | DELETE	HTTP method of the request.
url	String	Endpoint URL of the Web API being called.

Name	Possible values	Description
headers	JSON | omitted	When present, insert a JSON string; otherwise, use the word "omitted".
params	JSON | omitted	Send optional parameters in JSON format in this field, when necessary. If not, inform "omitted".
payload_api	JSON | omitted	Send the payload in JSON format, when necessary. If not, inform "omitted".

Example(s):

- And the API with method "POST"is called on the endpoint "http://website.com"with headers "omitted", parameters "omitted"and payload "{bodyInfo: "info"}"
- Then the API with method "GET"is called on the endpoint "http://website.com"with headers "{Accept: OK}", parameters "{id: 2}"and payload "omitted"

Check the status code of the last API call

Statement: the API will return "<status_code>"

Description: Validates the status code of the last API call.

Parameter(s)

Tabela A.2: *Parâmetros para checagem de status code*

Name	Possible values	Description
status_code	Number	Expected status code.

Example(s):

- And the API will return "200"
- Then the API will return "201"

Check specific fields in the response of the last API call

Statement: the value of the field(s) "`<api_returned_field>`"present in "`<api_return_origin>`"of the API response has the value(s) "`<comparison_type>`"to "`<api_response_expected_values>`"respectively

Description: Validates if the values of the fields returned by the API are equal to the expected values.

Parameter(s):

Tabela A.3: *Parâmetros para validação de resposta de API*

Name	**Possible - values**	**Description**
api_returned_field	String	Names of the fields returned by the API whose values will be compared to the expected values. When comparing more than one field, the names should be separated by double at symbols ("@@"). If the source of the value is the body of the API response, the entire path of the field must be given, considering the fields of type array.
api_return_origin	body | header	Source of the fields returned by the API whose values will be compared to the expected values. If "body", it indicates that the fields to be analyzed come from the response body. If "header", it indicates that the fields to be analyzed come from the response header.
comparison_type	equal | greater or equal | less or equal | different	The type of comparison that will be performed, in relation to the "api_response_expected_values" parameter.
api_response_expected_values	String	List of expected values that will be compared to the values of the parameterized fields in the API response. The comparisons will be positional in relation to the fields defined in the "api_returned_field"parameter. In other words, the first element of "api_returned_field"should have the value of "api_response_expected_values", and so on. For a list of values with more than one element, separate the values with double at symbols ("@@").

Example(s):

- And the value of the field "data.test" present in the "body"of the API response has the value "equal"to "123"
- Then the value of the field "data.test" present in the "body"of the API response has the value "greater or equal"to "123"

Check the presence or absence of specific fields in the response of the last API call

Statement: the field(s) "<api_returned_field>"is(are) "<field_situation>"in the "<api_return_origin>"of the API response

Description: Validates the existence of specific fields in the response of the last API call.

Parameter(s):

Tabela A.4: *Parâmetros para verificação de existência de campos no retorno de API*

Name	Possible - values	Description
api_returned_field	String	Names of the fields returned by the API whose existence is to be verified. When verifying the existence of more than one field, the names must be separated by double at symbols ("@@"). If the source of the value is the body of the API response, the entire path of the field must be given, considering the fields of type array.
field_situation	present | absent	When "present", it will validate if the fields defined in "api_returned_field"were found in the response of the last API call made. When "absent", it will validate if the fields defined in "api_returned_field"are not present in the response of the last API call made.
api_return_origin	body | header	Source of the fields returned by the API whose existence is to be verified. If "body", it indicates that the fields to be analyzed come from the response body. If "header", it indicates that the fields to be analyzed come from the response header.

Example(s):

- And the field(s) "data"is(are) "present"in the "header"of the API response
- And the field(s) "data.test"is(are) "absent"in the "body"of the API response

---

Check the values of fields returned in an API call located in an unordered structure

Statement: the value of field "<api_returned_field>"located on the body of the API response in an unordered structure that has the field(s) "<api_related_fields>"with value(s) "<api_related_values>"has value equal to "<api_returned_expected_values>"

Description: Validates if the values of the parameterized fields returned by the last API called are equal to the expected values, taking into account a value contained in an unordered structure like an array.

