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

Um Arcabouço para a Geração Automatizada de Testes Funcionais a partir de Cenários BDD

Nicholas N. Marques
Rafael A. Fernandes

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

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2020

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

Um Arcabouço para a Geração Automatizada de Testes Funcionais a partir de Cenários BDD

Nicholas N. Marques
Rafael A. Fernandes

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

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

do Bacharelado em Ciência da Computação

Brasília, 18 de dezembro de 2020

Dedicatória

Dedicamos este trabalho à nossos pais, que sempre nos proporcionaram condições para que continuássemos conseguindo sempre buscar o conhecimento.

Agradecimentos

Agradecemos aos nossos pais, por todo apoio para possibilitar que tivéssemos foco nos estudos. Agradecemos também Markus Schirp, criador da gema Mutant para a linguagem Ruby. Ele disponibilizou licenças da gema para que conseguíssemos validar com testes de mutação a ferramenta desenvolvida. E um agradecimento em especial à nossa orientadora, Prof.a Dr.a Genaína Nunes Rodrigues, que sempre nos auxiliou em toda a graduação e em particular também com este trabalho.

Resumo

Atualmente a complexidade dos sistemas vêm aumentando muito e com isso se faz cada vez mais necessário técnicas de Engenharia de Software para se ter sistemas com mais qualidade. Algumas das técnicas mais conhecidas e aplicadas para se ter qualidade de software são as que envolvem realizar testes automatizados para o sistema. Os testes se tornam ainda mais prementes quando se considera a expressiva adoção de metodologias ágeis de desenvolvimento de software. Neste trabalho, propomos uma abordagem de geração de testes funcionais a partir de cenários construídos conforme o conceito de Desenvolvimento Orientado à Comportamento (BDD). A partir da execução do conjunto de testes é feito um mapeamento, por meio da Programação Orientada a Aspectos (AOP), de quais funções mais internas do sistemas são chamadas para reproduzir aquele comportamento. Com base na identificação dessas funções, seus parâmetros e seus retornos, geramos testes funcionais automatizados relevantes. Estes geram uma cobertura de testes elevada, agregando na qualidade do sistema e auxiliando o trabalho do desenvolvedor. Isso possibilita o programador realizar uma gama de testes elevada de forma automática. A abordagem foi desenvolvida em arcabouço desenvolvido em Ruby on Rails e aplicado a dez projetos de software open-source com commits recentes. Os resultados evidenciaram que a nossa abordagem consegue gerar testes que não apenas correspondem aos testes originais como também é capaz de gerar outros testes funcionais não contemplados originalmente nos projetos. Tais testes se mostraram relevantes quanto aos requisitos de software especificados em BDD.

Palavras-chave: Behavior Driven Development, testes funcionais, programação orientada a aspectos, metaprogramação, geração de testes

Abstract

Nowadays the complexity of the systems has been increasing a lot and with this, Software Engineering techniques are more and more necessary to have systems with more quality. Some of the best known and applied techniques to have software quality are those that involve performing automated tests for the system (either before or after development). Tests become even more urgent when the expressive adoption of agile software development methodologies is considered. In this work, we propose a functional test generation approach based on scenarios built according to the concept of Behavior Driven Development (BDD). A mapping of the functions called when a behavior test is executed is made, through the Aspects Oriented Programming (AOP). Based on the identification of these functions, their parameters and their returns, we generate relevant automated functional tests which generates a high test coverage, improving the quality of the system and helping the developer's work to perform a high range of tests. The approach was developed in framework developed in Ruby on Rails and applied to ten open-source software projects with recent commits. The results showed that our approach can generate tests that not only correspond to the original tests but the approach was able to generate other functional tests not originally contemplated in those projects and that were relevant to the software requirements specified in BDD.

Keywords: Behavior Driven Development, functional testing, Aspect-Oriented Programming, metaprogramming, test generation

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A ideia de se testar software é algo que surge inerentemente à natureza do software. A importância de se testar foi com o tempo também tendo sua relevância percebida, principalmente com a expansão da metodologia Ágil de desenvolvimento de software. Conforme Ahamed [1], o teste de um software representa o último estágio de um design de software, onde sua existência deve moldar o próprio código que ele testa.

Dada a complexidade inerente ao teste de software e seus diferentes tipos, i.e. aceitação, funcional, unitário, integração e sistema, percebeu-se a importância de se fazer testes de forma mais sistemática e automatizada possível. O trabalho manual de se testar afeta diretamente a complexidade de realizar os testes, além de aumentar a chance de introduzir falhas na própria atividade de teste, conforme mostrado em [2].

Existem algumas definições na literatura para o conceito de teste de software. Segundo [2], um teste de software nada mais é do que verificar se o comportamento de um programa está de acordo com o esperado, com base em um conjunto de entradas específicos. Esses testes podem ser classificados de diferentes maneiras. Algumas dessas maneiras de se classificar os testes serão tema dos estudos propostos neste trabalho, como por exemplo testes funcionais, testes de comportamento e testes de mutação, contextualizados no Capítulo 2.

Tais conceitos de teste de software se fazem cada vez mais utilizados na atualidade, uma vez que as metodologias ágeis vem ganhando bastante popularidade no desenvolvimento de software. As metodologias ágeis são conceitos que visam tornar o desenvolvimento de software mais rápido [3]. Esses conceitos envolvem como iterações em um período de tempo determinado da equipe de desenvolvimento. Em geral, analisa-se o que será desenvolvido e o que foi desenvolvido em cada iteração, e a todo instante são feitas validações de qualidade e aceitabilidade do software.

Existem vários tipos de metodologias ágeis, sendo uma bem conhecida o Scrum [4]. O Scrum implementa o conceito de iterações a partir das chamadas sprints, que são

períodos determinados de tempo (em semanas). Em cada sprint, é planejado o que vai ser desenvolvido, é feito o desenvolvimento do que foi planejado, há uma certificação de qualidade no que foi desenvolvido e por fim há uma validação com o cliente referente a entrega.

Devido à sua natureza ágil, tais metodologias necessitam da contínua entrega, junto com o produto, de testes de software, para assegurar a qualidade a cada entrega. Esses testes são importantes para que seja possível escalar o produto final sem comprometer as entregas anteriores. Sabemos porém [2] que fazer a testagem de software pode ser custosa. Para contornar esse custo, técnicas de testagem de software são cada vez mais necessárias para se assegurar a qualidade do software em tempo hábil.

Para um tema tão relevante como o de geração de testes e cobertura de código como proposto neste trabalho, bases sólidas de conhecimentos foram pesquisadas e utilizadas. Como a proposta é gerar testes funcionais a partir de testes de comportamento, conceitos relativos à metaprogramação e à orientação à aspectos foram utilizados. No que tange ao primeiro conceito, estes consistem em fazer código que consiga gerar mais código. No que tange ao segundo conceito, este consiste em conseguir adicionar comportamentos a cada parte do sistema de forma individual. Dessa forma, por meio da aplicação desses dois conceitos (que serão mais contextualizados no Capítulo 2) foi possível mapear e instrumentar cada função acessada pelas funcionalidades, bem como seus parâmetros e retornos. A partir disto, se torna possível a geração de testes funcionais.

1.1 Problema de pesquisa

Apesar da importância de se realizar a atividade de teste de software, ela é uma atividade inerentemente desafiadora por dois principais motivos: o esforço em se produzir uma classe abrangente e satisfatória de testes de software e a importância de se garantir que os principais requisitos de um software estão devidamente cobertos por casos de testes.

Mesmo com tais desafios, testes são necessários para produtividade de uma equipe no longo prazo. Eles permitem que se aumente o sistema com uma confiabilidade de que os casos de uso do sistema se mantenham funcionais [5]. Com as ideias de desenvolvimento de software se tornando cada vez mais ágeis, a agilidade na produção de testes de software para o sistema também se faz necessária.

Visando a aumentar a otimização na geração de casos de teste, um dos principais problemas de pesquisa é justamente observar se é possível criar uma ferramenta para se gerar testes funcionais relevantes de forma automática a partir de cenários BDD. O BDD (Behavior Driven Development) é um técnica de desenvolvimento que consiste em fazer testes de comportamento previamente à implementação do código de uma funcionalidade

[6]. Cenários BDD nesse caso são os casos de uso que o teste de comportamento abrange. Tais conceitos são mais aprofundados no Capítulo 2.

Os testes funcionais, por um lado, são importantes para se garantir que o comportamento em seus níveis mais fundamentais estejam funcionando da maneira esperada. Eles testam as funções do sistema de forma modularizada. Por outro lado, torna-se importante saber quais requisitos são relevantes e que estão sendo exercitados a partir da especificação do comportamento do sistema (cenários BDD). Postulamos que a integração entre essas duas perspectivas de teste de forma automatiza poderá contribuir efetivamente em uma maior produtividade de se desenvolver um software, a medida que facilita o processo de se criar testes, conforme citado também em [5].

1.2 Objetivo geral

O objetivo principal desse trabalho é propor um arcabouço para mapear o comportamento do sistema de software especificado por meio da abordagem BDD e, a partir daí, gerar testes funcionais de forma automática.

A abordagem está baseada em três conceitos principais: (1) expressões regulares por meio das quais os cenários BDD são implementados; (2) programação orientada a aspectos para viabilizar a instrumentação do software e registro automatizado das execuções do software a partir dos cenários BDD executados; (3) metaprogramação para se extrair meta-informações sobre as classes, os métodos e seus respectivos parâmetros e variáveis que são invocados para gerar os testes funcionais.

1.3 Objetivos específicos

Este trabalho tem também como objetivo prover um arcabouço por meio do qual evidenciamos a viabilidade do funcionamento do processo na prática. Para isso, faz-se uso de métricas de cobertura de teste e de relevância dos casos de testes funcionais a partir de requisitos. Dessa forma, os objetivos específicos são os seguintes:

1. Explorar conceitos sólidos de linguagens de programação como uso de expressões regulares, orientação a aspecto e de metaprogramação no contexto de geração de casos de teste.

2. Implementar um arcabouço para geração automática de testes funcionais a partir de cenários BDD.

3. Avaliar a qualidade dos testes gerados por meio da nossa abordagem com estudos de caso.

Dados os objetivos gerais e específicos, este trabalho destaca, portanto, as seguintes contribuições: (1) Um processo para geração de testes funcionais a partir de cenários BDD. (2) Um framework que implementa o processo, para fins de amplo uso em projetos Ruby on Rails; (3) Estudos de caso em projetos open-source para validação da ferramenta em projetos reais que estão em uso e disponibilizados no Github.

Além disso, validou-se a qualidade dos testes gerados pelo framework com base nos testes de mutação para verificar a consistência e completude dos testes. Segundo Just et al., [7] a detecção de mutantes tem uma significativa correlação com a detecção de falhas num software, a partir da análise estatística feita pelos autores em diversos repositórios. O estudo mostra que 73% das falhas no software (das quais os autores já tem ciência previamente) tem mutantes as apontando. Isso mostra que esse indicador consegue explicitar as falhas reais no sistema, mesmo a partir de simples mutações.

1.4 Justificativa

A importância de se testar os softwares já é um conceito consolidado, e fazer isso de forma replicável e automatizada é essencial para garantir a escalabilidade de um sistema [1].

A hipótese de que é possível gerar testes funcionais relevantes a partir de cenários BDD se justifica devido às diferentes naturezas dos dois conceitos. Enquanto cenários BDD são especificados em linguagens próximas as linguagens naturais e tendem a ditar em mais alto nível qual será o comportamento da aplicação, testes funcionais são especificados em linguagens mais próximas as próprias linguagens de programação para os quais são escritos. Testes funcionais asseguram o comportamento das partes do sistema, que podem ser mapeados a partir das especificações dos cenários BDD.

A validação de tal hipótese vem por meio de critérios como o de cobertura de código dos testes funcionais gerados e da relevância dos mesmos, metrificados a partir dos testes de mutação. O critério de validação da ferramenta a partir da cobertura de código é justificado a partir da ideia de que a cobertura de código por parte dos testes consegue elucidar aos desenvolvedores onde ainda há necessidade de se testar um código. Ou seja, partes do sistema onde os casos de uso ainda não foram testados [8].

Além disso, sabemos que a cobertura de testes é uma métrica objetiva e quantificável, e que é de fácil obtenção na maioria dos softwares desenvolvidos. Diversas ferramentas de automação para obtenção dessa métrica existem nas linguagens e frameworks utilizados atualmente. No entanto, a literatura em teste de software reporta que apenas critérios de cobertura de testes podem ter distorções [8]. Por exemplo, uma cobertura de 100% do software não garante que todos os casos de uso tenham sido verificados de modo

confiável. Para isso, torna-se relevante obter-se meios de ter um controle sistemático do teste de software.

