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

Provendo manutenibilidade e testabilidade ao Feature-Trace

Vítor Ribas Bandeira

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Genaína Rodrigues

Brasília
2022

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

Provendo manutenibilidade e testabilidade ao Feature-Trace

Vítor Ribas Bandeira

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Genaína Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Gabriel Siqueira Rodrigues
CIC/UnB

do Curso de Engenharia da Computação

Brasília, 12 de maio de 2022

Dedicatória

Dedico este trabalho à minha família, mas em especial, aos meus pais Ivaneide e Cícero e à minha falecida avó Damiana, que sempre foram a minha base e fonte de ensinamentos durante a minha formação.

Agradecimentos

Agradeço primeiramente à Deus, por ter me guiado diariamente e me amparado em meio às dificuldades.

Agradeço do fundo de meu coração aos meu pais, Ivaneide e Cícero, por terem me oferecido todo o suporte necessário e orientações durante esta jornada na universidade. Ao meu irmão Vinícius e melhores amigos, com os quais posso contar a qualquer momento e que sempre me apoiaram e sonharam junto comigo. Aos meus avós, tios e tias, que sempre acreditaram no meu potencial e também me deram bons conselhos e palavras.

Agradeço também à minha orientadora, Prof.ª Dr.ª Genaína Nunes Rodrigues, pelos ensinamentos passados durante essa etapa de extrema importância de minha graduação. À Vanessa Nunes, pela boa vontade, alto astral e apoio entregues ao longo do desenvolvimento deste trabalho.

Não posso deixar de agradecer também aos amigos que fiz na universidade, com os quais sempre compartilhei os momentos de dificuldades e preocupações da graduação, mas também de diversão e amizade.

Resumo

A ferramenta *Feature-Trace* realiza uma análise estática e dinâmica em projetos que utilizam o BDD (*Behaviour Driven Development*). Seu objetivo é gerar um POS (*Perfil Operacional do Software*) a partir dos requisitos implementados pelos cenários descritos no BDD, que por sua vez permitirá uma priorização e seleção de casos de testes no projeto analisado.

Entretanto, a ferramenta possui aspectos que não facilitam a evolução do projeto e contribuições futuras. Um destes aspectos se trata da falta de uma suíte de testes automatizados que garanta certo grau de corretude e confiabilidade em sua implementação. Essa é uma questão relevante, uma vez que a proposta da ferramenta é a de se integrar com softwares dos mais variados tipos e proporções. Outro ponto delicado é o quanto a ferramenta está preparada para evoluções futuras, do ponto de vista do esforço necessário para implementação de novas funcionalidades ou manutenções. Este aspecto também está ligado à importância da presença de testes, para assegurar o funcionamento daquilo que já está implementado e detectar possíveis efeitos indesejados.

Desse modo, este trabalho propõe uma solução para esse cenário por meio do desenvolvimento de sua manutenibilidade e, consequentemente, também de sua testabilidade. Ou seja, torná-lo propício para contribuições e alterações futuras, bem como para a implementação de testes de código. De maneira subsequente, também é proposta a implementação de uma suíte de testes que impliquem em uma cobertura de testes satisfatória. Tais contribuições são aplicadas por meio de refatorações de código que visam melhorar sua qualidade do ponto de vista estrutural e de compreensão.

A validação da contribuição deste trabalho se dá por meio da avaliação de aspectos e métricas antes e depois da aplicação destas ações que compõem a metodologia proposta. Os resultados analisados mostram uma melhora significativa na qualidade do Feature-Trace como um projeto de software, isto devido ao esforço aplicado de aprimorar a sua manutenibilidade e testabilidade.

Palavras-chave: BDD, Engenharia de Software, Refatoração, Testabilidade, Cobertura de Código

Abstract

The Feature-Trace tool performs static and dynamic analysis on projects that use BDD (Behaviour Driven Development). Its objective is to generate a POS (Profile Operational Software) from the requirements implemented by the scenarios described in the BDD, which in turn will allow a prioritization and selection of test cases in the project analyzed.

However, the tool has aspects that do not facilitate the evolution of the project and future contributions. One of these aspects is the lack of an automated test suite that guarantee a certain degree of correctness and reliability in its implementation. This is very worrying, since the purpose of the tool is to integrate with software of the most varied types and proportions. Another delicate point is how much the tool is prepared for future evolutions, from the point of view of the necessary effort for implementing new features or maintenance. This aspect is also linked to the importance of the presence of tests, to ensure the functioning of what is already implemented and detect possible unwanted effects.

Thus, this work proposes a solution to this scenario through the development of its testability, that is, make it suitable for the implementation of tests of code. Subsequently, it is also proposed the implementation of a suite of tests that imply satisfactory test coverage. Such contributions also comprise code refactorings that aim to improve its quality from the point of view structural and understanding.

The validation of the contribution of this work takes place through the evaluation of aspects and metrics before and after the application of the actions that make up the proposed methodology. The analyzed results show a significant improvement in the quality of the Feature- Trace as a software project, this due to the effort applied to improve its manutenability and testability.

Keywords: BDD, Software Engineering, Code Refactoring, Testability, Code Coverage

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Vivemos em um contexto onde os softwares podem mudar a todo momento, seja devido a evoluções no projeto, adição de novas *features* ou mesmo manutenções preventivas. Segundo [4], o ciclo de vida do software é altamente dependente de manutenção, processo esse que comumente compreende a maior parte do custo envolvido no ciclo de desenvolvimento do software.

Por causa disso, a manutenibilidade, isto é, a facilidade com a qual um software pode ser modificado [4] [5], é um importante atributo de qualidade de software. Atributo este que está fortemente ligado ao processo de manutenção e atividades de refatoração [4]. De acordo com [6], a refatoração pode ser definida como o processo usado para alterar o código de um software sem produzir alterações no seu comportamento. Ela pode ser aplicada por engenheiros de software por meio da adesão de padrões de código, uma vez que segundo [4], a engenharia de software deve ser responsável por criar não somente código funcional, mas também bem estruturado e manutenível.

Dentro do ciclo de desenvolvimento, o teste de software é uma atividade fundamental para garantir a qualidade de um software. Porém, nem todos os sistemas são facilmente testáveis. A testabilidade de um software, que é um atributo de qualidade, é definida como a facilidade com a qual um software pode demostrar suas falhas por meio da atividade de teste [3]. Portanto, quanto maior o grau de testabilidade do software, maior a facilidade de implementação de casos de teste. Já um baixo grau de testabilidade resultará em um maior esforço para aplicação de testes em um determinado período de tempo

A cobertura de teste é um importante indicador da qualidade do software e parte essencial para a sua manutenção. Essa métrica auxilia na avaliação da eficácia dos testes implementados e aponta áreas de código que não foram cobertas [7], possibilitando uma análise de mais casos de teste que contribuam para aumentar essa cobertura.

A motivação para este trabalho parte da necessidade de tornar uma ferramenta *open source* mais manutenível e testável. Isto é justificado pois, como qualquer software mo-

derno, há a necessidade de realizar constantes modificações no mesmo, com o menor custo e esforço possíveis, assim como garantir um grau maior de confiabilidade naquilo que está implementado e entregue.

1.1 Problema

O Feature-Trace é um software de código aberto implementado em linguagem *Python* que propõe a geração do POS (Perfil Operacional do Software) de um software terceiro, utilizando como *inputs* os cenários de teste do BDD (*Behaviour Driven Development*) desta aplicação. Desse modo, com o POS em mãos, é possível tomar decisões de projeto como, por exemplo, a priorização de determinados casos de teste [2].

A ferramenta Feature-Trace é bastante promissora para guiar equipes de desenvolvimento de software na aplicação de seus esforços. Contudo, sua implementação de código carece de uma maior manutenibilidade e testabilidade. Isto é, a versão inicial que motiva este trabalho possui *bugs* isolados e seu código é confuso e difícil de ser modificado e testado. Sua estrutura não favorece a manutenibilidade do projeto. Mais especificamente, apresenta métodos longos, trechos de código que se repetem e métricas de código que podem ser melhoradas. Do mesmo modo, também não possui casos de teste implementados que possam garantir o correto funcionamento de sua implementação para análise de outros softwares.

Tais pontos negativos notados vão em contramão a características e atributos desejáveis de um bom e atual projeto de software. Para isso, a proposta deste trabalho se torna necessária.

1.2 Objetivo geral

O objetivo principal deste trabalho é aumentar o grau de manutenibilidade e testabilidade do Feature-Trace. Isto se dará pela aplicação de técnicas e abordagens pertencentes a cada um destes dois atributos de qualidade de software.

1.3 Objetivos Específicos

Considerando o objetivo geral proposto, este trabalho foca nos seguintes objetivos específicos:

- Tornar menos custoso o processo de manutenção e evolução do software por parte dos desenvolvedores.

- Atualizar a implementação de código do software de acordo com padrões de codificação seguidos pela comunidade de desenvolvimento.
- Prover casos de teste para a implementação das funcionalidades entregues pela ferramenta.

Cada um destes objetivos específicos traçados possui ligação com os esforços de manutenibilidade e testabilidade, como ficará mais claro ao longo dos próximos capítulos.

1.4 Descrição dos Capítulos

Este trabalho está dividido em mais quatro capítulos seguintes a este. O Capítulo 2 fornece um embasamento teórico necessário para a aplicação das contribuições propostas e uma contextualização sobre o Feature-Trace e conceitos ligados a ele.

O Capítulo 3 apresenta a formulação da proposta aplicada para contribuir na solução das limitações levantadas. Tal proposta inclui o levantamento do problema, bem como o estudo realizado para constatação do mesmo e seus pontos específicos de melhoria, seguindo as referências adotadas.