Parameter(s):

Tabela A.5: *Parâmetros para verificação de igualdade de retorno de API*

Name	Possible values	Description
api_returned_field	String	Contains the name or path of the field in the API that you want to store, where the unordered structure should be denoted with the syntax "[?]". The path "data[?].name", for example, aims to store the "name"of some data, which has other identifying values that will be passed in the parameters "api_related_fields"and "api_related_values".
api_related_fields	String	Contains the name or path of the field that will be used as an identifier for the value we want to find. If there are multiple levels of unordered structures, each level should be separated by "@@". If multiple values are needed as identifiers at the same level, each value should be separated by "&&&".

Tabela A.5: *Parâmetros para verificação de igualdade de retorno de API*

Name	Possible values	Description
api_related_values	String	Contains the expected values of the fields that will be used as identifiers in the "api_related_fields"parameter. If there are multiple identifier values at the same level, each value should be separated by "&&&". In the case of multiple levels of unordered structures, each level should be separated by "@@".
api_returned_expected_values	String	Expected value to be compared with the value returned by the API in the field passed by the "api_returned_field"parameter.

Example(s):

- And the value of field `data[?].name` located on the body of the API response in an unordered structure that has the field(s) `id&&&class` with value(s) `01&&&A2` has value equal to `John`
- And the value of field `data.dataGroup.groupTypes[?].groupId[?].name` located on the body of the API response in an unordered structure that has the field(s) `type&&&id@@surname` with value(s) `01&&&532@@Smith` has value equal to `Albert`

A.1.1 Controllers

Pause the test execution

Statement: the test waits "<time_in_seconds>"seconds

Description: Pauses the execution of the current test for the specified period of time.

Parameter(s):

Tabela A.6: *Parâmetros para pausar execução de testes*

Name	Possible values	Description
time_in_seconds	Number	Amount of waiting time in seconds.

Example(s):

- And the test waits "180"seconds

A.1.2 Value Generators

UUID Generator

Statement: a new uuid will be generated and stored in the field named "<storage_field>"

Description: Generates a random UUID and stores the generated value in the variable with the specified name.

Parameter(s):

Tabela A.7: *Parâmetros para geração de UUID*

Name	Possible values	Description
storage_field	String	Name of the variable in which the generated UUID will be stored.

Example(s):

- And a new uuid will be generated and stored in the field named "generatedUUID"

Time generator

Statement: a time "<time>"will be generated in the format "<format>"and stored in a field named "<storage_field>"

Description: Generates a time with a parameterized format and stores the generated value in the variable with the parameterized name.

Parameter(s):

Tabela A.8: *Parâmetros para geração de horário*

Name	Possible values	Description
time	String	Moment at which the time will be generated, can be "current", "{seconds} ago"or "{seconds} ahead"
format	String	Format in which the time will be generated, this value can be "epoch", for generating epoch values, datetime values as in "date-time@@%d/%m/%y %H:%M:%S.%f", which can have its format changed by changing the order of the parameters after "@@", the word zuluex can be passed to generate a time in Zulu format without the 'z' at the end, or the word zuluz can be passed to generate a time in zulu format with 'z' at the end.
storage_field	String	Name of the variable in which the generated time will be stored.

Example(s):

- And a time "current"will be generated in the format "epoch"and stored in a field named "time"
- And a time "60 ahead"will be generated in the format "datetime@@%d/%m/%y %H:%M:%S.%f"and stored in a field named "time"
- And a time "60 ago"will be generated in the format "zuluex"and stored in a field named "time"

CRC Generator

Statement: a CRC value is calculated with the polynomial "<poly>", initial value "<init>"e XOR value "<xorValue>", for the value stored in the field "<value_field>", and will be stored as a hexdigest in the field "<storage_field>"

Description: Generate the CRC encoding according to the parameterized values for a stored variable, and store the generated CRC hexdigest in the "storage_field"variable.

Parameter(s):

Tabela A.9: *Parâmetros para geração de CRC*

Name	Possible values	Description
poly	String	Polynomial that will be used to perform the CRC.