Tais conceitos como cobertura de código, testes de mutação e outros conceitos são mostrados no capítulo 2, como base da fundamentação teórica presente neste trabalho. Conceitos estes que foram utilizados para o desenvolvimento do processo de criação de testes apresentada e para a criação da ferramenta utilizada para validação de toda ideia presente nesta pesquisa.

1.5 Organização do trabalho

Os demais capítulos desta monografia estão organizados da seguinte forma:

No Capítulo 2 abordamos toda fundamentação teórica relativa aos conceitos utilizados neste trabalho. As bases construídas neste capítulo são utilizadas para a criação do processo e da ferramenta de automação da geração dos testes funcionais.

No Capítulo 3 fazemos uma revisão da literatura, mostrando alguns conceitos e ideias similares a ideia deste trabalho, e que são utilizados como referência.

No Capítulo 4 apresentamos o processo para criação da ferramenta. Dividimos o processo em etapas e apresentamos um diagrama que o explica.

No Capítulo 5 mostramos o desenvolvimento da ferramenta proposta na linguagem Ruby. A ferramenta foi desenvolvida a partir do processo proposto no Capítulo 4. As ferramentas utilizadas, como os frameworks de teste e o framework Ruby On Rails, são contextualizadas neste capítulo.

No Capítulo 6 mostramos a validação da ferramenta a partir dos estudos de caso. Mostramos aqui as métricas obtidas com a ferramenta nas questões de cobertura de código e relativas aos testes de mutação. Uma discussão acerca dos resultados também é apresentada neste capítulo.

Por fim, no Capítulo 7 fazemos uma conclusão do trabalho.

Capítulo 2

Fundamentação Teórica

2.1 Aspect Oriented Programming

Aspect Oriented Programming (AOP) [9] é um paradigma de programação que aumenta a modularidade com base nos aspectos. Aspectos são funções que tiram a responsabilidade de uma determinada classe e pode ser acoplada em diversas partes do código. Um exemplo onde o uso de um aspecto é ideal são os logs de sistema, onde a responsabilidade de geração de outputs dos métodos pode ser compreendida como além das responsabilidades das classes monitoradas.

Esse paradigma de programação veio num contexto de muita popularidade da programação orientada a objetos (OOP). Esse paradigma ficou muito popular devido a resolver vários problemas de modelagem de dados e de arquiteturas de software em geral, permitindo maior qualidade e agilidade no desenvolvimento de grandes sistemas[10]. Apesar de bastante útil, esse paradigma possui algumas limitações, e é justamente por algumas dessas limitações que surge a programação orientada a aspectos (AOP).

Um dos problemas que a programação orientada a objetos não consegue resolver de forma tão simples é a modularização de responsabilidades ao invés de modularizar estruturas. Por exemplo, em uma modelagem de orientação a objetos, modelar uma classe que represente uma entidade de uma pessoa pode ser feito de forma direta. Porém, não é tão simples modelar uma ideia de um módulo de logs para o sistema de software, onde são mapeados todos os outputs das funções que estão sendo executadas para o sistema. Na orientação a aspectos tal ideia decorre de forma automática por meio dos aspectos.

Os aspectos são módulos que possuem funções que realizam determinado comportamento e podem ser acopladas a cada parte do sistema. Um exemplo disso é justamente um módulo de logs, que pode ser adicionado a todos os módulos de um sistema (inclusive classes numa modelagem de orientação a objetos). Eles realizam um determinado comportamento necessário e compartilhado entre todos esses módulos. Assim, os Aspec-

tos observados podem ser utilizados para registar quais os métodos executados ao serem invocados.

2.2 Ruby e a metaprogramação

Ruby [11] é uma linguagem de programação de alto nível interpretada dinâmica e inteiramente orientada a objetos. Pode apresentar diversos paradigmas de programação, como funcional, orientada a objetos e até mesmo orientada a aspectos. O paradigma de aspectos possibilitado no Ruby se torna especialmente interessante para o projeto, pois dessa forma conseguimos mapear os métodos invocados durante a execução da suíte de testes de comportamento.

Outra possibilidade interessante que o Ruby permite é seu aspecto de metaprogramação, uma das principais possibilidades da linguagem. Metaprogramação consiste em produzir código que pode operar sobre outros códigos, ou, nesse caso, sobre o próprio código [12]. Outro aspecto importante da metaprogramação é o seu uso em sistema complexos e presentes no dia a dia para qualquer programa funcionar, como sistemas operacionais, compiladores, assemblers, linkers, etc. Essa ideia se tornou muito importante para o surgimento de softwares cada vez mais complexos.

Um exemplo de metaprogramação abordado neste trabalho é a gema feita, que por meio da execução de testes de comportamento, gera outros testes (testes funcionais). Essa ideia é justamente a ideia de um código gerar mais código.

2.3 Testes de comportamento

Testes de comportamento, conhecidos em inglês por *Behavior Tests*, são testes que, conforme o próprio nome já propõe, testam o comportamento de uma aplicação. Com a popularização da metodologias ágeis, esse tipo de teste começou a ser cada vez mais necessário [6]. Em geral estão associados aos testes de aceitação do sistema, uma vez que é necessário garantir um determinado comportamento para se obter a aceitação de uma funcionalidade. Muitas vezes são utilizados em uma metodologia de desenvolvimento de software chamada *Behavior Driven Development (BDD)*. Tal metodologia propõe a criação de testes de cenários (cenários BDD) para cada funcionalidade do sistema, para que seu comportamento seja testado.

Testes de aceitação envolvem geralmente a utilização de uma aplicação por seu usuário final, onde certos requisitos do sistema são exercitados. Se esses requisitos são atendidos, então as funcionalidades estão aceitas. Porém, testar softwares de forma manual com um

usuário final não é prático e repetível em larga escala [2], daí a necessidade de se criar os testes de comportamento.

Testes de comportamento visam simular um comportamento do usuário de forma automatizada. Como propõe Dan North [13], os testes de comportamento podem ser vistos como cenários BDD. Estes contribuem para que a equipe de desenvolvimento possa focar os esforços na identificação e implementação das funcionalidades do software [14], garantindo uma maneira eficiente de se checar se o sistema faz o que a sua especificação diz. Por este motivo, utilizamos os testes comportamento como documento base para a geração dos testes funcionais do sistema.

Os testes de comportamento geralmente podem ser escritos em linguagens naturais (ou próximas das naturais) e automatizados com linguagens de programação. Um exemplo disso é o seguinte trecho de código:

```
@posts
Funcionalidade: visualizar postagens
  Como leitor
  Eu quero visualizar as postagens do blog
  Para que eu possa me manter atualizado

  #index
  Cenario: listar todos as postagens
    Dado que existam 2 postagens criadas
    Dado que eu esteja na pagina inicial
    Quando eu clicar no link para "postagens"
    Entao eu devo ver a lista de postagens com 2 itens
```

Listing 2.1: Exemplo de testes de comportamento em Gherkin

No cenário exemplificado acima, foi utilizada a linguagem Gherkin, que tem palavras reservadas como "Dado", "Quando", "Entao". Quando combinadas com o português dão uma legibilidade alta para o que o teste quer dizer em linguagem natural. O seu entendimento se dá até por pessoas que não possuem muito conhecimento em testes ou de programação. O teste escrito em linguagem natural traz uma facilidade no momento de saber quais são os testes de aceitação a serem realizados (guiados pelo comportamento da aplicação).

Para automatizar esses testes, pode-se usar expressões regulares de uma linguagem de programação, como no exemplo a seguir que automatiza o código em Gherkin exemplificado anteriormente:

```
#index steps
Dado("que existam {int} postagens criadas") do |number_of_posts|
  number_of_posts.times do |i|
    Post.create(
```

```
5         title: "Post #{i}",
6         description: "Esse e o post #{i}"
7     )
8   end
9 end
10
11 Dado("que eu esteja na pagina inicial") do
12   visit root_path
13 end
14
15 Quando("eu clicar no link para {string}") do |link_name|
16   click_link link_name
17 end
18
19 Entao("eu devo ver a lista de postagens com {int} itens") do
20 |number_of_posts|
21   number_of_posts.times do |i|
22     expect(page).to have_content "POST #{i}"
23   end
24 end
```

Listing 2.2: Exemplo de teste de comportamento implementado em Ruby

No exemplo conseguimos ver a implementação das especificações na linguagem Ruby. Podemos ver que mesmo na implementação no nível da linguagem de programação, a ideia dos comportamentos e da linguagem mais próxima à natural se mantém. Isto se observa na especificação de cada trecho do teste, que continua utilizando as palavras reservadas Dado, Quando e Então.

Os testes de comportamento facilitam a maior confiabilidade no que está sendo produzido. Apenas uma cobertura alta de testes pode não ser suficiente e dar uma falsa impressão de que seu sistema está funcionando perfeitamente. Porém, testar de forma automática o comportamento garante também que faz sentido tudo que está sendo produzido uma vez que são os requisitos especificados pelo usuário que estão sendo rastreados e testados [15].

2.4 Testes funcionais

Testes Funcionais são técnicas onde o programa a ser testado é considerado uma caixa-preta (não se sabe o conteúdo do código fonte do mesmo, apenas o que recebe como entrada e a saída esperada) [2]. O teste funcional envolve uma chamada de uma função da aplicação, passando parâmetros (válidos ou inválidos, a depender de que resultado se espera do teste) e verificando a resposta da função.

Em geral esse tipo de teste tem como principal importância a correta utilização dos parâmetros de entrada, para se garantir uma maior cobertura de possíveis situações que o teste prevê. Por exemplo, numa função de soma de inteiros, é importante verificar se ele funciona corretamente para duas entradas positivas, bem como para duas entradas negativas. Em particular, esse conceito pode ser bem analisado pelas classes de equivalência citadas acima.

Esses testes são denominados de funcionais por que em geral são testes caixa-preta que testam funções (ou métodos). Recebem como entrada os parâmetros da função ou do método e testam se as saídas estão de acordo com o esperado. Esse tipo de teste tem um custo considerado baixo para ser escrito, devido à testar mais separadamente as partes de um sistema, permitindo escrever testes mais sucintos e mais específicos [2]. Em compensação o ganho por se ter esse tipo de testes num sistema é alto, uma vez que ele garante que o sistema em seus níveis mais fundamentais esteja validado e funcionando da maneira que deveria.

São um dos tipos de testes mais comuns nas aplicações, sendo o tipo de garantia de qualidade de software mais executado [16]. Funcionam como uma espécie de "equipamento de proteção"do código, visto que eles tendem a dizer se as funções da aplicação estão a todo momento fazendo o que elas deveriam de fato fazer. Permitem a inserção de mais códigos sem que comprometa o correto funcionamento da aplicação.

2.5 Testes de mutação

Outro conceito de testes bastante utilizado são os testes de mutação. Um teste de mutação é uma técnica utilizada para aferir a relevância dos testes feitos [2].

A técnica de teste de mutação envolve modificar o programa que está sendo testado diversas vezes, criando um conjunto de programas alternativos (mutantes) [2]. Para cada um desses programas alternativos, é necessário analisar quais desses programas alternativos tem o comportamento esperado diferente do programa original. Para tais programas, rodam-se a suíte de testes, e como o comportamento do programa foi modificado, é esperado que esses testes apresentem falhas. Quando os testes rodados nos programas alternativos apresentam falhas (indicando que o programa está com comportamento diferente do esperado) é entendido que um mutante foi morto, ou seja, aquela versão do programa não passou pelos testes feitos previamente na aplicação. Isso indica que a suíte de testes está de fato testando o correto comportamento do sistema.

É uma técnica de testagem considerada bem custosa em termos de tempo de desenvolvimento, como explicitado em [17], já que necessita de uma mudança nas partes do código fonte a ser testado, de forma a preservar o código original.

Por exemplo, suponha que em um sistema exista função que checa se um usuário é maior de idade (tem mais que 18 anos), conforme o seguinte pseudo-código:

```
1    bool checkUserAge(user){
2        return user.age >= 18
3    }
```

Listing 2.3: Exemplo de função para checar idade do usuário

E o seguinte teste que testa o correto funcionamento dessa função:

```
1    test('#checkUserAge'){
2        valid_user = new User(age: 21)
3        invalid_user = new User(age: 16)
4
5        expect(checkUserAge(valid_user)).to eq true
6        expect(checkUserAge(invalid_user)).to eq false
7    }
```

Listing 2.4: Exemplo de teste para a função de checar idade do usuário

Para aplicar os conceitos de teste de mutação, que são usados para testar o teste, precisamos fazer modificações no programa original (neste caso, na função `checkUserAge`). Por exemplo, trocar determinados operadores ou condicionais, para então verificar se o teste associado à ela vai indicar algum erro.

