Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

BDD-CLI: Um arcabouço para geração automática de artefatos de testes unitários de software

Heitor L. Belém
Icaro N. Rezende

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Genaina Nunes Rodrigues

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

BDD-CLI: Um arcabouço para geração automática de artefatos de testes unitários de software

Heitor L. Belém
Icaro N. Rezende

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Genaina Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Breno Miranda
CIn/UFPE

Prof. Rafael Fazzolino Pinto Barbosa
FGA/UnB

do Bacharelado em Ciência da Computação

Brasília, 12 de novembro de 2021

Dedicatória

Dedicamos este trabalho, primeiramente, a nossos pais, que sempre nos deram todas as condições para que tudo isso fosse possível. Dedicamos também aos nossos familiares, amigos e professores que participaram desse longa jornada de aprendizado e amadurecimento.

Agradecimentos

Agradecemos às nossas famílias, que forneceram apoio em todos os momentos dessa jornada. Também a todos os amigos e colegas que passaram por nossas vidas ao longo de todos esses anos. Agradecemos à Prof.a Dr.a Genaina Nunes Rodrigues por todos os ensinamentos dados durante esse período universitário, em especial, durante a orientação para este trabalho e à Prof.a Dr.a Vanessa Tavares Nunes por todos os conselhos e revisões para a escrita deste documento. Por fim, agradecemos a todos os profissionais da área da saúde, que nos últimos anos lutaram por nossas vidas dia após dia.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Com a evolução do mercado de *software*, exigências relacionadas a entregas de produtos de qualidade e com rapidez só aumentaram e, portanto, surgiram necessidades de criação de ferramentas facilitadoras do processo de desenvolvimento e produção de um *software*. Dito isso, este projeto se baseia em uma ferramenta desenvolvida por estudantes do grupo de Engenharia de *Software* da Universidade de Brasília, o *TestGenerator*. Tal ferramenta utiliza conceitos como a Programação Orientada a Aspectos para mapear as funções e classes do sistema que são chamadas durante a execução dos cenários descritos conforme a metodologia de Desenvolvimento Orientado a Comportamento e, com isso, gerar testes unitários de forma automática através da metaprogramação. A partir daí, surge a proposta do trabalho atual: modularizar a ferramenta já existente para ser capaz de agregar a geração de diversos artefatos de *software* no futuro além de adaptar a geração automática de testes unitários já implementada para a nova arquitetura. Com base nessa ferramenta, na proposta da nova arquitetura e nos estudos realizados para validar o funcionamento da proposta, foi possível verificar que a aplicação *CLI* facilitou a utilização da ferramenta em projetos com estrutura mais complexa e, além disso, a funcionalidade de geração automática dos testes unitários foi adaptada de forma a gerar os mesmos resultados fornecidos pela implementação original. Os resultados obtidos evidenciaram que a abordagem tornou mais simples e eficaz tanto a fase da análise da execução dos testes de comportamento quanto a fase de geração automática de artefatos, no caso, os testes unitários.

Palavras-chave: Artefatos de software, Desenvolvimento Orientado a Comportamento, Ruby on Rails, Metaprogramação, TestGenerator, CLI

Abstract

With the evolution of the software development industry, demands related to the quickly delivery of good products only increased and, therefore, there were needs to create tools that facilitate the development and production process of a software. In this way, this project is based on a tool developed by students from the Software Engineering group at the University of Brasília, TestGenerator. This tool uses concepts such as Aspect-Oriented Programming to map the system functions and classes that are called during the execution of the scenarios described according to the Behavior-Oriented Development methodology and, therefore, generate unit tests automatically using metaprogramming. From there, the proposal of the current work arises: to modularize the existing tool to be able to aggregate the generation of several software artifacts in the future, in addition to adapting the automatic generation of unit tests already implemented to the new architecture. Based on this tool, on the proposal of the new architecture and on the studies carried out to validate the operation of the proposal, it was possible to verify that the CLI application facilitated the use of the tool in projects with more complex structure and, in addition, the automatic generation functionality of the unit tests was adapted in order to generate the same results provided by the original implementation. The results obtained showed that the approach made the analysis of the execution of behavior tests and the automatic generation of artifacts (unit tests in this case) simpler and more effective.

Keywords: Software Artifacts, Behavior Driven Development, Ruby on Rails, Metaprogramming, TestGenerator, CLI

Sumário

1 Introdução **1**
1.1 Problema da pesquisa 1
1.2 Objetivo geral 2
1.3 Objetivos específicos 3
1.4 Organização do trabalho 4

2 Fundamentação Teórica **5**
2.1 Desenvolvimento de software 5
2.2 *Behavior Driven Development* (*BDD*) 6
2.3 *Ruby* e os paradigmas de programação 8
2.3.1 *Ruby* 8
2.3.2 Programação Orientada a Aspectos 8
2.3.3 Metaprogramação 9
2.4 Testes automatizados e cobertura de código 9
2.5 *TestGenerator* 10

3 Trabalhos Relacionados **12**
3.1 Geração automática de artefatos de *software* 12
3.1.1 *Pickles* 12
3.1.2 *Visual Narrator* 13
3.2 Utilização de *CLI* para agilizar processos 14
3.2.1 ASDF 14

4 Arcabouço de geração de artefatos **15**
4.1 Visão Geral 15
4.1.1 Correção da execução 15
4.1.2 Mudança de organização 16
4.1.3 Desenvolvimento da *CLI* 18
4.1.4 Evoluções 18

4.2 Desenvolvimento 19
4.2.1 Correções 19
4.2.2 Reorganização dos módulos 20
4.2.3 *CLI* 21
4.2.4 Evoluções 22
4.2.5 Análise final da implementação 27

5 Validação da Ferramenta **29**
5.1 Planejamento 29
5.2 Projetos selecionados 30
5.2.1 *Zlp-scheduler* 31
5.2.2 Doei 32
5.2.3 *Facebook Clone* 33
5.2.4 *Diaspora* 33
5.2.5 Chiquinho 35
5.3 Resultados gerais 35

6 Conclusão **37**

Referências **40**

Lista de Figuras

4.1 Relação entre os módulos do *TestGenerator* [20]. 16
4.2 Proposta de nova organização de módulos. 17

5.1 Erro na execução do *Diaspora* com o *TestGenerator* 34
5.2 Erro na execução do *Diaspora* com o *TestGenerator* 34
5.3 Comparação dos tempos de execução 36

Lista de Tabelas

4.1 Análise final da implementação 28

5.1 Projetos escolhidos para realização dos testes 31
5.2 Comparativo entre execução das ferramentas para o Zlp-Scheduler 32
5.3 Comparativo entre execução das ferramentas para o Doei 32
5.4 Comparativo entre execução das ferramentas para o *Facebook Clone* 33
5.5 Comparativo entre execução das ferramentas para o *Diaspora* 34
5.6 Comparativo entre execução das ferramentas para o Chiquinho 35

Capítulo 1

Introdução

Este trabalho se concentra no estudo dos desafios relacionados à geração de artefatos dentro de um processo de desenvolvimento de *software* utilizando metodologias ágeis. A utilização do *BDD* dentro deste contexto se mostrou uma solução interessante a partir do momento em que os cenários que descrevem os testes de comportamento podem ter a execução rastreada para servir como uma espécie de "fonte única da verdade", do inglês *Single Source of Truth*, e então fornecer informações relevantes e consistentes para a geração automática dos artefatos.

Nas próximas seções serão descritos, detalhadamente, os problemas que levaram ao início desta pesquisa bem como os objetivos definidos para delinear a proposta da solução exposta ao decorrer do trabalho.

1.1 Problema da pesquisa

O levantamento de requisitos é um dos principais problemas relativos à criação de um *software*. Além disso, outros erros que ocorrem durante o processo de desenvolvimento têm grande chance de serem derivados de falhas no processo de elicitação de requisitos [16]. Com o passar dos anos e a evolução do mercado de *software*, a busca por abordagens que facilitassem alterações constantes nos requisitos e também proporcionassem maior agilidade no desenvolvimento se tornou primordial. Então, em 2001, foi publicado o manifesto ágil, que reuniu a essência de tais abordagens: focar na entrega contínua de valor para o cliente final, deixando de lado a burocracia envolvida no processo [5].

Planejamento durante as etapas de desenvolvimento, em um curto prazo de tempo, reuniões de equipe todos os dias, histórias de usuário bem definidas, contato contínuo com o cliente, automatização de testes e integração contínua são características importantes das metodologias ágeis aplicadas ao desenvolvimento de *software* [9].

O surgimento das metodologias ágeis se deu como uma forma de abordar os principais desafios no desenvolvimento de *software*, como, por exemplo, a diferença de entendimento do escopo do sistema e a falta de sincronia entre código, requisitos e documentação. Entretanto, existe uma grande barreira para introduzir o desenvolvimento ágil na gestão de projetos e é justamente a dificuldade de comunicação entre as diversas partes envolvidas no processo [10].

Uma das abordagens para mitigar o problema da comunicação entre os *stakeholders* e as equipes de desenvolvimento é a aplicação do Desenvolvimento Orientado a Comportamento, do inglês *Behavior Driven Development* (*BDD*). Para tal problema o *BDD* traz, de forma estruturada, uma maneira de representar os comportamentos e validar as histórias de usuário por meio de uma linguagem ubíqua, compartilhada entre todos os envolvidos no ecossistema do projeto [22].

Entretanto, durante o processo de produção do sistema, outros artefatos de *software* podem, facilmente, perder a sincronização com as histórias descritas através dos cenários dos testes de comportamento e acarretar muitas inconsistências em relação aos requisitos [24]. Dito isso, é interessante aproveitar a descrição fornecida pelos cenários *BDD* para a geração de novos artefatos de *software* de forma automática, diminuindo a incongruência entre todas as peças fundamentais do produto.

1.2 Objetivo geral

Para agilizar o processo de produção e maximizar a possibilidade de geração de novos artefatos de *software* de maneira automática a partir de outro artefato, este trabalho propôs a evolução da arquitetura do *TestGenerator*, uma ferramenta desenvolvida por alunos do laboratório de Engenharia de *Software* da Universidade de Brasília.

Com o intuito de incentivar o desenvolvimento de novas pesquisas baseadas nessa ferramenta, foi proposta uma arquitetura que tornasse possível a incorporação de novas funcionalidades. Muitas vezes são utilizadas as interfaces de linha de comando, do inglês *Command Line Interfaces CLI's*, para acelerar processos de desenvolvimento. Estas aplicações têm a responsabilidade de identificar comandos de entrada, analisar parâmetros, opcionais ou não, e executar as ações definidas para cada tipo de entrada [25].

Especialmente para este projeto, a ideia de construir uma *CLI* baseada no *TestGenerator* foi justamente para facilitar o processo de geração de artefatos através da divisão de responsabilidades que são disparadas através de comandos dados à ferramenta.

Como visto em Pereira et al. [21], a implantação do *BDD* apresenta alguns desafios. Cenários mal escritos por falta de experiência, curva de aprendizado para a utilização de ferramentas durante o desenvolvimento e a dificuldade de convencer os desenvolvedores e

clientes que o valor agregado compensa todo o esforço feito para a implantação do desenvolvimento orientado a comportamentos são exemplos dos grandes desafios de empregar tal metodologia.

Tendo em vista o que é apresentado em Marques and Fernandes [20], a ferramenta *TestGenerator* mostra uma possibilidade de diminuir a diferença entre o esforço empregado para utilização do *BDD* e o valor que é agregado pela sua implantação, através da geração de testes automatizados a partir da análise da execução dos cenários *BDD*.

Entretanto, o *TestGenerator* ainda apresenta alguns problemas relacionados à sua utilização em projetos de grande porte e, além disso, se encontra muito atrelado à geração de testes unitários. A partir daí, foi identificada a possibilidade de ampliar a ferramenta para abordar a geração de outros artefatos de *software* que agregassem valor ao uso da ferramenta.

O principal objetivo deste trabalho é apresentar uma ferramenta que analisa a execução de cenários *BDD* para mapear o comportamento do sistema e, com isso, gerar artefatos de *software* relevantes para o processo de produção do produto.

Essa proposta é baseada nos conceitos apresentados no Capítulo 2, os testes comportamentais descritos em forma de cenários BDD utilizando a ferramenta *Cucumber* [3], a programação orientada a aspectos, que realiza o papel de rastrear as execuções dos cenários BDD, e a metaprogramação, que possibilita a obtenção de informações fundamentais sobre métodos ou classes que foram utilizados durante cada execução.