Tabela A.9: *Parâmetros para geração de CRC*

Name	Possible values	Description
init	String	Initial value from which the CRC will be calculated.
xorValue	String	Final value with which XOR will be used in CRC.
value_field	String	Value with which the CRC encoding will be performed, the value will be interpreted with ASCII encoding.
storage_field	String	Name of the variable in which the generated CRC hexdigest will be stored.

Example(s):

- And a CRC value is calculated with the polynomial "0x0011", initial value "0xDDDD", and XOR value "0xAAAA", for the value stored in the field "crcValue", and will be stored as a hexdigest in the field "calculatedCRC"

Random string generator

Statement: a random string of size "<string_size>"will be generated and stored in a field named "<storage_field>"

Description: Generates a random string of a parameterized size and stores the generated string in the variable with the parameterized name.

Parameter(s):

Tabela A.10: *Parâmetros para geração de string aleatória*

Name	Possible values	Description
string_size	Number	Size of the random string to be generated
storage_field	String	Name of the field in which the generated string will be stored.

Example(s):

- And a random string of size "15"will be generated and stored in a field named "randomString"
- And a random string of size "30"will be generated and stored in a field named "randomString"
- And a random string of size "8"will be generated and stored in a field named "randomString"

A.1.3 Database Interactions

Define a database connection

Statement: a database connection named "<name>"with host "<host>", port "<port>", user "<user>", password "<password>"and database "<database>"is defined

Description: Defines a connection to a database, specifying the host, port, user, password, and database name.

Parameter(s):

Tabela A.11: *Parâmetros para chamadas a bancos de dados*

Name	Possible values	Description
name	String	Name of the connection.
host	String	Host of the database.
port	Number	Port of the database.
user	String	User of the database.
password	String	Password of the database.
database	String	Name of the database.

Example(s):

- And a database connection named "students_local"with host "localhost", port "5432", user "admin", password "admin"and database "students"is defined
- And a database connection named "users_local"with host "localhost", port "5432", user "admin", password "admin"and database "users"is defined

Check value of a column in a database table

Statement: on database "`<database_name>`"on table "`<database_table>`"the column "`<database_expected_column>`"must contain the value "`<equals_or_differs>` `<expected_value>`", in the row where the column "`<database_column_where>`"of type "`<database_column_where_type>`"has the value "`<where_value>`"from the field "`<field_name>`"originated from "`<field_origin>`"

Description: Checks if the content of a column in a specific table of a database is the expected value, using parametrization for the database, table, expected column, comparison column, and field that contains the expected value.

Parameter(s):

Tabela A.12: *Parâmetros para validação de valores no banco de dados*

Name	**Possible values**	**Description**
database_name	String	Name of the database where the query should be performed.
database_table	String	Name of the table where the query should be performed.
database_expected_column	String	Name of the column in the database table to be compared. This parameter can be used on a column of type "JSON"when you want to compare the value of a specific field in the JSON. To do this, simply define this parameter as "column_name>-> "JSON_field_path". For example: "data->user.name". In this case, the "expected_value"would be compared with the value of the "name"field, which is inside the "user"object in the "data"column.
equals_or_differs	equals to | differs from	When "equals to", a comparison of equality will be made between the value returned by the query and the one parameterized in the "expected value"field. When "differs from", it will be analyzed if the value returned by the query is different from the value defined in the "expected_value"parameter.
expected_value	String | equal to where | empty	What is the expected value in the column. If it is a string, it will check if the value returned by the query is equal to the value of this parameter. When "equals to where", the expected value is the same value defined in the "where_value"parameter described below. When "empty", it will analyze if the select returned "NULL"for the query.

Tabela A.12: *Parâmetros para validação de valores no banco de dados*

Name	**Possible values**	**Description**
database_column_where	String	Name of the column in the database table that will be used for comparison (where clause) in the generated selection.
database_column_where_type	uuid | varchar | number	Type of the comparison column. This parameter is necessary for the selection command to be constructed correctly, as the type of the comparison variable influences how the query should be written.
where_value	String | equals	What is the value to be used in the comparison of the selection command. If it is a string, the parameterized value will be used in the comparison query (where clause). When "equals", it indicates that the value to be used should be defined in the "field_name"parameter described below.
field_name	String | omitted	Name of the field where the value to be used in the comparison will be obtained. If it is a string, it indicates the name of the field in question. When "omitted", it indicates that the value will not come from a field, but from the "where_value"variable described above.