O Capítulo 4 evidencia os resultados alcançados e observações feitas a partir do desenvolvimento da contribuição.

Finalmente, as conclusões acerca do trabalho aplicado são denotadas no Capítulo 5.

Capítulo 2

Fundamentação Teórica

Neste capítulo são introduzidos os conceitos de *Perfil Operacional* e de *Behaviour Driven Development*, bases que sustentam a ferramenta Feature-Trace. É também necessário ter uma breve noção do propósito e funcionamento do Feature-Trace. Por fim, são apresentados em mais detalhes dois atributos de qualidade de software, a testabilidade e a manutenibilidade de código, objetos da contribuição deste trabalho.

2.1 Perfil Operacional

O Perfil Operacional do Software (POS), para [8] consiste na quantificação da ocorrência das entidades que compõem esse software, sob utilização dos usuários. É caracterizado por uma variedade de ações que o software executa ligado às probabilidades de ocorrência dessa ação ou funcionalidade. Ainda segundo [8], o POS reflete como o software será utilizado na prática e, frequentemente, apresenta as maiores probabilidades ligadas a um número pequeno de classes. Dessa forma, não se trata de uma característica estática, mas sim de algo que se altera conforme o software é utilizado.

Uma alternativa para simular a utilização de um software por um usuário final é a adoção do *Behaviour Driven Development*, que será caracterizado na Seção seguinte.

2.2 Behaviour Driven Development

O *Behaviour Driven Development* (BDD) se trata de um desenvolvimento orientado a comportamento, que tem como foco principal a escrita de testes de aceitação por meio de uma linguagem natural. Essa metodologia envolve a colaboração de *stakeholders* devido à linguagem fácil e próxima à linguagem humana e também colabora para a escrita dos requisitos do software produzido [9].

Testes de aceitação são usados para validar e exercitar os requisitos de um sistema, porém, este tipo de teste é realizado de forma manual com o usuário final, não sendo muito prático e nem viável em larga escala. Desse modo, o BDD visa replicar de forma automatizada o comportamento de um usuário final [10].

Comumente, é utilizada a linguagem *Gherkin* [11], que permite a escrita de cenários que cubram métodos do código fonte e, principalmente, executados de forma automatizada para validar e assegurar o devido comportamento da aplicação.

A Figura 2.1 mostra um exemplo de uma *feature* escrita utilizando a linguagem *Gherkin*. Uma *feature*, isto é, uma descrição em alto nível de uma funcionalidade do sistema, deve ser escrita em um arquivo *.feature*, possuindo um ou mais cenários, que por sua vez descrevem um uso específico do sistema ou regra de negócio envolvida. Estes cenários devem possuir os chamados *steps*, que detalham passo a passo do cenário em questão.

Os *steps* possuem outras palavras-chave relacionadas a si como o *Given* para caracterizar um estado inicial do cenário, *When* para definir uma ação executada no sistema, *Then* para detalhar um resultado esperado dessa ação e *And/But* que servem como sinônimos para as palavras-chave anteriores evitando repetições.

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

  # The second example has three steps
  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

Figura 2.1: Exemplo de um arquivo *.feature* [1]

Uma vez entendidos os conceitos de POS e como o BDD se relaciona com eles, é possível compreender como o Feature-Trace os utiliza para seu funcionamento e objetivo.

2.3 Feature-Trace

O Feature-Trace [1] é um software de código aberto focado em gerar o POS de uma aplicação terceira empregando baixo esforço para tal objetivo e permitindo uma priorização do teste de software. Para tal tarefa, ele aproveita o melhor oferecido pelo POS e pela rastreabilidade entregue pelo BDD. Assim, uma vez de posse do POS é possível fazer uma priorização ou seleção de casos de teste [2].

Estruturalmente, o Feature-Trace é composto por três módulos. São eles: *Automated Runtime Module (ARM)*, *Analyser* e *Visualizer*, mostrados na Figura 2.2 abaixo [2].

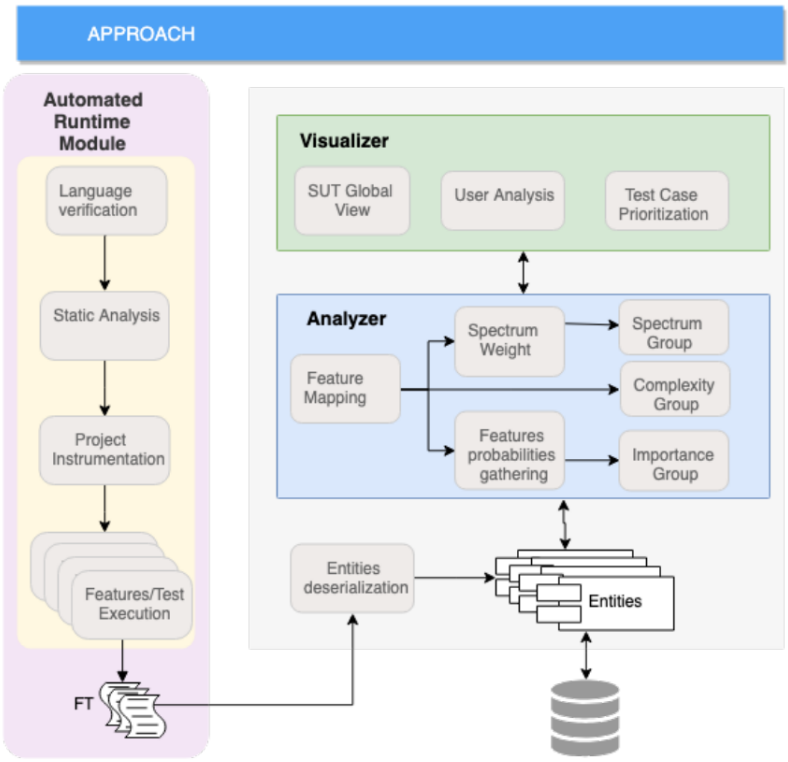


Figura 2.2: Arquitetura do Feature-Trace [2]

O ARM realiza a leitura estática e dinâmica do código do software sob análise, extraindo detalhes como linguagem utilizada, classes, módulos e métodos, bem como os casos de teste presentes e cenários de teste BDD. Todo o seu trabalho se baseia na leitura dos artefatos BDD e do próprio código fonte. Também extrai métricas de complexidade de código como a Complexidade Ciclomática, *ABC score* e número de linhas de código por método. Após essa extração de métricas e relacionamentos, todos os dados são enviados a uma base de informações para a análise e trabalho dos módulos *Analisador* e *Visualizador*.

[1] `https://github.com/BDD-OperationalProfile/trace_feature`

Por sua vez, o módulo *Analisador* mapeia as entidades existentes no software aos métodos da aplicação, assim gerando o POS buscado. Também é responsável por buscar no banco de dados todas as informações geradas pelo trabalho do ARM. O principal *output* deste módulo é um grafo que explicita a relação entre todos os métodos da aplicação com seus cenários e suas *features*.

Finalmente, o módulo *Visualizador* entrega de forma visual todos os dados extraídos pelo módulo Analisador. Por meio dele, é possível constatar as métricas que ajudarão na priorização de casos de teste pela equipe de desenvolvimento.

2.3.1 Arquitetura de Implementação do ARM

Como parte primordial do Feature-Trace, existe o *Automated Runtime Module* (ARM), em destaque vertical na Figura 2.2, responsável por verificar a linguagem do projeto analisado, analisar estaticamente o código e executar a suíte de testes e *features* do projeto em análise.

Na arquitetura interna de módulos do ARM, nota-se a presença de uma organização pensada em facilitar a evolução do software para novas linguagens alvo [2]. A nível de código, cada um desses módulos é representado por uma classe. Tem-se módulos focados na configuração, leitura e execução de projetos em linguagem *Ruby*, *parsing* de arquivos *.feature* e entidades de modelos que dão suporte à persistência de todo o *output* do processo.

Como qualquer análise estática ou dinâmica depende diretamente da linguagem utilizada pela aplicação sob análise [2], o primeiro passo da ferramenta consiste na detecção da linguagem de implementação e da checagem e configuração de dependências. Este papel é desempenhado pelo módulo *RubyConfig*, que leva em consideração alguns padrões do framework *Ruby On Rails* [2], como estrutura de pastas e arquivos específicos de configuração. Tal módulo realiza leitura e escrita em arquivos de configuração da aplicação sob análise e utiliza recursos de metaprogramação para executar comandos no mesmo.

O papel de fazer a leitura dos métodos da aplicação sob análise e extração de informações como conteúdo do método, número da linha do arquivo em que é definido, quantidade de linhas e métricas como ABC *score* e Complexidade Ciclomática, fica a cargo do módulo *ReadMethods*. A partir de todos os parâmetros e métricas levantados, o módulo instancia objetos relativos a cada método presente na aplicação, que são enviados para a base de dados.

O módulo *RubySpecExecution* desempenha a função de percorrer todos os arquivos de testes da aplicação sob análise, lendo e instanciando objetos para cada teste implementado. Estarão presentes nesses objetos, detalhes como caminho do arquivo, número da linha em que é declarado, descrição do caso de teste e métodos impactados.

[2] `https://rubyonrails.org/`

Finalmente, um dos principais módulos da ferramenta, o *RubyExecution*, contém métodos que executam a análise dinâmica da aplicação sob análise. Esse módulo conta com o auxílio dos outros módulos já mencionados e seus métodos compreendem as seguintes tarefas, que são executadas a depender da intenção do usuário:

- Execução de todos os testes da aplicação e levantamento de métodos impactados;
- Execução de um teste em específico e levantamento dos métodos impactados;
- Execução de todas as *features* e levantamento dos métodos impactados;
- Execução de uma *feature* em específico e levantamento dos métodos impactados; e
- Execução de um cenário em específico e levantamento dos métodos impactados.