1.3 Objetivos específicos

Para contemplar o objetivo principal descrito na seção anterior, este trabalho é composto por objetivos específicos que pontuam as contribuições realizadas na ferramenta *TestGenerator*, utilizada como base para o desenvolvimento deste projeto, sendo eles:

1. Correção do *TestGenerator* para possibilitar a sua utilização em aplicações com arquiteturas mais complexas.

2. Introduzir uma nova arquitetura para a ferramenta *TestGenerator* para funcionamento com Interface de Linha de Comando (*CLI*) para utilização em múltiplos projetos.

3. Modularização da ferramenta e padronização do arquivo de saída do módulo de análise dos testes de comportamento para facilitar a incorporação de novas funcionalidades no futuro.

4. Adaptar a geração de testes já existentes no *TestGenerator* para se adequar à nova arquitetura, transformando em um módulo a parte que utiliza o arquivo resultado da análise dos cenários *BDD* para a geração dos testes unitários.

1.4 Organização do trabalho

Este trabalho é composto por mais cinco capítulos e sua organização é descrita a seguir.

No Capítulo 2 é apresentada toda a fundamentação teórica utilizada para embasar a proposta do trabalho. Os conceitos e as ferramentas expostas neste capítulo foram essenciais para a evolução da ferramenta alvo deste trabalho.

No Capítulo 3 é realizada uma revisão de literatura e apresenta alguns projetos que serviram de referência para a idealização da ferramenta da proposta.

No Capítulo 4 é apresentada a forma como a ferramenta foi evoluída considerando os quatro objetivos específicos. Além disso, neste capítulo também são comparadas as diferenças entre a antiga e a nova arquitetura do *TestGenerator*.

No Capítulo 5 são mostrados os dados obtidos através dos estudos de caso para validar a proposta deste trabalho. Nesse capítulo são apresentadas as métricas utilizadas para realizar a comparação entre a execução do *TestGenerator* para as duas arquiteturas, antiga e nova, além de comparar estas com a execução pura do *framework Cucumber*.

Já no Capítulo 6 é apresentada a conclusão obtida a partir do desenvolvimento deste trabalho além da indicação de possíveis trabalhos futuros, que podem dar continuidade e mais visibilidade ao que vem sendo desenvolvido.

Capítulo 2

Fundamentação Teórica

Neste capítulo serão abordados os temas e tópicos necessários para realizar o embasamento teórico da pesquisa, bem como as ferramentas e materiais utilizados para a implementação do projeto.

2.1 Desenvolvimento de software

O desenvolvimento de um artefato de *software* é baseado em processos. Processo de desenvolvimento de *software* é um conjunto de tarefas e ações que devem ser tomadas para a criação do *software* como, por exemplo, criação de testes para o sistema, planejamentos, interações com o cliente e escrita de documentação e código. Cada uma dessas tarefas compõe uma metodologia em que se relacionam umas com as outras e com o processo por completo. Para a engenharia de *software*, o processo é uma abordagem adaptável que possibilita a escolha e definição de tarefas de maneira eficaz, sempre visando entregar um *software* de qualidade dentro do prazo estabelecido [23].

O desenvolvimento ágil, no contexto da engenharia de *software*, é um grupo de métodos interativos e incrementais de desenvolvimento de *software* empregado no processo de produção de um artefato. Ser ágil envolve um conjunto de valores e princípios que proporcionam a entrega de um bom *software* ao consumidor final [15].

Essa metodologia surgiu para suprir as necessidades dos modelos de processos existentes à época: flexibilização das etapas de desenvolvimento e interação com as pessoas interessadas no produto (*stakeholders*) [11]. Um dos modelos anteriores ao ágil, conhecido como Cascata, é baseado em cronogramas muito rígidos, com documentações extensas e pouca interação com o cliente, de tal forma que este só receberia um vislumbre do produto ao final de todo o projeto [23].

Um dos principais benefícios da adoção da metodologia ágil no processo de desenvolvimento de *software* é a forma de lidar com mudanças de requisitos [15]. O modelo de

Cascata exige que o cliente estabeleça todas as necessidades atuais relativas ao *software* e qualquer mudança à medida que o projeto evolui pode provocar confusão no desenvolvimento. Já com o ágil, a fase de planejamento foi aprimorada porque existe um *Product Owner* (*PO*), representante de todas as partes interessadas no desenvolvimento do produto, diretamente envolvido com o projeto, assim há maior controle dos processos por meio de interações que refletem diretamente as necessidades atuais dos usuários [27]. Dessa forma, ao final de cada etapa, o cliente possui uma ideia mais concreta do produto podendo, inclusive, ter partes do *software* já em produção [23].

Outro ponto importante, favorável à adoção da metodologia ágil, é que a alta qualidade da entrega é garantida porque, ao contrário do modelo anterior, em que os testes eram feitos após a conclusão do desenvolvimento, na metodologia ágil, testes e desenvolvimento andam juntos. Com a utilização de técnicas como o Desenvolvimento Orientado a Comportamento, explicado na próxima seção, ao decorrer de cada iteração, as falhas e funcionalidades incorretas são detectadas de forma mais imediata [15].

2.2 *Behavior Driven Development* (*BDD*)

BDD é um conjunto de práticas da engenharia de *software* desenvolvido para ajudar os times a construir e entregar produtos de maior qualidade e com mais rapidez [26]. Para Irshad et al. [13], *BDD* é uma metodologia de desenvolvimento orientado a testes que proporciona o alinhamento das necessidades técnicas e de negócio de um *software*. Em geral, essa técnica incentiva a compreensão compartilhada do problema, o que diminui a diferença de entendimento do produto entre as equipes de desenvolvimento e os *stakeholders* [3]. Para alcançar os objetivos citados anteriormente, é comum utilizar os testes de comportamento nos ciclos de desenvolvimento.

Pragmaticamente, os testes de comportamento testam o comportamento de um sistema de forma automatizada. Estes testes abordam, através de uma linguagem natural, a especificação de uma funcionalidade do sistema e os cenários ligados à ela [28].

Para padronizar a escrita dos cenários dos testes de comportamento, é comum utilizar uma linguagem de domínio específico (*Domain Specific Language* - *DSL*) que é composta por um conjunto de palavras chave que dão estrutura e significado às especificações.

Para as equipes de desenvolvimento de *software* é comum a utilização de ferramentas que analisam estes cenários, através de expressões regulares, para então gerar os testes na linguagem de programação de forma automatizada. Um exemplo disso é mostrado na Listagem 2.1, que utiliza o *framework Cucumber* para elaborar os testes a partir da análise dos cenários descritos com uma linguagem de domínio específico *Gherkin* [3], baseada em três palavras chave: “dado”, “quando” e “então” que, quando combinadas com o texto

escrito em português, expressam claramente quais são os comportamentos esperados para o cenário descrito.

```
1 Funcionalidade: Listar todas as salas do sistema
2   Como administrador do sistema,
3   Quero poder visualizar todas as salas no sistema
4   Para verificar quais salas estao disponiveis para reserva
5   Contexto:
6     Dado que eu esteja cadastrado como "admin@admin.com", "123123", "
    Engenharia", "11/0111111", "admin"
7     E que esteja logado
8     E que esteja na pagina inicial
9     E eu clicar no link "Salas Existentes"
10  Cenario: Lista de salas aparece adequadamente
11    E exista a sala "Pat-45", "50", "PAT-AT" cadastrada no sistema
12    Quando eu clicar no botao "Salas Existentes"
13    Entao eu devo estar em uma pagina com uma tabela com os dados:
14    | name     | Pat-45 |
15    | location | PAT-AT |
16    | capacity | 50     |
```

Listing 2.1: Exemplo de descrição de cenário utilizando Gherkin

Já na Listagem 2.2, é possível ver o cenário mostrado na Listagem 2.1 escrito na linguagem de programação *Ruby*. O maior objetivo para manter a semelhança entre a linguagem natural e o código escrito em *Ruby* é facilitar o entendimento do cenário descrito, de forma a alinhar as expectativas de todos os interessados no produto. À medida que o *Product Owner* (*PO*) está interessado na descrição do escopo do projeto, a equipe de desenvolvimento busca descrever os cenários e pensar nos detalhes de cada requisito [3]. Dessa forma, utilizando uma linguagem como *Gherkin*, é possível, sem grandes esforços, mitigar o problema de comunicação entre a parte técnica e as regras de negócios do *software*.

```
1 Dado("que eu esteja cadastrado como {string}, {string}, {string}, {
    string}, {string}") do |email, password, course, registration,
    username|
2   @user = User.create(username: username, password: password,
3   email: email, registration: registration, course: course, is_admin:
    true)
4 end
5 Dado("que esteja logado") do
6   visit new_user_session_path
7   fill_in :user_email, with: @user.email
8   fill_in :user_password, with: @user.password
9   click_button "Log in"
```

```
10 end
11 Dado("que esteja na pagina {string}") do |string|
12   visit(backoffice_path)
13 end
14 Dado("eu clicar no link {string}") do |string|
15   click_link "#{string}"
16 end
17 Quando("eu preencher o campo nome, capacidade maxima e localidade com:")
     do |table|
18   table.rows_hash.each do |field, value|
19     fill_in field, :with => value
20   end
21 end
22 Entao("eu devo ver uma mensagem escrita {string}") do |string|
23   page.has_content?("#{string}")
24 end
```

Listing 2.2: Cenário descrito utilizando Ruby

2.3 *Ruby* e os paradigmas de programação

Para este trabalho, a escolha de *Ruby* como a principal linguagem para o desenvolvimento da proposta descrita no Capítulo 4, se deu por seu suporte a diferentes paradigmas de programação como, por exemplo, a programação orientada a aspectos e a metaprogramação. A utilização destes paradigmas está muito presente no desenvolvimento do módulo de aspectos, descrito na seção 4.2.4, que observa a chamada dos métodos através da execução dos casos de testes de comportamento.

2.3.1 *Ruby*

Ruby [8] é uma linguagem de programação interpretada, *open source* e multiplataforma, portanto está disponível para qualquer ambiente, independente do sistema operacional. Ela é totalmente orientada a objetos, com tipagem dinâmica e forte, ou seja, é capaz de definir os tipos de uma variável ou função de acordo com os valores que recebe em tempo de execução e também conta com gerenciamento de memória automático, o que otimiza a performance de suas aplicações.

2.3.2 Programação Orientada a Aspectos

Para desenvolver um sistema que busca solucionar um problema, é comum dividir este problema em partes menores. Ao projetar o sistema, são trabalhados diversos interesses

que, na programação orientada a objetos, são divididos em unidades chamadas de classes, as quais podem conter mais de um interesse. Em projetos muito grandes, é comum surgir fenômenos de entrelaçamento de interesses, quando mais de um está combinado em uma única classe, e interesses transversais, que não podem ser modularizados em classes como, por exemplo, gerenciamento de transações, *logs* de sistema e acesso concorrente [14].

A Programação Orientada a Aspectos (*Aspect Oriented Programming* - *AOP*) surge para tentar resolver os problemas relacionados à implementação dos interesses transversais e à separação de interesses. Ela é um paradigma de programação que permite separar o código de acordo com a sua importância para a aplicação, possibilitando que o programador encapsule o código secundário em módulos separados do restante da aplicação, os aspectos [14].

Aspectos são trechos de código que tiram responsabilidades indevidas de outros módulos do programa e que podem alterar o comportamento de um código através da aplicação de uma funcionalidade adicional sobre um ponto de execução. Por exemplo, para a geração de *logs* de uma aplicação, é necessário inserir mensagens informativas ao longo de todo o programa. Entretanto, a geração de métricas do sistema não faz parte da regra de negócio de um *software* ou dos modelos e classes que o compõem, portanto não é responsabilidade deles se preocupar com isso. Para tal, a *AOP* recomenda que esse interesse transversal seja encapsulado em outro módulo separado do código principal [1].

2.3.3 Metaprogramação

Metaprogramação é a escrita de programas de computador, metaprogramas, que escrevem ou manipulam outros programas (ou a si próprios) como se fossem dados [4]. Essa técnica permite que esses metaprogramas gerem novos códigos ou modifiquem os já existentes. A linguagem em que o metaprograma é escrito é chamada de metalinguagem, e a gerada pela metaprogramação é chamada de linguagem objeto [17].