Tabela A.12: *Parâmetros para validação de valores no banco de dados*

Name	Possible values	Description
field_origin	response header | response body | feature file	Where the field defined in the "field_name"parameter should be retrieved from. If "response header", the field will be retrieved from the headers object of the response of the last executed API. If "response body", the field will be retrieved from the body object of the response of the last executed API. If "feature file", it indicates that the value of the field will be the value defined in the "where_value"parameter defined above.

Example(s):

- Check if the value of the "name"column in the user database is equal to "John". Using the "class"column as the comparison base and the content of the "Accept"header from the previous API call as the comparison value.
  - Then on database "students"on table "user"the column "name"must contain the value "equals toJohn", in the row where the column "class"of type "number"has the value "equals"from the field "Accept"originated from "response header"
- Check if the value of the "class"column in the user database is equal to the value present in the "Accept"header from the previous API call.
  - Then on database "students"on table "user"the column "class"must contain the value "equals toequals to where", in the row where the column "class"of type "number"has the value "equals"from the field "Accept"originated from "response header"
- Check if the value of the "class"column in the user database is equal to "3"which is defined in the examples table of the feature file.
  - Then on database "students"on table "user"the column "class"must contain the value "equals toequals to where", in the row where the column "class"of type "number"has the value "3"from the field "omitted"originated from "feature file"
- Check if the value of the "class"column in the user database is not empty. Using the "class"column as the comparison base and the content of the "Accept"header from the previous API call as the comparison value.

– Then on database "students"on table "user"the column "class"must contain the value "differs fromempty", in the row where the column "class"of type "number"has the value "equals"from the field "Accept"originated from "response header"

Update value of a column in a database table

Statement: on database "<database_name>"on table "<database_table>"the column "<update_column>"of type "<update_column_type>"must update the value of field "<update_field_column>"to the value "<update_value>"of type "<update_value_type>", in the row where the column "<database_column_where>"of type "<database_column_where_type>"has the value "<where_value>"from the field "<field_name>"originated from "<field_origin>"

Description: Update the content of a column in a specific table of a database, parameterizing the column to update, comparison column for the command, and the value to be used in the update.

Parameter(s):

Tabela A.13: *Parâmetros para atualização de valores no banco de dados*

Name	**Possible values**	**Description**
database_name	String	Name of the database where the update should be performed.
database_table	String	Name of the database table where the update should be performed.
update_column	String	Name of the column in the database table where the value update should be performed.
update_column_type	uuid | varchar | number | JSON	Type of the update column. This parameter is necessary for the update command to be constructed correctly, as the type of the comparison variable influences how the query should be written.
update_field_column	String | omitted	If the "update_column_type"is "JSON", this parameter is used to specify which field of the JSON should be updated. If the type is not JSON, this parameter should be "omitted".
update_value	String	Value to which the field should be updated.

Tabela A.13: *Parâmetros para atualização de valores no banco de dados*

Name	**Possible values**	**Description**
update_value_type	uuid | varchar | number	What is the type of the value provided in the "update_value"parameter. This parameter is necessary for the update command to be constructed correctly, as the type of the comparison variable influences how the query should be written.
database_column_where	String	Name of the column in the database table that will be used for comparison (where clause) in the generated update.
database_column_where_type	uuid | varchar | number	Type of the comparison column. This parameter is necessary for the update command to be constructed correctly, as the type of the comparison variable influences how the query should be written.
where_value	String | equal	What is the value to be used in the comparison of the update command. If it is a string, the parameterized value will be used in the comparison query (where clause). When "equals", it indicates that the value to be used should be defined in the "field_name"parameter described below.
field_name	String | omitted	Name of the field where the value to be used in the comparison will be obtained. If it is a string, it indicates the name of the field in question. When "omitted", it indicates that the value will not come from a field, but from the "where_value"variable described above.