Após entender do que se trata o Feature-Trace e sua arquitetura de implementação, é necessário introduzir os conceitos que sustentam a contribuição deste trabalho de conclusão de graduação.

2.4 Atributos de Qualidade do Software

Um atributo de qualidade do software é a propriedade mensurável ou testável de um sistema e é utilizado para indicar o quão bem este sistema satisfaz as necessidades de seus *stakeholders* [3]. De forma mais prática, é possível pensar em um atributo de qualidade como uma medição de utilidade do produto em determinada perspectiva de interesse do *stakeholder*. A ISO/IEC 9126 [12] classifica os atributos de qualidade de software dentre os seguintes: *Funcionalidade*, *Manutenibilidade*, *Usabilidade*, *Eficiência*, *Confiabilidade* e *Portabilidade*.

Essas características são atributos de qualidade que podem descrever um sistema de software. Esses atributos de qualidade são desdobrados ainda em subcaracterísticas com mais atributos. E essas subcaracterísticas são refinadas a atributos ou propriedades mensuráveis usando várias métricas.

Ainda seguindo a ISO/IEC 9126 [12], a manutenibilidade é composta pelas seguintes subcaracterísticas: *expansibilidade*, *modificabilidade* e *testabilidade*.

A partir disso, é apropriado descrever em mais detalhes a manutenibilidade, como é feito no subtópico a seguir.

2.4.1 Manutenibilidade

A *Manutenibilidade* é definida como a facilidade com a qual um software pode ser modificado [4] [5]. Este atributo de qualidade do software está diretamente ligado a atividades

de manutenção e refatoração, que constituem a maior parte do custo total do ciclo de vida de um sistema [4]. Sendo assim, a indústria de software se encontra em uma árdua busca por meios de aumentar a manutenibilidade de seus produtos.

A manutenção, componente da manutenibilidade, consiste em quatro categorias, sendo elas [4] [13]: corretiva, perfeita, adaptativa e preventiva. Assim, observando o contexto da ferramenta Feature-Trace e os objetivos traçados para esse trabalho, os tipo de manutenção que mais se encaixam são a manutenção corretiva e a preventiva. Isto pois, a manutenção corretiva, como sugere o nome, foca em reparar defeitos presentes no software. Já a manutenção preventiva visa melhorar a estrutura do software para também melhorar sua futura manutenibilidade [4].

Existem diversas técnicas para manutenção de código, porém a ação de fato aplicada em todas as atividades de manutenção é a refatoração, processo esse que consiste na mudança da estrutura de código sem mudar a funcionalidade que este implementa. Desse modo, a refatoração contínua aumenta a manutenibilidade [4].

Há duas maneiras de aplicar mudanças em um software [14]: *Editar e rezar* ou *Cobrir e modificar*. No primeiro caso, que ocorre na maioria das vezes, é necessária uma familiarização com a parte do código-fonte que se deseja modificar, modificá-lo e validar manualmente se nenhuma funcionalidade foi quebrada (o que é difícil de ser garantido). Finalmente a aplicação é atualizada e apenas resta rezar para que tudo continue em perfeito funcionamento após as modificações feitas. Obviamente, esta não é uma abordagem usual para softwares atuais e de grandes proporções. Em uma via totalmente oposta, a abordagem *Cobrir e modificar* demanda a criação de testes ou ampliação da suíte de testes já existente de modo a cobrir o código que será modificado. Desse modo, é possível detectar qualquer comportamento inesperado pelas alterações aplicadas.

Finalmente, se o desenvolvedor se depara com um código de difícil entendimento ou não testado, [4] propõe seguir os seguintes passos:

1. Identificar os pontos passíveis de mudança. Para isso, pode ser executada a aplicação localmente replicando histórias de usuários, examinando o relacionamento entre as principais classes ou quantificar a qualidade do código e a cobertura de testes.

2. Desenvolver *testes de caracterização*, responsáveis por registrar como esta porção do software se comporta atualmente.

3. Determinar se os pontos a serem mudados necessitam de refatoração para torná-los mais testáveis ou comportarem as mudanças planejadas.

4. Aplicar as mudanças após as refatorações e estabelecida uma cobertura de testes satisfatória.

Para auxiliar na identificação dos pontos passíveis de mudança, métricas de software e *code smells*, que são indícios de possíveis problemas de código, são úteis. Uma vez identificados, serão *inputs* para as refatorações, que quando implementadas, melhorarão as métricas e eliminarão tais *code smells* [4]. Testes de caracterização serão responsáveis por assegurar o comportamento atual do sistema. Durante o processo, trechos de código podem ser logo refatorados, de modo a se tornarem mais testáveis e facilitarem as alterações futuras. E, finalmente, uma cobertura de testes ajudará a assegurar um software mais confiável e manutenível.

Nos subtópicos a seguir, são fundamentados alguns conceitos relacionados a esses passos mencionados.

Métricas de software

Código bem estruturado é mais fácil e menos custoso para manter, afirmam Fox e Patterson [4]. Por sua vez, a qualidade de um código pode ser analisada quantitativamente por meio da análise de *métricas de software*, sendo parâmetros muito úteis para avaliar a complexidade e tamanho de uma aplicação.

As métricas de software são usualmente medidas em relação à complexidade do código e que estimam a dificuldade de testar determinado trecho. Dentre as métricas mais utilizadas, encontram-se a *Complexidade Ciclomática* e o *ABC score*.

Proposta em 1976 por McCabe [15], a Complexidade Ciclomática está fundada na teoria de grafos e mede a quantidade de caminhos lineares e independentes em um trecho de código. Quanto mais alta essa medida, maior a complexidade do código analisado. Para este trabalho, foi adotado um valor alvo para a Complexidade Ciclomática de até 10 por método, com base no trabalho de [4]. Ou seja, métodos com uma Complexidade Ciclomática que ultrapasse um valor 10, são considerados complexos.

Outra métrica de software muito útil é o *ABC score*. Ela provê uma medida relativa ao tamanho do software e foi proposta para substituir as desvantagens de se analisar puramente a quantidade de linhas de código [16]. Tal pontuação leva em conta três componentes que podem fazer parte de um trecho de código: atribuições, saltos e trocas de contexto do programa. Na prática, as atribuições tomam forma de declaração ou atualização do valor de uma variável, saltos como chamada de funções e condicionais por diretivas *if, else* e similares. Essa métrica pode ser representada por meio de um vetor de três componentes A (*Assignments*, B (*Branches*) e C (*Conditionals*), ou simplesmente por um valor escalar obtido pelo cálculo da Magnitude deste vetor. Para este trabalho, foi adotado um valor alvo para o *ABC score* de até 20 por método, com base no trabalho de [4]. Ou seja, métodos com um *ABC score* em Magnitude maior que este valor podem ser considerados complexos e devem ter a sua implementação revista.

Code Smells

Qualitativamente, também é possível analisar um software por meio da procura por *code smells*, que chamam a atenção para possíveis pontos de problema [4]. Tal termo foi pela primeira vez sugerido por Martin Fowler [6] em seu livro dedicado ao tema de refatoração, onde o autor lista 22 tipos de *code smells*. *Code smells* são características estruturais do código não capturadas pelas métricas quantitativas mencionadas e indicam sintomas de um código com más decisões de design e implementação, afetando assim negativamente a manutenibilidade de um software.

Quatro desses *code smells* listados por Matin Fowler são enfatizados por [4], por se tratarem de sintomas que podem ser solucionados por meio de refatorações simples. Destas quatro derivam o acrônimo em inglês *SOFA*, que estabelecem que um método adequando necessita:

- ser pequeno (S), para manter seu propósito e foco;
- executar apenas uma (O) responsabilidade, para que o teste possa focar em executar apenas isto;
- receber poucos (F) argumentos , para que todas as combinações de argumentos possam ser testadas;
- manter um grau de abstração (A) consistente, para que o mesmo não seja responsável por dizer "o que fazer"e "como fazer".

Testes de caracterização

Testes de caracterização descrevem o comportamento atual do sistema ou trecho da aplicação, mesmo se o mesmo apresentar "*bugs*", pois o intuito é que seja assegurado que as mudanças planejadas no software não alterem o cenário atual do sistema [14].

Os testes de caracterização podem ser feitos em nível de integração, que levam em conta apenas o comportamento prático, sem se preocupar em como esse comportamento é implementado. Por outro lado, também podem ser pensados em nível unitário, que requerem um maior entendimento da implementação do código testado.

Feathers [14] menciona uma técnica muito útil de "engenharia reversa"para testar um fragmento de código difícil de se entender. Ela consiste em criar um teste com uma asserção que provavelmente falhará, executar o teste e utilizar a informação retornada no erro para alterar o teste e deixá-lo correto. Essa abordagem ajuda a alcançar uma cobertura de testes ideal próxima de 100%, já que o principal objetivo é cobrir o comportamento atual da aplicação.

Cobertura de testes

Cobertura de código consiste em quantificar o grau de abrangência dos testes em relação ao código fonte que por ele é executado. Isto é, a porcentagem de código do sistema que possui um teste associado [17].

Existem várias ferramentas, para inúmeras linguagens de programação, que permitem levantar essa métrica de cobertura de código testado [17].

Um dos principais objetivos para a codificação de testes é assegurar que o código que implementa uma funcionalidade tenha seu comportamento mantido ao longo de todo o projeto. Para verificar isso, deve-se ter a certeza de que o trecho de código considerado possui um teste que o exercita [18]. Para isso, utiliza-se as ferramentas de cobertura de código.