Caso a metalinguagem e a linguagem objeto sejam as mesmas, então é dito que a metaprogramação é homogênea mas, se forem diferentes, a metaprogramação é heterogênea [17]. Para este trabalho, foram utilizados os dois tipos de metalinguagem, ao passo que, para a geração dos testes funcionais, escritos em *Ruby*, a partir da análise de testes de comportamento, também em *Ruby*, existe a geração de um arquivo *JSON* para realizar essa transição.

2.4 Testes automatizados e cobertura de código

Para identificar o correto funcionamento de um *software* é comum que sejam realizados testes. Quando são realizados testes manuais, o testador assume o papel do Oráculo,

aquele que sabe as condições corretas do funcionamento do sistema. À medida em que alterações são feitas no *software*, o processo de testes manuais fica cada vez mais repetitivo e isso leva à utilização de testes automatizados. Nesse modelo, o papel de Oráculo é atribuído ao próprio teste, que já possui as condições esperadas de funcionamento do *software* [12].

Testes funcionais são técnicas de teste em que a unidade de *software* a ser testada é tratada como uma caixa preta, dados de entrada são fornecidos, o teste é executado sob esses dados, fornecendo um resultado que é comparado a um resultado esperado que já é previamente conhecido. O componente de código a ser testado pode ser um componente, um conjunto de programas, um método e/ou funções [12].

Testes unitários concentram-se em testar a menor unidade do projeto de *software*, assim são escritos testes para cada módulo do projeto, com o intuito de encontrar erros de lógica e implementação. Os testes unitários complementam os testes funcionais por permitir identificar em que parte do código os erros estão localizados, auxiliando no processo de correção do código [12].

Ao se fazer testes para um *software*, um dos principais objetivos almejados é verificar se cada trecho de código tem um teste associado, para garantir que, ao longo do tempo, os comportamentos individuais continuam funcionando da maneira correta [29]. Para isso, é utilizado o conceito de cobertura de código, que visa verificar a quantidade de código desenvolvido que possui um teste associado [19].

Para este trabalho, esses conceitos relacionados à técnica de testes funcionais e cobertura de código serão utilizados para verificar a corretude da proposta de nova arquitetura para a ferramenta *TestGenerator*, de forma que o módulo de gerador de testes unitários, a partir da mesma suite de testes que a versão anterior, deve proporcionar o mesmo grau de cobertura de código previamente obtido.

2.5 *TestGenerator*

O *TestGenerator* [20] é o alvo das contribuições deste trabalho e consiste em uma ferramenta que monitora testes de comportamento com o objetivo de gerar testes unitários de forma automática.

Os testes escritos em forma de cenários (*features*), que utilizam a linguagem de domínio específico *Gherkin*, são executados e monitorados. Esse monitoramento consiste na verificação e avaliação de quais métodos, classes e funções são utilizadas pelo cenário em questão. Após a análise inicial, a ferramenta gera testes unitários de forma automática, considerando a execução de cada teste de comportamento.

Para realizar tais tarefas, o *TestGenerator* é dividido em duas partes. A primeira utiliza os testes de comportamento, a programação orientada a aspectos e a metaprogramação para mapear as entidades do sistema que são acessadas durante a execução do cenário. Com isso, são gerados arquivos de *logs* que contêm informações relativas a cada método, função ou classe utilizada como, por exemplo, atributos das funções ou propriedades das classes.

Já a segunda etapa consiste na geração de códigos de teste unitário a partir das informações obtidas na fase anterior. Para a concepção dos arquivos de testes, é realizada a leitura e análise dos *logs* gerados na etapa de verificação dos testes de comportamento em conjunto com uma gramática responsável por dar significado e estrutura a cada teste criado.

Os autores ressaltam que o sistema também pode ser monitorado através de interação do usuário, porém a inicialização do *TestGenerator* em qualquer ambiente que não o de teste, atrapalha na utilização dos comandos **db** do *Rails*. Caso o *TestGenerator* seja instalado antes da execução do comando `rails db:create`, a instrução presente na linha 7 da Listagem 2.3 irá causar um erro, pois o *Application Record*[1] é uma camada que faz contato direto com o banco de dados.

```
1  # config/application.rb
2  # {...}
3
4  config.after_initialize do
5    require 'test_generator'
6    Rails.application.eager_load!
7    ApplicationRecord.descendants.each do |model|
8      model.send(:include, TestGenerator::Observer)
9      model.observe
10   end
11
12   puts 'Observing models.'
13 end
```

Listing 2.3: Código necessário para a inicialização da gema

Mais detalhes do *TestGenerator*, bem como sua arquitetura e funcionamento, serão abordados em capítulos posteriores deste trabalho.

[1] https://guides.rubyonrails.org/active_record_basics.html

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados alguns trabalhos relacionados que utilizam parte de um *software* para gerar artefatos de forma automática e também buscam agilizar o processo de desenvolvimento por meio de ferramentas facilitadoras.

Este trabalho foi baseado em implementar ou possibilitar a implementação de funcionalidades abordadas nas ferramentas descritas nas próximas seções para o contexto da análise da execução dos testes de comportamento.

3.1 Geração automática de artefatos de *software*

Nesta seção são apresentados exemplos de *frameworks* que geram artefatos de *software* de forma automática. Esses projetos contribuíram de forma significativa com o trabalho atual pois inspiraram a ideia de expandir o *TestGenerator* para além da geração de testes funcionais.

Ao verificar que é possível gerar documentação e modelos conceituais a partir da análise da linguagem dos testes de comportamento, houve o interesse em agrupar em uma única ferramenta funcionalidades que facilitariam todo o processo de desenvolvimento e produção do *software* como, por exemplo, documentação do código, diagramas de classes e relacionamentos e testes funcionais.

3.1.1 *Pickles*

Pickles [6] é uma ferramenta *open source* utilizada para gerar, de forma automática, a documentação atualizada do *software*. A partir de cenários (*features*) *BDD* escritos na linguagem de domínio específico *Gherkin*, é gerada a documentação do estado atual do *software* em diversos formatos.

Este *framework* foi desenvolvido para utilização em conjunto com o *Spec Flow* [7], ferramenta para a plataforma .NET que possibilta a escrita de testes de comportamento em forma de cenários usando como base a linguagem *Gherkin*. O *Pickles* é capaz de gerar a documentação em quatro diferentes formatos:

- *HTML* estático
- Documento de texto
- Planilha *Excel*
- *JSON*

O objetivo dessa ferramenta é traduzir os arquivos *.feature* de testes comportamentais para uma linguagem mais amigável aos negócios. Seguindo os formatos citados acima, os *stakeholders* passam a ter acesso facilitado às especificações, através de páginas web, planilhas do *Excel* e documentos escritos no *Word* sem necessariamente entrar em contato com a equipe de desenvolvimento. Essa facilidade fornecida pelo *framework*, que disponibiliza formatos comumente utilizados por outras aplicações, serviu de modelo para a alteração do arquivo gerado pelo módulo *Logger*, descrito na seção 4.2.4, afim de descomplicar a comunicação entre *plugins* futuros com os módulos produzidos neste trabalho.

3.1.2 *Visual Narrator*

Visual Narrator [18] é uma ferramenta que tem como objetivo gerar, de forma automática, modelos conceituais a partir de histórias de usuário. Os cenários de cada história devem ser descritos utilizando palavras chave "como", "como um", "eu quero"e "eu gostaria", semelhante à linguagem *Gherkin* citada em exemplos anteriores.

Esta ferramenta é dividida em duas partes principais:

- Processador
  Responsável pela análise do conjunto de histórias de usuário para extrair, a partir de cada história, os *tokens*, suas classes e relacionamentos com outros *tokens*. Esses *tokens* são armazenados em uma estrutura para que sejam determinadas, na fase de construção, as entidades relativas a cada *token* e seus relacionamentos com outras entiadades.
- Construtor
  O construtor identifica todos os padrões nas histórias de usuário e analisa, em relação ao que foi armazenado na estrutura gerada pelo processador, quais são os termos que têm recorrência na estrutura e quais relacionamentos existem entre eles, para então gerar o modelo conceitual.

O objetivo principal da ferramenta é gerar modelos que forneçam uma identidade visual para as entidades e seus relacionamentos, de forma a facilitar a descoberta de inconsistências durante o desenvolvimento do produto.

Para este trabalho, a principal contribuição do *Visual Narrator* foi a proposta da arquitetura dividida em módulos com responsabilidades específicas, que incentivou a realização de tal alteração na arquitetura do *TestGenerator*. Mapeamos o módulo Processador como o Analisador, responsáveis por extrair informações do código, e o Construtor como o Gerador, que geram artefatos com base nos resultados obtidos no módulo anterior. Tais módulos são aprofundados na seção 4.2.4.

3.2 Utilização de *CLI* para agilizar processos

Esta seção apresenta um software de linha de comando que facilita o desenvolvimento de artefatos através do gerenciamento de versão de outras ferramentas. Para este trabalho, a contribuição de uma aplicação *CLI* é muito significativa, visto que é possível padronizar a comunicação dos módulos existentes na implementação atual e também fornecer uma interface para facilitar a incorporação de módulos com outras funcionalidades no futuro.

3.2.1 ASDF

O ASDF [2] é uma ferramenta de controle de versão. Ele é capaz de gerenciar várias versões de *frameworks*, linguagens e bibliotecas como *Ruby*, *Python* e *NodeJS* utilizando apenas uma ferramenta de linha de comando que é extensível por meio de *plugins*.

Para utilizar o ASDF, é necessário configurar, antes de tudo, o módulo principal da ferramenta (*core*), que é responsável por interagir com o usuário, receber comandos de entrada e executar ações baseadas neles. Durante a utilização da ferramenta, o usuário pode desejar adicionar *plugins*, mudar a versão do *plugin* em uso ou até mesmo removê-lo da máquina, e tudo isso é feito através da interação do usuário com a linha de comando da ferramenta.

Por ser uma ferramenta *open source* construída como uma interface de linha de comando, a quantidade de *frameworks* versionados pelo ASDF pode ser facilmente expandida. Durante o desenvolvimento de novos *plugins*, uma das tarefas do módulo principal da ferramenta é definir interfaces que devem ser implementadas por essas extensões, ou seja, há a padronização dos *plugins* para estabelecer um alinhamento de comunicação entre as extensões e o *core* da aplicação. Essa característica do ASDF foi essencial para a idealização da *CLI* desenvolvida neste trabalho, aprofundada na seção 4.2.3.

Capítulo 4

Arcabouço de geração de artefatos

Neste capítulo são apresentadas as diferenças entre a antiga e a nova arquitetura do *TestGenerator* a fim de relatar os principais desafios encontrados durante o processo de reorganização da arquitetura, bem como as contribuições realizadas pelo desenvolvimento do trabalho atual.

4.1 Visão Geral

Para tornar o *TestGenerator* mais expansível, adicionamos mais módulos, de modo que a sua execução fosse facilitada. Antes de adicionar novos módulos, foi necessário realizar correções para que o *TestGenerator* fosse executado corretamente em projetos mais complexos.

4.1.1 Correção da execução

Para iniciar o nosso desenvolvimento, clonamos o projeto *TestGenerator* e realizamos testes nos projetos Doei e *Zlp-Scheduler*, que também foram utilizados nos testes realizados pelos autores. Adicionamos o projeto *Diaspora* como um novo caso de teste, por ser um projeto mais maduro e com uma boa suite de testes de comportamento.

Para executar a gema *TestGenerator* nos projetos citados, algumas modificações foram necessárias. Essas modificações corrigem a instalação da gema, adicionando comandos para a inicialização dos módulos, filtrando métodos que não são processados corretamente e corrigindo o funcionamento em projetos mais complexos, que possuem seus recursos aninhados em *namespaces*, como é o caso do *Diaspora*.

4.1.2 Mudança de organização

A organização dos módulos do *TestGenerator* pode ser vista na Figura 4.1. Os testes de comportamento servem de entrada para a gema e devem estar configurados para execução no *framework Cucumber*. O módulo de aspectos é responsável por injetar o código que gera os *logs* de execução dos testes. Os *logs* gerados são utilizados como entrada para a geração de testes. O módulo de geradores de código faz a leitura dos *logs* e, utilizando reflexão, fornece a saída esperada para cada método e, com isso, gera casos de teste.