Tabela A.13: *Parâmetros para atualização de valores no banco de dados*

Name	Possible values	Description
field_origin	response header | response body | feature file	Where the field defined in the "field_name"parameter should be searched. If "response header", the field will be searched in the headers object of the last executed API response. If "response body", the field will be searched in the body object of the last executed API response. If "feature file", it indicates that the field value will be the value defined in the "where_value"parameter defined above.

Example(s):

- Update the value of the "name"field in the "user"column of the "students"table in the "school"database to the value "Albert"
  - And on database "school"on table "students"the column "user"of type "string"must update the value of field "name"to the value "Albert"of type "string", in the row where the column "id"of type "varchar"has the value "igual"from the field "header"originated from "header"

Run an arbitrary query in a database

Statement: on database "<database_name>"run query "<query>"whose expected returned value must be equal to "<expected_value>"

Description: Run the query passed as a parameter in the parameterized database.

Parameter(s):

Tabela A.14: *Parâmetros para execução de query no banco de dados*

Name	Possible values	Description
database_name	String	Name of the database where the query should be executed.
query	String	Query that should be executed. Supports "select", "update", and "delete"commands, but they must always include the "where"clause.

Tabela A.14: *Parâmetros para execução de query no banco de dados*

Name	Possible values	Description
expected_value	String | omitted	When the parameterized query is of type "update"or "delete", this value should be filled as "omitted". In queries of type "select", it can be used to validate if the return value of the command is equal to the value defined in this parameter. Important: The "select"command that will be executed will return the value of only one column and one tuple from the database that meets the condition criteria, it does not support the return of values from multiple columns or multiple tuples.

Example(s):

- And on database "students"run query "SELECT name FROM user where id = '3';"whose expected returned value must be equal to "John"
- And on database "students"run query "DELETE FROM user where id = '2'"whose expected returned value must be equal to "omitted"

A.1.4 Storage

This section refers to storing values in a storage variable that exists at runtime only. These values can be stored using the following statements:

Store the response of an API

Statement: the value of field "<api_field>"from "<field_origin>"of the response will be stored on field named "<storage_field>"

Description: In some scenarios, it is necessary to store a certain value returned by an API call to use it in the subsequent test in another statement. The purpose of this statement is precisely that: to store a returned value in a variable whose name is defined via a parameter.

Parameter(s):

Tabela A.15: *Parâmetros para armazenamento de valores*

Name	Possible values	Description
api_field	String	Contains the name of the API field that should be stored. If the field originates from the body of the API response, the entire path must be defined, considering fields of type array. For example, the path "data.classes.students[0].parents[0].name"aims to store the "name"of the first "parent", of the first "student"defined within the "class", which is located within the root "data".
field_origin	body | header	Indicates where to locate the field that you want to store: in the "body"or "header"of the response from the last API call made.
storage_field	String	Name of the variable that will be created and will store the value present in the "api_field"field.

Example(s):

- And the value of field "data.classes.students[0].parents[0].name"from "body"of the response will be stored on field named "name"
- And the value of field "data.classes.students[0].siblings[0].age"from "body"of the response will be stored on field named "age"

Store the response of an API for unordered data structures

Statement: the value of field "<api_returned_field>"present in the API body response related to the field(s) "<api_related_fields>"with value(s) "<api_related_values>"stored in field named "<storage_field>"

Description: In some scenarios, it is necessary to store a certain value returned by an API call to use it in the subsequent test in another statement. However, there are cases where we do not know exactly in which position of an unordered structure, such as an array, this value is located. Therefore, the purpose of this statement is precisely that: to store a returned value in a variable where we do not know exactly in which position of a certain structure it is located, using adjacent values as identifiers to find it.

Parameter(s):

Tabela A.16: *Parâmetros para armazenamento de valores não ordenados*

Name	Possible values	Description
api_returned_field	String	Contains the name or path of the API field that you want to store, where the unordered structure should be denoted with the syntax "[?]". For example, the path "students[?].name"aims to store the "name"of a student that has other identifying values, which will be passed in the "api_related_fields"and "api_related_values"parameters.
api_related_fields	String	Contains the name or path of the field that will be used as an identifier to find the desired value. If there are multiple levels of nested unordered structures, each level should be separated by "@@". If multiple values are needed as identifiers at the same level, each value should be separated by "&&&".
api_related_values	String	Contains the expected values of the fields that will be used as identifiers in the "api_related_fields"parameter. If there are multiple identifier values at the same level, each value should be separated by "&&&". In the case of multiple levels of nested unordered structures, each level should be separated by "@@".
storage_field	String	Name of the variable that will be created and will store the value present in the "api_returned_field"field.