Neste caso, uma mudança seria trocar a parte que se compara user.age >= 18 por um operador de menor que (user.age < 18). Neste caso, é esperado que o teste associado não esteja válido. Caso isso ocorra, significa que um mutante foi morto (um teste falhou e identificou um comportamento irregular no sistema). Como essas mudanças no código fonte são injetadas propositalmente para serem erradas, é esperado que os testes associados em sua maioria não sejam mais válidos para aquela mudança (mutação).

A automatização da criação de testes de mutação consiste em uma análise de trechos dos códigos do sistema, de modo a identificar operações aritméticas, comparações, entre outras operações. Essa automatização tem como objetivo mudar semanticamente essas partes do código [17]. Por exemplo, pode-se utilizar tal técnica para uma operação aritmética de soma ser trocada por uma subtração, e com isso gerar uma mutação no código fonte.

Essa automatização se faz útil para agilizar o processo de se utilizar testes de mutação.

Pode-se dizer que testes de mutação são uma boa boa alternativa para se "testar testes", já que ele tem como objetivo aferir se os testes conseguem identificar o correto funcionamento do sistema. Este critério inclusive será utilizado nesta pesquisa para validação da ferramenta criada.

2.6 Cobertura de código

Cobertura de Código é um conceito que visa quantificar o grau de abrangência dos testes em relação aos códigos que eles testam. Ou seja, a quantidade de código do sistema que tem um teste associado a ele [18].

Esse critério é muito utilizado em projetos de software devido a existirem várias ferramentas para diversas linguagem atualmente que conseguem trazer essa métrica de cobertura de código, seja por linhas cobertas, seja por blocos, qualquer critério [18].

Um dos principais objetivos quando se faz testes é verificar se cada trecho de código a ser testado tem seu comportamento mantido ao longo do projeto. Uma das formas de se verificar isso é saber se cada trecho de código tem um teste associado [19]. Por exemplo, numa branch condicional verificar se o bloco de if e o else tem testes associados. Nesse caso, o objetivo de se verificar cada parte do teste é explicitado, e essa verificação é possível de ser feita justamente com ferramentas de cobertura de código.

Muitas vezes há partes de um software que devem ter maior prioridade e que possivelmente requerem mais atenção num sistema. Com critérios de cobertura de código, é possível verificar o quanto de testes existem para cada parte do seu sistema, tornando possível priorizar esforços para se criar mais testes para partes mais cruciais do sistema [5].

Algumas classificações formais para diferentes rigores que uma cobertura de código pode ter são definidas em [20], sendo representada pelos seguintes questionamentos quando olhamos para os testes feitos no sistema:

1. Cobertura a nível de Métodos: Cada método é executado pelo menos uma vez pelo conjunto de testes?
2. Cobertura a nível de Chamadas ou Cobertura de Entrada/Saída: Cada método foi chamado de todos os locais onde poderia ser chamado?
3. Cobertura a nível de Declarações: Cada declaração do código é executada pelo menos uma vez pelo teste de um condicional como uma única declaração?
4. Cobertura a nível de ramos: Cada ramo foi tomado em cada direção pelo menos uma vez?
5. Cobertura a nível de Caminho: Foram executados todos os percursos possíveis através do código?

O conceito de cobertura a nível de métodos será utilizado neste trabalho para verificar o grau de abrangência dos testes. O trabalho visa abranger todas as funções e partes de código que os testes de comportamento testam.

Além disso, como um dos conceitos abordados é o monitoramento de funções por meio dos testes de comportamento, é possível a partir do mapeamento pelo módulo de aspectos (discutido no Capítulo 4) obter uma cobertura a nível de ramos também com o método. Para isso, os testes de comportamento precisam abordar todas as possibilidades de casos no sistema.

Capítulo 3

Trabalhos Relacionados

O tema de automatizar o processo de fazer testes já é discutido à algum tempo e estudado por alguns autores. Alguns casos como os de [21] mostram que casos de uso podem ser levados em consideração quando se tem o objetivo de gerar testes de forma automática.

Isso já é feito pelos engenheiros de software, cujo papel é projetar o funcionamento da aplicação, tendo como base o comportamento esperado e com isso gerar testes que descrevam tais comportamentos. Porém, como citado no artigo, a distância entre mapear um comportamento da aplicação e gerar testes relevantes para a mesma de forma automática pode ser grande. Porém, se feita, pode otimizar muito a qualidade de um software e a produtividade de uma equipe, por isso a importância de seu estudo.

3.1 Alguns métodos

3.1.1 Geração de testes baseada em modelagem UML

Um dos métodos citados em [21] é o de utilizar a Unified Modeling Language (UML). A UML é uma linguagem que possui padrões para definir a modelagem de um sistema. A linguagem se utiliza de palavras chave e uma representação por diagramas para facilitar a transformação destes requisitos em códigos de programação (como uma representação em grafos). Isso permite mapear melhor os casos de uso de um sistema e extrair as informações mais relevantes para gerar os testes de forma automática.

A partir desses UMLs são gerados Objetos de Teste, que são sentenças lógicas que traduzem o comportamento de uma determinada parte do diagrama UML. Com base nesses Objetos de Teste e na ferramenta desenvolvida por [21] se tem a implementação dos testes.

Essa ideia da geração de testes a partir de casos de uso é algo que se assemelha a ideia deste trabalho. Nesse caso se utilizam diagramas UMLs e sentenças lógicas como base da

geração de testes. No caso desta pesquisa utiliza-se testes de comportamento, que já nos dão esse comportamento da aplicação.

3.1.2 Geração de testes baseada em máquinas de estados

Outro método encontrado na literatura é a utilização de máquinas de estado (autômatos) [22]. Transforma-se uma representação UML numa máquina de estados especializada, onde é possível saber as transições que ela possui e, por consequência, o comportamento geral da aplicação.

Com base nessa ideia, os autores propõem construir uma máquina de estados para cada funcionalidade do sistema. Com base em cada uma dessas máquinas é proposto uma ferramenta desenvolvida pelos autores (ParTeg). Essa ferramenta tem como entrada essas máquinas de estado codificadas e dá como saída testes unitários na linguagem Java, utilizando o framework JUnit.

A ideia por trás dessa ferramenta é testar as funções que são representadas como arestas das máquinas de estado (levando a aplicação de um estado pra outra e gerando saídas que podem ser identificados pela ferramenta). Esse funcionamento é similar a ideia proposta neste trabalho no que tange o mapeamento por meio de aspectos das funções de cada classe do sistema.

3.1.3 Geração de testes baseada em requisitos

Segundo [23], modelos de sistemas definidos utilizando máquinas de estados finitos tem que ser manualmente criados e mantidos por designers de testes de alta habilidade. Dessa forma, os modelos criados manualmente não são frequentemente precisos e não refletem o sistema de software por conta de erros humanos no processo. A automações completa das atividades relacionadas ao teste, desde a especificação do software até a execução dos testes é desejada, podendo reduzir muito o custo do teste. [23]

A premissa dessa abordagem, de geração de testes baseada em requisitos da especificação do software, é utilizar a linguagem de especificação formal SDL - *Specification and Description Language*. Esta é aceita como uma forma de se especificar o comportamento do sistema [23], converter essa especificação para modelos de máquinas de estados finitos e utilizar esse modelo como entrada no gerador de testes de caixa preta.

O gerador de testes de caixa preta pode utilizar várias estratégias já conhecidas como cobertura dos estados, das transições, do caminho, etc. Dependendo da estratégia escolhida, o gerador gera um conjunto de caminhos que, um a um, são mapeados em cada requisito do sistema. Dessa forma, um conjunto de requisitos da especificação do software é associada a cada caso de teste.

3.2 Frameworks para geração de testes

Nesta seção, apresentaremos dois frameworks para exemplificar duas abordagens de como testes podem ser gerados tanto de um nível de abstração mais baixo, como é o caso do KLOVER[24], como de uma abstração mais alta, como o [25].

3.2.1 KLOVER

Uma ferramenta proposta em [24] é o KLOVER. Esta é uma ferramenta para geração de testes de forma automática para a linguagem C++. A ferramenta utiliza-se do programa fonte transformado em bytecode para executar o programa. A partir disso obtém-se os testes para o programa, numa aproximação de modo a criar uma máquina virtual própria para execução dos códigos C++.

A ideia de execução do programa para obtenção dos testes foi um design utilizado na criação da solução proposta nesse trabalho.

A validação dessa solução se deu por meio do testes da ferramente em projetos disponíveis no site sourceforge. A ferramenta foi testada em alguns repositórios e foi constatado um aumento significativo da cobertura dos testes unitários para os programas em C++. Esses aumentos chegaram a ser de mais de 40% de cobertura em alguns arquivos.

Esse método de validação se mostra eficiente e foi utilizado também na proposta de validação da solução proposta em nossa pesquisa. Critérios de cobertura de código são amplamente utilizados para se determinar a qualidade de um software [26].

3.2.2 GUITAR

O GUITAR [25] é um framework inovador e flexível para o desenvolvimento de ferramentas de testes para GUI - *Graphical User Interface*. Ele suporta o desenvolvimento de ferramentas customizadas para a tarefa de se testar software. O design e a implementação do GUITAR se baseia nos seguintes aspectos:

- Automação: o GUITAR automatiza atividades chaves do teste de GUI enquanto se adapta bem à ferramentas de terceiros.
- Reuso: o código núcleo do GUITAR não é limitado a uma aplicação, plataforma ou técnica, mas maximiza o uso independente dos aspectos dos algoritmos.
- Baseado em modelos: o GUITAR é nativa, mas não exclusivamente baseado em modelos.
- Modularidade: a arquitetura do GUITAR é baseada em plugins.

O framework GUITAR perite o desenvolvimento de quatro ferramentas principais. Elas são:

- *Ripper*: uma ferramenta para a geração de modelos estruturais da Interface Gráfica de Usuário de uma aplicação sob teste.
- *Graph Converter*: uma ferramenta para converter o modelo estrutural, gerado pelo *Ripper* em um grafo, conhecido como *Event-Flow Graph*.
- *Test Case Generator*: uma ferramenta para a geração automatizada de casos de testes baseada no grafo gerado pelo *Graph Converter*
- *Replayer*: uma ferramenta para execução automatizada dos casos de testes gerados pelo *Test Case Generator*.

O módulo *Test Case Generator* automaticamente gera os casos de teste baseado no grafo gerado pelo *Graph Converter*. Embora o testador seja livre para implementar qualquer algoritmo para a geração dos testes, o GUITAR provê suporte para, utilizar o grafo como entrada e executar algoritmos de travessia em grafos para automaticamente gerar os casos de teste.

Capítulo 4

Visão Geral do Processo

Neste capítulo apresentamos os conceitos estudados condensados em um processo para se criar testes funcionais a partir de testes de comportamento.

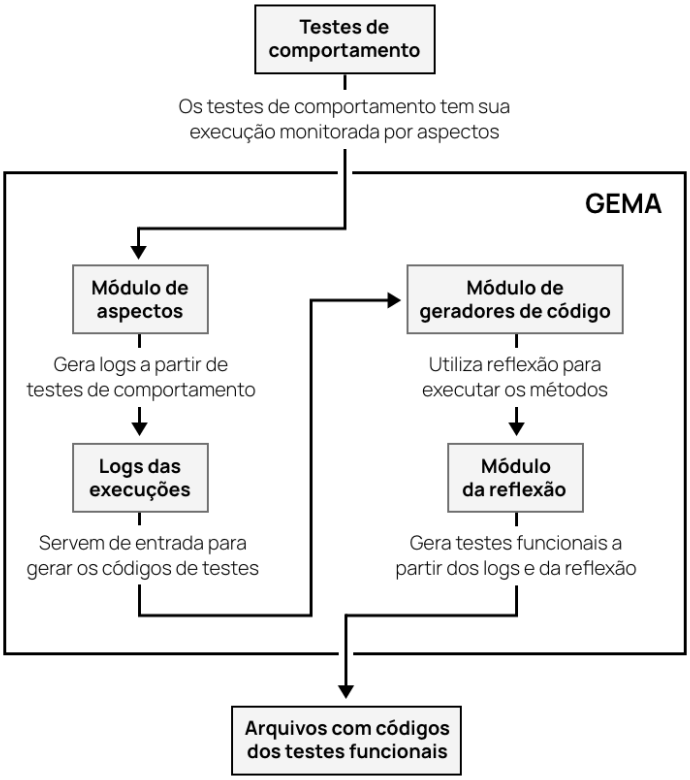

Figura 4.1: Visão dinâmica sobre a relação entre os módulos conceituais do processo

Na Figura 4.1 provemos uma perspectiva de como os principais módulos conceituais da nossa ferramenta se relacionam para a geração dos testes funcionais a partir dos testes de comportamento. Inicialmente, conforme a Figura 4.1 ilustra, os cenários BDDs são as entradas para o módulo de aspectos, onde os métodos executados nos cenários são

monitorados. Esses cenários mapeiam o comportamento da aplicação em uma camada de abstração de mais alto nível.