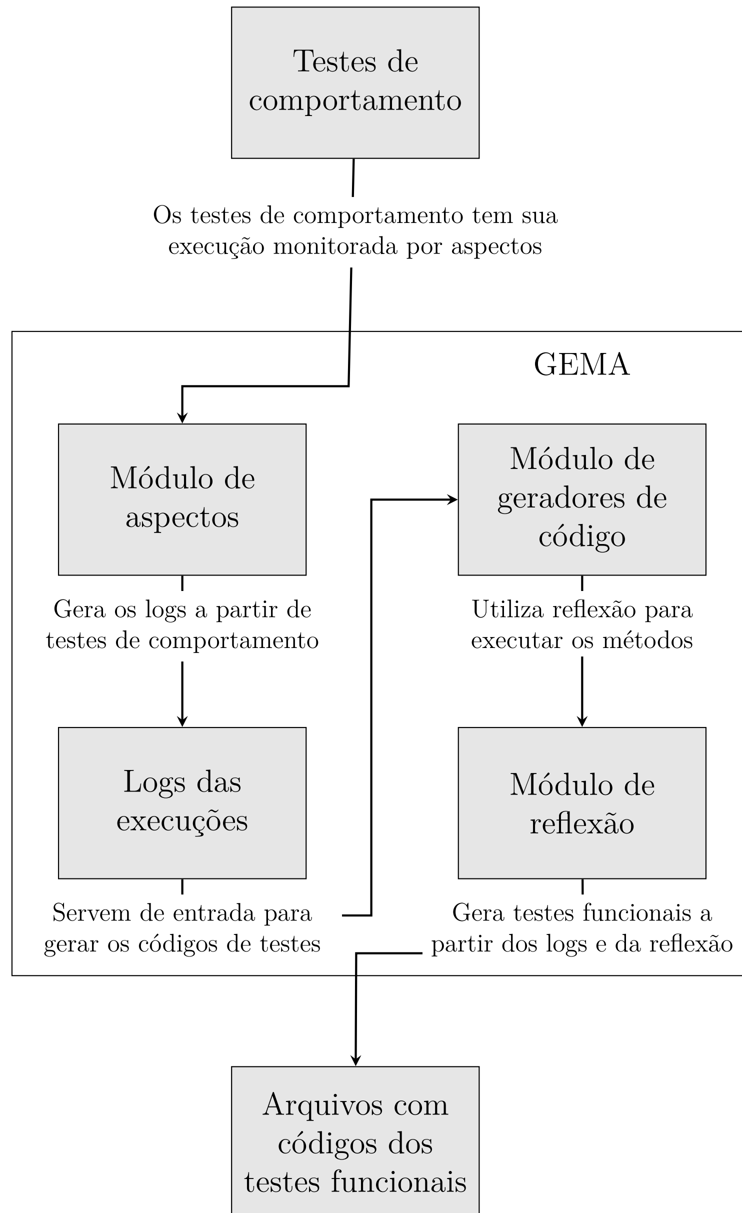

Figura 4.1: Relação entre os módulos do *TestGenerator* [20].

Para aprimorar a configurabilidade do *TestGenerator* e promover o amplo uso da ferramenta, separamos o fluxo de execução da gema em duas etapas. A primeira chamamos

de Analisador e a segunda de Gerador, como pode ser visto na Figura 4.2. O Analisador é responsável por realizar a análise de execução dos testes de comportamento e salvar os *logs* em um arquivo *JSON* com uma estrutura que pode ser reaproveitada para diversos fins, como geração de casos de teste e geração de documentação de código. O Gerador é responsável, nesse caso, por gerar testes funcionais com base nos arquivos *JSON* utilizando reflexão para fornecer os resultados esperados, assim como a implementação original.

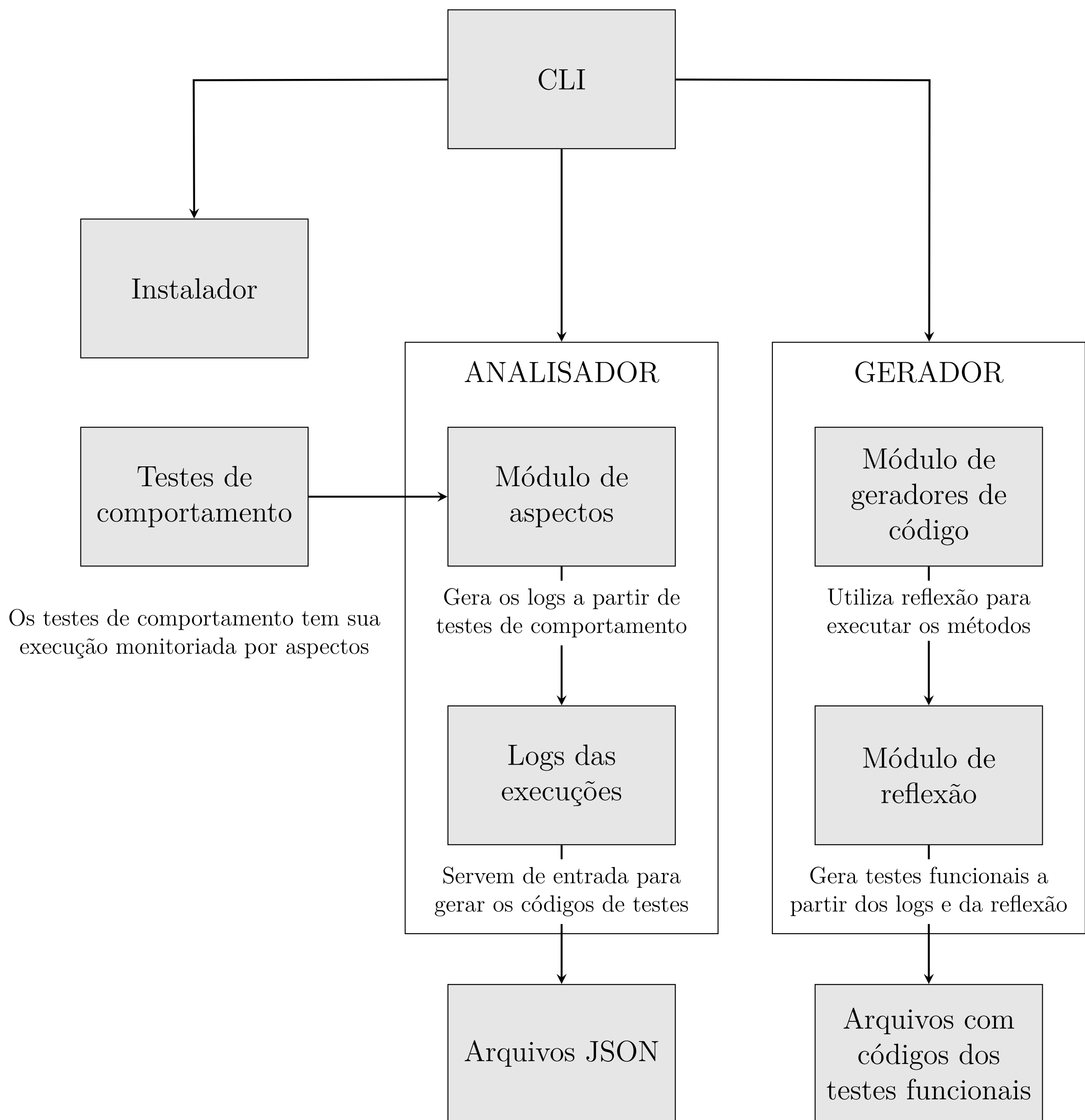

Figura 4.2: Proposta de nova organização de módulos.

Para automatizar o processo de instalação e facilitar o uso dos novos módulos propostos, adicionamos uma *CLI*. Um novo módulo, chamado Instalador, foi adicionado para realizar o processo de instalação em um simples comando, simplificando um processo em

que, atualmente é necessário criar arquivos de configuração e alterar arquivos já existentes. Também foram adicionados comandos para executar os dois módulos, Analisador e Gerador, separadamente.

Essa nova proposta de organização permite que novos módulos sejam inseridos no projeto sem a necessidade de alteração dos módulos já existentes. Novos comandos também podem ser adicionados facilmente à *CLI*, facilitando a integração de novas tarefas em um fluxo de trabalho já existente.

4.1.3 Desenvolvimento da *CLI*

A *CLI* foi desenvolvida utilizando o padrão do *framework Ruby on Rails*. Nela temos os seguintes comandos:

- `install` - Comando responsável pela instalação das dependências do projeto. Cria um inicializador para o módulo de análise e insere os códigos necessários para o módulo de geração de testes.
- `run` - Comando que faz a execução do *Cucumber* e realiza a análise, gerando os *logs* em formato *JSON*.
- `generate` - Recebe o nome do gerador como argumento para gerar os artefatos com base nos arquivos *JSON* salvos.
- `clean` - Comando para remover os arquivos temporários criados pela gema.

4.1.4 Evoluções

O que impacta o tempo de execução do *TestGenerator* é a quantidade de operações de arquivos realizadas. No processo original, sempre que um método é executado, é feita a análise, gerado o *log* e o conteúdo é escrito no arquivo correspondente. Esse processo é ineficiente pelo fato de cada cenário de teste executar vários métodos. Para reduzir a quantidade de interações com arquivos, alteramos o processo para armazenar os dados dos *logs* e apenas realizar as operações de arquivos ao final de cada cenário de teste, como descrito na seção 4.2.4, o que reduz sensivelmente o tempo de execução se comparado ao *TestGenerator* original.

Além disso, as alterações feitas no módulo *Logger* resultaram em arquivos mais simples e fáceis de entender, com tamanhos menores mas que disponibilizavam as mesmas informações. Essas alterações consistem na utilização do *hook* do *Cucumber* que, ao final de cada cenário, fazem a verificação de duplicidade entre as informações armazenadas em *cache* e nos arquivos.

4.2 Desenvolvimento

Para alcançar os objetivos já expostos, foram feitas alterações em diversos módulos do projeto, reorganizando sua estrutura, a fim de deixá-lo mais modularizado. A aplicação utilizada como base para a validação das novas funcionalidades foi o *Diaspora*, por se tratar de um projeto mais complexo.

O processo de desenvolvimento da ferramenta *BDD-CLI*, a partir do *TestGenerator*, se iniciou com as correções no processo de instalação. Em seguida foram feitas evoluções nos módulos já existentes, para que mais projetos pudessem atendidos por essa solução. Para transformar, de fato, o *TestGenerator* em *BDD-CLI*, reorganizamos o projeto para que mais funcionalidades, além do gerador de testes pudessem ser adicionadas, e desenvolvemos uma *CLI* para facilitar a execução da ferramenta.

4.2.1 Correções

O *framework Ruby on Rails* é conhecido pela sua filosofia de convenção acima de configuração, isso também se reflete quando falamos de módulos externos sendo adicionados no formato de gemas. Quando uma gema é instalada no projeto, é comum que ela instale um arquivo com as configurações de inicialização e execução. Esses arquivos de inicialização das gemas ficam na pasta `config/initializers` e são detectados automaticamente quando o servidor é executado.

Quando seguimos as instruções de instalação, ainda é necessário realizar alterações em dois arquivos, o `config/application.rb` para adicionar as configurações de inicialização e o `spec/spec_helper.rb` para adicionar as configurações da gema *Shoulda Matchers*, uma das dependências do *TestGenerator*.

Ao adicionar o código de inicialização apresentado na Listagem 2.3, o pacote do *TestGenerator* é importado no projeto e após o carregamento da aplicação é feita a injeção de um *Observer* em cada classe de Modelo do projeto. Essa configuração permite que o *TestGenerator* observe a execução dos métodos da camada de Modelo da aplicação durante sua execução. Porém, da forma como a inicialização é feita, o registro das execuções é feito sempre, independentemente do ambiente de execução.

No *Ruby on Rails* as aplicações são organizadas em três ambientes: produção, desenvolvimento e teste. O registro de execução em ambiente de produção pode expor dados sensíveis, além de aumentar o tempo de resposta de todas as ações. O ambiente de desenvolvimento também é impactado pelo aumento no tempo de resposta das ações, o que é indesejável durante o desenvolvimento.

Esperamos que a execução seja feita apenas em ambiente de testes, pois o foco da análise são os testes de comportamento. Caso o *TestGenerator* seja executado em outra

suíte de testes, como a de testes unitários, não é esperado que as análises sejam corretas. O código injetado para o funcionamento do *TestGenerator* interfere na execução de outros testes que não sejam os de comportamento. Por isso adicionamos uma variável de ambiente que controla quando o *TestGenerator* deve ser executado. Isso também permite que os desenvolvedores possam escolher entre executar os testes de comportamento com ou sem a análise de execução.

```
1  # config/application.rb
2  # {...}
3  config.after_initialize do
4    if ENV['BDD\_OBSERVE'] then
5     # { mesmo codigo do exemplo anterior }
6    else
7      puts "If you want to generate logs for future analysis, run:
   testgenerator run"
8    end
9  end
```

Listing 4.1: Adição da variável de ambiente 'BDD_OBSERVE'

Após essa adição, quando o usuário quiser executar o analisador do *BDD-CLI* é necessário que seja informado o valor *true* para a variável 'BDD_OBSERVE'. Caso a variável não seja inicializada a aplicação será executada normalmente, sem a execução da gema.