Example(s):

- And the value of field "students[?].name"located on the body of the API response in an unordered structure that has the field(s) "id"with value(s) "001"stored in field named "name"
- And the value of field "data.school.classes[?].students[?].name"located on

the body of the API response in an unordered structure that has the field(s) "id&&&meanGrades@@age with value(s) "001&&&10@@15"stored in field named "name"

Store the response of a database query

Statement: a value of database "<database_name>"is adquired by the query "<query>"and stored in field named "<storage_field>"

Description: In some scenarios, it is necessary to store a certain value returned by a query to a specific database, in order to use it in another statement. The purpose of this statement is precisely that: to store a value, or a composite structure of values (for cases where the select retrieves more than one value), returned in a variable whose name is defined via a parameter.

Parameter(s):

Tabela A.17: *Parâmetros para armazenamento de valores retornados pelo banco de dados*

Name	Possible values	Description
database_name	String	Name of the database where the query should be executed.
query	String	Query that should be executed. Supports only the "select"command and must always include the "where"clause.
storage_field	String	Name of the variable that will be created and will store the value returned by the query "query".

Example(s):

- And a value of database "school"is adquired by the query "select meanGrade from students where name = 'John' order by age desc"and stored in field named "grade"
- And a value of database "school"is adquired by the query "select meanGrade from students where name = 'Joseph' and stored in field named "grade"

A.1.5 How to use the stored values

Once a value is stored in the desired variable, it can be used in subsequent API call statements whenever needed. To do this, simply indicate the variable name enclosed in hashtags (`"#"`) in the feature file parameters. The automation tool will replace the

variable name with its stored value during test execution. The replacement can be done in parameters that represent:

- Path
- Payload
- Headers
- Params
- Expected values in comparison statements

For example, for a simple value: A path parameter defined as "/dogFacts/-fact/#factNumber#"will be replaced with the previously stored value during test execution, resulting in something like "/dogFacts/fact/01". For example, for a composite value: A path parameter defined as "/dogFacts/fact/#factNumber[0]#"will be replaced with the first value from the stored structure during test execution, resulting in something like "/dogFacts/fact/01".

In the case of composite structures, the order of the indexes will be the same as the values returned by the "select"query.

Anexo B

Anexo 2 - Exemplos da Dialética Diegética

Neste anexo, iremos apresentar com mais detalhes as sintaxes das máscaras e dicionários da Dialética Diegética.

B.1 Máscaras

As máscaras foram criadas com o intuito de serem substitutas dos Feature Files genéricos, para que a escrita e manutenção dos cenários fosse mais simples e intuitiva. Temos na Figura B.1 um exemplo de uma máscara.