Existem algumas classificações formais de cobertura de código definidas por [4]. Uma delas, a cobertura a nível de Métodos, que avalia se cada método é executado pelo menos uma vez pelo conjunto de testes. Este conceito foi aplicado neste trabalho, para buscar um grau de cobertura satisfatório, visando abranger o maior número possível de métodos implementados na ferramenta Feature-Trace.

Porém, antes da implementação de fato da cobertura de testes, é necessário analisar se o software é testável ou se possui um bom grau de testabilidade.

2.4.2 Testabilidade

A testabilidade de um software se refere à facilidade com que tal software pode ser concebido para demonstrar suas falhas por meio de testes [3]. Desse modo, um sistema é considerado testável se o mesmo revela suas falhas facilmente, ou seja, se uma falha está presente no software, é desejável que ele falhe durantes os testes o mais cedo possível. Não somente isso, a arquitetura do software pode aprimorar a testabilidade fazendo com que seja mais fácil replicar uma falha ou mesmo chegar à raiz da causa da falha.

Para um sistema ser propriamente testável, deve ser possível controlar cada componente de entrada e observar cada saída produzida [3]. A figura abaixo mostra um exemplo simples de modelo de teste, onde o programa processa entradas e produz saídas. Um oráculo é uma entidade, humana ou computacional, que determinada se a saída é correta comparando esta com o resultado esperado. A saída pode não se tratar apenas do valor produzido da funcionalidade, mas também pode incluir outras medidas derivadas como até mesmo quanto levou para produzir a saída.

Uma abordagem possível, como mostrado na Figura 2.4, é que a arquitetura interna do programa pode ser mostrada para o oráculo, assim ele pode decidir se o estado está

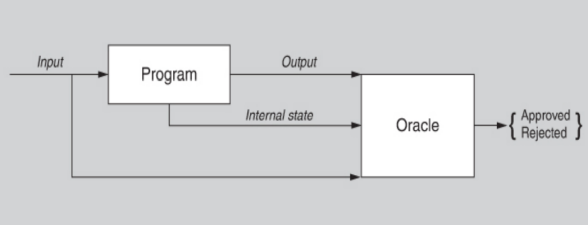


Figura 2.3: Modelo de processo de teste [3]

correto, isto é, se é possível detectar se o programa acessou um estado incorreto e sinalizar um julgamento para a correção do programa.

Mais especificamente, há duas categorias de estratégias para desenvolver a testabilidade de um software, tornando o software mais fácil e eficiente de ser testado [3]. A seguir, é detalhada melhor a categoria selecionada para a contribuição deste trabalho.

Teste de software não se trata apenas de fazer o software falhar, mas também sobre achar o erro que causa a falha para que possa ser removido. Assim, um software complexo é mais difícil de ser testado [3], pois apresentando contextos extensos, é mais difícil de replicá-los do que no caso de um contexto pequeno.

Desse modo, evitar ou resolver dependências exageradas entre componentes, isolar e encapsular dependências em ambiente externo, pode ajudar a reduzir a complexidade. Por exemplo, em um contexto de sistemas orientados a objeto é possível:

- Limitar o número de classes a partir da qual outras classes são derivadas.
- Limitar a profundidade da árvore de herança e o número de filhos de uma classe.
- Limitar o polimorfismo e chamadas dinâmicas.

Isto é, a aplicação de técnicas de modificabilidade como aumentar a coesão, diminuir o acoplamento e separar responsabilidades, também podem auxiliar na testabilidade do software [3]. Essas características limitam a complexidade do elementos presentes, uma vez que limitam suas interações e aumentam seu foco.

Esse conceito está bastante ligado a alguns parâmetros de complexidade e alguns deles adotados para este trabalho, são definidos a seguir.

Acoplamento

Uma mudança que afeta apenas um módulo é mais fácil e menos custosa do que uma mudança que afete mais de um módulo [3]. O *acoplamento* entre dois módulos caracteriza o quanto estes têm responsabilidades sobrepostas e, dessa forma, serão afetados por uma mudança. Um acoplamento alto é inimigo da modificabilidade, ou seja, ao reduzir o

acoplamento entre dois módulos também será reduzido o custo de qualquer modificação que afete qualquer um destes módulos.

Para reduzir o acoplamento entre módulos, é sugerido, entre outras táticas, o *encapsulamento* [3], que foi adotado neste trabalho na refatoração de classes e métodos muito extensos.

Coesão

Este parâmetro indica o quanto as responsabilidades de um módulo estão relacionadas. Desse modo, a coesão de um módulo é a probabilidade de que a mudança de um cenário que afete uma responsabilidade deste módulo também afete outras responsabilidades. Assim, quanto maior a coesão, menor a probabilidade de que uma determinada mudança afete múltiplos módulos. Isto acaba sendo muito positivo para uma boa modificabilidade do software [3].

Porém, se um módulo possui uma baixa coesão, então ela pode ser aprimorada por meio da remoção de responsabilidades não afetadas por mudanças antecipadas. Algumas táticas para aumento de coesão são mencionadas e foram também aplicadas neste trabalho, como a *divisão de módulos* e a *redistribuição de responsabilidades*.

Portanto, se um módulo a ser modificado possui muitas responsabilidades que não possuem coesão, então o custo para modificação será alto [3]. Assim, refatorar este módulo em outros módulos mais coesos irá reduzir o custo médio de alterações futuras. De modo análogo, se responsabilidades similares estão espalhadas por diferentes módulos, é oportuno reuni-las. Esta refatoração pode consistir na criação de um novo módulo, ou na migração dessas responsabilidades para módulos já existentes.

Tamanho de um módulo

Módulos muito grandes são difíceis e mais custosos para aplicar alterações, além de mais propícios a conterem *bugs* [3].

Esta característica também é mencionada por [6]. Fowler define que quando um módulo ou classe assume muitas responsabilidades, frequentemente aparecem variáveis de instância demais. Quando esta classe possui variáveis de instância demais, código duplicado também certamente estará presente.

A partir da descrição de quais as características de um software testável, cabe também descrever o próprio processo de implementação de testes, como se dá a seguir.

Testes Caixa-branca

O teste de software é definido como o processo de executar o programa com a intenção de encontrar erros, ou seja, uma atividade destrutiva, afirma Myers [19]. Por outro lado, o teste de software eventualmente é usado para estabelecer um grau de confiabilidade de que o software faz aquilo que deveria fazer e não faz aquilo que não se espera que faça [19].

Uma vez entendido o conceito de teste de software, é necessário determinar se é possível testar um programa para encontrar todos os seus erros. Em geral, isso é impraticável e impossível, segundo Myers [19]. Assim, esse problema tem impactos no custo dessa atividade, premissas que o testador deve assumir e a maneira como os casos de teste serão modelados [19]. Portanto, uma estratégia de teste de software deve ser escolhida desde o início. Dentre as mais predominantes estão o teste de caixa-preta e o teste de caixa-branca.

A estratégia de teste caixa-branca parte de examinar a estrutura interna do software [19]. Isto é, os dados de teste são derivados da examinação da lógica do problema e aplicados por meio de casos de teste que exercitem ou cubram determinado trecho de código do software em questão. Uma vez que testar todos os caminhos possíveis em um software é impossível, de acordo com Myers [19], deve-se aplicar metodologias que permitam testar o mais completamente possível.

Dentro da estratégia de testes caixa-branca, existem as seguintes metodologias, listadas por Myers [19]: Cobertura de Instrução (*statement*), Cobertura de Decisão, Cobertura de Condição, Cobertura de Decisão/Condição e Cobertura de Múltipla Condição. Todos esses se baseiam na cobertura da lógica do código-fonte do software.

Algumas dessas metodologias foram selecionadas e combinadas, assim como sugerido por [19], com o objetivo de proporcionar um rigoroso teste de programa. Elas são definidas a seguir.

A chamada *cobertura de decisão* estabelece que deve-se escrever casos de testes suficientes para que cada decisão resulte em, pelo menos uma saída positiva e outra negativa. Isto é, cada caminho da aplicação deve ser percorrido ao menos uma vez. Frequentemente, a cobertura de decisão também implica em uma cobertura de instrução, desde que cada instrução faça parte de algum sub caminho.

Na *cobertura de condição* deve-se escrever casos de teste suficientes para garantir que cada condição em uma decisão assuma todos os casos possíveis pelo menos uma vez, sendo essa abordagem muitas vezes mais consistente que uma cobertura de decisão.

Capítulo 3

Fundamentação da Proposta

Este capítulo apresenta os detalhes da proposta deste trabalho, compreendendo o ponto de partida, metodologia adotada e sua devida aplicação.

3.1 Contexto inicial

A ferramenta Feature-Trace se dedica a fazer a análise de softwares codificados na linguagem Ruby. Essa é uma tarefa complexa, uma vez que é necessário ler e interpretar o código de outra aplicação. Isso se traduz então em uma implementação do software Feature-Trace que não é de trivial entendimento.

Nesse cenário, é ainda mais desafiador, por exemplo, realizar qualquer tipo de manutenção ou melhoria no projeto, pois o software não está idealmente alinhado com boas práticas de projeto e padrões de codificação que facilitariam tais modificações. Desse modo, notou-se que o software carece de um maior grau de manutenibilidade.

Tal cenário se agrava devido ao fato do software também não possuir qualquer tipo de suíte de testes que ajude a reforçar a confiança na implementação de suas funcionalidades.

A seguir é proposta uma abordagem para mitigar as dificuldades mencionadas acima.