Para ser possível interceptar as entidades do software (e.g. classes, métodos, parâmetros, etc) que são chamadas na execução de cada cenário BDD, utilizamos os recursos viabilizados por aspectos. Eles conseguem instrumentar cada função chamada pelo teste de comportamento para então gerar logs com as informações sobre as entidades executadas e que serão posteriormente utilizadas como informação para a geração dos casos de testes funcionais.

A partir das informações registradas nos logs, faz-se então a chamada para os módulos geradores de código para produzir os testes funcionais a partir da gramática de testes funcionais. A reflexão completa então a última parte do teste funcional, que é a saída esperada para as funções exercitadas a partir dos cenários BDDs e seus respectivos parâmetros. A partir desse módulo obtém-se finalmente a saída com os testes funcionais.

4.1 Etapas

A premissa inicial do processo é a de que existam testes de comportamento no formato de cenários (features) na linguagem Gherkin previamente especificados no sistema de software de interesse. Em muitas linguagens de programação, há um framework de testes de comportamento chamado Cucumber [27], que permite essa automatização dos testes de comportamento. Essa ferramenta está disponível para linguagens como Java, Ruby, JavaScript, PHP, dentre outras.

4.1.1 Monitoramento dos testes de comportamento

A primeira etapa deste processo se dá pelo monitoramento dos testes de comportamento. Esse monitoramento pode ser entendido como verificar, a cada execução dos testes de comportamento, os métodos, classes e funções aos quais aquele comportamento se utiliza. Por exemplo, em um teste relacionado ao comportamento de um usuário se autenticar num sistema, é de se esperar que ele acesse funções para busca do email num banco de dados, que ele se utilize de uma função de verificação se a senha está correta, e assim por diante.

A ideia desta etapa é identificar, a partir do teste de comportamento, todas as entidades do sistema que são acessadas. Para esse monitoramento, utiliza-se o conceito de Aspectos, citados anteriormente no Capítulo 2. Para todas as entidades invocadas pelos cenários BDD, cria-se um sistema de monitoramento que indique quando um método é chamado. O conceito de Aspectos apresenta-se bastante adequado para esse requisito no sistema.

Para cada teste de comportamento tem-se funções gerais do sistema sendo chamadas para atender o requisito do mesmo. Cada uma dessas funções pode ser monitorada por meio de uma injeção de um aspecto, ou seja, adicionar na função que ela deve, por exemplo, gerar um log com seu nome e seus parâmetros passados. Dessa forma, tem-se um relacionamento direto entre os testes de comportamento e os aspectos a serem inseridos, de maneira que, as funções utilizadas nos testes de comportamento serão justamente onde os aspectos deverão ser inseridos.

Esse monitoramento dos testes de comportamento então, se dá pela injeção dos aspectos nos testes de comportamento. Em uma orientação à objetos, isso poderia ser feito por meio da injeção de aspectos em cada classe do sistema, para mapear cada um de seus métodos.

Tal princípio é implementado para a linguagem de programação Ruby, e expressa no Capítulo 5. Para outras linguagens e ferramentas a ideia é análoga, precisando-se encontrar as ferramentas equivalentes da linguagem a ser trabalhada.

Partindo do exemplo de um comportamento de autenticação de usuário, podemos supor, para uma linguagem orientada a objetos, que as funções de verificação do email do usuário e verificação de senha são métodos em uma classe de usuário:

```
1  class Usuario
2  {
3    funcaoParaChecarEmail(argumentos){
4      ...
5    }
6    funcaoParaChecarSenha(argumentos){
7      ...
8    }
9  }
```

Listing 4.1: Exemplo de classe de usuário

```
1  class Usuario
2  {
3    funcaoParaChecarEmail(argumentos){
4      ...
5      GerarLog(argumentos)
6    }
7    funcaoParaChecarSenha(argumentos){
8      ...
9      GerarLog(argumentos)
10   }
11 }
```

Listing 4.2: Classe de usuário com aspectos injetados

Nesse exemplo, a função GerarLog é responsável por criar um arquivo com o nome da função chamada e os argumentos passados [1].

Vale ressaltar que é importante que se gere um arquivo com os Logs, uma vez que esse arquivo será necessário para etapas posteriores. Esse arquivo será uma forma adequada de se persistir informações relativas a quais partes do sistema (ou entidades) cada cenário BDD acessa. Esses arquivos podem ser separados de diferentes formas: por relacionamento com os testes que os geram; por classes que foram acessadas; etc. Um exemplo de arquivo com esses Logs pode ser visto a seguir, em pseudo-código:

```
1  {Classe: "Usuario", funcao_acessada: "funcaoParaChecarEmail",
    argumentos: ["email@exemplo.com"]},
2  {Classe: "Usuario", funcao_acessada: "funcaoParaChecarSenha",
    argumentos: ["123456"]}
```

Listing 4.3: Exemplo de arquivo de Logs gerados

Aqui, vale a observação de que tais arquivos de Logs poderiam ser gerados a partir do monitoramento de outras partes do sistema que não os testes de comportamento. Um exemplo disso seria o monitoramento pelo próprio uso do sistema por usuários. Porém, por critérios de replicabilidade conforme citados no Capítulo 2, utilizar os testes de comportamento pra gerar estes logs se torna mais replicável.

4.1.2 Reflexão dos métodos

As três partes principais dos arquivos de log são: nome da classe, nome da função acessada e os argumentos utilizados. Cada uma dessas partes tem sua relevância explicada a seguir.

A parte da classe será importante para saber corretamente qual a função a ser testada. Funções de mesmo nome serão diferenciadas pelo nome da classe, presente no arquivo de Logs.

A parte da função acessada dos logs será útil pois para escrevermos o teste precisamos justamente chamar essa função. A maneira de chamar ela no código do teste será pelo seu nome, conforme mostrado no próximo capítulo.

A parte dos argumentos será útil para chamarmos a função passando os argumentos corretos para ela. Para cada conjunto de argumentos diferentes serão criados diferentes casos de teste.

Uma vez gerados os arquivos de Log, precisa-se saber qual seria a saída para os testes a serem criados. Para isso, utiliza-se o conceito de reflexão. Para fazer as reflexões das funções e seus argumentos, podemos ter duas abordagens:

[1] Essa exata implementação para a linguagem Ruby é mostrada na Seção 5.3.2, podendo servir de inspiração para implementações em outras linguagens de programação.

A primeira abordagem consistem em utilizar diretamente os argumentos mapeados na primeira etapa para se fazer a reflexão. Ou seja, utilizar argumentos que já tenham sido utilizados nos testes de comportamento. No caso do exemplo na Listagem 4.3, seria utilizar o email email@exemplo.com para se fazer a reflexão na função `funcaoParaChecarEmail`.

Essa reflexão se dá a partir de uma instanciação da classe que se tenha a função que se quer saber a resposta. Por isso é importante que, em uma orientação a objetos, mapear a função a qual aquele método pertence. A partir dessa instanciação, pode-se chamar a função com os argumentos mapeados, para se obter o resultado esperado. Ilustramos essa ideia no seguinte pseudo-código:

```
...
classeDeUsuario = Usuario()
classDeUsuario.chamarFuncao(funcaoParaChecarEmail, "email@exemplo
  .com")
// deve retornar, por exemplo, o valor verdadeiro, caso o usuario
   exista no banco de dados
```

Listing 4.4: Exemplo de reflexão

Os resultados gerados nessa etapa de reflexão são adicionados ao arquivo de Logs. Essa nova parte do arquivo será utilizada, em conjunto com o nome da classe, nome da função e argumentos para gerar o teste funcional. Um exemplo de arquivo de Log com os resultados da reflexão pode ser visto a seguir:

```
{Classe: "Usuario", funcao_acessada: "funcaoParaChecarEmail",
  argumentos: ["email@exemplo.com"], reflexao: true},
{Classe: "Usuario", funcao_acessada: "funcaoParaChecarSenha",
  argumentos: ["123456"], reflexao: "Senha Incorreta."}
```

Listing 4.5: Exemplo de arquivo de Logs gerados

Na listagem 4.5 vemos que é esperado que o método `funcaoParaChecarEmail` retorne o valor verdadeiro para o argumento email@exemplo.com. Esse é o valor do campo de reflexão no arquivo de log.

4.1.3 Geração dos códigos de teste

A última etapa consiste na geração dos códigos de teste. Sabemos que um teste funcional testa se para determinadas entradas a saída esperada ocorre, geralmente testando-se funções. A ideia desta etapa é a partir dos Logs, montar a suíte de testes.

Para se gerar esses códigos é necessário definir uma gramática a ser utilizada como base para que os testes na linguagem específica sejam criados. Em geral, uma gramática para se fazer códigos de teste irá necessitar de suporte à chamadas de funções, chamadas à operadores de comparação (para testar se as saídas são as esperadas) e regras básicas

de uma gramática com operações aritméticas, operações lógicas, etc. Um exemplo de gramática que possui esses suportes pode ser vista a seguir:

```
1  S -> declarations
2  declarations -> declarations declaration
3  declaration -> describe <string> test_code | context test_code
4  test_code -> test_code operations | test_code expectations |
    operations | expectations
5  operations -> <operacoes aritm ticas e logicas>
6  expectations -> expect(expressao)."to" assert expressao
7  expressao -> numero | string | operations
8  assert -> equal | be_higher | be_lower
```

Listing 4.6: Parte de uma gramática com suporte a linguagem de testes

Essa gramática em pseudo código mostra uma parte de uma gramática de uma linguagem de programação que permite suporte para sintaxe de testes. O princípio consiste em utilizar funções chave como `describe` e `context` (linha 3) para descrever a função a ser testada. Palavras chave `expect` e `assert` (linha 6) são usadas para comparar 2 expressões, se são iguais, se são diferentes, se uma é maior que outra, etc. Dessa forma podemos montar nossa gama de testes.

Tendo em vista essa sintaxe, é necessário se obter a partir dos Logs a função a ser testada, os argumentos a serem utilizados no teste e a resposta esperada da função. Para isso, pode-se fazer um parser do arquivo de Logs. Vale lembrar que a formatação do arquivo de Logs já é definida previamente na primeira etapa, quando se cria esse arquivo. Uma formatação comum para esse tipo de arquivo é a de JSON [28]. Essa formatação é reconhecida pela maioria das linguagens de programação moderna. Isso facilita a criação de um *parser*. Tal formatação foi utilizada na ferramenta proposta.

```
1  teste para classe Usuario
2  {
3    teste para funcao funcaoParaChecarEmail
4    {
5      descricao do teste "a funcao retorna verdadeiro caso o
    usuario com este email exista"
6      {
7        espere(Usuario.funcaoParaChecarEmail("email@exemplo.com"))
    ser igual a verdadeiro
8      }
9    }
10 }
```

Listing 4.7: Exemplo de arquivo de teste gerado

Esta etapa pode ser considerada a mais simples de se implementar, já que os componentes a serem utilizados nela já foram obtidos em etapas anteriores. Geralmente, faz-se

uso de um modelo com o formato em que os testes devem ser gerados. Essa ideia é demonstrada no Capítulo 5 a seguir.

Capítulo 5

Arcabouço de Geração de Testes Funcionais

Neste capítulo apresentamos o arcabouço que implementa o processo descrito no capítulo anterior. São feitas as análises sobre os testes gerados e critérios previamente explicitados. A ideia geral do projeto foi desenvolver uma biblioteca que fosse facilmente acoplável a qualquer sistema desenvolvido em Ruby, como forma de evidenciar a viabilidade da abordagem proposta. Para tanto, foi desenvolvida uma gema (biblioteca para Ruby) naquela linguagem, dentro do framework Rails. A gema foi denominada Test Generator (TG). Nela se encontra o código todo necessário para fazer todas as etapas da geração dos testes.

Tal ferramenta pode ser encontrada no repositório no GitHub:

https://github.com/NickNish09/TestGenerator. Neste repositório também se encontram as instruções de instalação da mesma em um projeto Ruby on Rails.