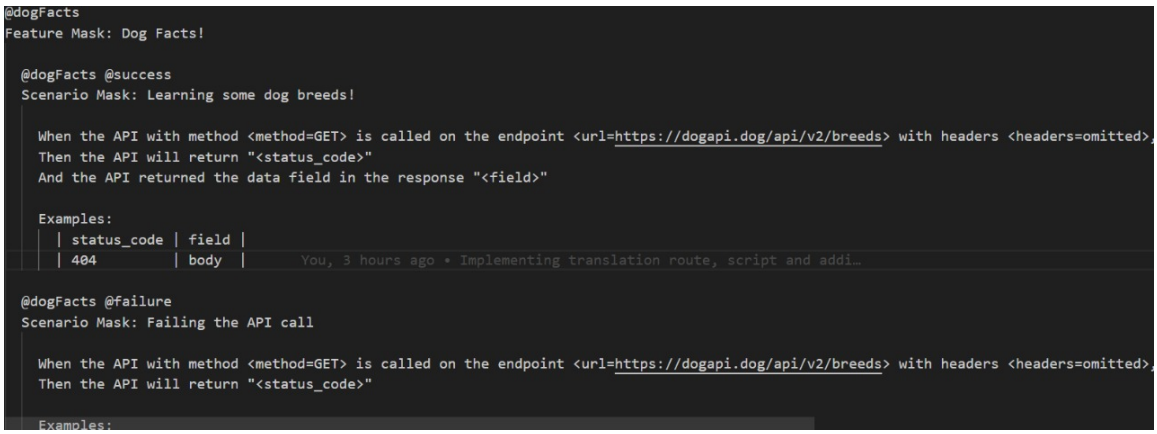

Figura B.1: *Exemplo de máscara*

Como podemos notar, a sintaxe é bem parecida com o Gherkin, com a única diferença sendo a adição de injeção de parâmetros diretamente nos Statements, onde o nome no lado esquerdo do sinal de igual representa o nome da coluna que este valor irá ocupar, e o no lado direito temos o valor em si. Nestes casos de injeção, é necessário que não haja aspas entre o parâmetro, para que a interpretação seja feita corretamente.

Estas pequenas mudanças sintáticas fazem uma grande diferença na legibilidade e manutenção dos cenários, pois diminui a parametrização da tabela de examples, como podemos ver comparando a máscara da Figura B.1 com o Feature File da Figura B.2.

```
@dogFacts
Feature: Dog Facts!

  @dogFacts @success
  Scenario Outline: Learning some dog breeds!

    When the API with method "<method>" is called on the endpoint "<url>" with headers "<headers>", parameters "<params>" and payload "<pay
    Then the API will return "<status_code>"        You, 2 months ago • TCC-3: …
    And the field(s) "<api_returned_field>" is(are) "<field_situation>" in the "<api_return_origin>" of the API response

    Examples:
      | method | url                            | headers | params  | payload_api | status_code | api_returned_field | field_situation |
      | GET    | https://dogapi.dog/api/v2/breeds | omitted | omitted | omitted     | 200         | data               | presente        |
```

Figura B.2: *Exemplo de Feature File genérico*

Além disso, podemos notar que a máscara apresenta um último Statement que não pertence ao conjunto de Statements genéricos, ou seja, um Statement customizado. Para que este Statement seja reconhecido, é preciso que haja um dicionário que contenha sua descrição, como veremos a seguir.

B.2 Dicionários

Os dicionários foram criados com o intuito de dar ao QA a capacidade de definir seus próprios Statements de maneira resumida e reutilizável. O dicionário possui sintaxe similar ao Gherkin, no entanto os Scenarios e Scenario Outlines foram substituídos por Statements customizados, em que estes são descrito. Já a tabela de Examples foi substituída pela tabela de Statement Params, que contém os valores que os Statements sendo resumidos devem ter ao utilizar o Statement Customizado.

Além disso, valores podem ser injetados ao adicionar parâmetros no Statement customizado e utilizá-los na tabela de Statement Params entre dois @, como podemos ver no exemplo da Figura B.3.

```
Dictionary: Dog Facts!

  Statement: the API returned the data field in the response "<field>"

    the field(s) "<api_returned_field>" is(are) "<field_situation>" in the "<api_return_origin>" of the API response

    Statement Params:       You, 3 hours ago • Implementing translation route, script and addi…
      | api_returned_field | field_situation | api_return_origin |
      | data               | presente        | @field@           |
```

Figura B.3: *Exemplo de dicionário*

Logo, temos que a Dialética Diegética consegue através das máscaras e dicionários, preservar a linguagem natural e a comunicação do comportamento de um software, apesar da alta parametrização do modelo de Statements genéricos.

Referências

[Beck 2002] Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. isbn: 0321146530 (citado na pg. 7).

[K. Beck *et al.* 2001] Kent Beck *et al. Manifesto for Agile Software Development*. 2001. url: agilemanifesto.org (acesso em 15/09/2024) (citado na pg. 7).

[North 2006] Dan North. *Introducing BDD*. 2006. url: dannorth.net/introducing-bdd/ (acesso em 15/09/2024) (citado na pg. 8).