3.2 Proposta

Para atingir e minimizar as dificuldades citadas a respeito da manutenibilidade e testabilidade do Feature-Trace, é proposto um fluxo de trabalho baseado nas técnicas e métricas descritas na Seção 2. Ao final é esperado constatar o aumento da manutenibilidade e testabilidade de seu ARM (*Automated Runtime Module*). Desse modo, tem-se como principais objetivos práticos da contribuição os seguintes pontos:

- Tornar menos custoso o processo de manutenção e evolução do software por parte de novos desenvolvedores;
- Atualizar a implementação de código do software de acordo com padrões de codificação seguidos pela comunidade de desenvolvimento;
- Prover casos de teste para a implementação das funcionalidades entregues pela ferramenta.

Assim, uma vez que forem alcançados esses objetivos propostos, pode-se trazer uma melhora importante para a ferramenta à nível de desenvolvimento.

3.3 Metodologia adotada

A metodologia elaborada para este trabalho visa trazer para a ferramenta os dois objetivos gerais propostos de aprimorar a manutenibilidade e a testabilidade do Feature-Trace. Para isso, foram aplicados os seguintes passos:

1. Definiram-se os objetivos práticos listados na seção 3.2 anterior, a partir do contexto inicial de baixa manutenibilidade e testabilidade da ferramenta;
2. Adotou-se a abordagem *Cobrir e modificar* proposta para manutenção visando a manutenibilidade por [14] [4]. Isto é, identificar pontos de mudança, criar testes que cubram o comportamento atual e aplicar refatorações que melhorem a testabilidade;
3. Para os casos com necessidade de aumento da testabilidade, aplicou-se a abordagem de limitar a complexidade do código, sugerido por [3], observando os parâmetros de acoplamento, coesão e tamanho de módulo;
4. Implementou-se mais testes para completar uma suíte automatizada seguindo a modalidade de testes caixa-branca e uma cobertura de decisão e condição, ambos descritos por [19];
5. Verificou-se o impacto alcançado pela proposta de melhoria da manutenibilidade e testabilidade por meio da comparação de aspectos e métricas antes e depois.

Por fim, a melhoria da manutenibilidade e testabilidade da ferramenta é essencial para a longevidade do projeto, mesmo que isso não seja claro em primeiro momento.

3.4 Abordagens e resultados esperados

Uma vez descritos os objetivos práticos pretendidos e a metodologia adotada, é também interessante traçar um paralelo entre esses objetivos e as ações de fato aplicadas. Isso é melhor estabelecido na Tabela 3.1 abaixo.

Tabela 3.1: Sumário geral dos objetivos, ações e resultados esperados

Objetivos práticos	Atributos	Contribuições	Resultados Esperados
Tornar menos custoso o processo de manutenção e evolução do software por parte de novos desenvolvedores.	• Manutenibilidade; • Testabilidade.	• Diminuição da complexidade do código; • Eliminação de *code smells*.	Verificar a melhora das métricas e qualidade do código.
Atualizar a implementação de código do software de acordo com padrões adotados pela comunidade de desenvolvimento.	• Manutenibilidade; • Testabilidade.	• Eliminação de *code smells*; • Implementação de suíte de testes.	Código atendendo a padrões e boas práticas, além de diminuir dificuldade de contribuições.
Prover maior confiabilidade na implementação das funcionalidades entregues pela ferramenta.	• Testabilidade	• Diminuição da complexidade do código; • Implementação de suíte de testes.	Software com implementação confiável e verificada.

Uma vez correlacionadas as contribuições aplicadas com os objetivos práticos e atributos de qualidade correspondentes, é necessário detalhar melhor cada uma dessas contribuições aplicadas no software.

3.5 Diminuição da complexidade

Como já foi ressaltado, o Feature-Trace não possui um estudo a respeito do seu grau de manutenibilidade, o que dificulta a evolução do software e a implementação de possíveis mudanças por parte de novos desenvolvedores. Além disso não há uma certeza de que o software foi codificado com a preocupação e foco em manter uma baixa complexidade.

Visando esse problema e para iniciar a contribuição deste trabalho, assim como sugerido por [4], foram analisadas as métricas de software do Feature-Trace, neste caso a Complexidade Ciclomática e o *ABC score*, para determinar possíveis pontos de melhoria. Assim como explicado no Capítulo 2, uma vez que essas métricas de software possam ser melhoradas, haverá uma evolução na manutenibilidade e testabilidade do software [4] [3].

A próxima sessão detalha o processo de levantamento e avaliação das métricas de Complexidade Ciclomática e *ABC score* do código que implementa a ferramenta Feature-Trace.

3.5.1 Avaliação da Complexidade Ciclomática

Para o levantamento da métrica de Complexidade Ciclomática a linguagem *Python* possui um pacote chamado *Radon*[1]. Este pacote consegue analisar as funções, métodos e até mesmo classes presentes no código, retornando valores que os caracterizam quanto a sua complexidade de acordo com a tabela 3.2 a seguir, disponível na documentação de referência do pacote.

Tabela 3.2: Radon: Intervalos de complexidade

Intervalo de notas	Complexidade
1 - 5	baixa - bloco simples
6 - 10	baixa - bloco bem estruturado
11 - 20	moderada
21 - 30	mais que moderada
31 - 40	alta
41+	muito alta

Uma vez instalada a ferramenta *Radon*, a mesma foi executada de uma só vez em todo o projeto, revelando assim a complexidade de cada classe e método que compõem o

[1] `https://pypi.org/project/radon/`

Feature-Trace. O *output* desse processo de avaliação está claro na Tabela 3.3, que mostra a contagem de ocorrência de tais notas. Nesse caso, foram consideradas apenas as notas relativas a métodos e funções.

Tabela 3.3: Contagem de ocorrências de notas para Complexidade Ciclomática

Intervalo de notas	Número de ocorrências
1 - 5	65
6 - 10	6
11 - 20	1

Tendo como referência a Tabela 3.2, observa-se que quase a totalidade dos métodos apresentam uma Complexidade Ciclomática considerada como baixa, refletida pela presença de blocos simples ou bem estruturados. Apenas uma estrutura de código apresentou uma complexidade considerada moderada, que se trata da função *trace*, presente no módulo principal da aplicação. Com um valor de Complexidade Ciclomática igual a 12, esse método também pode ser considerado complexo, baseando-se no limitar sugerido por [4], que é de no máximo 10 por método. Analisando-se a natureza dessa função, a mesma é responsável por receber os argumentos do usuário, via linha de comando, para execução das funcionalidades da ferramenta. Por esse motivo, se trata de uma função que implementa muitas responsabilidades, o que eleva a sua complexidade.

3.5.2 Avaliação do *ABC score*

De modo semelhante ao caso da Complexidade Ciclomática, a linguagem Python também possui um pacote, chamado *Python ABC* [2] , que permite o levantamento da métrica de *ABC score* do código de um software, para cada módulo presente.

Utilizando este pacote para análise do *ABC score*, foi produzida a tabela 3.4 abaixo, que demonstra a quantidade de *Assignments* (A), *Branches* (B) e *Conditionals* (C) por módulo do Feature-Trace.

Tabela 3.4: *ABC score* por módulos do Feature-Trace

Módulo	<A, B, C>	Magnitude
RubyExecution	<79, 109, 31>	138,1
ReadMethods	<49, 65, 22>	84,3
RubyConfig	<43, 53, 42>	75,7
GherkinParser	<33, 35, 2>	48,1
Models	<38, 16, 0>	41,2
RubySpecExecution	<13, 10, 5>	17,1
SpecModels	<11, 4, 0>	11,7

[2] `https://github.com/eoinnoble/python-abc`

Olhando em mais detalhes cada módulo e levando-se em conta o limiar máximo de *ABC score* por método sugerido por [4], podem ser considerados complexos os seguintes métodos mostrados na Tabela 3.5 abaixo, sendo portanto fortes candidatos a refatorações.

Tabela 3.5: Métodos complexos de acordo com o *ABC score*

Módulo	Método	Magnitude
ReadMethods	read_methods	24,61
RubyConfig	*check_environment*	23,6
-	*trace*	37,76

3.5.3 Acoplamento e coesão

Como descrito na Seção 2.5.2, o acoplamento representa o grau de dependência entre módulos e, portanto, está diretamente ligado à complexidade do software.

Para realizar esse levantamento a respeito da Feature-Trace, foi utilizado o pacote *Pydeps* [3]. Uma vez instalado, é possível fazer uma leitura do software e representar cada módulo e seus relacionamentos por meio de um grafo. Assim, a partir desse grafo produzido, é possível ter uma ideia sobre o relacionamento dos módulos da aplicação. Finalmente, a Figura 3.5 apresenta a configuração de acoplamento dos módulo do Feature-Trace.