Na tecnologia escolhida para a implementação do framework (Ruby on Rails), é seguido o modelo MVC (Model-View-Controller). Como citado em [29], o modelo MVC consiste basicamente na divisão de um software em três camadas, sendo o *M (Modelo)* a cama de modelagem dos dados, onde estão os dados utilizados no sistema em geral e as formas de consumir esses dados, como a conexão com um banco de dados e todas as regras de negócio. O *V (Visualização)* é a camada de visualização, que permite exibir esses dados na forma mais apropriada ao cliente (podendo ser, por exemplo, em uma aplicação web, uma página HTML), e por fim o *C (Controlador)*, a camada que integra as duas anteriores, onde são processadas as requisições de um cliente, realizadas chamadas à camada de Modelo e a resposta ao cliente se dá por meio da Visualização.

Como a lógica e e as regras de negócio da aplicação se concentram basicamente na camada de modelagem (os Modelos) [30], a gema monitora as chamadas a métodos das

classes representadas pelos Models. Portanto utiliza-se a ideia de Aspectos para se realizar esse monitoramento.

Aspectos possibilitam a injeção de métodos nas classes da camada de Modelo para mapear as chamadas de métodos pertencentes àquelas classes. Dessa forma, temos a possibilidade de chamar um método toda vez que qualquer outro método da classe for executado. Isso permite saber, de imediato, todos os métodos executados na aplicação a partir de um determinado comportamento do usuário. Por exemplo, se um usuário se cadastrar num sistema, os aspectos irão mapear todas os métodos que são chamados nesse comportamento. Estes métodos podem ser validação do email, funções para salvar no banco de dados os dados o usuário, etc.

A gema utiliza o conceito de reflexão, citada no tópico 4.1.2, após o mapeamento de uma chamada à método de algum Model da aplicação. Deste modo, sabem-se quais são os retornos dos métodos já mapeados, por meio do nome do método e argumentos recebidos na hora que o método foi chamado.

A partir do mapeamento dos métodos da parte lógica do sistema e da reflexão desses métodos, é possível obter todos os componentes necessários para a geração dos testes (o nome do método a ser testado, seus argumentos, e o seu retorno).A gema pode ser utilizada tanto a partir de cenários BDD, onde ocorrem chamadas aos métodos do comportamento a ser testado, quanto a partir da utilização do sistema por um usuário. Em ambos os casos, a ferramenta monitora os métodos chamados e já realiza a reflexão.

Na abordagem de mapear a utilização da aplicação pelo usuário, consideramos uma aplicação em produção que utiliza a gema. Nesse caso, quando o servidor da aplicação é colocado em execução, automaticamente os aspectos começam a mapear todas as chamadas à métodos das requisições que os usuários fazem. Isso gera um arquivo de logs, contendo a classe, o nome, os argumentos e retorno do méto, como exemplificado a seguir.


```
{
    "klass": "Post", "method": "upcase_title", "args": [], "attrs":
    {"id":117,"title":"Post 0","description":"Esse e o post 0 post
   0 post 0 post 0 post 0 post 0 post 0", "response": "\"POST 0\""
}

{
    "klass": "Post", "method": "truncated_desc", "args": [30], "
   attrs": {"id":117,"title":"Post 0","description":"Esse e o post
   0 post 0 post 0 post 0 post 0 post 0 post 0", "response": "\"
   Esse    o post 0 post 0 post...\""
}

{
```

```
10      "klass": "Post", "method": "create_date", "args": [], "attrs":
        {"id":117,"title":"Post 0","description":"Esse e o post 0 post 0
         post 0 post 0 post 0 post 0 post 0", "response": "\"14/10/20\""
11  }
12
13  {
14      "klass": "Post", "method": "upcase_title", "args": [], "attrs":
        {"id":118,"title":"Post 1","description":"Esse    o post 1 post
        1 post 1 post 1 post 1 post 1 post 1", "response": "\"POST 1\""
15  }
```

Listing 5.1: Exemplo de arquivo com Logs

5.1 Ferramentas utilizadas

Para a construção da gema algumas ferramentas foram necessárias, que são explicitadas a seguir.

5.1.1 Ruby on Rails

Ruby on Rails [31] é um framework que tem como base a linguagem de programação Ruby e utilizado para construção de sistemas web. O framework possui uma forte arquitetura MVC [29], o que permite a modularização das diversas partes do sistema.

Como o Ruby on Rails utiliza o modelo MVC, grande parte da lógica de funcionamento do sistema e das regras de negócio estão concentradas no Model *(M do MVC)*. O foco da ferramenta foi na geração de casos de testes para os Models dos sistemas testados. Isso permitiu uma maior geração de casos de teste e consequentemente uma maior cobertura de testes.

O framework também possui outras vantagens que auxiliaram na criação da gema, como os geradores de código [32]. Esses geradores de código permitem gerar códigos padronizados do framework (como um Modelo com seus atributos, por exemplo) através da linha de comando, e com base nesse módulo do Rails, um módulo da gema foi criado para ser o gerador de testes funcionais.

A ideia desse gerador é gerar os testes funcionais a partir dos arquivos de logs anteriormente gerados pelos aspectos (no padrão da ferramenta RSPEC, explicitada abaixo). Isso facilita o processo de automação da geração dos testes.

5.1.2 RSPEC

RSPEC [33] é um framework de testes para a linguagem Ruby. Essa ferramente apresenta uma sintaxe composta por palavras chaves como `context`, `it`, `should` para definir casos de teste para uma aplicação escrita em Ruby, além de operadores de comparação para verificar os resultados dos testes.

A gema fez uso do RSPEC para a geração dos testes funcionais, aproveitando-se da modularidade e organização de código dos arquivos de teste do framework. Cada classe da camada de Modelo possui um arquivo único de testes, desse modo a gema consegue agrupar os testes gerados para uma determinada classe em um único arquivo.

Um exemplo de arquivo de testes funcionais para a classe `Usuario` é demonstrado abaixo.

```
1 RSpec.describe User, :type => :model do
2   let(:user) { create(:user, {email: 'nnmarques97@gmail.com',
     password: '123456'}) }
3
4   it "is valid with valid attributes" do
5     expect(user).to be_valid
6   end
7 end
```

Listing 5.2: Exemplo da sintaxe do RSPEC

No exemplo acima há, na linha 1, uma declaração do tipo de teste para a classe da camada Modelo `User`. Em seguida, o método `let` define uma instância da classe `User`, que será utilizada como argumento no caso de teste descrito por `it` na linha 4. Neste caso de teste, é empregado o método `expect` que pode ser entendido como uma esperança de que ao se instanciar a classe `User` com os atributos da linha 2, esse objeto seja valido.

5.1.3 FactoryBot

FactoryBot é uma gema para auxílio na criação de instâncias de uma classe em Ruby. Ela foi utilizada para se definir instâncias dos Models que foram mapeadas pelos aspectos. A partir dos logs gerados na etapa de monitoramento, define-se uma fábrica com base nos atributos do logs (attrs).

Essa fábrica auxilia a melhor validação dos testes, uma vez que padroniza as entradas a serem utilizadas. Isso evita eventuais problemas de entradas irregulares, como por exemplo datas, horários, etc.

```
1 FactoryBot.define do
2   factory :post do
3     title { "Post 1" }
```

```
4     description { "Esse e o post 1 post 1 post 1 post 1 post 1 post
      1 post 1"}
5     created_at {12-12-2020 13:54:13 -03}
6   end
7 end
```

Listing 5.3: Exemplo de fábrica de postagens numa aplicação de Blog

A listagem 5.3 mostra uma factory de postagens, numa aplicação de postagens. Tais factories foram utilizadas na composição dos testes funcionais, criados para o framework de testes RSPEC. Sua utilização pode ser vista na listagem 5.2, onde a variável user foi criada a partir de uma factory de usuários.

5.1.4 Cucumber

O Cucumber é uma ferramenta que utiliza a linguagem Gherkin para descrever o comportamento de aplicações e gerar seus respectivos testes automatizados. A sua sintaxe conta com palavras chave como cenário, funcionalidade e possui suporte para diversas linguagens naturais como português e inglês, por exemplo. O Cucumber facilita o processo de automação de testes de comportamento e aceitação e se torna um suporte prático ao Desenvolvimento Orientado à Comportamento (BDD) [27].

Um teste de comportamento do Cucumber pode ser conferido no exemplo abaixo.

```
1 #language: pt
2 #encoding: utf-8
3
4 Funcionalidade: criar posts no blog
5   Como usuario
6   Para que eu crie um post
7   Eu quero criar um post
8
9   #index
10  Cenario: Listar todos os posts
11    Dado que existam 2 posts criados
12    Dado que eu esteja na pagina inicial
13    Quando eu clicar no link para "posts"
14    Entao eu devo ver a lista de postagens com 2 itens
15
16  Cenario: Listar posts com truncamento diferente
17    Dado que existam 2 posts criados
18    Dado que eu esteja na pagina inicial
19    Quando eu clicar no link para "posts"
20    E eu clicar no link para "Ver com truncamento 10"
```

```
21     Entao eu devo ver a lista de postagens com 2 itens
```

Listing 5.4: Exemplo de teste de comportamento em Gherkin

Observa-se que a linguagem se assemelha muito a linguagens naturais, o que permite uma construção mais direta do mapeamento do comportamento da aplicação por meio de algo formal e reproduzível em código. Isso se torna uma vantagem sobre métodos como os de UML e diagramas de estado.

O teste do Listing 5.4 é composto pela descrição da funcionalidade e dois cenários, que são conhecidos como cenários BDD quando desenvolvidos utilizado a abordagem BDD.

O Cucumber para a linguagem Ruby utiliza os chamados *step definitions* para relacionar o cenário de teste em linguagem natural com trechos de códigos em Ruby que devem realizar o que o cenário estabelecer. Por exemplo, na linha 21 do teste acima, o *step definition* correspondente é:

```
1 E("eu clicar no link para ver {string}") do
2     click_button(string)
3 end
```

Listing 5.5: Exemplo de step definition do Cucumber

Para a gema proposta neste trabalho, o Cucumber se torna útil por conta da sua relação de ao exercitar testes de comportamento, executar os métodos utilizados nos *step definitions*. Dessa forma, a gema consegue mapear os métodos das classes de Modelo que foram chamados a partir da exercitação dos cenários BDD.

5.2 Arquitetura do sistema

A Figura 5.1 ilustra a arquitetura da gema em Ruby, com seus dois grandes módulos: um de geradores de testes, o `Generators`, e outro de suporte à geração de teste, o `TestGeneratorLib`. O módulo `Generators` engloba os pacotes de geração de teste, responsáveis pela criação dos arquivos de testes funcionais. O módulo `TestGeneratorLib` possui os outros pacotes necessários para o funcionamento da ferramenta. Tais módulos são explicitados nas próximas seções.

5.2.1 Monitoramento dos testes de comportamento

O monitoramento dos métodos executados na exercitação dos cenários BDD é responsabilidade do módulo `Observer` da gema. Este módulo injeta o aspecto às classes da camada de Modelo da aplicação, desse modo todos os métodos executados executará também o aspecto.