4.2.2 Reorganização dos módulos

Com o intuito de aumentar a modularização do projeto, alguns módulos foram reorganizados para converter a estrutura da gema ilustrada na Figura 4.1 na estrutura apresentada na Figura 4.2.

Atualmente, o aspecto injetado pelo *observer* realiza uma chamada ao método *reflect*, que executa o método observado e, em seguida, faz o *log* dessa execução. Esse fluxo permite que os *logs* tenham os dados de chamada dos métodos e os resultados da execução dos mesmos. Em alguns casos, o método *reflect* pode fazer a execução de um método que logo em seguida faz a execução do método *reflect*, causando uma repetição infinita.

A abordagem utilizada para lidar com esse problema, foi alterar o fluxo, fazendo com que o aspecto faça a chamada do método *log* diretamente. Assim o método *reflect* passa a ser executado apenas ao executar o comando de geração de testes.

Essa mudança faz com que dois fluxos de execução sejam criados:

1. **Fluxo de análise**: Iniciando na execução dos testes de comportamento, monitoramento e geração dos arquivos de *log*;

2. **Fluxo de geração**: Utilizando o arquivo de *log*, obtém os resultados de execução dos métodos através de reflexão e realiza a geração dos artefatos que, para a implementação atual, são os testes unitários.

Assim, os arquivos foram reorganizados de modo que o fluxo de análise corresponde ao módulo analisador e o fluxo de geração corresponde ao módulo gerador, ambos descritos na seção 4.1.2.

4.2.3 *CLI*

Para manter todos os comando necessários para a execução em um só lugar, adicionamos uma *CLI*. Com ela, o fluxo de utilização da ferramenta fica mais intuitivo. Em geral, para utilizar uma ferramenta, o usuário faz a instalação da mesma, em seguida insere os comandos correspondentes ao objetivo esperado. Utilizando a *CLI* também conseguimos criar uma interface comum para executar novos módulos geradores com pouco acréscimo de código.

A interface de gemas[1] da linguagem *Ruby* permite que um arquivo executável esteja presente no pacote da gema. Utilizamos essa utilidade para desenvolver a *CLI*, mantendo os padrões da linguagem *Ruby*. Por ser uma ferramenta projetada para o *Ruby on Rails*, utilizamos a estrutura de geradores[2] disponibilizada pelo *framework*.

A versão inicial da *CLI* conta com quatro comandos:

install

O comando checa a existência de configurações já existentes, para evitar que haja duplicidade de arquivos de configuração. Em seguida, utilizando um gerador, cria os dois arquivos necessários para a configuração da ferramenta:

- `config/initializers/bdd_logger.rb` - Arquivo com as configurações de inicialização da ferramenta. Por estar localizado na pasta *config/initializers* é carregado automaticamente sempre que um servidor *Ruby on Rails* é iniciado.
- `features/support/env.rb` - Arquivo que contém o código adicionado ao *hook* do *Cucumber* para melhorar o processo de escrita dos *logs*. Será exposto com mais detalhes na seção 4.2.4.

[1] *Make your own gem*, disponível em: https://guides.rubygems.org/make-your-own-gem/
[2] *Creating and Customizing Rails Generators Templates*, disponível em: https://guides.rubyonrails.org/generators.html

run

O comando *run* ativa a variável de ambiente 'BDD_OBSERVE', responsável por indicar quando o analisador deve monitorar a execução dos testes de comportamento, e executa o *Cucumber*. O comando, por padrão, executa todos os casos de teste presentes na pasta `features`, a pasta padrão do *Cucumber*. Ele pode receber os mesmos argumentos que o *Cucumber* receberia em sua execução.

Após a conclusão da análise o comando ainda confere se os arquivos em formato *JSON* foram criados corretamente.

generate

O comando `generate` é responsável por chamar os geradores e passar as opções de execução. Antes de executar um gerador, primeiro é feita a checagem de existência dos arquivos intermediários que servem de entrada. Em seguida, os argumentos recebidos são passados para o gerador correspondente.

clean

Comando para remover os arquivos temporários criados pela gema. Por padrão remove os arquivos criados após a análise de execução dos testes de comportamento. Cada gerador cadastrado na ferramenta pode declarar os caminhos para os arquivos que gera, de forma que esses arquivos possam ser deletados pelo comando `clean`, quando informado o nome do gerador como argumento.

4.2.4 Evoluções

Para que a ferramenta pudesse ser executada em mais projetos, o módulo *Observer* precisou de mudanças em seu processo de criação, e para melhorar o desempenho da ferramenta, foram feitas melhorias no processo de geração dos *logs*, em busca de deixá-los mais padronizados e de fácil integração com módulos futuros.

***Observer* em módulos aninhados**

Na versão original do *TestGenerator*, são injetados *Observers*, como o da Listagem 4.2, em cada classe de Modelo da aplicação, utilizando `model.send(:include , TestGenerator :: Observer)` como pode ser visto no trecho das linhas 7 a 10 da Listagem 2.3. Através do recurso de metaprogramação são injetados aspectos em cada método presente na classe de Modelo, como pode ser visto no trecho das linhas 7 a 11 da Listagem 4.3 [20].

```
1    def self.included(klass)
```

```ruby
    klass.extend(ClassMethods)
    observer = const_set("#{klass.name}Observer", Module.new)
    klass.prepend observer
  end
```

Listing 4.2: Trecho de código que define o *Observer*

```ruby
def observe
    klass = self
    methods = klass.instance_methods(false) - DENYLIST
    observer = const_get "#{klass.name}Observer"

    observer.class_eval do
      methods.each do |method_name|
        define_method(method_name) do |*args, &block|
          reflect(klass, method_name, args, self)
          super(*args, &block)
        end
      end
    end
  end
```

Listing 4.3: Trecho de código que define método *observe* do *Observer*

No trecho de código apresentado, os *observers* são inicializados com base no nome da classe que está sendo observada, isso é definido utilizando interpolação na cadeia de caracteres `"#{klass.name}Observer"`. Essa abordagem gera um problema quando o projeto utiliza *namespaces*, como no caso do *diaspora*. Na linguagem *Ruby*, os *namespaces* são acessados utilizando o operador "`::`".

Em projetos que possuem *namespaces* na camada de modelos, podemos acessar uma classe de dentro desse módulo utilizando a estrutura `Namespace::Classe`. O *Observer* dessa classe será nomeado `Namespace::ClasseObserver`, que é um nome não permitido por conter o operador "`::`".

Para corrigir esse problema, duas possibilidades de alteração foram levantadas: gerar o *observer* apenas com o nome da classe, descartando os nomes dos módulos pais, ou substituir o operador "`::`" por outro caractere válido na nomeação de variáveis. A primeira abordagem faria o *observer* `Namespace::ClasseObserver` se transformar em `ClasseObserver`. Isso resolve o problema do caractere inválido, mas levanta outro possível problema: a geração de *observers* com nomes repetidos caso existam classes com mesmo nome em *namespaces* diferentes.

Logo, a solução foi a troca do operador "`::`" pelo caractere "`_`". Isso pode ser feito facilmente ao alterar a cadeira de caracteres `"#{klass.name}Observer"`, presente nos métodos `observe` e `self.included` do arquivo `observer.rb` para `"#{klass.name.gsub("::",`

"_")}Observer"`. Com essa alteração, o *observer* `Namespace::ClasseObserver` será nomeado `Namespace_ClasseObserver`, que agora é válido e não levanta o problema da duplicidade na nomeação dos *observers*.

Geração de arquivos intermediários

Os arquivos intermediários gerados pelo *TestGenerator* são arquivos *Ruby*, em que cada entrada do *log* é escrita em um *Hash*[3], com a seguinte estrutura de chaves e valores:

- *klass*: recebe o nome da classe atual;
- *method_name*: recebe o nome do método sendo observado;
- *args*: recebe um *array* com os argumentos da chamada da função observada;
- *attrs*:recebe os atributos da instância da classe atual;
- *response*: recebe a resposta da execução do método após a execução do *reflector* no método chamado.

Um exemplo de arquivo pode ser visto na Listagem 4.6. Ela representa o *log* dos métodos da classe de modelo *Friendship* do projeto *Facebook Clone*. Podemos notar que a informação de nome da classe se repete em todas as linhas. Também, nesse caso, todos os valores da chave *response* foram *null*.

```
1  { "klass": "Friendship", "method": "autosave_associated_records_for_user", "args": [], "attrs": {"id":20,"user_id":89,"friend_id":90,"confirmed":true,"created_at":"2021-10-27T02:44:13.319Z","updated_at":"2021-10-27T02:44:13.319Z"}, "response": "null" }
2  { "klass": "Friendship", "method": "autosave_associated_records_for_friend", "args": [], "attrs": {"id":20,"user_id":89,"friend_id":90,"confirmed":true,"created_at":"2021-10-27T02:44:13.319Z","updated_at":"2021-10-27T02:44:13.319Z"}, "response": "null" }
3  { "klass": "Friendship", "method": "autosave_associated_records_for_user", "args": [], "attrs": {"id":2,"user_id":4,"friend_id":5,"confirmed":true,"created_at":"2021-10-27T02:45:41.101Z","updated_at":"2021-10-27T02:45:41.101Z"}, "response": "null" }
4  { "klass": "Friendship", "method": "autosave_associated_records_for_friend", "args": [], "attrs": {"id":2,"user_id":4,"friend_id":5,"confirmed":true,"created_at":"
```

[3]Na linguagem *Ruby*, *Hash* é a estrutura de chave-valor equivalente ao *HashMap* da linguagem *Java*, ou ao Objeto da linguagem *Javascript*.

```
2021-10-27T02:45:41.101Z","updated_at":"2021-10-27T02:45:41.101Z"}, "
response": "null" }
```

Listing 4.4: Arquivo intermediário do TestGenerator: `Friendship.rb`

Outro fato que pode ser notado é que o nome do arquivo gerado é o mesmo nome da classe monitorada. Isso nos permite retirar a chave *klass* das entradas do arquivo sem perder informações. Também foi removida a chave *response*, devido à alteração arquitetural descrita na Subseção 4.2.2

Para reduzir entradas duplicadas[4] no arquivo intermediário, o nome dos métodos passou a ser utilizado como chave, cujo valor são os dados de atributos da classe e argumentos da chamada de função. Também foi alterada a forma como os arquivos são nomeados. Originalmente os arquivos são nomeados apenas com o nome da classe acrescido da extensão `rb`, que designa arquivos da linguagem *Ruby*. O novo esquema de nomeação conta com o prefixo `BDD_` e a extensão de arquivo passa a ser `.json`. Esse novo formato de arquivo permite que os arquivos sejam utilizados, inclusive em outras linguagem que possuam suporte a esse formato. O arquivo resultante, com as alterações propostas pode ser visto na Listagem 4.5.

```
1  [
2   {"autosave_associated_records_for_user": {"args":"[]","attrs":"{\"id
   \":null,\"user_id\":182,\"friend_id\":181,\"confirmed\":true,\"
   created_at\":null,\"updated_at\":null}"}},
3   {"autosave_associated_records_for_friend": {"args":"[]","attrs":"{\"
   id\":null,\"user_id\":182,\"friend_id\":181,\"confirmed\":true,\"
   created_at\":null,\"updated_at\":null}"}}
4   ]
```

Listing 4.5: Arquivo intermediário do BDD-CLI: `BDD_Friendship.json`

Redução de operações de escrita

Na arquitetura original do *TestGenerator*, as operações de escrita nos arquivos de *log* são realizadas a cada vez que o método ou classe é utilizado, através do método *log*, mostrado na Listagem 4.6. Isso acarreta grandes complicações ao tratar do tempo de execução da ferramenta, já que operações de leitura e escrita em arquivos são muito custosas para o sistema operacional.

```
1  def log(klass, method_name, args, attrs, response)
2      # Create temp model file
3      file_name = Rails.root.join("tmp/#{klass}.rb")
```

[4] Dizemos que as entradas são duplicadas por conta da forma como o arquivo de *log* é utilizado posteriormente pelo gerador de testes. Nele apenas a primeira entrada de cada método é utilizada e as entradas restantes são descartadas.

```
    puts "Logger inicializado em #{file_name}"
    file_line = "{ \"klass\": \"#{klass}\", \"method\": \"#{
method_name}\", \"args\": #{args.inspect}, \"attrs\": #{attrs.to_json
}, \"response\": #{response.inspect.gsub('nil', 'null').to_json} }"

    file = File.new(file_name, 'a')

    unless File.open(file_name).each_line.any? { |line| line.include?(
file_line) }
      file.puts file_line
    end

    file.close
  end
```

Listing 4.6: Método *log* do *TestGenerator*

Para a proposta da nova arquitetura, com o intuito de diminuir o custo de tempo das operações de escrita, os métodos e classes passaram a ser mapeados para um recurso fornecido pelo *framework Rails*: o *Cache*. Como pode ser visto na Listagem 4.7

```
  def log(klass, method_name, args, obj)

    attrs = obj.instance_variable_get("@attributes").to_hash

    file_line = {
      "args": args.to_json,
      "attrs": attrs.to_json
    }

    Rails.cache.write("#{klass}##{method_name}",file_line)
  end
```

Listing 4.7: Método *log* do *BDD-CLI*

Uma das bibliotecas que compõem o *framework Ruby on Rails* é o *Active Support*[5]. Essa biblioteca provê algumas implementações de *Cache* para uso. A implementação utilizada foi o *Memory Store*[6], por armazenar os dados em memória e possuir um tamanho satisfatório para armazenar os dados de cada cenário executado.