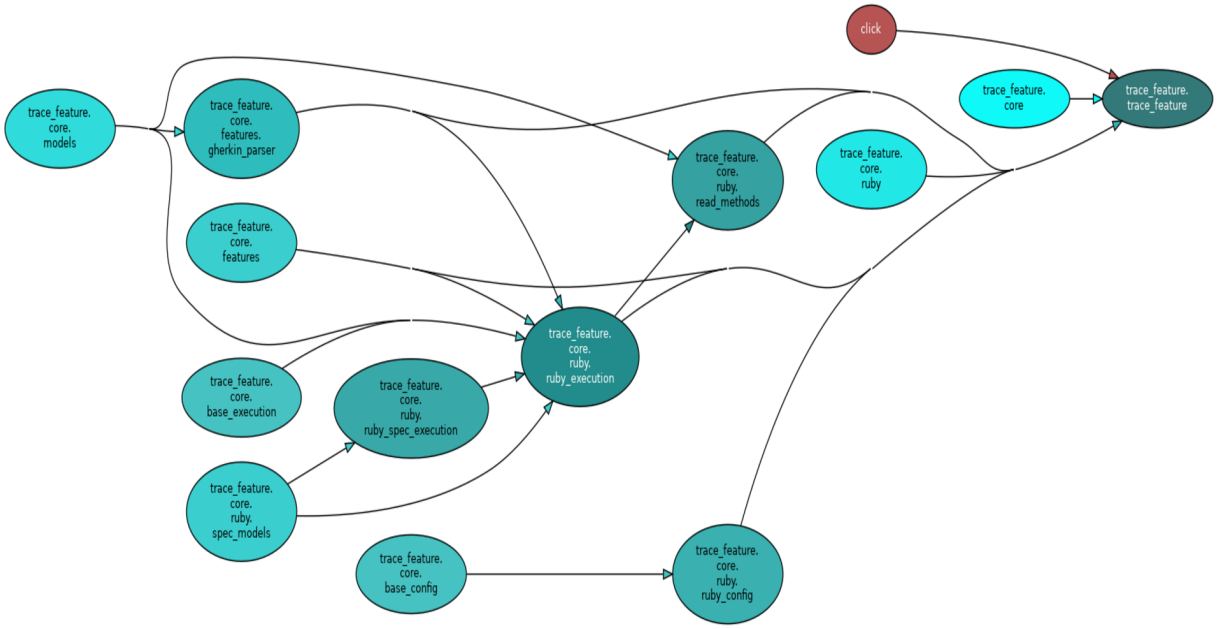


Figura 3.1: Acoplamento inicial do Feature-Trace

[3] `https://github.com/thebjorn/pydeps`

Esforços foram aplicados para reavaliar as responsabilidades de cada módulo e seus métodos, o que também está ligado à coesão. Uma vez aplicadas refatorações com esse foco, é esperado que o acoplamento diminua e a coesão aumente. Tal tarefa também contribui para a redução do tamanho em linhas de código dos módulos, outro aspecto de complexidade também já discutido.

3.6 Eliminação de *code smells*

Para a análise de *code smells* na aplicação, foi utilizado o pacote *Pylint* [4], uma ferramenta de linha de comando disponível para a linguagem Python. O Pylint também tem como princípios adicionais checar por erros no código e encorajar o uso de um padrão de codificação [6], que se baseia no guia de estilo *PEP 8* [5], muito adotado pela comunidade de desenvolvimento em linguagem Python.

A execução do Pylint pode ser feita por meio do comando abaixo, que fará a análise de todos os arquivos do software e retornará uma série de mensagens, classificadas em alertas ou erros, e seu número de linha de código correspondente, bem como uma nota geral para a qualidade do código.

```
1 pylint trace_feature
```

Listing 3.1: Executando o Pylint

Executando o Pylint no repositório de origem do Feature-Trace disponível no Github[6], foram retornadas 191 linhas de erros ou alertas referente ao código e uma nota geral igual a 7,38. Tais mensagens tinham relação com os seguintes aspectos e padrões recomendados pela *PEP 8*:

- Linha de código muito longa;
- Linha com caractere em branco;
- Falta de descrição para módulos;
- Falta de descrição para classes;
- Falta de descrição para métodos;
- Muito argumentos por método;
- Classe muito longa;

[4] https://pypi.org/project/pylint/
[5] https://www.python.org/dev/peps/pep-0008/
[6] https://github.com/BDD-OperationalProfile/trace_feature

- Redefinição de variável;
- Biblioteca importada, mas não utilizada;
- Variável declarada, mas não utilizada;
- Nome de variável fora do padrão *snake case*;
- Indentação incorreta; e
- Trechos de código duplicados

Para auxiliar no entendimento do esforço aplicado na eliminação de *code smells*, a Tabela 3.6 abaixo reúne, por módulo, quais os *code smells* relevantes e qual a abordagem utilizada para eliminar. Isto é feito pois nem todos os *code smells* encontrados implicam em maior complexidade de código, foco da análise deste trabalho.

Tabela 3.6: *Code smells* presentes por módulo e abordagem para solução

Code smell	Módulos	Abordagem de solução
Muitos argumentos para um método.	*TraceFeature*	Reduzir a quantidade de argumentos recebidos pelo método.
Muitos atributos de instância para uma classe.	*Models*	Reduzir o número de atributos de instância por classe.
Método poderia ser uma função.	*RubyConfig* *RubyExecution*	Migrar alguns métodos para funções, uma vez que não dependem da classe que os implementa.
Muitas variáveis locais.	*RubyConfig*	Migrar alguns métodos para funções, uma vez que não dependerem da classe que os implementa.
Classe muito longa.	*RubyConfig* *RubyExecution* *Models* *GherkinParser*	Redistribuir as responsabilidades entre módulos, tornando-os menores e mais focados.

Trechos de código duplicados.	*RubyConfig* *RubyExecution*	Eliminar a duplicidade, deixando apenas um e reutilizá-o.

Assim, esforços foram direcionados para eliminar tais *code smells* do projeto, bem como deixá-lo de acordo com as convenções da *PEP 8*, que também compreende a resolução de outros *code smells* ligados a padrões de código.

3.7 Implementação de suíte de testes

Dado que a ferramenta Feature-Trace não possuía uma suíte de testes automatizados, foi necessária a implementação de uma cobertura de testes, composta por testes unitários e de integração, com o intuito de verificar a corretude dos métodos implementados e garantir o comportamento original do software antes de qualquer refatoração [14].

A abordagem utilizada aqui foi a de *cobertura de decisão* e *cobertura de condição*, descritas no final da Seção 2.4.2, uma vez que ambas quando combinadas oferecem uma boa variedade de casos de testes para validação do código.

Como solução para a implementação de testes utilizando a linguagem Python, foi adotado o framework *Pytest*[7], que permite uma escrita simplificada de testes desde os mais simples aos mais complexos. Aliado a este, também foi utilizado o plugin *pytest-cov*[8], que permite a produção de relatórios acerca da cobertura de testes atingida a partir da suíte de testes implementada.

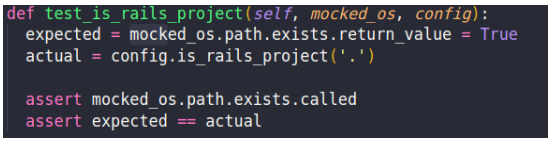
Após o estudo das estratégias utilizadas pelo Pytest para a descoberta de testes a serem executados, foi necessária a adoção de uma convenção para organização dos testes em módulos.

Sendo assim, cada módulo/classe do ARM possui uma classe de teste correspondente, que é prefixada pelo termo *Test* e cada método implementado por esse módulo/classe, também possui seu método de teste correspondente, que é prefixado pelo termo *test_*. Tal *pattern* está exemplificado nas Figuras 3.6 e 3.7.

Uma vez definido o *pattern* e implementados os testes para classes e métodos da aplicação, é necessário executar a suíte de testes presente. Essa é feita pelo comando abaixo, que executará os testes implementados e fornecerá um *output* da cobertura de testes atingida, por meio do plugin *pytest-cov*.

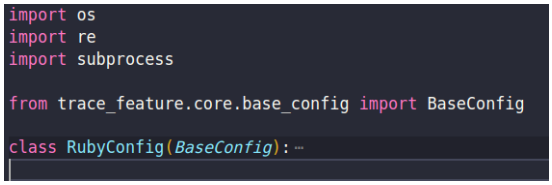
[7] `https://docs.pytest.org/en/6.2.x//`
[8] `https://pypi.org/project/pytest-cov/`

(a) Método *is_rails_project*



(b) Método *test_is_rails_project*

Figura 3.2: *Pattern* para implementação do teste de métodos



(a) Classe *RubyConfig*

(b) Classe *TestRubyConfig*

Figura 3.3: *Pattern* para implementação do teste de classes

```
1 pytest trace_feature -v --cov -q
```

Listing 3.2: Executando a suíte de testes com o Pytest

Capítulo 4

Resultados Alcançados

Este capítulo apresenta os resultados obtidos a partir da implementação mostrada no capítulo anterior e dificuldades encontradas durante o processo.

4.1 Solução de *Code Smells*

Como mencionado no capítulo anterior, a execução do pacote *Pylint* para análise do projeto Feature-Trace, antes de qualquer alteração no código, apresentou um nota geral igual a 7,38, considerando todos os módulos em conjunto. Assim, mostrou-se que o projeto apresentava *code smells* relevantes a serem removidos. Tais *code smells* representavam tanto a falta de boas práticas de código, quanto aspectos que acarretavam em maior complexidade.

Nesse processo de refatoração para a remoção dos alertas indicados pelo *Pylint*, todos os módulos foram modificados, ou seja, cada um deles possuía pelo menos um dos pontos listados acima. Portanto, após essas melhorias, a execução do *Pylint* indicou uma nota geral para o código igual a 10, evidenciando a remoção destes *code smells*.

A Tabela 4.1 a seguir sumariza todos os *code smells* eliminados, bem como seu impacto provocado.

Tabela 4.1: *Code smells* eliminados e impacto provocado

Code smells eliminados	Impacto
• Linha de código muito longa; • Falta de descrição para módulos; • Falta de descrição para classes; • Falta de descrição para métodos; • Biblioteca importada, mas não utilizada; • Variável declarada, mas não utilizada; • Nome de variável fora do padrão *snake case*; • Indentação incorreta;	Adequação ao *style guide* da linguagem e a boas práticas de código.
• Muitos argumentos por método; • Método muito longo; • Classe muito longa; • Redefinição de variável; • Trechos de código duplicados.	Diminuição da complexidade atrelada ao código.