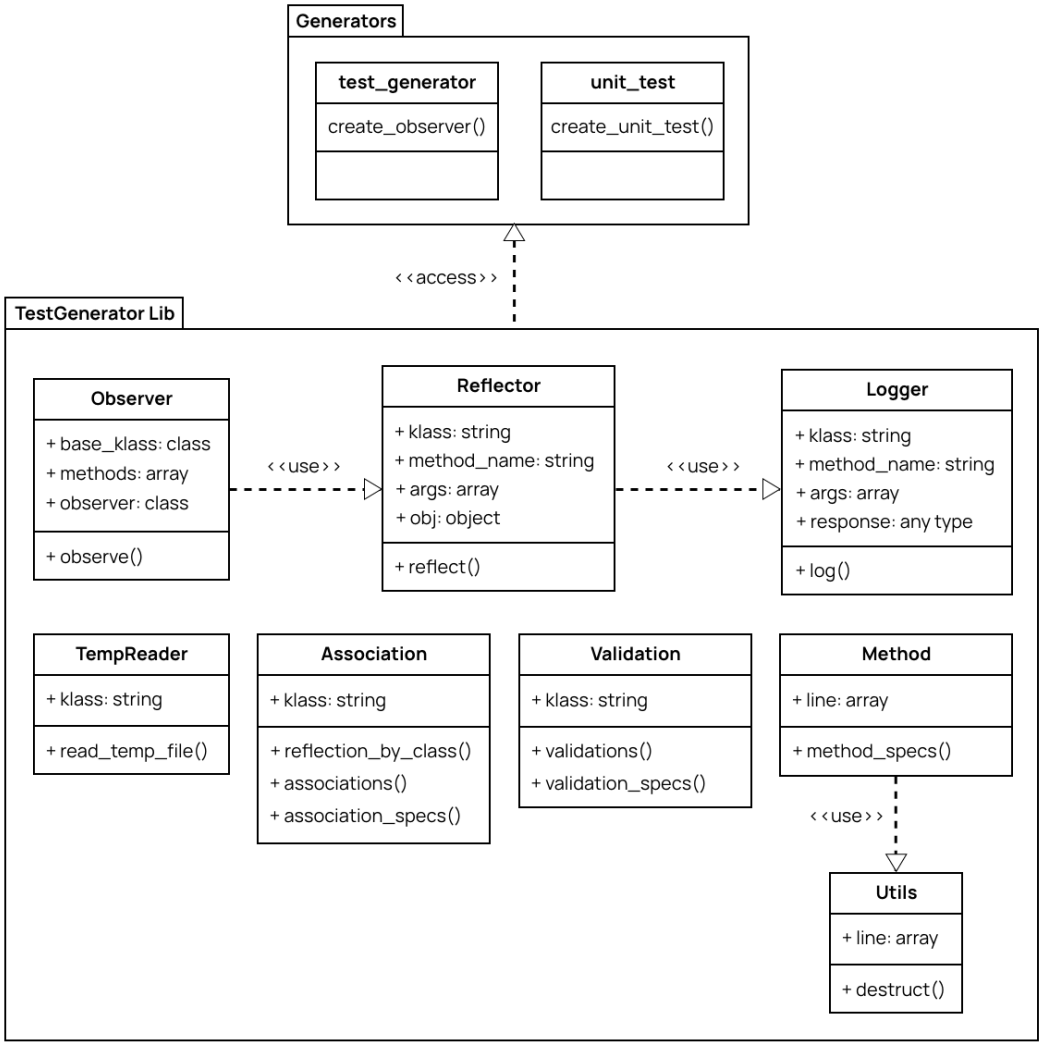

Figura 5.1: Diagrama de pacotes da gema

O aspecto é definido pelo método `observe` no módulo `Observer` que realiza a reflexão dos métodos executados, apresentado no código abaixo.

```
1  def observe
2    klass = self
3    observer = const_get "#{klass.name}Observer"
4    methods = klass.instance_methods(false) - DENYLIST
5
6    observer.class_eval do
7      methods.each do |method_name|
8        define_method(method_name) do |*args, &block|
9          reflect(klass, method_name, args, self)
10         super(*args, &block)
11       end
12     end
```

```
13      end
14    end
```

Listing 5.6: Trecho de código do módulo Observer

O método `observe` é injetado graças à inerente orientação à objetos da linguagem Ruby, sendo possível a criação dinâmica do módulo `Observer` especifico para cada classe da camada de Modelo e também a inserção dinâmica de trechos de código nas classes da aplicação.

Ao se iniciar a exercitação de um conjunto de testes de comportamento, o seguinte trecho de código do Listing 5.7 é executado. Esse código é criado pelo módulo `test_generator`.

```
1  ApplicationRecord.descendants.each do |model|
2    model.send(:include, TestGenerator::Observer)
3    model.observe
4  end
```

Listing 5.7: Trecho de código responsável por observar os models

Esse trecho inclui dinamicamente o módulo Observer da gema em cada classe da camada de Modelo e também executa o método `observe`, o aspecto da gema.

5.2.2 Reflexão dos módulos

Como observado no Listing 5.6, o método `observe` utiliza o método `reflect` na sua definição. Tal método é responsabilidade do módulo `Reflector`, como mostrado no código abaixo.

```
1  def reflect(klass, method_name, args, obj)
2    attrs = obj.instance_variable_get("@attributes").to_hash
3
4    response = if args.any?
5      obj.method(method_name).super_method.call(*args)
6    else
7      obj.method(method_name).super_method.call
8    end
9
10   log(klass, method_name, args, attrs, response)
11 end
```

Listing 5.8: Trecho de código responsável por realizar a reflexão

O Listing 5.8 traz um trecho o código do método `reflect` do módulo `Reflector` Nesse método, a gema utiliza os conceitos de metaprogramação em Ruby para se obter o retorno de qualquer método de qualquer classe da camada de Modelo.

O método `reflect` recebe como parâmetros: o nome da classe, o nome do método observado, os argumentos do método observado e o objeto, no qual o método foi utilizado.

Primeiramente a gema recupera os atributos do objeto no qual o método observado foi chamado. Esses atributos são utilizados posteriormente para o registro do método. A reflexão ocorre chamando o método observado de forma dinâmica a partir do método `call` e passando à ele os seus argumentos. Há um método utilizado previamente ao `call` para se chamar o próprio método da classe da camada de Modelo, e não o método modificado pelo módulo `Observer`.

Com o retorno do método recuperado, um log é criado, utilizando o nome da classe, o nome do método observado, os seus argumentos e retorno e também os atributos do objeto no qual o método observado foi utilizado.

Esse log é salvo num arquivo com o nome da classe na pasta `tmp` (arquivos temporários) da aplicação.

5.2.3 Geração dos códigos de testes

Após a criação dos logs dos métodos observados, executa-se o gerador de testes funcionais da gema. O gerador é composto por um único método que lê os logs do arquivo temporário e cria os testes funcionais. Ele cria também testes de validação de atributos e testes de associação (relacionamento) entre as classes da camada de Modelo.

```
1 def create_unit_test
2   ...
3   lines = read_temp_file(@klass)
4   ...
5 end
```

Listing 5.9: Trecho de código do gerador de testes funcionais: leitura do arquivo de log

O Listing 5.9 traz um trecho de código do gerador de testes funcionais que é responsável pela leitura do arquivo temporário. O método para leitura, definido no módulo `TempReader`, recebe o nome da classe da camada de Modelo como parâmetro e retorna todas as linhas do arquivo temporário de métodos que são daquela classe.

```
1 def create_unit_test
2   ...
3   associations = associations(@klass.camelize.constantize)
4
5   associations.each do |kind, names|
6     @associations_specs << association_specs(kind, names)
7   end
8
9   @associations_specs = @associations_specs.flatten.uniq.compact
```

```
10    ...
11  end
```

Listing 5.10: Trecho de código do gerador de testes funcionais: associações

O Listing 5.10 traz um trecho de código que é responsável pela criação dos testes funcionais de associação (relacionamento) da classe da camada de Modelo, por exemplo, uma classe que esteja relacionada com outra através do relacionamento `has_many`, significa que para cada chave primária de uma classe, na outra há chaves estrangeiras que apontem para essa chave primária. O módulo `ActiveRecord` do Ruby on Rails é responsável por implementar os relacionamentos entre as classes da camada de Modelo.

```
1  def create_unit_test
2    ...
3    validations = validations(@klass.camelize.constantize)
4
5    validations.each do |kind, attrs|
6      @validations_specs << validation_specs(kind, attrs)
7    end
8
9    @validations_specs = @validations_specs.flatten.uniq.compact
10   ...
11 end
```

Listing 5.11: Trecho de código do gerador de testes funcionais: validações

O Listing 5.11 traz um trecho de código que é responsável pela criação dos testes funcionais de validação dos atributos da classe da camada de Modelo, por exemplo, uma classe que tenha o método de validação `validate_presence_of` sob o atributo `name`, significa que essa classe não pode ser instanciada sem passar o atributo `name`. O módulo `ActiveModel` do Ruby on Rails é responsável por implementar as validações dos atributos das classes da camada de Modelo.

```
1  def create_unit_test
2    ...
3    lines.each do |line|
4      @methods_specs << method_specs(line)
5    end
6
7    template "model_spec.rb", Rails.root.join("spec/models/#{
       class_name.tableize.singularize}_spec.rb")
8    ...
9  end
```

Listing 5.12: Trecho de código do gerador de testes funcionais: métodos

Por fim, o Listing 5.12 traz um trecho de código que é responsável pela criação dos testes funcionais dos métodos observados da classe da camada de Modelo. A gema utiliza um *template* para criar o arquivo de teste do RSPEC, como mostrado no Listing 5.13

```
...
RSpec.describe <%= @klass %>, type: :model do<% @validations_specs.
    each do |validation| %>
  <%= validation %>
<% end %><% @associations_specs.each do |association| %>
  <%= association %>
<% end %>
<%= @subject_spec %>
<% @methods_specs.each do |method| %>
  <%= method %>
<% end %>end
```

Listing 5.13: Template do arquivo de teste do RSPEC

5.3 Exemplo de execução da ferramenta

Nesta seção apresentamos um exemplo de execução da ferramenta, partindo de um teste de comportamento do Cucumber, passando pelos módulos da gema, de reflexão, geração de logs e geração dos testes funcionais.

5.3.1 Teste de comportamento

O Listing 5.14 exemplifica um cenário BDD de um teste de comportamento de uma aplicação de Blog, desenvolvida com o framework Ruby on Rails.

```
#language: pt
#encoding: utf-8

Cenario: Listar todos os posts
  Dado que existam 2 posts criados
  Dado que eu esteja na pagina inicial
  Quando eu clicar no link para "posts"
  Entao eu devo ver a lista de postagens com 2 itens
```

Listing 5.14: Cenário BDD para listagem de posts

Na exercitação do cenário do Listing 5.14, cada passo executará um *step definition* relacionado. O passo da linha 7 executará o seguinte *step definition*:

```
Quando("eu clicar no link para {string}") do |link_name|
  click_link link_name
```

```
3 end
```

Listing 5.15: Step definition de um passo de cenário BDD

Semanticamente, ao clicar no link "posts", o usuário da aplicação será redirecionado à página que contém todos os posts do Blog.

Como a aplicação é desenvolvida no Ruby on Rails e como já explicitado na seção 5.1, após a requisição do cliente, a aplicação faz uma chamada à camada Controladora, realiza as consultas necessárias à camada de Modelo e retorna uma página HTML utilizando a camada de Visualização. A página retornada no exemplo do Listing 5.15 executa o seguinte trecho de código:

```
1  <% @posts.each do |post| %>
2    <tr>
3      <td><%= post.upcase_title %></td>
4      <td><%= post.truncated_desc((params[:truncate_size] || 30).to_i
       ) %></td>
5      <td><%= post.create_date %></td>
6      <td><%= link_to 'Show', post %></td>
7      <td><%= link_to 'Edit', edit_post_path(post) %></td>
8      <td><%= link_to 'Destroy', post, method: :delete, data: {
       confirm: 'Are you sure?' } %></td>
9    </tr>
10 <% end %>
```

O trecho de código do Listing 5.3.1 executa os métodos `upcase_title`, `truncated_desc` e `create_date` sobre o método `post`, uma instância da classe `Post` da camada de modelo da aplicação.

5.3.2 Monitoramento dos métodos

Como demonstrado na seção 5.3.1, antes de se exercitar os testes de comportamento, o código do Listing 5.7 é executado, injetando o código do método `observe` em cada método de cada classe da camada de Modelo, incluindo a classe `Post`, nesse exemplo. O método `observe` realiza a reflexão do método observado sobre o objeto da classe `Post`.

5.3.3 Reflexão e geração de logs

Após a reflexão dos métodos observados `upcase_title`, `truncated_desc` e `create_date`, os seguintes logs são gerados no arquivo `Post.rb` dentro da pasta `tmp` da aplicação:

```
1 {
2     "klass": "Post", "method": "upcase_title", "args": [], "attrs":
      {"id":117,"title":"Post 0","description":"Esse e o post 0 post
```

```
    0 post 0 post 0 post 0 post 0 post 0","created_at":"2020-10-14
    T15:58:06.866Z","updated_at":"2020-10-14T15:58:06.866Z"}, "
    response": "\"POST 0\""
}

{
     "klass": "Post", "method": "truncated_desc", "args": [30], "
    attrs": {"id":117,"title":"Post 0","description":"Esse e o post
    0 post 0 post 0 post 0 post 0 post 0 post 0","created_at":"
    2020-10-14T15:58:06.866Z","updated_at":"2020-10-14T15:58:06.866Z
    "}, "response": "\"Esse    o post 0 post 0 post...\""
}

{
     "klass": "Post", "method": "create_date", "args": [], "attrs":
    {"id":117,"title":"Post 0","description":"Esse e o post 0 post 0
     post 0 post 0 post 0 post 0 post 0","created_at":"2020-10-14T15
    :58:06.866Z","updated_at":"2020-10-14T15:58:06.866Z"}, "response
    ": "\"14/10/20\""
}
```

Listing 5.16: Logs gerados pelo método observe

5.3.4 Geração dos testes funcionais

Para a geração dos testes funcionais, é necessário a execução do comando que executará o código do módulo `unit_test`, dentro do módulo `TestGenerator`.

```
rails g unit_test Post

Running via Spring preloader in process 47351
      create  spec/models/post_spec.rb
      create  spec/factories/posts.rb
```

Listing 5.17: Execução do módulo unit test

O módulo `unit_test` faz uso dos módulos `association`, `validation` e `method` para se gerar os testes funcionais de associação (relacionamento entre classes), validação dos atributos da classe e os testes dos métodos da classe.