O *Cucumber* possui *hooks*[7] que funcionam como pontos de entrada para execução de código externo durante a execução dos testes de comportamento. O *hook* utilizado foi o *After*, que é ativado após a execução de cada cenário de teste. Assim, o analisador

[5]https://guides.rubyonrails.org/active_support_core_extensions.html
[6]https://api.rubyonrails.org/classes/ActiveSupport/Cache/MemoryStore.html
[7]https://cucumber.io/docs/cucumber/api/#hooks

realiza os *logs* no *Cache* e, após cada cenário, os dados que estão em *cache* são escritos nos arquivos *json* correspondentes.

Adaptação do gerador de testes

Com a alteração do arquivo intermediário gerado, também é necessário adaptar os módulos que consomem esse arquivo. Atualmente, o arquivo é utilizado no módulo gerador de testes. A primeira alteração necessária foi no método `read_temp_file`. Em sua implementação original, o método abre o arquivo de extensão `rb` e faz a leitura linha a linha do arquivo, executando o método `JSON.parse` em cada linha encontrada, como pode ser visto na Listagem 4.8.

```
1    def read_temp_file(klass)
2      file_path = Rails.root.join("tmp/#{klass}.rb")
3      return false unless File.exists?(file_path)
4      File.readlines(file_path).map { |line| JSON.parse(line.strip) }
5    end
```

Listing 4.8: Leitor de arquivo temporário do *TestGenerator*

Para realizar a leitura do novo formato de arquivo, o método agora busca por um arquivo de extensão `json` e com o prefixo `BDD_`. O novo arquivo não necessita que a leitura seja feita linha a linha, então executamos o método `JSON.parse` apenas uma vez, retornando a lista de métodos com os atributos de classe e argumentos da chamada do método. O método `read_temp_file` alterado é mostrado na Listagem 4.9.

```
1    def read_temp_file(klass)
2      file_path = Rails.root.join("tmp/BDD_#{klass}.json")
3      return false unless File.exists?(file_path)
4      JSON.parse(File.read(file_path))
5    end
```

Listing 4.9: Leitor de arquivo temporário do *BDD-CLI*

A última alteração foi no esquema de nomeação das *factories* e dos testes gerados. Para criar um padrão que pudesse ser gerenciado pela *CLI*, os arquivos agora são gerados dentro de uma pasta `BDD`, criada nos diretórios `spec/factories` e `spec/models`.

4.2.5 Análise final da implementação

Na Tabela 4.1 são relacionados os objetivos do trabalho com as correções, reorganizações e evoluções feitas durante o desenvolvimento do projeto, com o intuito de assinalar as contribuições realizadas pela proposta da nova arquitetura.

Objetivos	Contribuições	Resultados
Correção do *TestGenerator* para execução com arquiteturas mais complexas	• Correções da instalação • Correção no *Observer* gerado	Instalação das dependências feita em um comando, possibilidade de monitar projetos com arquiteturas mais complexas, como o uso de *namespaces*.
Nova arquitetura e adição da CLI	• CLI • Módulo Analisador • Módulo Gerador • Módulo Instalador	CLI que unifica os comandos de execução e permite adição de novas funcionalidades. Modularização do projeto, melhorando a separação de responsabilidades.
Modulariação e padronização do arquivo intermediário	• Arquivo em formato JSON • Nova nomenclatura para os arquivos e chaves • Arquivos menores	Arquivo intermediário em formato JSON, amplamente utilizado no mercado, arquivos mais legíveis e com tamanho reduzido.
Adaptação do *TestGenerator* para se adequar à nova arquitetura	• Novo leitor de arquivos • Adaptação do gerador de *factories* e casos de teste	O gerador de testes agora funciona com o arquivo JSON como entrada e possui um novo esquema de nomeação para as factories e testes. Mantendo os resultados do *TestGenerator*

Tabela 4.1: Análise final da implementação

Capítulo 5

Validação da Ferramenta

Este capítulo apresenta a validação dos objetivos da pesquisa através da coleta de métricas geradas pela utilização da ferramenta proposta em projetos reais encontrados em repositórios públicos do *GitHub*, que serviram como provas de conceito para este trabalho.

5.1 Planejamento

A fim de validar a proposta do trabalho, de evolução da ferramenta *TestGenerator* para uma arquitetura que torne mais fácil a utilização em projetos maiores e com organizações mais robustas, foi preciso avaliar qualitativamente a usabilidade da ferramenta, tanto para a sua arquitetura original quanto para a nova que foi proposta. Além disso, foram avaliados também os arquivos resultantes da análise de execução dos testes de comportamento para ambas arquiteturas, com o intuito de verificar a legibilidade e objetividade de cada *log* gerado.

Para validar a proposta de entrega de uma nova arquitetura, com um módulo gerador de testes unitários a partir dos *logs* obtidos do módulo analisador, foi preciso comparar o resultado obtido pela utilização dessa proposta com o que já é fornecido pela arquitetura original do *TestGenerator*. Para tanto, foram consideradas algumas métricas que avaliassem a evolução da ferramenta e contrastassem os resultados das duas arquiteturas, sendo elas:

1. Média do tamanho dos arquivos gerados pelo módulo Analisador do *BDD-CLI* em relação aos *logs* gerados pelo *TestGenerator*.

2. Tempo de execução para a antiga e nova arquitetura.

3. Cobertura de código fornecida pelo módulo gerador de testes unitários do *BDD-CLI* em comparação ao que é fornecido pelo *TestGenerator*.

Com o propósito de realizar as coleta das métricas descritas anteriormente, foram selecionados alguns projetos *open source* publicados na plataforma *GitHub*. Para cada projeto selecionado, foi aplicada a mesma sequência de passos com o intuito de preservar a metodologia de obtenção das métricas, sendo eles:

1. **Clonar o repositório do projeto alvo do *Github*.**
2. **Determinar o grupo de testes a ser executado.**
3. **Para a versão original do projeto:**
   (a) Executar o *Cucumber* para anotar o tempo de execução da ferramenta.
   (b) Caso a gema *simplecov* não esteja instalada no projeto, então ela deve ser adicionada.
   (c) Executar a ferramenta que gera métricas relativas à cobertura de código, neste caso o *SimpleCov*, para obter a cobertura inicial dos testes unitários já existentes no projeto.
4. **Realizar a instalação da gema *TestGenerator* no projeto.**
5. **Realizar a instalação da gema *BDD-CLI* no projeto**
6. **Configurar o ambiente de teste.**
7. **Para cada ferramenta (*TestGenerator* e *BDD-CLI*) seguir as etapas:**
   (a) Executar a análise dos testes de comportamento e avaliar o arquivo de *log* gerado por essa análise para comparar a estrutura e tamanho entre os resultados das duas ferramentas.
   (b) No caso do *BDD-CLI*, é necessário executar o módulo de gerador de testes unitários, para gerar os testes a partir dos *logs* obtidos na etapa anterior.
   (c) Executar o *SimpleCov* para obter a cobertura de código oferecida pelos novos testes.

5.2 Projetos selecionados

Os projetos citados nesta seção compõem o conjunto utilizado para a realização dos testes de validação da arquitetura proposta. O processo de seleção foi separado em dois tipos, sendo eles:

1. Escolha de projetos já utilizados no estudo de caso em Marques and Fernandes [20].

2. Mineração de repositórios no *GitHub*.

Para o tipo (1), os projetos selecionados foram os que possuíam a maior quantidade de cenários *BDD* com a execução correta. Alguns projetos citados no estudo de caso do *TestGenerator* apresentavam erros de terceiros e fugiam do escopo definido para este trabalho.

Já para o tipo (2), foram minerados repositórios de projetos que utilizassem o *Ruby on Rails* para o desenvolvimento e possuíssem as gemas *cucumber-rails* e *rspec-rails* instaladas para a geração dos cenários *BDD* e execução dos testes unitários, respectivamente. Além disso, para refinar ainda mais os resultados da mineração, foram estipulados critérios relativos ao número de contribuidores e quantidade de *commits* feitos.

A partir desses critérios de seleção, foram determinados cinco projetos para a realização da validação da proposta da ferramenta, sendo eles:

Projeto	**Commits**	**Contribuidores**	**Repositório**
Zlp-Scheduler	381	12	tamu-zlp/zlp-scheduler
Doei	322	6	limabia/doei
Facebook Clone	81	2	johnsonsirv/facebook-clone
Diaspora	20.472	368	diaspora/diaspora
Chiquinho	240	6	EngSwCIC/Chiquinho

Tabela 5.1: Projetos escolhidos para realização dos testes

5.2.1 *Zlp-scheduler*

Para este projeto a suite de testes escolhidas excluiu o fluxo de autenticação porque existiam erros na implementação dos cenários que ocasionavam em mais erros na análise feita. Os testes relativos à validação da proposta foram baseados em 40 cenários *BDD* já existentes.

Como é possível verificar nas Listagens 5.1 e 5.2, os arquivos de teste unitário gerados pela análise da execução dos cenários BDD possuem a mesma estrutura para ambas arquiteturas, o que contempla o objetivo de adaptar a geração de testes unitários do *TestGenerator* para a arquitetura proposta no *BDD-CLI*.

```
1 describe '#email=' do
2   it 'should' do
3     user = create(:bdd_user)
4     expect(user.email=(abel.satterfield@tamu.edu)).to eq 'abel.
  satterfield@tamu.edu'
5   end
6 end
```

Listing 5.1: Teste gerado pelo *BDD-CLI* para o método *setter* do *email*

```
describe '#email=' do
  it 'should' do
    user = create(:user, {uin: '', lastname: 'Keebler', firstname: 'Corgi_Dog', email: 'tiana@tamu.edu', password_digest: '$2a$04$nFP/n6mTcg02.rSjjileJedElMPTdOWmJxeHQw.5ZNIRtUQOOZkja' })
    expect(user.email=(tiana@tamu.edu)).to eq 'tiana@tamu.edu'
  end
end
```

Listing 5.2: Teste gerado pelo *TestGenerator* para o método *setter* do *email*

Com a execução dos passos da metodologia descrita na seção anterior, foram obtidos os resultados mostrados na Tabela 5.2, em que é possível verificar que a execução de toda a análise e geração de testes feitas pela nova arquitetura foi bastante otimizada em relação ao que é fornecido pelo *TestGenerator*, ao passo que o número médio de linhas dos arquivos de *log* diminuiu e a cobertura de código permaneceu a mesma.

Ferramenta	Nº de linhas	Tempo de Execução	Cobertura de Código
Cucumber	-	14 segundos	51.52%
TestGenerator	614	28 segundos	53.79%
BDD-CLI	5	16 segundos	53.79%

Tabela 5.2: Comparativo entre execução das ferramentas para o Zlp-Scheduler

5.2.2 Doei

Neste projeto, os testes de validação da ferramenta foram realizados com base nos 60 cenários *BDD* já implementados.