Aliado a isso, algumas lógicas puderam ser encapsuladas, permitindo um reuso de código eficiente. Como pode ser visto na Figura 4.1 com a criação do método *execute_command*, que encapsula a lógica de execução de comandos em um projeto analisado. Dessa forma, foi removida a repetição de código que existia em vários pontos do software para essa mesma funcionalidade.

```
346     def execute_command(command):
347         process = subprocess.Popen(command.split(), stdout=subprocess.PIPE)
348 
349         test_message = process.communicate()[0]
350         test_message = test_message.decode('utf-8')
351         print(test_message)
352 
353         return test_message
```

(a) Implementação do método *execute_command*

```
124     def execute_scenario(self, feature_name, scenario):
125         """This Method will execute only a specific scenario
126         :param feature_name: define the feature that contains this scenario
127         :param scenario: contains a key to get a scenario
128         :return: a json file with the trace.
129         """
130         print('Executing Scenario: ', scenario.scenario_title)
131 
132         self.execute_command("bundle -v")
133         self.execute_command("bundle exec rake cucumber FEATURE=" + feature_name + ":" +
134                              str(scenario.line))
```

(b) Uso do método *execute_command*

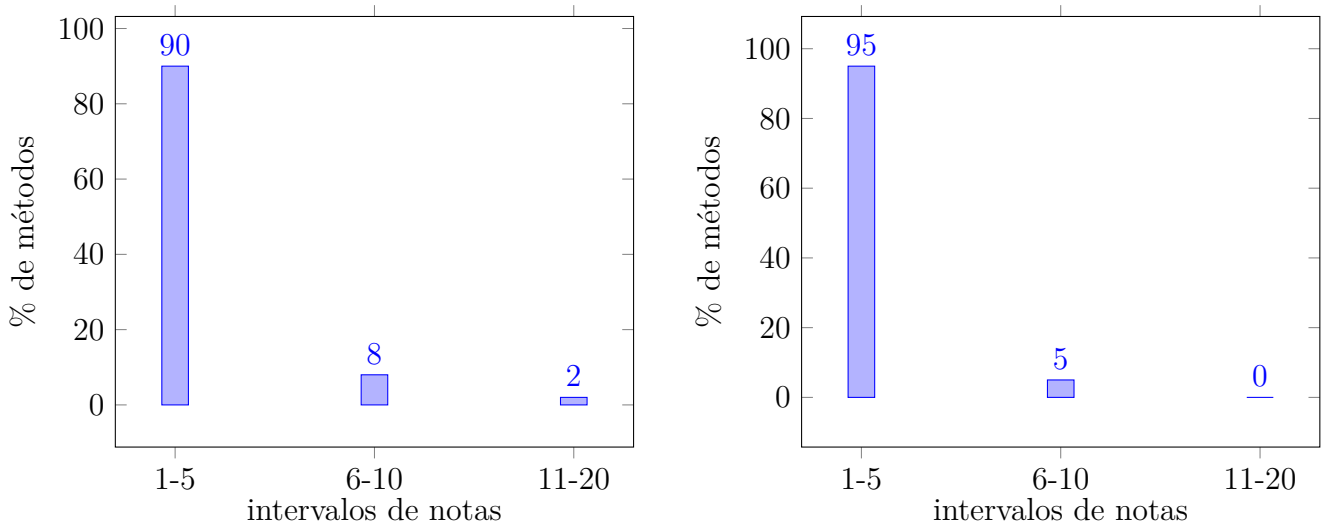
Figura 4.1: Exemplo de encapsulamento de código

4.2 Melhoria das métricas de código

Como sugerido na Seção 3.5, a análise das métricas de código *ABC score* e Complexidade Ciclomática foram um parâmetro de referência para as refatorações aplicadas. Assim, ao aplicar modificações que aprimorem essas métricas, espera-se também uma diminuição da complexidade do código.

4.2.1 Complexidade Ciclomática

Após todas as refatorações aplicadas, foi analisado novamente o valor de Complexidade Ciclomática para cada método do software, com auxílio da ferramenta *Radon*. A Figura 4.2 a seguir resume o cenário antes e depois das modificações, retratando a porcentagem de métodos do ARM presentes em cada intervalo de notas de Complexidade Ciclomática.



(a) Antes

(b) Depois

Figura 4.2: Comparativo de Complexidade Ciclomática antes e depois das contribuições

Nota-se que a porcentagem de métodos com nota entre 1 a 5 aumentou em 5%, de forma relacionada, a porcentagem de métodos com notas entre 6 a 10 e 11 a 20 diminuíram. A porcentagem de métodos com notas entre 6 a 10 foi reduzida em 3% e entre 11 a 20 foi zerada. Isso pode ser justificado pelo esforço aplicado em refatorar métodos complexos em outros mais encapsulados, o que contribui para um código menos complexo e de manutenção menos custosa.

4.2.2 *ABC score*

De modo similar, também pode ser feita uma comparação do *ABC score* após aplicadas refatorações nos módulos do Feature-Trace. Como sugerido pela análise prévia desta métrica, baseada no limiar máximo de 20 por método, esforços foram direcionados principalmente na refatoração dos métodos *read_methods* (*ReadMethods*), *check_environment* (*RubyConfig*) e *trace*. Porém, como quase a totalidade do software recebeu alterações, também é válido analisar essa métrica globalmente.

A Tabela 4.2 a seguir mostra o *ABC score* por módulo após as refatorações realizadas.

Tabela 4.2: *ABC score* por módulos do Feature-Trace

Módulo	Magnitude antes	Magnitude depois
RubyExecution	138,1	110,1
ReadMethods	84,3	56,44
RubyConfig	75,7	43,1
GherkinParser	48,1	28,4
RubySpecExecution	17,1	15,9
SpecModels	11,7	7,0
Project	-	6,4
Method	-	9,8
SimpleScenario	-	9,8
Scenario	-	4,0
StepBdd	-	3,0
Feature	-	9,0

Comparando-se com os valores iniciais antes das mudanças aplicadas, mostrados pela comparação da Tabela 4.2, nota-se uma diminuição do *ABC score* de cada módulo, além também dos valores baixos para os novos módulos criados.

De forma mais específica, os valores de *ABC score* para os métodos *read_methods* (*ReadMethods*), *check_environment* (*RubyConfig*) e *trace* também diminuíram, como mostra a Tabela 4.3 a seguir.

Tabela 4.3: Métodos considerados complexos de acordo com o *ABC score* antes das modificações

Módulo	Método	Magnitude
ReadMethods	read_methods	18,9
RubyConfig	*check_environment*	22,1
-	*trace*	23,7

De acordo com a tabela e comparando-a com a 3.5 da seção anterior, foi possível também diminuir o *ABC score* destes métodos. Porém, vale notar que apenas para o método *read_methods* ficou abaixo do valor 20 de limiar máximo considerado na análise inicial.

De modo geral, foi importante a redução da magnitude de *ABC score* nos diversos módulos e métodos do Feature-Trace. Isso entrega ao software estruturas de código mais coesas e com menos trocas de contextos ou fluxos alternativos.

4.3 Acoplamento e coesão

Após as refatorações aplicadas, é possível analisar novamente a coesão dos módulos do ARM por meio do *Pydeps*. A Figura 4.3 mostra esse cenário por meio do grafo.

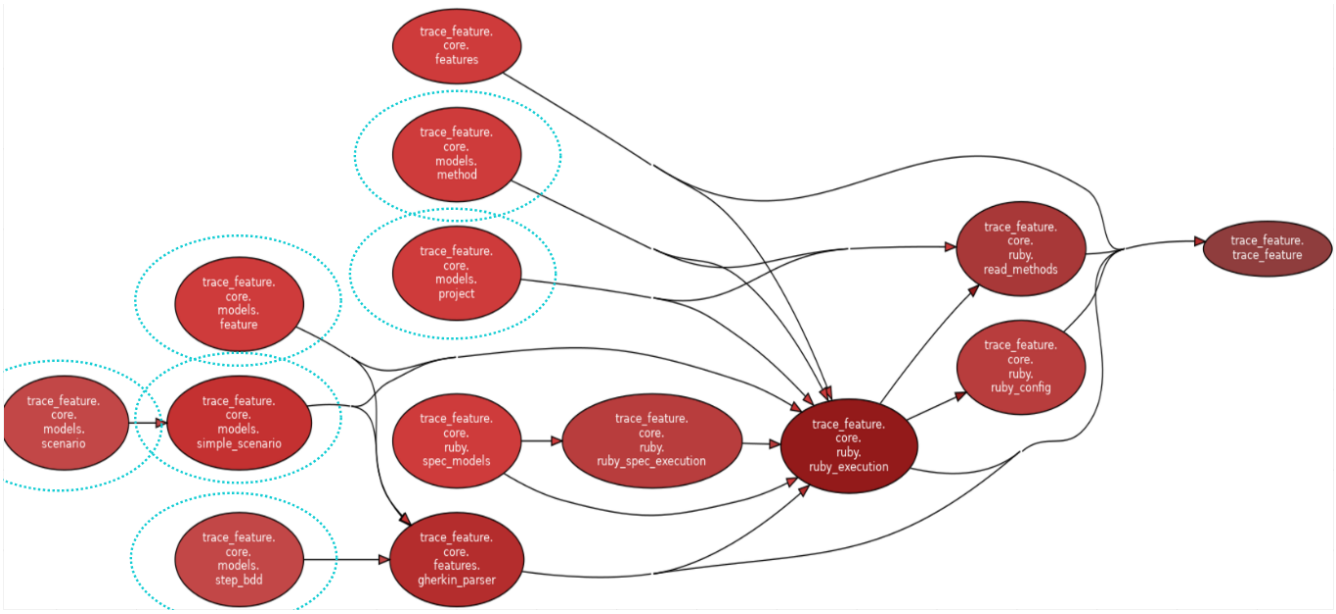


Figura 4.3: Acoplamento e coesão após refatorações

Nota-se que foi possível diminuir o acoplamento do antigo módulo *Models* em relação a outros, por meio de sua subdivisão em outros módulos menores e mais particulares, destacados pelo pontilhado azul na Figura 4.3. Desse modo, cada um deles é importado por no máximo dois outros módulos distintos. Por outro lado, os módulos restantes não puderam ser mais subdivididos, ao constatar-se que já possuíam uma responsabilidade bem definida.