Os testes funcionais gerados para a classe `Post` da camada de Modelo estão listados no Listing 5.18.

```
RSpec.describe Post, type: :model do
  let(:post) { create(:post) }
  it { should validate_presence_of(:title) }

```

```
5   describe '#upcase_title' do
6     it 'should' do
7       expect(post.upcase_title()).to eq "POST 0"
8     end
9   end
10
11  describe '#truncated_desc' do
12    it 'should' do
13      expect(post.truncated_desc(30)).to eq "Esse e o post 0 post 0
      post..."
14    end
15  end
16
17  describe '#create_date' do
18    it 'should' do
19      expect(post.create_date()).to eq "14/10/20"
20    end
21  end
22 end
```

Listing 5.18: Testes funcionais gerados para a classe Post

Capítulo 6

Estudo de Caso

Para validação dos objetivos da pesquisa alguns estudos de caso foram feitos. A ideia é validar o quão relevante pode ser a gerações de testes de forma automática e a percepção de melhora no código promovida pela mesma.

Para tanto, foram selecionados projetos hospedados em repositórios no GitHub que utilizam a gema Cucumber e que implementados em Rails, já que o framework atualmente dá suporte a projetos Ruby on Rails. A partir disso, alguns fatores foram levados em consideração para analisar os resultados obtidos:

1. Cobertura antes e depois da geração dos testes.
2. Testes de mutação (mutantes mortos) antes e depois da geração dos testes.

Alguns casos de teste criados pela ferramenta também foram analisados qualitativamente. Esses casos mostram uma semelhança em testes criados pela ferramenta e testes criados por programadores.

Para os critérios de seleção dos repositórios, levou-se em conta projetos que tinham atualizações no último ano, e que possuíam versão do Rails maior do que o 4.0. Além disso, foram selecionados manualmente repositórios que tinham ao menos dois métodos nas classes da camada de Modelo e um teste comportamento associado à classe. Esses repositórios são apresentados na Tabela 6.1

O processo para se testar os repositórios pode ser descrito nos seguintes passos:

1. Mineração dos projetos no Github que utilizam a gema `cucumber-rails` (gema utilizada no Rails para gerar testes de comportamento com o Cucumber).
2. Dos projetos encontrados, fez-se uma seleção baseada na versão do Rails utilizada, no histórico de atualizações e na quantidade de Models testáveis.

3. Se o projeto atende aos critérios, ele é clonado para uma máquina local e instala-se a gema Test Generator nele, configurando os requisitos necessários para fazê-la funcionar no projeto.

4. Instalada a gema, os testes de comportamento são executados, por meio do Cucumber, gerando os logs no sistema, que são utilizados pela gema para criar os testes unitários e as factories para testes.

5. Verificando a cobertura dos novos testes, incluindo os novos testes gerados por meio da nossa abordagem. Ressalta-se que previamente exercitam-se os testes que haviam antes da gema gerar os novos testes, para obter-se a cobertura antes também.

6. Por fim, são gerados e exercitados os testes de mutação sobre os testes gerados para se verificar a quantidade de mutantes mortos.

Para cada um dos repositórios testados seguindo tal procedimento, obtivemos a cobertura antes e depois da utilização da gema e também o grau de relevância dos testes por meio dos testes de mutação (quantidade de mutantes mortos).

O objetivo principal deste estudo de caso consistiu em avaliar a eficácia da nossa abordagem por meio da gema implementada neste trabalho. Essa eficácia foi medida em relação à sua aplicabilidade e à viabilidade de gerar testes funcionais em projetos reais. Para tanto, levantamos as seguintes perguntas a serem exploradas neste estudo de caso:

Q1. Os resultados gerados estão corretos?

Q2. A cobertura dos testes aumentou após o uso da ferramenta?

Q3. Qual a qualidade dos testes gerados?

Tais questões guiaram a metodologia de pesquisa descrita na seção anterior e permitiram um maior rigor à pesquisa.

Repositório	**Commits**	**Contribuidores**	**BDD Features**
lrusso96/Minerva	82	2	26
Omeshwor/finance-tracker-6	38	1	3
limabia/doei	322	6	29
brandaoplaster/portfolio-creator/	396	1	16
wearefuturegov/outpost	1024	6	3
ahomayouni/XChange	1034	7	5
tamu-zlp/zlp-scheduler	381	12	10
davidaniel91/Blog	46	1	3
CWISoftware/enlighten	459	7	7
johnsonsirv/facebook-clone	81	2	10

Tabela 6.1: Tabela com os repositórios selecionados para o estudo de caso

6.1 Resultados

A partir dos 10 repositórios testados com a gema pode-se obter alguns resultados que respondem as questões de pesquisa.

6.1.1 Q1. Os resultados gerados estão corretos?

Para verificar se os testes gerados eram corretos, rodou-se a ferramenta RSPEC sobre os testes gerados. Todos os testes geradores pela gema se apresentaram corretos, ou seja, não apresentaram falhas. Para cada entrada utilizada no teste a saída correta ocorreu, como mostrado no Listing 6.1. Esse comportamento se manteve em todos as classes da camada de Modelo de todos os repositórios testados.

```
$ rspec spec/models/user_spec.rb

User
  should validate that :email cannot be empty/falsy
  should validate that :name cannot be empty/falsy
  should validate that :password cannot be empty/falsy
  should have many notifications
  should have many borrow_requests
  should have many live_searches
  should have many memberships
  should have many listings
  should have many messages
  should have one person
  should have one location
  #forget
    should
  #activation_token
    should

Finished in 0.37481 seconds (files took 3.67 seconds to load)
13 examples, 0 failures
```

Listing 6.1: Comando para rodar os testes no repositório XChange

6.1.2 Q2. A cobertura dos testes aumentou após o uso da ferramenta?

Para todos as classes da camada de Modelo testadas, a cobertura final com os testes gerados aumentou.

Uma comparação entre a cobertura de código nos testes já existentes em cada classe da camada de Modelo e a cobertura de código após serem gerados os testes pela gema foi feita. O gráfico 6.1 mostra essa comparação.

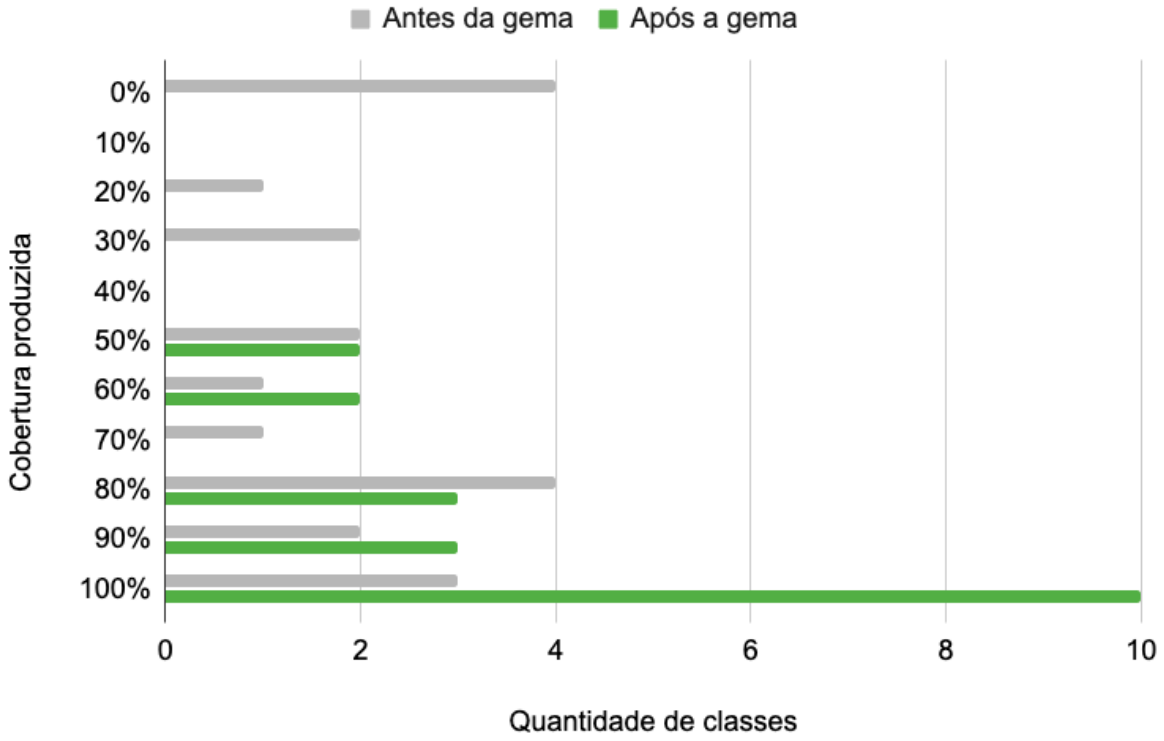

Figura 6.1: Gráfico com comparação entre cobertura de código com testes existentes e testes gerados pela ferramenta

Observa-se que o número de classes da camada de Modelo com coberturas na faixa de 80, 90 e 100% aumentou (barras verdes) em comparação com o mesmo número para os testes que existiam antes do uso da ferramente (em cinza).

A partir desses dados, obteve-se a cobertura média das classes da camada de Modelo para os repositórios estudados. Com testes existentes antes do uso da ferramenta, esse número foi de **51,5%**. Ou seja, em média, a cobertura de código foi de 51,5% para cada classe da camada de Modelo. Para os testes gerados com a ferramenta, esse número foi de **81,5%**.

A partir desses dados pode-se obter a média de aumento de cobertura por classe. Esse valor foi de **30,0%**. Ou seja, em média, para cada classe, houve um aumento de 30% na quantidade de testes, após serem gerados os testes pela ferramenta.

6.1.3 Q3. Qual a qualidade dos testes gerados?

Para aferir a qualidade, utilizamos a comparação entre a quantidade de mutantes mortos com testes feitos pelos programadores e a quantidade de mutantes mortos com testes feitos pela ferramenta.

Para cada repositório, executou-se os testes de mutação em cada classe da camada de Modelo para verificar a porcentagem de mutantes mortos para aquela suíte de testes.

O mesma execução dos testes de mutação foram feitos para os testes gerados pela ferramenta. Para cada classe, obteve-se a porcentagem de mutantes mortos. Estas classes foram agrupados em faixas de porcentagem de mutantes mortos, conforme mostrado no gráfico 6.2.

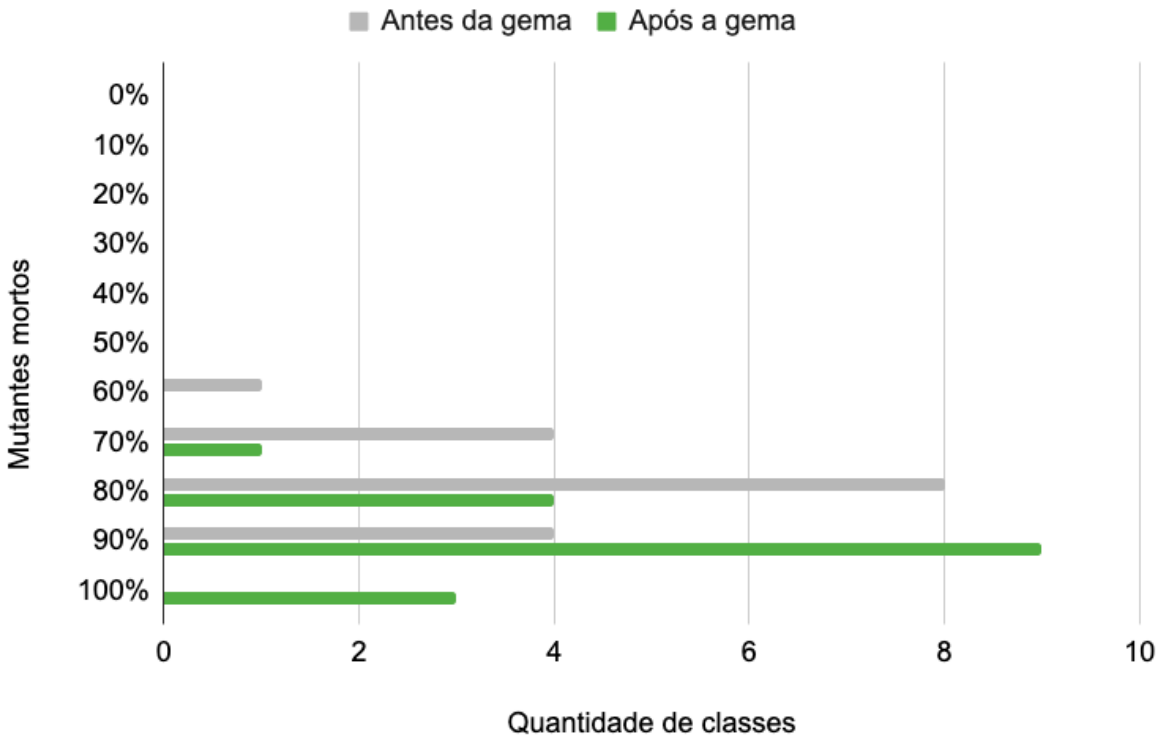

Figura 6.2: Gráfico com comparação entre porcentagem de mutantes mortos antes e depois da geração dos testes pela gema

A partir desses dados, obteve-se o coeficiente de mutantes mortos (CMM) para os testes feito por programadores. Esse número é a média aritmética da porcentagem de mutantes mortos para cada classe da camada de Modelo. O CMM foi de **78,82%** para os testes feitos por programadores (testes anteriores).