Com a aplicação dos passos da metodologia, foram obtidos os dados apresentados na Tabela 5.3. Esses dados mostram a diminuição do tamanho médio dos arquivos gerados pela análise feita pelo *BDD-CLI*, em relação à arquitetura anterior, e a manutenção do índice de cobertura de código oferecido pelo *TestGenerator*, o que indica que, neste projeto, houve um ganho pela utilização da ferramenta *BDD-CLI*.

Ferramenta	Nº de linhas	Tempo de Execução	Cobertura de Código
Cucumber	-	16 segundos	98.05%
TestGenerator	197	27 segundos	87.42%
BDD-CLI	12	18 segundos	87.42%

Tabela 5.3: Comparativo entre execução das ferramentas para o Doei

5.2.3 *Facebook Clone*

A suite de testes de comportamento utilizada neste projeto para a validação da ferramenta contém 19 cenários *BDD*. Durante a configuração do ambiente de teste não houve nenhum erro que impossibilitasse a execução das ferramentas.

Por meio do que é apresentado na Tabela 5.4, é possível verificar que há uma melhoria de desempenho na utilização da nova arquitetura em relação à antiga, visto que a mesma cobertura de código foi atingida por ambas, mesmo com a diminuição significativa na média de quantidade de linhas por arquivo de *log* gerado.

Ferramenta	Nº de linhas	Tempo de Execução	Cobertura de Código
Cucumber	-	2 segundos	68.85%
TestGenerator	183	7 segundos	75.08%
BDD-CLI	7	3 segundos	75.08%

Tabela 5.4: Comparativo entre execução das ferramentas para o *Facebook Clone*

5.2.4 *Diaspora*

Durante o teste realizado com a ferramenta *TestGenerator*, ocorreram alguns erros na execução da análise feita pela ferramenta, como mostrado na figura 5.1. Um dos *namespaces* da camada de modelos é o *Services*.

Para acessar uma classe de dentro desse módulo, como a classe *Tumblr*, a estrutura `Services::Tumblr` é utilizada. A estrutura `Services::TumblrObserver` corresponde ao *Observer* gerado para essa classe. Esse processo lança o erro de **nome de constante incorreto**, pois foi gerado um nome não permitido por conter o operador "`::`" . A solução para esse erro foi a alteração no padrão de nomenclatura dos *Observers*, como exposto na subseção 4.2.4.

Ao corrigir esse problema, outro impedimento apareceu, a execução do *TestGenerator* impacta nos comandos de criação do banco de dados, gerando o erro apresentado na figura 5.1 já exposto na subseção 4.2.1.

Corrigidos tais erros para possibilitar a execução da análise feita pelo *TestGenerator*, foram obtidos os dados mostrados na Tabela 5.5, em que é possível verificar uma diferença significativa tanto na cobertura gerada pelos testes unitários quanto no número médio de linhas dos arquivos gerados pelo analisador do *BDD-CLI* em relação ao que é fornecido pelo *TestGenerator*.

Mesmo com essas correções, ainda resta o problema que o Aspecto executa a função de reflexão ao invés da função de *log*, o que leva à ocorrência do erro citado na seção 4.2.2,

```
→ bundle exec rails db:create
/home/icaro/.asdf/installs/ruby/2.6.0/lib/ruby/gems/2.6.0/gems/pry-byebug-3.8.0/lib/pry-byebug/control_d_handler.rb:5: warning
: control_d_handler's arity of 2 parameters was deprecated (eval_string, pry_instance). Now it gets passed just 1 parameter (p
ry_instance)
/home/icaro/.asdf/installs/ruby/2.6.0/lib/ruby/gems/2.6.0/gems/pry-byebug-3.8.0/lib/pry-byebug/control_d_handler.rb:5: warning
: control_d_handler's arity of 2 parameters was deprecated (eval_string, pry_instance). Now it gets passed just 1 parameter (p
ry_instance)
**************************************************
⛔ WARNING: Sidekiq testing API enabled, but this is not the test environment.  Your jobs will not go to Redis.
**************************************************
[2021-10-26T18:21:53] INFO  PID-14959 TID-47023587482000 DiasporaFederation: successfully configured the federation library
root  ..........................................  *debug      -T
- <Appenders::Stdout name="stdout">
- <Appenders::RollingFile name="file">
  ActionView::Base  .............................  *debug  +A  -T
  ActiveRecord::Base  ...........................  *debug  +A  -T
  ActiveSupport::Cache::NullStore  ..............   debug  +A  -T
  DiasporaFederation  ...........................   debug  +A  -T
    DiasporaFederation::Salmon::MagicEnvelope  ...  *info  +A  -T
  Logging  ......................................    *off  -A  -T
  Rails  ........................................   debug  +A  -T
rails aborted!
NameError: wrong constant name Api::OpenidConnect::AuthorizationObserver
/home/icaro/Projects/TG/ESTUDO/diaspora/vendor/TestGenerator/lib/test_generator/observer.rb:9:in `const_set'
/home/icaro/Projects/TG/ESTUDO/diaspora/vendor/TestGenerator/lib/test_generator/observer.rb:9:in `included'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/application.rb:112:in `include'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/application.rb:112:in `block (2 levels) in <class:Application>'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/application.rb:111:in `each'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/application.rb:111:in `block in <class:Application>'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/environment.rb:7:in `<top (required)>'
bin/rails:4:in `require'
bin/rails:4:in `<main>'
Tasks: TOP ⇒ db:create ⇒ db:load_config ⇒ environment
(See full trace by running task with --trace)
```

Figura 5.1: Erro na execução do *Diaspora* com o *TestGenerator*

```
→ bundle exec rake db:create
/home/icaro/.asdf/installs/ruby/2.6.0/lib/ruby/gems/2.6.0/gems/pry-byebug-3.8.0/lib/pry-byebug/control_d_handler.rb:5: warning: contr
ol_d_handler's arity of 2 parameters was deprecated (eval_string, pry_instance). Now it gets passed just 1 parameter (pry_instance)
**************************************************
⛔ WARNING: Sidekiq testing API enabled, but this is not the test environment.  Your jobs will not go to Redis.
**************************************************
[2021-10-26T19:05:57] INFO  PID-22425 TID-47218045573560 DiasporaFederation: successfully configured the federation library
root  ..........................................  *debug      -T
- <Appenders::Stdout name="stdout">
- <Appenders::RollingFile name="file">
  ActionView::Base  .............................  *debug  +A  -T
  ActiveRecord::Base  ...........................  *debug  +A  -T
  ActiveSupport::Cache::NullStore  ..............   debug  +A  -T
  DiasporaFederation  ...........................   debug  +A  -T
    DiasporaFederation::Salmon::MagicEnvelope  ...  *info  +A  -T
  Logging  ......................................    *off  -A  -T
  Rails  ........................................   debug  +A  -T
rake aborted!
ActiveRecord::NoDatabaseError: FATAL:  database "diaspora_development" does not exist
/home/icaro/Projects/TG/ESTUDO/diaspora/app/models/person.rb:125:in `<class:Person>'
/home/icaro/Projects/TG/ESTUDO/diaspora/app/models/person.rb:7:in `<top (required)>'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/application.rb:110:in `block in <class:Application>'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/environment.rb:7:in `<top (required)>'
/home/icaro/.asdf/installs/ruby/2.6.0/bin/bundle:23:in `load'
/home/icaro/.asdf/installs/ruby/2.6.0/bin/bundle:23:in `<main>'

Caused by:
PG::ConnectionBad: FATAL:  database "diaspora_development" does not exist
/home/icaro/Projects/TG/ESTUDO/diaspora/app/models/person.rb:125:in `<class:Person>'
/home/icaro/Projects/TG/ESTUDO/diaspora/app/models/person.rb:7:in `<top (required)>'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/application.rb:110:in `block in <class:Application>'
/home/icaro/Projects/TG/ESTUDO/diaspora/config/environment.rb:7:in `<top (required)>'
/home/icaro/.asdf/installs/ruby/2.6.0/bin/bundle:23:in `load'
/home/icaro/.asdf/installs/ruby/2.6.0/bin/bundle:23:in `<main>'
Tasks: TOP ⇒ db:create ⇒ db:load_config ⇒ environment
(See full trace by running task with --trace)
```

Figura 5.2: Erro na execução do *Diaspora* com o *TestGenerator*

em que alguns métodos caem em uma chamada circular, ocasionando inconsistências nos testes gerados.

Um dos pontos que mais impactou os testes foi o tempo de execução. Nesse projeto, os casos de teste de comportamento são dividos em dois escopos, *web* e *mobile*. Executando a suíte *mobile*, apenas com o *Cucumber*, obtemos um tempo próximo dos 3 minutos, como podemos ver na Tabela 5.5. Ao adicionarmos o *TestGenerator*, o tempo de execução dessa mesma suíte foi de 9 minutos.

Ferramenta	**Nº de linhas**	**Tempo de Execução**	**Cobertura de Código**
Cucumber	-	3 minutos e 12 segundos	97.52%
TestGenerator	1156	9 minutos	51.75%
BDD-CLI	13	3 minutos e 38 segundos	64.78%

Tabela 5.5: Comparativo entre execução das ferramentas para o *Diaspora*

5.2.5 Chiquinho

Neste projeto foram removidos os fluxos de avaliação de matérias e definição de grades curriculares, pois existiam implementações incorretas, o que ocasionava alguns erros na análise da execução dos cenários. Diante disso, foram utilizados 41 cenários para realizar a validação da ferramenta proposta.

Para este projeto, é possível perceber, através da Tabela 5.6, que houve o aprimoramento da ferramenta *TestGenerator* com a aplicação da nova arquitetura fornecida pelo *BDD-CLI*, porque a cobertura de código é a mesma para as duas ferramentas, mas o tempo de execução da nova proposta é consideravelmente menor e o arquivo de *log* gerado pela análise é mais sucinto.

Ferramenta	**Nº de linhas**	**Tempo de Execução**	**Cobertura de Código**
Cucumber	-	2 segundos	70.33%
TestGenerator	516	15 segundos	76.57%
BDD-CLI	11	3 segundos	76.57%

Tabela 5.6: Comparativo entre execução das ferramentas para o Chiquinho

5.3 Resultados gerais

De acordo com os resultados obtidos através dos testes de validação da ferramenta, foi possível observar que:

- **O processo de instalação e execução do *BDD-CLI* foi simplificado quando comparado ao do *TestGenerator***.
  Em todos os projetos, foram executados apenas dois comandos para configurar as dependências e executar o monitoramento dos testes de comportamento. Enquanto para o *TestGenerator* era necessário uma série de correções e configurações manuais.
- **O tempo de execução da análise dos testes de comportamento diminuiu significativamente**.
  Como mostrado na Figura 5.3, para cada projeto testado, o tempo de execução necessário para realizar a análise da execução dos testes de comportamento utilizando a ferramenta *BDD-CLI* foi bem próximo ao tempo despendido pela ferramenta *Cucumber*, que apenas executa os testes sem mapear o comportamento do sistema. Já com o *TestGenerator*, o tempo de execução para realizar as análises dos testes é consideravelmente maior que os tempos das outras duas ferramentas. Essa diferença no tempo de execução é mais expressiva em projetos maiores e mais complexos.

- **O tamanho dos arquivos de *logs* diminuíram em relação à quantidade de linhas, mantendo a mesma relevância das informações**.
  A nova abordagem para escrita de *logs* acabou com a duplicidade de dados nos arquivos, reduzindo a complexidade para entendimento e leitura do arquivo, o que facilita o desenvolvimento de novas funcionalidades. Mesmo com a alteração da estrutura do *log*, os resultados dos testes gerados pelo módulo gerador foram semelhantes aos da abordagem original. No caso do projeto *Diaspora*, mais casos de teste foram gerados.