4.4 Testes e cobertura de testes

Com o auxílio do *framework* de teste *Pytest*, foi possível implementar testes unitários para cada um dos módulos existentes do Feature-Trace. A seguir, é detalhada a execução da suíte de testes para cada módulo, juntamente com seu percentual de cobertura atingido.

GherkinParser

Foi possível cobrir 100% do código implementado por esse módulo. Seu módulo de teste correspondente, o *TestGherkinParser* implementa um teste de caracterização para cada método do *GherkinParser*. Esse módulo de teste conta com o auxílio de um arquivo *.feature* de exemplo.

Project, *Feature*, *SimpleScenario*, *Method* e *StepBdd*

Estes módulos implementam classes que auxiliam em todo o fluxo de análise do FeatureTrace, e por sua vez, tiveram seus métodos testados e 100% cobertos pelas classe de teste *TestFeatureInstance*, *TestMethodInstance*, *TestProjectInstance*, *TestSimpleScenarioInstance* e *TestStepBddInstance*.

ReadMethods

Este módulo teve todos seus métodos testados pelo módulo de teste *TestReadMethods*, porém algumas linhas de código do método *send_all_methods* não puderam ser cobertas pelo teste *test_send_all_methods*, pois o método apresenta um bloco de exceção, que não é trivial de ser simulado. Apesar disso, apresentou uma boa cobertura igual a 94%.

Este módulo de teste conta com um arquivo de métodos proveniente de um projeto *Ruby on Rails* como *input* para os testes realizados.

RubyConfig

Com todos os seus métodos testados pelo módulo de teste *TestRubyConfig*, este módulo apresentou uma cobertura muito satisfatória e igual a 97%.

Este módulo conta com um arquivo *Gemfile* e *env.rb* importados de um projeto *Ruby on Rails* para a simulação dos testes implementados.

RubySpecExecution

A partir de todos seus métodos testados pelo módulo de teste *TestRubySpecExecution*, este módulo alcançou uma cobertura completa de 100%.

Além disso, conta com a presença do arquivo *user_spec.rb* importado de um projeto *Ruby on Rails*, que fornece casos de teste exemplo para a análise e simulação de parte da súite de testes de um software terceiro em análise.

SpecModels

Por meio da implementação de testes pelo módulo *TestSpecModels*, foi possível cobrir 100% de código do módulo *SpecModels*.

RubyExecution

O módulo *RubyExecution* funciona como uma espécie de centralizador da execução do software, apresentando métodos que chamam outros métodos da aplicação. Sendo assim, alguns métodos não puderam ser refatorados para proporcionar maior testabilidade.

Apesar disso, o módulo de teste *TestRubyExecution*, implementou testes para metade do módulo testado, resultando em uma cobertura de 55%. Tal cobertura está abaixo de uma cobertura que se possa ser considerada satisfatória, sendo portanto um bom indicativo de futuras contribuição para a continuação deste trabalho.

Cobertura total

Levando em consideração todos os módulos de teste, a suíte de testes criada possui 70 testes implementados e proporcionou uma cobertura de código igual a 84%, como mostra a Figura 4.3. Uma vez que a aplicação não possuía testes implementados, essa cobertura representa uma boa entrega de valor à ferramenta Feature-Trace. Por outra perspectiva, *branches* e decisões lógicas importantes foram cobertas, sendo bastantes relevante por contemplarem funcionalidades chave para o funcionamento do software.

```
----------- coverage: platform linux, python 3.9.5-final-0 -----------
Name                                                                                                 Stmts   Miss  Cover
------------------------------------------------------------------------------------------------------------------
env/lib/python3.9/site-packages/trace_feature/core/base_config.py                                        4      0   100%
env/lib/python3.9/site-packages/trace_feature/core/base_execution.py                                     8      0   100%
env/lib/python3.9/site-packages/trace_feature/core/features/gherkin_parser.py                           58      0   100%
env/lib/python3.9/site-packages/trace_feature/core/models/feature.py                                    15      0   100%
env/lib/python3.9/site-packages/trace_feature/core/models/method.py                                     19      0   100%
env/lib/python3.9/site-packages/trace_feature/core/models/project.py                                    14      0   100%
env/lib/python3.9/site-packages/trace_feature/core/models/scenario.py                                   11      0   100%
env/lib/python3.9/site-packages/trace_feature/core/models/simple_scenario.py                            23      0   100%
env/lib/python3.9/site-packages/trace_feature/core/models/step_bdd.py                                    5      0   100%
env/lib/python3.9/site-packages/trace_feature/core/ruby/read_methods.py                                127      7    94%
env/lib/python3.9/site-packages/trace_feature/core/ruby/ruby_config.py                                  128      4    97%
env/lib/python3.9/site-packages/trace_feature/core/ruby/ruby_execution.py                               218     99    55%
env/lib/python3.9/site-packages/trace_feature/core/ruby/ruby_spec_execution.py                           27      0   100%
env/lib/python3.9/site-packages/trace_feature/core/ruby/spec_models.py                                   13      0   100%
------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                                   670    110    84%
```

Figura 4.4: Cobertura de testes na ferramenta

Capítulo 5

Conclusão

O objetivo principal deste trabalho consistia em tornar a ferramenta Feature-Trace mais manutenível e testável. Para cumprir tal proposta, foram seguidas importantes referências da literatura de Engenharia de Software, que sugerem técnicas e parâmetros para desenvolver a avaliar o software a respeito desses pontos.

As refatorações aplicadas aumentaram o grau de manutenibilidade, isto é, a facilidade com que a ferramenta pode ser alterada e evoluída por outros desenvolvedores. Desse modo, a ferramenta também segue padrões de codificação importantes da linguagem, que permitem garantir uma boa qualidade de código.

Não somente isso, com uma melhora também da testabilidade do software, uma suíte de testes foi implementada e automatizada utilizando o *framework Pytest*, disponível para linguagem *Python*.

Para trabalhos futuros, o acoplamento e coesão dos módulos pode ser mais explorado, a fim de obter módulos mais específicos e focados em uma responsabilidade. É interessante também buscar a melhoria dos critérios de cobertura dos testes, além de também dedicar esforços em tornar testáveis e mais encapsulados os métodos complexos restantes. Na mesma medida em que, refatorações serão sempre bem vindas e úteis durante a expansão dessa ferramenta.

Referências

[1] Cucumber: *Gherkin reference*. `https://cucumber.io/docs/gherkin/reference/#keywords.html`, acesso em 2021-10-11. ix, 5

[2] Fazzolino, R. e Rodrigues: *Feature-trace: Generating operational profile and supporting testing prioritization from bdd features*. página 22, 2019. ix, 2, 6, 7

[3] Bass, Len, Paul Clements e Rick Kazman: *Software Architecture in Practice*. Addison-Wesley Professional, fourth edição, 2021. ix, 1, 8, 12, 13, 14, 17, 19

[4] Fox, Armando e David Patterson: *Engineering Software as a Service: An Agile Approach Using Cloud Computing Second Edition*. 2021. 1, 8, 9, 10, 11, 12, 17, 19, 20, 21

[5] *Ieee standard glossary of software engineering terminology*. IEEE Std 610.12-1990, páginas 1–84, 1990. 1, 8

[6] Fowler, Martin, Kent Beck, John Brant, William Opdyke e Don Roberts: *Refactoring: Improving the Design of Existing Code*. janeiro 1999. 1, 11, 14, 22

[7] Shahid, Dr Muhammad, Suhaimi Ibrahim e Mohd Mahrin: *A study on test coverage in software testing*. janeiro 2011. 1

[8] Musa, J. D.: *The operational profile in software reliability engineering: an overview. in proceedings third international symposium on software reliability engineering*. IEEE Computer Society, páginas 140—-141, 1992. 4

[9] Wynne M., Hellesoy A. e Tooke S.: *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017. 4

[10] Smart, John Ferguson: *Bdd in action*. Manning, 2014. 5

[11] Gherkin: *Gherkin sintax*. `https://cucumber.io/docs/gherkin/`, acesso em 2021-10-14. 5

[12] ISO/IEC/IEEE: *Systems and software engineering – architecture description*. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000), páginas 1 –46, janeiro 2011. 8

[13] B. P. Lientz, E. B. Swanson e G. E. Tompkins: *Characteristics of application software maintenance. Communications of the ACM*. 1978. 9

[14] Feathers, Michael C.: *Trabalho Eficaz com Código Legado*. Bookman, 2013. 9, 11, 17, 24

[15] McCabe, T.J.: *A complexity measure*. IEEE Transactions on Software Engineering, SE-2(4):308–320, 1976. 10

[16] Fitzpatrick, Jerry: *Applying the abc metric to c, c++, and java*. 1997. 10

[17] Malaiya, Yashwant, Michael Li, James Bieman, Senior Member e Rick Karcich: *Software reliability growth with test coverage*. IEEE Transactions on Reliability, 51, fevereiro 2003. 12

[18] Zhu, Hong, Patrick Hall e J. May: *Software unit test coverage and adequacy*. ACM Computing Surveys - CSUR, janeiro 2000. 12

[19] Myers, Glenford J., Corey Sandler e Tom Badgett: *The art of software testing*. John Wiley & Sons, 3rd ed edição, 2012. 15, 17