Obteve-se também o CMM para os testes produzidos pela ferramenta. Esse valor foi de **83,23%**. Isso mostra que na média, o número de mutantes mortos foi maior nos testes criados pela gema, para cada classe da camada de Modelo.

Uma análise qualitativa de um dos testes gerados ajuda também a entender a relevância dos testes feitos. A seguir mostramos um teste gerado pela ferramenta no repositório Minerva, no classe da camada de Modelo de usuários:

```
1 describe '#fullname' do
2     it 'should' do
3       user = create(:user, { email: 'cocomero@email.com', name: '
    Cocomero', surname: 'Registrato', birthdate: '2000-10-15',
    nickname: '', country: '', website: '', admin: 'false' } )
4       expect(user.fullname()).to eq "Cocomero Registrato"
5   end
6 end
```

Listing 6.2: Caso de teste gerado pela ferramenta

A função testada é a `fullname`, mostrada a seguir:

```
1   def fullname
2     name + ' ' + surname
3   end
```

Listing 6.3: Caso de teste gerado pela ferramenta

Vemos na linha 4 da listagem 6.2 que a expectativa ao se chamar esse método `fullname` num usuário de nome 'Cocomero' e sobrenome (surname) 'Registrato' é o retorno do nome e sobrenome concatenados, 'Cocomero Registrato', segundo o teste.

Observa-se pela implementação do método `fullname` que esse é justamente o comportamento da função, que concatena os 2 atributos. Isso mostra que o teste foi capaz de identificar qual deveria ser o comportamento da função.

6.2 Discussão

A partir dos resultados obtidos pode-se perceber o quanto a nossa abordagem acrescentou em quantidade e em qualidade os testes funcionais gerados. Em vários dos repositórios analisados percebeu-se uma carência considerável de testes funcionais, apesar de inicialmente se ter a falsa impressão de que todo o sistema está sendo testado. Os resultados gerados pela nossa ferramenta por meio de testes funcionais mostram que houve aumento de cobertura em um número significativo de Models. Aumentos de mais de 50% em alguns casos na cobertura total da classe da camada de Modelo foram identificados. Percebe-se que 16 dos Models testados atingiram mais de 80% de cobertura de testes apenas com os testes gerados pela ferramenta. Isso indica que soluções como as propostas podem obter um número satisfatório de cobertura. Tal processo também indica quais funções estão sendo utilizadas no sistema e quais dessas são as mais relevantes em termos de necessi-

dade de se testar. Esse resultado decorre diretamente dos arquivos de teste gerados pela gema.

Quanto à parte dos mutantes, observou-se alto grau de mutantes mortos durante os testes nas classes as quais a ferramenta criou testes. Esse resultado mostra a coerência dos testes gerados.

Além disso, observamos que, em média, a quantidade de mutantes mortos por classe da camada de Modelo foi bastante considerável: 83%. Para fins de comparação, foram testados algumas desses classes com os testes feitos por programadores previamente que estavam presentes nos repositórios. O resultado de 78% mostrado elucida que é difícil se obter um grau de mutantes mortos alto. Mesmo nos testes previamente feito por programadores, um grau de mutantes mortos superior a 80% não foi atingido, em média.

Essa diferença na média de mutantes mortos entre os testes feitos pela ferramenta e testes feitos por programadores mostra que é possível se gerar testes de modo automático. Testes estes que se assemelham, ou ainda se mostram superiores aos testes feitos de maneira manual. Estes resultados podem indicar um futuro promissor pra metodologia e pro próprio projeto da ferramenta.

Uma ameaça à validade de construção dos nossos experimentos foram as dificuldades de reproduzir o ambiente de execução dos projetos. Em algumas classes da camada de Modelo houveram problemas de compatibilidade na hora da execução dos testes de mutação, que eram uma dependência de terceiros. Por conta dessa limitação, o número de classes da camada de Modelo testadas com os testes de mutação (17) foi menor do que o número de classes testadas pela gema criada (20). Futuros experimentos deverão ser realizados para fins de generalização dos resultados preliminarmente reportados neste trabalho.

Capítulo 7

Conclusão

A partir do estudo algumas conclusões puderam ser obtidas a respeito da ferramenta proposta.

A partir do estudo realizado de conceitos como programação orientada a aspectos, meta programação, reflexão de funções pudemos obter o processo proposto para criação da ferramenta. Esse processo é explicitado no Capítulo 4.

Outro resultado importante foi a própria ferramenta, analisada no capítulo 5. Desenvolvida em Ruby, a gema proposta se mostrou capaz de produzir testes funcionais a partir dos cenários BDD.

A ferramenta mostrou resultados positivos, uma vez que aumentou a cobertura de testes dos Models em que ela foi aplicada. Esse aumento na cobertura mostra que é possível gerar testes de forma automática de modo a aumentar esse critério.

Além desse aumento de cobertura, era preciso mostrar também que os testes gerados possuíam um bom grau de relevância. Isso foi mostrado a partir das métricas de testes de mutação e a partir de estudos de caso específico com comparações qualitativas.

Como mostrado em alguns trechos de códigos de testes gerados, mostrados no capítulo 5, vemos que alguns testes gerados conseguem mapear exatamente o comportamento esperado da função testada. Isso mostrou que a metodologia tem, em suas bases, pensamentos utilizados pelos próprios programadores na hora de se realizar testes e identificar comportamentos dos métodos.

A partir de toda pesquisa elucidada nesse trabalho, do processo apresentado e dos resultados obtidos para ferramenta feita, pudemos concluir que a partir dos conceitos estudados é possível gerar a partir de testes de comportamento, testes funcionais que aumentam significativamente a cobertura de códigos e que possuem alto grau de relevância, comparáveis a testes feitos por programadores.

Trabalhos futuros envolvem a generalização do processo descrito no Capítulo 4, para fim de se tornar mais fácil a criação de ferramentas como a proposta neste trabalho.

Melhorias na ferramenta proposta, o *TestGenerator*, também fazem parte de um aprimoramento do trabalho, para que se possa utilizar a ferramenta em uma grande quantidade de repositórios.

Referências

[1] Ahamed, SS: *Studying the feasibility and importance of software testing: An analysis.* arXiv preprint arXiv:1001.4193, 2010. 1, 4

[2] Márcio Eduardo Delamaro, José Carlos Maldonado, Mario Jino: *INTRODUÇÃO AO TESTE DE SOFTWARE*, volume 4. Elsevier Editora Ltda, 2007. 1, 2, 8, 9, 10

[3] Talby, David, Arie Keren, Orit Hazzan e Yael Dubinsky: *Agile software testing in a large-scale project.* IEEE software, 23(4):30–37, 2006. 1

[4] Schwaber, Ken e Jeff Sutherland: *The scrum guide.* Scrum Alliance, 21:19, 2011. 1

[5] Blackburn, Joseph D, Gary D Scudder e Luk N Van Wassenhove: *Improving speed and productivity of software development: a global survey of software developers.* IEEE transactions on software engineering, 22(12):875–885, 1996. 2, 3, 12

[6] Mathias Soeken1, Robert Wille1 e Rolf Drechsler: *Assisted behavior driven development using natural language processing.* Lecture Notes in Computer Science, 7304(2):131–147, 2003. 3, 7

[7] Just, René, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes e Gordon Fraser: *Are mutants a valid substitute for real faults in software testing?* Em *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, páginas 654–665, 2014. 4

[8] Horgan, Joseph R., Saul London e Michael R Lyu: *Achieving software quality with testing coverage measures.* Computer, 27(9):60–69, 1994. 4

[9] *What is aspect-oriented programming.* `https://docs.jboss.org/aop/1.0/aspect-framework/userguide/en/html/what.html`, acesso em 2020-03-03. 6

[10] Elrad, Tzilla, Robert E Filman e Atef Bader: *Aspect-oriented programming: Introduction.* Communications of the ACM, 44(10):29–32, 2001. 6

[11] Wikipédia: *Ruby language.* `https://en.wikipedia.org/wiki/Ruby_(programming_language)`, acesso em 2020-03-04. 7

[12] *Metaprogramming.* `https://cs.lmu.edu/~ray/notes/metaprogramming/`, acesso em 2020-03-03. 7

[13] North, Dan: *Introducing bdd*, 2015. `https://dannorth.net/introducing-bdd`. 8

[14] Smart, John Ferguson: *Bdd in action*. página 12, 2014. 8

[15] Zen, Roberto: *A new test case path composition approach to testing ruby on rails web applications*. 2013. 9

[16] Reichert, Amy: *Common functional testing types explained, with examples*, 2020. `https://searchsoftwarequality.techtarget.com/tip/Common-functional-testing-types-explained-with-examples`. 10

[17] Papadakis, Mike, Nicos Malevris e Maria Kallia: *Towards automating the generation of mutation tests*. Em *Proceedings of the 5th Workshop on Automation of Software Test*, páginas 111–118, 2010. 10, 11

[18] Malaiya, Yashwant K, Michael Naixin Li, James M Bieman e Rick Karcich: *Software reliability growth with test coverage*. IEEE Transactions on Reliability, 51(4):420–426, 2002. 12

[19] Zhu, Hong, Patrick AV Hall e John HR May: *Software unit test coverage and adequacy*. Acm computing surveys (csur), 29(4):366–427, 1997. 12

[20] Fox, Armando, David A Patterson e Samuel Joseph: *Engineering software as a service: an agile approach using cloud computing*. Strawberry Canyon LLC, 2013. 12

[21] Nebut, Clementine, Franck Fleurey, Yves Le Traon e J M Jezequel: *Automatic test generation: A use case driven approach*. IEEE Transactions on Software Engineering, 32(3):140–155, 2006. 14

[22] Weißleder, Stephan, Dehla Sokenou e Bernd Holger Schlingloff: *Reusing state machines for automatic test generation in product lines*. Em *1st workshop on model-based testing in practice (MoTiP)*, página 19. Citeseer, 2008. 15

[23] Tahat, Luay H., Boris Vaysburd, Bogdan Korel e Atef J. Bader: *Requirement-based automated black-box test generation*. 0-7695-1372-7101, 2001. 15

[24] Li, Guodong, Indradeep Ghosh e Sreeranga P Rajan: *Klover: A symbolic execution and automatic test generation tool for c++ programs*. Em *International Conference on Computer Aided Verification*, páginas 609–615. Springer, 2011. 16

[25] Nguyen, Bao N., Bryan Robbins, Ishan Banerjee e Atif Memon: *Guitar: an innovative tool for automated testing of gui-driven software*. Springer Science+Business Media New York 2013, 2013. 16

[26] Burr, Kevin e William Young: *Combinatorial test techniques: Table-based automation, test generation and code coverage*. Em *Proc. of the Intl. Conf. on Software Testing Analysis & Review*. Citeseer, 1998. 16

[27] Bui, Trong: *Testes de automação com cucumber bdd em times Ágeis*, nov 2018. `https://www.infoq.com/br/articles/cucumber-bdd-automation-testing/`. 19, 29

[28] North, Dan: *O que é json*, 2020. https://www.hostinger.com.br/tutoriais/o-que-e-json/. 23

[29] Deacon, John: *Model-view-controller (mvc) architecture*. Online[Citado em: 10 de março de 2006.] http://www.jdl.co.uk/briefings/MVC.pdf, 2009. 25, 27

[30] Pop, Dragos Paul e Adam Altar: *Designing an mvc model for rapid web application development*. Procedia Engineering, 69:1172–1179, 2014. 25

[31] Guides, Rails: *What is ruby on rails*, jul 2012. https://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark. 27

[32] Woods, Jelani: *Ruby on rails: Generators*, maio 2019. https://medium.com/@jelaniwoods/ruby-on-rails-generators-91bdebc4ca6d. 27

[33] Alameda, Eldon: *Introduction to testing with rspec*. Foundation Rails 2, páginas 253–289, 2009. 28