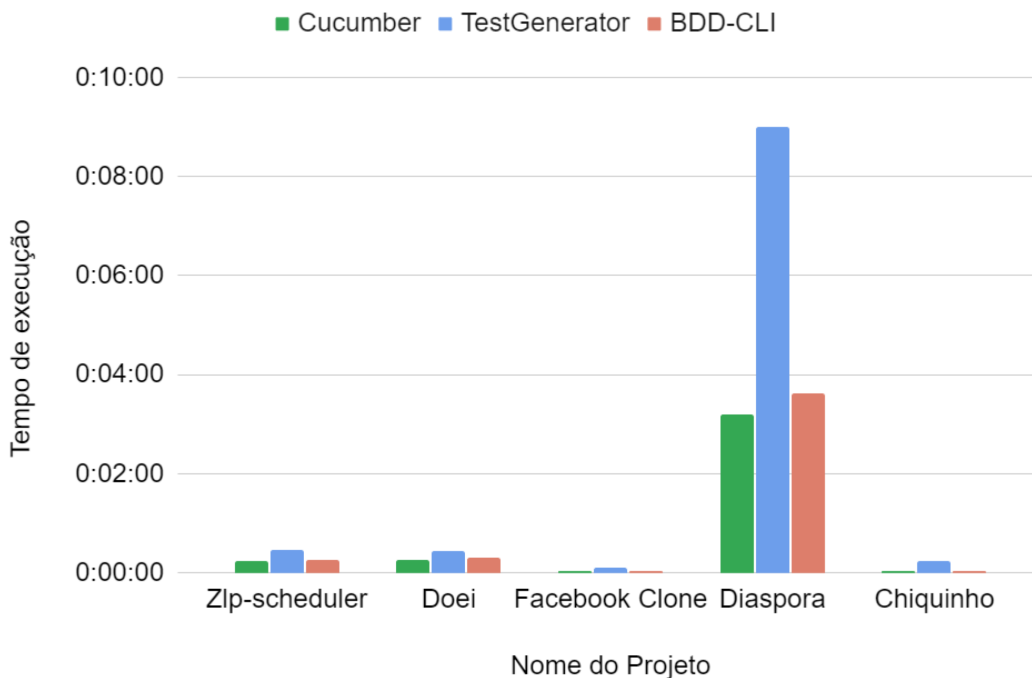

Figura 5.3: Comparação dos tempos de execução

Capítulo 6

Conclusão

A evolução do mercado de desenvolvimento de *software* traz consigo muitos desafios relativos a organização dos processos de criação de um sistema. Problemas de entendimento dos requisitos do sistema e entregas de baixa qualidade se tornaram muito comuns. Neste contexto, surgiram algumas metodologias que buscavam reduzir os impactos de tais problemas, como é o exemplo da Metodologia Ágil, abordada no presente trabalho. Ferramentas que auxiliam a organização de processos de desenvolvimento de *software* e aceleram as entregas de produtos de alta qualidade se fizeram cada vez mais presentes.

Este trabalho apresentou uma proposta de geração automática de artefatos de *software* a partir da análise da execução de cenários *BDD*. Para tanto, foi utilizado como ponto de partida a ferramenta *TestGenerator*, uma implementação que analisa a execução de testes de comportamento para a geração automática de testes unitários.

A implementação da proposta proporcionou o entendimento de conceitos como Programação Orientada a Aspectos, Metaprogramação e Desenvolvimento Orientado a Comportamento, que foram utilizados para realizar as correções e evoluções na ferramenta original. Além disso, com as pesquisas de trabalhos relacionados, foi possível analisar abordagens semelhantes ao que foi explicitado neste trabalho, o que contribuiu de forma significativa para algumas tomadas de decisão no decorrer do desenvolvimento do projeto.

Mediante os testes feitos para a validação da proposta, foi possível perceber as contribuições da nova arquitetura em relação a original. A utilização da *CLI* deixou o processo de configuração do ambiente menos suscetível a erros e com mais portabilidade para projetos reais, uma vez que, com poucos comandos, é possível começar a utilizar os módulos de análise e geração do *BDD-CLI*.

Além disso, com a simplificação e padronização dos arquivos resultantes do módulo analisador, o desenvolvimento de ferramentas futuras foi facilitado. A utilização do formato *JSON*, amplamente difundido e utilizado no mercado, possibilita o entendimento mais fácil das informações disponibilizadas em tais arquivos. Os resultados obtidos atra-

vés dos testes feitos mostram que a ferramenta *BDD-CLI* otimizou a execução da análise dos testes de comportamento quando comparado à execução do *TestGenerator* e, também, mesmo realizando o mapeamento do comportamento do sistema em arquivos de *log*, a execução da ferramenta para a nova arquitetura não apresentou grande mudança, ainda tratando de tempo de execução, em comparação ao que é obtido apenas com a execução do *Cucumber*.

Da mesma forma, os estudos feitos para validar a ferramenta apontaram que a adaptação da funcionalidade de geração de testes unitários do *TestGenerator*, quando adaptada para a nova estrutura do *BDD-CLI*, não surtiu efeitos negativos ao tratar de cobertura de código e qualidade dos testes gerados. Ao passo que a cobertura de código se manteve a mesma para os projetos testados e também os testes das classes da camada de Modelos possuem a mesma estrutura dos que eram gerados anteriormente.

Ademais, os resultados obtidos também trazem alguns pontos de atenção sobre possíveis, e importantes, aspectos que podem ser tratados e melhorados para a contínua evolução da ferramenta proposta, sendo eles:

- **Avaliar a recompensa da utilização do módulo gerador de testes unitários do *BDD-CLI* em projetos de grande porte**.
  Como visto nas Tabelas 5.3 e 5.5 , a cobertura de código oferecida pela ferramenta foi menor, em quantidade, se comparada à cobertura original. Dito isso, é interessante fazer um estudo para análise qualitativa dos testes gerados, com o intuito de verificar se os testes criados pelo *BDD-CLI* aumentariam a cobertura original de alguma forma, gerando novos testes que não haviam sido gerados originalmente.

- **Implementar novos módulos geradores**.
  Para centralizar a fonte de informações e diminuir as incongruências entre as diversas partes de um *software* e também aumentar o valor agregado pela utilização da ferramenta, é interessante incorporar mais funcionalidades ao projeto.

- **Realizar um estudo de caso envolvendo outros desenvolvedores**.
  Este aspecto é de extrema importância para verificar o interesse do mercado real em utilizar ferramentas deste tipo. Com esse estudo, será possível avaliar se, no estado atual, o custo benefício da aplicação do *BDD-CLI* em projetos interessa os desenvolvedores que vão utilizá-lo.

- **Ampliar a cobertura do módulo analisador para mapear a classe de Controladoras**.
  Atualmente, o módulo analisador avalia apenas as classes de Modelos do sistema. Para fornecer uma quantidade maior de informações relativas ao comportamento do

sistema, é interessante ampliar o mapeamento feito na análise dos testes de comportamento para abranger também a classe das *Controllers*.

- **Ampliar ferramenta para analisar os testes de comportamento para mais de um framework**.
  Um primeiro passo seria analisar projetos em *Ruby* que não usem o *framework Rails* mas que possuem o *Cucumber* como o ferramenta de testes de comportamento, já que na mineração de repositórios, descrita na seção 5.1, foram encontrados projetos com tal estrutura.

Referências

[1] Chapter 1. What Is Aspect-Oriented Programming? URL https://docs.jboss.org/aop/1.0/aspect-framework/userguide/en/html/what.html. 9

[2] Introduction to asdf. URL http://asdf-vm.com/guide/introduction.html#how-it-works. 14

[3] Behaviour-Driven Development - Cucumber Documentation. URL https://cucumber.io/docs/bdd/. 3, 6, 7

[4] Metaprogramming. URL https://cs.lmu.edu/~ray/notes/metaprogramming/. 9

[5] Manifesto for Agile Software Development. URL http://agilemanifesto.org/. 1

[6] Pickles - the open source Living Documentation Generator. URL https://www.picklesdoc.com/. 12

[7] BDD Framework for .NET - SpecFlow - Find Bugs before they happen. URL https://specflow.org/. 13

[8] Ruby-Doc.org: Documenting the Ruby Language. URL https://ruby-doc.org/. 8

[9] Scott Ambler. *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, 2002. ISBN 0-471-27190-X. 1

[10] Juyun Cho. Issues and challenges of agile software development with scrum. *Issues in Information Systems*, 9(2):188–195, 2008. 2

[11] Mike Cohn. *Desenvolvimento de software com Scrum: aplicando métodos ágeis com sucesso.* Grupo A - Bookman, 2000. ISBN 978-85-7780-819-9. URL http://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=3235954. OCLC: 923757001. 5

[12] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mário Jino. Introdução ao teste de software, 2007. URL http://www.sciencedirect.com/science/book/9788535226348. Archive: /z-wcorg/ ISBN: 9788535267495 8535267492 Library Catalog: http://worldcat.org. 10

[13] Mohsin Irshad, Ricardo Britto, and Kai Petersen. Adapting Behavior Driven Development (BDD) for large-scale software systems. *Journal of Systems and Software*, 177:110944, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.110944. URL https://www.sciencedirect.com/science/article/pii/S0164121221000418. 6

[14] John Irwin, Gregor Kickzales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, and J Loingtier. Aspect-oriented programming. *Proceedings of ECOOP, IEEE, Finland*, pages 220–242, 1997. 9

[15] Gaurav Kumar and Pradeep Bhatia. Impact of Agile Methodology on Software Development Process. *International Journal of Computer Technology and Electronics Engineering (IJCTEE)*, 2:2249–6343, August 2012. 5, 6

[16] Dean Leffingwell. Calculating Your Return on Investment from More Effective Requirements Management. page 5. URL `http://public.dhe.ibm.com/software/rational/web/whitepapers/2003/roi1.pdf`. 1

[17] YANNIS LILIS and ANTHONY SAVIDIS. A survey of metaprogramming languages. *ACM Computing Surveys*, 52(6):1 – 39, 2020. ISSN 03600300. URL `https://search-ebscohost-com.ez54.periodicos.capes.gov.br/login.aspx?direct=true&db=iih&AN=142465087&lang=pt-br&site=ehost-live`. 9

[18] Garm Lucassen, Marcel Robeer, Fabiano Dalpiaz, Jan Martijn EM Van Der Werf, and Sjaak Brinkkemper. Extracting conceptual models from user stories with visual narrator. *Requirements Engineering*, 22(3):339–358, 2017. 13

[19] Y.K. Malaiya, M.N. Li, J.M. Bieman, and R. Karcich. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51(4):420–426, 2002. doi: 10.1109/TR.2002.804489. 10

[20] Nicholas Nishimoto Marques and Rafael Alves Fernandes. Um arcabouço para a geração automatizada de testes funcionais a partir de cenários bdd. 2020. 3, 10, 22, 30

[21] Lauriane Pereira, Helen Sharp, Cleidson de Souza, Gabriel Oliveira, Sabrina Marczak, and Ricardo Bastos. Behavior-driven development benefits and challenges: reports from an industrial study. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, pages 1–4, 2018. 2

[22] Lauriane Pereira, Helen Sharp, Cleidson de Souza, Gabriel Oliveira, Sabrina Marczak, and Ricardo Bastos. Behavior-Driven Development Benefits and Challenges: Reports from an Industrial Study. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, XP '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-6422-5. doi: 10.1145/3234152.3234167. URL `https://doi.org/10.1145/3234152.3234167`. event-place: Porto, Portugal. 2

[23] Roger S. Pressman and Bruce R. Maxim. *Engenharia de Software: Uma abordagem profissional.* AMGH Editora Ltda., 8ª edition, 2016. ISBN 978-85-8055-534-9. 5, 6

[24] Thiago Rocha Silva and Brian Fitzgerald. Empirical findings on BDD story parsing to support consistency assurance between requirements and artifacts. 2021. doi: 10.1145/3463274.3463807. URL `https://ulir.ul.ie/handle/10344/10308`. Accepted: 2021-07-06T11:51:52Z Publisher: Association for Computing Machinery. 2

[25] Daniel Rohde. Implementation of a command-line interface for the vizzanalyzer, 2008. 2

[26] John Ferguson Smart. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Manning Publications, 1 edition, 2014. ISBN 161729165X,9781617291654. URL http://gen.lib.rus.ec/book/index.php?md5=c542c9ebfbaa751a5486a160fbd48a2c. 6

[27] Hrafnhildur Sif Sverrisdottir, Helgi Thor Ingason, and Haukur Ingi Jonasson. The Role of the Product Owner in Scrum-comparison between Theory and Practices. *Procedia - Social and Behavioral Sciences*, 119:257–267, 2014. ISSN 1877-0428. doi: https://doi.org/10.1016/j.sbspro.2014.03.030. URL https://www.sciencedirect.com/science/article/pii/S1877042814021211. 6

[28] Fiorella Zampetti, Andrea Di Sorbo, Corrado Aaron Visaggio, Gerardo Canfora, and Massimiliano Di Penta. Demystifying the adoption of behavior-driven development in open source projects. *Information and Software Technology*, 123:106311, 2020. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106311. URL https://www.sciencedirect.com/science/article/pii/S095058492030063X. 6

[29] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL https://doi.org/10.1145/267580.267590. 10